

# Flow Delegation: Flow Table Capacity Bottleneck Mitigation for Software-defined Networks

zur Erlangung des akademischen Grades eines  
Doktors der Ingenieurwissenschaften

von der KIT-Fakultät für Informatik  
des Karlsruher Instituts für Technologie (KIT)  
genehmigte

Dissertation

von

Dipl.-Inform. Robert Bauer  
aus Eckernförde

Tag der mündlichen Prüfung: 18. Mai 2020

Erste Referentin: Professor Dr. Martina Zitterbart  
Karlsruher Institut für Technologie (KIT)

Zweiter Referent: Professor Dr. Wolfgang Kellerer  
Technische Universität München (TUM)



This document is licensed under a Creative Commons Attribution 4.0 International License (CC BY 4.0): <https://creativecommons.org/licenses/by/4.0/deed.en>

## Danksagung

---

Allen voran gebührt mein Dank Frau Prof. Dr. Martina Zitterbart, die mir durch die Anstellung am Institut für Telematik und als Doktormutter die Möglichkeit zum Anfertigen meiner Thesis gegeben hat. Ich habe die anregenden Diskussionen genossen, und auch die vielen Freiheiten, die mir bei der Ausgestaltung meiner Forschung eingeräumt wurden. Ferner gebührt mein Dank Herrn Prof. Dr. Wolfgang Kellerer, der sich sofort bereit erklärte, das Korreferat für meine Promotion zu übernehmen. Besonders dankbar bin ich außerdem dafür, dass meine beiden Betreuer und die KIT-Fakultät für Informatik dafür gesorgt haben, dass ich mein Promotionsvorhaben trotz der Covid-19-Krise wie geplant abschließen konnte.

Mein Dank gilt ferner meine Kolleginnen und Kollegen, die sich für keine Diskussion zu schade waren. Meinen zahlreichen Bachelor- und Masterarbeitern, die mich bei meinen Forschungen unterstützten. Meinen Freunden, die mich auf andere Gedanken brachten, wenn es darauf ankam.

Mein ganz spezieller Dank gilt Matthias Flittner – und zwar nicht nur für das viele, viele Korrekturlesen. Wenn es etwas gab, das ich in meiner Zeit am Institut mehr genossen habe, als alles andere, dann die Ehre, mit Matthias zusammenarbeiten zu dürfen. Also FliTTi (mit zwei großen T): Es war mir die größTTe Freude mir dir zu forschen, zu reisen, zu lehren, zu leiden, zu philosophieren und herumzualbern und ich hoffe inständig, dass ich irgendwann wieder die Gelegenheit dazu bekomme.

Ich danke Hans-Christian Bandholz und Leonard Masing, die immer ein offenes Ohr für meine Probleme hatten. Roland Bless, der meine Diplomarbeit betreute und mich ans Institut holte. Hauke Heseding, der mich immer motivierte und unterstützte. Hans Wippel, der mir in der Anfangszeit zur Seite stand. Florian Martin, der mir ein Vorbild war (und mir seine Wohnung in Karlsruhe überließ, als er nach Berlin zog). Außerdem danke ich Herrn Heinz Wüstenberg, der mir durch seine Stiftung das Studium der Informatik ermöglicht hat. Und ich danke natürlich auch all meinen anderen Kollegen und Freunden. Nicht nur für eure langjährige Unterstützung, sondern vor allem auch für euer Vertrauen und die vielen freundlichen Gespräche und Ratschläge.

Zuletzt möchte ich meiner Familie danken. Vor allem meiner Mutter und meinen beiden Schwestern und ihren Familien. Ihr allein habt mir die Kraft gegeben, die nötig war, um mein Studium und später dann die Promotion erfolgreich abzuschließen. Dafür stehe ich tief in eurer Schuld.

Robert Bauer  
Karlsruhe, 27.07.2020





# Contents

---

<b>Introduction and Background</b>	<b>1</b>
<b>1 Introduction</b>	<b>3</b>
1.1 Software-defined Networking . . . . .	4
1.2 Problem Statement . . . . .	5
1.3 Use Cases and Benefits . . . . .	6
1.4 Flow Delegation . . . . .	8
1.4.1 Goals . . . . .	9
1.4.2 Example Scenario . . . . .	9
1.4.3 Core Mechanism . . . . .	10
1.4.4 Workflow . . . . .	12
1.5 Structure and Challenges . . . . .	13
1.5.1 System Design . . . . .	13
1.5.2 Algorithms . . . . .	15
1.5.3 Evaluation . . . . .	17
1.6 Main Contributions . . . . .	17
1.7 Outline . . . . .	19
<b>2 Background</b>	<b>21</b>
2.1 General Definitions . . . . .	21
2.1.1 Time Slots . . . . .	21
2.1.2 Delegation and Remote Switch . . . . .	23
2.2 Software-defined Networking (SDN) . . . . .	24
2.2.1 Definitions . . . . .	24
2.2.1.1 Packet Header Field . . . . .	24
2.2.1.2 Flow . . . . .	25
2.2.1.3 Software-defined Networking . . . . .	27
2.2.2 SDN Architecture and Interfaces . . . . .	28
2.2.2.1 Northbound Interface . . . . .	29
2.2.2.2 Southbound Interface . . . . .	30
2.2.3 Application Areas . . . . .	31
2.2.4 Important Concepts of SDN . . . . .	31
2.2.4.1 Match . . . . .	31
2.2.4.2 Action . . . . .	32

2.2.4.3	Flow Rule . . . . .	32
2.2.4.4	Flow Table . . . . .	35
2.2.4.5	Remote Control . . . . .	38
2.2.4.6	Proactive and Reactive Flow Rule Installation . . . . .	40
2.2.4.7	Packet Processing inside the Switch . . . . .	42
2.3	Flow Table Capacity . . . . .	43
2.3.1	Some Numbers for Hardware SDN Switches . . . . .	43
2.3.2	High Demand for Flow Rules . . . . .	44
2.3.2.1	Independent Actions . . . . .	45
2.3.2.2	Multistage Processing . . . . .	45
2.3.2.3	Fine-grained Control and Visibility . . . . .	46
2.3.3	Conclusion . . . . .	46
2.4	Flow Table Capacity Bottlenecks . . . . .	46
2.4.1	Bottleneck Definition . . . . .	46
2.4.2	Existing Research on Flow Table Capacity Bottlenecks . . . . .	48
2.4.2.1	Resource-aware Network Applications . . . . .	48
2.4.2.2	Flow Rule Eviction . . . . .	49
2.4.2.3	Software Offloading . . . . .	50
2.4.2.4	Flow Rule Distribution . . . . .	52
2.4.2.5	Flow Table Compression . . . . .	53
2.4.3	Further Reading . . . . .	54
2.4.4	Summary . . . . .	54

## **System Design** **57**

<b>3</b>	<b>Architecture and Abstractions</b>	<b>59</b>
3.1	Architecture . . . . .	60
3.2	Core Concepts and Abstractions . . . . .	61
3.2.1	Aggregation Rules . . . . .	63
3.2.2	Aggregation Example . . . . .	63
3.2.3	Symbolic Set and Cover Set . . . . .	66
3.2.4	Rule Conflict Problem . . . . .	67
3.2.5	Conflict-free Cover Set . . . . .	69
3.2.6	Calculating the Conflict-free Cover Set . . . . .	71
3.2.7	Delegation Templates . . . . .	73
3.3	Conclusion . . . . .	75
<b>4</b>	<b>Monitoring System</b>	<b>77</b>

---

4.1	Overview . . . . .	78
4.1.1	Static Parameters . . . . .	79
4.1.2	Time Slot Dependent Parameters . . . . .	80
4.2	Monitoring Approach . . . . .	82
4.2.1	Translation from Timestamps to Time Slots . . . . .	83
4.2.2	Lambda Notation . . . . .	85
4.2.3	Further Translations . . . . .	86
4.3	Conclusion . . . . .	88
<b>5</b>	<b>Rule Aggregation Scheme</b>	<b>89</b>
5.1	Different Approaches for the Rule Aggregation Scheme . . . . .	90
5.1.1	New Controller Functionality . . . . .	90
5.1.2	Based on Monitored Parameters . . . . .	90
5.1.3	Inferred from Context . . . . .	91
5.2	Indirect Rule Aggregation Schemes . . . . .	92
5.2.1	Prerequisites . . . . .	93
5.2.2	Scheme Examples . . . . .	94
5.2.2.1	Tagging Scheme . . . . .	94
5.2.2.2	Slicing Scheme . . . . .	95
5.2.2.3	Ingress Port Scheme . . . . .	96
5.2.2.4	Summary . . . . .	97
5.2.3	Algorithm for Indirect Rule Aggregation . . . . .	97
5.3	Remarks on Aggregation Priority . . . . .	99
5.4	Conclusion . . . . .	100
<b>6</b>	<b>Detour Procedure</b>	<b>103</b>
6.1	Flow Rule Terminology . . . . .	105
6.2	Packet-level Metadata . . . . .	105
6.2.1	Flow Delegation Indicators . . . . .	106
6.2.1.1	Remote Rule Indicator . . . . .	106
6.2.1.2	Ingress Port Indicator . . . . .	108
6.2.1.3	Backflow Indicator . . . . .	108
6.2.2	Transport Packet Header Fields . . . . .	109
6.2.2.1	Index function . . . . .	109
6.2.2.2	Transport Options . . . . .	110
6.3	Detour Procedure . . . . .	111
6.4	Control Message Generation . . . . .	114
6.4.1	State Management . . . . .	115
6.4.2	Handling of Backflow Rules . . . . .	115
6.4.3	Handling of Aggregation Rules . . . . .	117

6.4.4	Handling of Remote Rules . . . . .	119
6.5	Conclusion . . . . .	121
<b>7</b>	<b>Control Message Interception</b>	<b>123</b>
7.1	Proxy Layer . . . . .	125
7.1.1	Periodic Part . . . . .	126
7.1.2	Asynchronous Part . . . . .	127
7.2	Control Message Processors . . . . .	127
7.2.1	Interface . . . . .	128
7.2.2	XID Management . . . . .	129
7.3	Processor Logic for Flow Delegation . . . . .	129
7.3.1	Conflict Resolution . . . . .	130
7.3.2	Required State . . . . .	131
7.3.3	Installation of a New Flow Rule . . . . .	131
7.3.4	Update of an Existing Flow Rule . . . . .	133
7.3.5	Asynchronous Event from a Switch . . . . .	134
7.4	Conclusion . . . . .	136
<b>8</b>	<b>Prototype Implementations</b>	<b>137</b>
8.1	Existing Prototypes . . . . .	138
8.1.1	Middleware Prototype . . . . .	139
8.1.2	Proxy Prototype . . . . .	140
8.1.3	Other Prototypical Work . . . . .	140
8.2	Functional Evaluation of the Prototypes . . . . .	141
8.2.1	Setup . . . . .	141
8.2.2	Result . . . . .	142
8.3	Monitoring Overhead . . . . .	145
8.4	Detour Overhead . . . . .	146
8.4.1	Setup . . . . .	146
8.4.2	Result . . . . .	147
8.5	Conclusion . . . . .	148
<b>Algorithms</b>		<b>149</b>
<b>9</b>	<b>Decomposition and Problem Formulations</b>	<b>151</b>
9.1	Requirements . . . . .	153
9.2	Preliminaries . . . . .	154
9.3	Problem Decomposition . . . . .	155

9.4	Delegation Template Selection (DT-Select) . . . . .	157
9.4.1	Modeling Approach . . . . .	157
9.4.2	Decision Variables . . . . .	158
9.4.3	Utilization Coefficients . . . . .	158
9.4.4	Cost Coefficients . . . . .	160
9.4.4.1	Table Overhead . . . . .	160
9.4.4.2	Link Overhead . . . . .	161
9.4.4.3	Control Message Overhead . . . . .	161
9.4.5	Single Period Problem . . . . .	162
9.4.6	Multi Period Problem . . . . .	164
9.4.7	Problem Analysis for DT-Select . . . . .	165
9.5	Remote Switch Allocation (RS-Alloc) . . . . .	168
9.5.1	Modeling Approach and Additional Terminology . . . . .	168
9.5.1.1	Allocation Job . . . . .	169
9.5.1.2	Remote Set . . . . .	171
9.5.2	Decision Variables . . . . .	172
9.5.3	Utilization Coefficients . . . . .	172
9.5.4	Cost Coefficients . . . . .	174
9.5.4.1	Flow Table Capacity . . . . .	174
9.5.4.2	Link Capacity . . . . .	175
9.5.4.3	Control Messages . . . . .	175
9.5.4.4	Static Cost . . . . .	175
9.5.5	Single Period Problem . . . . .	176
9.5.6	Multi Period Problem . . . . .	179
9.5.7	Problem Analysis for RS-Alloc . . . . .	179
9.6	Conclusion . . . . .	181
<b>10</b>	<b>Multi Period Delegation Template Selection</b>	<b>183</b>
10.1	Problem . . . . .	184
10.2	DT-Select with Assignments (Select-Opt) . . . . .	185
10.2.1	General Idea . . . . .	185
10.2.1.1	Assignment-based Decision Variables . . . . .	185
10.2.1.2	Periodic Optimization . . . . .	188
10.2.2	Decision Variables . . . . .	192
10.2.3	Utilization Coefficients . . . . .	193
10.2.4	Cost Coefficients . . . . .	195
10.2.4.1	Rule Overhead . . . . .	198
10.2.4.2	Link Overhead . . . . .	199
10.2.4.3	Control Message Overhead . . . . .	200
10.2.5	Problem Formulation for Select-Opt . . . . .	201

10.2.6	Algorithm for Select-Opt . . . . .	202
10.2.7	Summary . . . . .	204
10.3	DT-Select with Restricted Assignments (Select-CopyFirst) . . . . .	204
10.3.1	General Idea . . . . .	205
10.3.2	Decision Variables . . . . .	205
10.3.3	Utilization Coefficients . . . . .	206
10.3.4	Cost Coefficients . . . . .	207
10.3.4.1	Rule Overhead . . . . .	207
10.3.4.2	Link Overhead . . . . .	207
10.3.4.3	Control Message Overhead . . . . .	208
10.3.5	Problem Formulation for Select-CopyFirst . . . . .	208
10.3.6	Algorithm for Select-CopyFirst . . . . .	209
10.3.7	Summary . . . . .	210
10.4	DT-Select with Greedy Strategy (Select-Greedy) . . . . .	210
10.4.1	General Idea . . . . .	210
10.4.2	Utilization Coefficients . . . . .	211
10.4.3	Cost Coefficients . . . . .	212
10.4.4	Algorithm for Select-Greedy . . . . .	212
10.5	Conclusion . . . . .	214
<b>11</b>	<b>Multi Period Remote Switch Allocation</b>	<b>215</b>
11.1	Problem . . . . .	215
11.2	RS-Alloc with Assignments . . . . .	217
11.2.1	General Idea . . . . .	217
11.2.1.1	Additional Terminology . . . . .	217
11.2.1.2	Allocation Assignments . . . . .	219
11.2.1.3	Periodic Optimization . . . . .	221
11.2.2	Decision Variables . . . . .	222
11.2.3	Utilization Coefficients . . . . .	223
11.2.4	Cost Coefficients . . . . .	226
11.2.5	Problem Formulation for RS-Alloc . . . . .	227
11.2.6	Algorithm for RS-Alloc . . . . .	228
11.3	Pre-processing for Allocation Assignments . . . . .	229
11.3.1	General Idea . . . . .	230
11.3.2	Allocation Interval . . . . .	231
11.3.3	Stability Metric . . . . .	232
11.3.4	Pre processing Algorithm . . . . .	234
11.4	Conclusion . . . . .	237

<b>Evaluation</b>	<b>239</b>
<b>12 Preliminaries and Methodology</b>	<b>241</b>
12.1 Network Model and Assumptions . . . . .	242
12.2 Important Terminology . . . . .	244
12.2.1 Scenario . . . . .	244
12.2.2 Scenario Set . . . . .	245
12.2.3 Capacity Reduction Factor . . . . .	245
12.2.4 Failure Rate . . . . .	247
12.2.5 Flow Delegation Performance . . . . .	248
12.2.6 Flow Table Utilization Ratio . . . . .	249
12.2.7 Other Terminology . . . . .	249
12.3 Evaluation Methodology . . . . .	250
12.3.1 Scenario Generation . . . . .	251
12.3.1.1 Topology Generation . . . . .	252
12.3.1.2 Flow Rule Set Generation . . . . .	254
12.3.1.3 Examples . . . . .	260
12.3.2 Pre-Processing . . . . .	265
12.3.2.1 Parameter Selection . . . . .	265
12.3.2.2 Construction of Scenario Sets . . . . .	266
12.3.3 Experiment Execution . . . . .	269
12.3.3.1 Environment . . . . .	269
12.3.3.2 Execution of an Experiment Series . . . . .	270
12.3.3.3 Parameterization . . . . .	273
12.3.4 Post-Processing . . . . .	274
<b>13 Case Study</b>	<b>275</b>
13.1 Candidate Selection . . . . .	275
13.2 Description of Investigated Scenarios . . . . .	275
13.3 Experiment Setup . . . . .	279
13.4 Results . . . . .	280
13.4.1 Capacity Reduction and Failure Rate . . . . .	280
13.4.2 Flow Table Utilization Ratio . . . . .	284
<b>14 Performance</b>	<b>291</b>
14.1 Performance without Failures . . . . .	292
14.1.1 Experiment Setup . . . . .	292
14.1.2 Results . . . . .	293
14.2 Performance with Small Failure Rates . . . . .	296

14.2.1 Experiment Setup . . . . .	296
14.2.2 Results . . . . .	296
14.3 Performance with Different Utilization Ratios . . . . .	297
14.3.1 Experiment Setup . . . . .	298
14.3.2 Results . . . . .	300
14.4 Over- and Underutilization . . . . .	301
14.4.1 Experiment Setup . . . . .	302
14.4.2 Results for Overutilization . . . . .	305
14.4.3 Results for Underutilization . . . . .	307
<b>15 Overhead</b>	<b>313</b>
15.1 Experiment Setup . . . . .	314
15.2 Overhead Example . . . . .	315
15.3 Results for Table Overhead . . . . .	318
15.4 Results for Link Overhead . . . . .	320
15.5 Results for Control Overhead . . . . .	322
<b>16 Runtime and Scalability</b>	<b>325</b>
16.1 Runtime . . . . .	326
16.1.1 Experiment Setup . . . . .	326
16.1.2 Results for DT-Select . . . . .	326
16.1.3 Results for RS-Alloc . . . . .	331
16.2 Number of Delegation Templates . . . . .	333
16.2.1 Experiment Setup . . . . .	333
16.2.2 Results . . . . .	334
16.3 Number of Switches . . . . .	340
16.3.1 Experiment Setup . . . . .	340
16.3.2 Results . . . . .	342
<b>Conclusion and Appendices</b>	<b>345</b>
<b>17 Conclusion and Outlook</b>	<b>347</b>
17.1 Summary of Contributions . . . . .	348
17.2 Perspectives for Future Work . . . . .	349
<b>Appendices</b>	<b>351</b>
<b>A Parameter Study</b>	<b>353</b>
A.1 Impact of Look-ahead Factor on Algorithm Runtime . . . . .	353



---

A.2	Impact of Look-ahead Factor on Overhead . . . . .	361
A.3	Weights for Multi-Objective Optimization . . . . .	366
A.4	Used Parameters . . . . .	374
<b>B</b>	<b>Scenario Set Characteristics</b>	<b>375</b>
B.1	Parameter Distribution . . . . .	375
B.2	Correlation between Important Characteristics . . . . .	375
B.3	Examples for Bottlenecked Situations . . . . .	379
<b>C</b>	<b>Additional Performance Results</b>	<b>383</b>
C.1	Performance without Failures . . . . .	383
C.2	Performance with small Failure Rates . . . . .	386
<b>D</b>	<b>Reproducibility</b>	<b>389</b>
D.1	Code . . . . .	389
D.2	Datasets . . . . .	390
D.3	Dataset Details . . . . .	392
D.4	Scripts . . . . .	393
<b>E</b>	<b>Terminology</b>	<b>395</b>
	<b>Bibliography</b>	<b>401</b>



# Part I

## Introduction and Background



## Chapter 1

---

# Introduction

---

Software-defined Networking (SDN) is an ongoing trend to manage, control and optimize communication networks by means of software programmability. It has gained significant traction in recent years with many real world applications. Google's private backbone (B4), for example, is based on SDN since 2011 with great success [Jai+13; Hon+18]. The SDN paradigm has several advantages, but the most important one is increased flexibility [Jar+14; Kre+15]. Simply put: SDN allows it to add new functions to the network easier and faster. This is possible because higher level network control tasks such as routing, traffic engineering or access control are implemented as *network applications*, i.e., as a piece of software. Network applications are executed in a logically centralized SDN controller connected to a set of SDN switches which can be remotely programmed based on centralized decisions.

Each SDN switch is equipped with a flow table where decisions from network applications are stored in the form of so-called flow rules. Larger networks and more complex tasks need more flow rules. Reactive network applications, for example, create flow rules in response to events from the network such as number of active users or active connections which can lead to high flow table utilization. The maximum capacity of such a flow table, however, is limited because the required TCAM chips (Ternary Content Addressable Memory) are expensive, have a high energy footprint and can therefore only be used in small quantities [Ars+18].

Limited flow table capacity is a well known scalability and performance limitation for SDN that was intensively studied [Cur+11; YTG13; JMD14; Kat+14b]. But despite the fact that SDN is around for more than a decade, the situation has not yet changed significantly. Current hardware switches are still limited to a couple of thousand flow

table entries and capacity-related constraints are still present in the research community [Jan+19; Wan+19b; Li+19a].

Several approaches exist that address this limitation. The B4 engineers, for example, were forced to implement complex rule management strategies to deal with capacity limitations [Jai+13]. Other researchers have proposed compression schemes to reduce the number of flow rules before they are installed [KB13] or caching strategies where only the most important flow rules are kept in the hardware switch [Kat+14b]. However, most existing approaches make changes to the infrastructure [Cur+11; Yu+10; Kat+14b] or require that network applications are aware of the performed optimization [KB13; Len+15; KHK13]. These solutions are not well suited for scenarios where bottlenecks only occur in parts of the network or within certain time frames (or exceed the available capacity only slightly). In summary: it is difficult to utilize existing countermeasures for limited flow table capacity in an *on-demand* fashion without side effects regarding the controller, the network applications or even the design of the infrastructure itself.

This thesis presents a novel concept to address flow table capacity bottlenecks called **flow delegation** [BZ16; BD17; BDZ19]. Flow delegation can be used on-demand in a transparent fashion, without changes to network applications or any other part of the infrastructure – something that cannot be achieved today with existing solutions. Core idea is it to automatically relocate flow rules from a bottlenecked switch to neighboring switches with spare capacity.

The introduction is structured as follows. First, the SDN workflow is explained in Sec. 1.1, followed by the problem statement in Sec. 1.2, and the two main use cases in Sec. 1.3. Flow delegation is introduced in Sec. 1.4. The structure of the thesis and the addressed challenges are discussed in Sec. 1.5. The contributions are summarized in Sec. 1.6. And Sec. 1.7 finally presents an outline of the rest of the document.

## 1.1 Software-defined Networking

This section briefly introduces the basic SDN workflow which makes it easier to understand the flow delegation approach outlined below. Fig. 1.1 shows a single network application and four SDN switches  $s_1$  to  $s_4$  connected to the SDN controller.

In step ①, the network application makes a high level control decision. This could be, for example, a routing decision for a pair of active users in the network. This decision is communicated towards the SDN controller (e.g., via a programming interface) that calculates one or more low level *flow rules* ②. These flow rules are sent to the switches in step ③ where they are stored locally inside a hardware flow table in step ④. The third step requires that control messages are exchanged between the controller and the

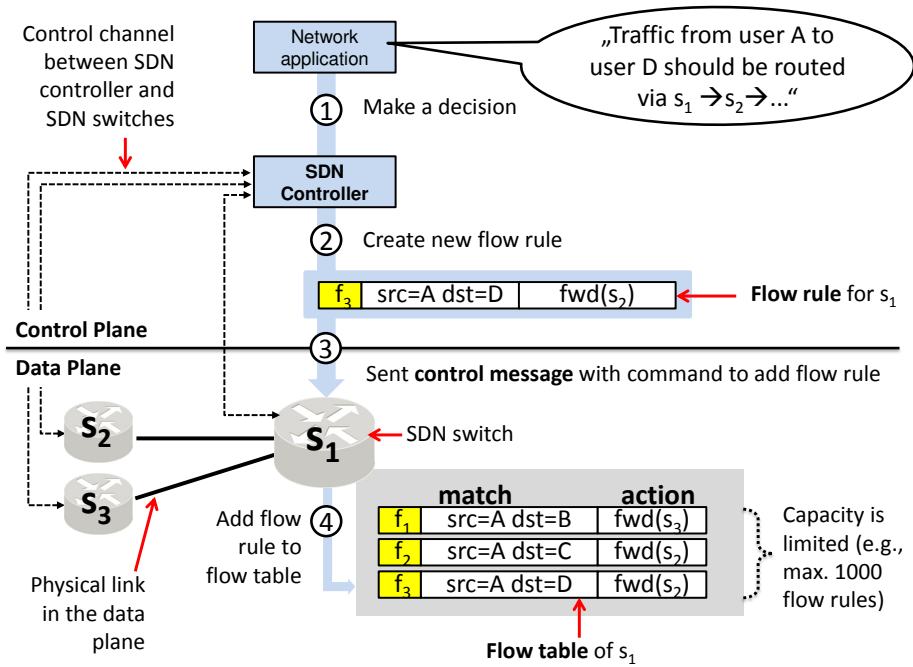


Figure 1.1: Basic SDN workflow

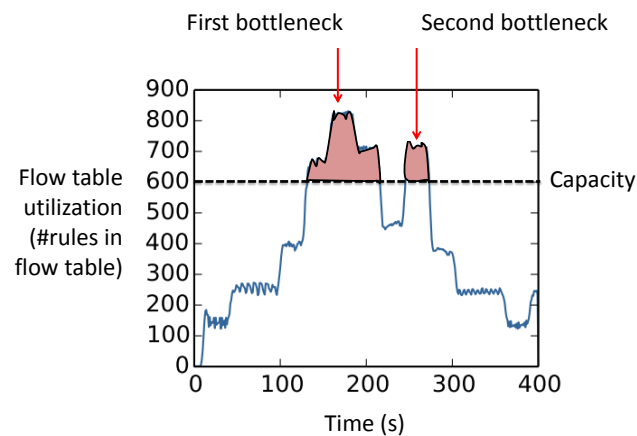
switches via a control channel. The number of flow rules that can be stored in the flow table are limited: current hardware SDN switches support between 1.000 and 5.000 flow rules [Ryg+17; Cos+17; CYP18; Van+19].

A flow rule defines how a group of similar packets (e.g., all packets with the same IP address) is processed by the switch. It therefore consists of two important parts. A match part that selects a group of packets and an action part that is executed for each matched packet. Flow rule  $f_3$  in Fig. 1.1, for example, selects every packet that is sent from user A to user D based on source and destination (IP) addresses and forwards them to switch  $s_2$ .

## 1.2 Problem Statement

An SDN switch suffers from a flow table capacity bottleneck if the amount of flow rules determined by the controller for that switch is higher than the capacity of its flow table. Without appropriate countermeasures, such a bottleneck can have a significant negative impact. In the best case, some of the installed flow rules are optional (e.g., for better monitoring quality) and the controller can mitigate the bottleneck in exchange for slightly decreased performance (e.g., less monitoring accuracy). In the worst case, the functionality of the network is affected which may result in loss of connectivity, security issues or violation of service agreements. Both cases will probably result in loss of revenue for the network operator.

A visual example of a (single) switch suffering from a flow table capacity bottleneck taken from [BZ16] is given in Fig. 1.2. It shows the utilization of the flow table over time measured for a period of 400 seconds. The experiment was executed inside an emulated network with a real SDN controller and 200 users (iperf sessions). The switch supports a maximum of 600 flow rules (the dashed black capacity line). The used network application creates two flow rules per active user, one per direction. More details regarding the setup can be found in [BZ16].



**Figure 1.2:** Example for a flow table capacity bottleneck from [BZ16]

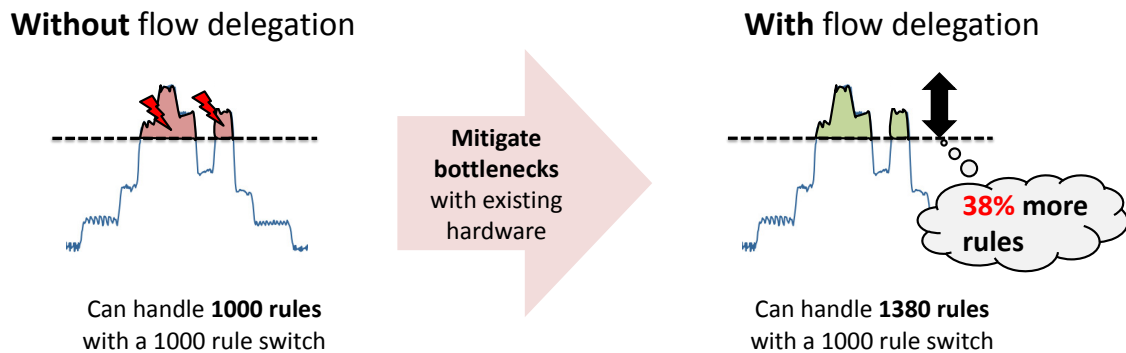
Within the first 130 seconds, the number of active users that communicate via the switch is small and the flow table provides sufficient capacity. But between the 130s and the 210s mark and between 250s and 270s, the number of users as well as the number of required flow rules increases which results in a bottleneck (red area). Such bottleneck situations can easily occur, e.g., when the number of users is unexpectedly high due to a popular video streaming event like the Red Bull Stratos jump [Pet14] or the SpaceX’s Falcon launch [Mic18].

In the following, a switch that suffers from a flow table capacity bottleneck is referred to as delegation switch. A switch that is connected to the delegation switch and has enough spare capacity to mitigate the bottleneck is called remote switch.

### 1.3 Use Cases and Benefits

Before flow delegation is explained in more detail, this section highlights two important use cases where network operators can expect benefits from using flow delegation. The **first use case is flow table capacity bottleneck mitigation with existing hardware** which is illustrated in Fig. 1.3.



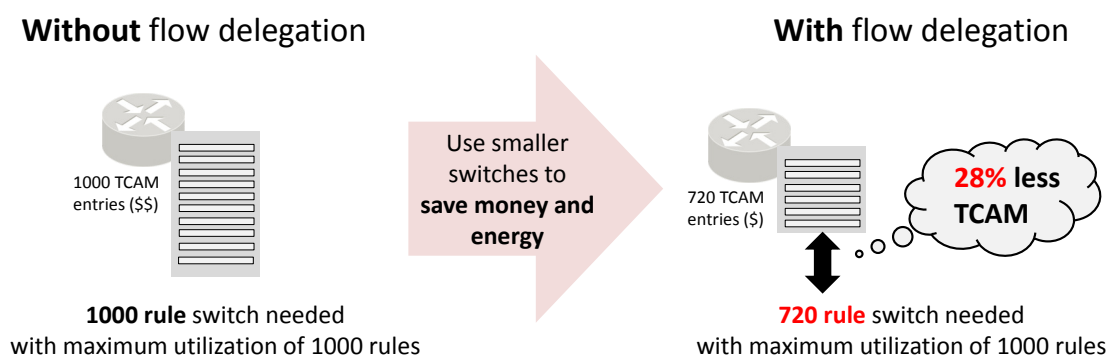


**Figure 1.3:** First use case for flow delegation

The left side represents the situation without flow delegation. The example assumes a network equipped with switches that support 1000 flow rules, i.e., the network fails when more than 1000 rules must be installed at the same time (red area). The situation with flow delegation is shown on the right side. In this case, the network will not fail if the flow table utilization exceeds the capacity as long as there is at least one remote switch with enough spare capacity that can store the excess flow rules (green area).

The thesis applies flow delegation to 5000 different bottleneck scenarios with various different characteristics and shows: flow delegation can mitigate bottlenecks which exceed the maximum flow table capacity by **up to 38%**. This means flow delegation can handle situations with 1380 concurrently installed flow rules while all (!) switches in the infrastructure have a capacity of 1000 rules.

The **second use case is reduction of operational and capital expenditure**, i.e., operators buy switches with less TCAM to save money and energy, illustrated in Fig. 1.4.



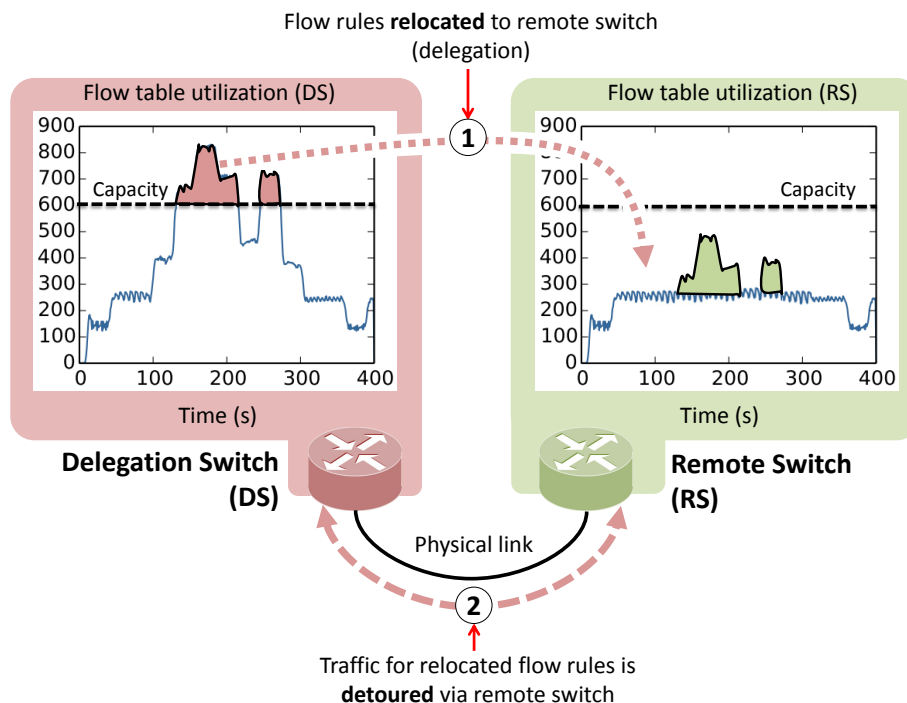
**Figure 1.4:** Second use case for flow delegation

The left side represents the situation without flow delegation where a network operator expects a maximum utilization of 1000 rules and buys switches with 1000 rules. With flow delegation, the network operator can include the mitigation potential of the flow delegation approach into the calculation and use cheaper and more energy efficient switches with smaller TCAM chips. It is shown in the thesis that flow delegation allows for a 28% reduction in TCAM size which can lead to significant savings.

Note that both values reported here – 38% and 28% – are median values and are thus not achieved in all scenarios. Scenarios with more free capacity achieve better results (up to 52% reduction of required TCAM capacity) while scenarios with limited free capacity perform worse (heavily depends on the scenario).

## 1.4 Flow Delegation

Given the two main use cases, this section will now further elaborate how flow delegation works. The basic idea is as follows: In case of a flow table capacity bottleneck, a subset of the flow rules of the bottlenecked delegation switch is “relocated” to a neighboring remote switch with spare capacity. If a flow rule is relocated, all traffic for that rule is redirected towards and processed within the remote switch (cf., Fig. 1.5).



**Figure 1.5:** *Flow delegation approach*

The red part on the left side shows the flow table utilization over time of the delegation switch with two bottlenecks (same as in Fig. 1.2). The green part on the right side shows the flow table utilization of the remote switch. Flow delegation will now **relocate some of the flow rules** from the left to the right – indicated by the red dashed arrow in the top of the figure labeled as ① – so that the two bottlenecks are avoided. The second red arrow in the bottom indicates that a portion of the overall traffic – all packets that have to be processed by the relocated rules – has to be **detoured via the remote switch** ②. This means some packets are sent to the remote switch, processed there, and sent back to the delegation switch afterwards. The detoured traffic can be seen as the price or overhead to be paid for flow delegation.

Further on in this section, it will be explained in more detail how the above concept works. First, the goals are briefly summarized followed by a simple example scenario. Afterwards, it is explained how flow rules can be relocated to a remote switch. The section concludes with an illustration of the workflow.

### 1.4.1 Goals

Flow delegation was designed with two goals in mind: it has to be applicable to existing SDN-enabled networks in an on-demand fashion and it must be possible to consider estimations of future network situations. This means flow delegation has to work with established protocols and the infrastructure – switches, controller, and network applications – cannot be changed. As a result, the flow delegation system has to be hidden from controller and network applications. If a control message (from the controller) deals with one of the installed flow rules, it should be executed as expected from the network application regardless of whether the flow rule is currently delegated to a remote switch or not.

### 1.4.2 Example Scenario

Consider the example network in Fig. 1.6 that consists of an SDN controller, a network application, four SDN switches, and six users ( $A, B, C, D, Y, Z$  – the grey monitor symbols).  $A$  to  $D$  are considered receivers,  $Y$  and  $Z$  are senders.

The red part on the left side shows the flow table of the delegation switch. The green part on the right side shows the flow table of the remote switch. The flow tables of switches  $s_3$  and  $s_4$  are not shown. The network application in this example creates a new flow rule every time there is a new connection between two users. Connection here means a user is sending one or more packets to another user for a period of time. The six flow rules  $f_1$  to  $f_6$  in the flow table of the delegation switch represent six connections. The yellow highlighting for rules  $f_3$  to  $f_6$  is used further below and can be ignored for now.

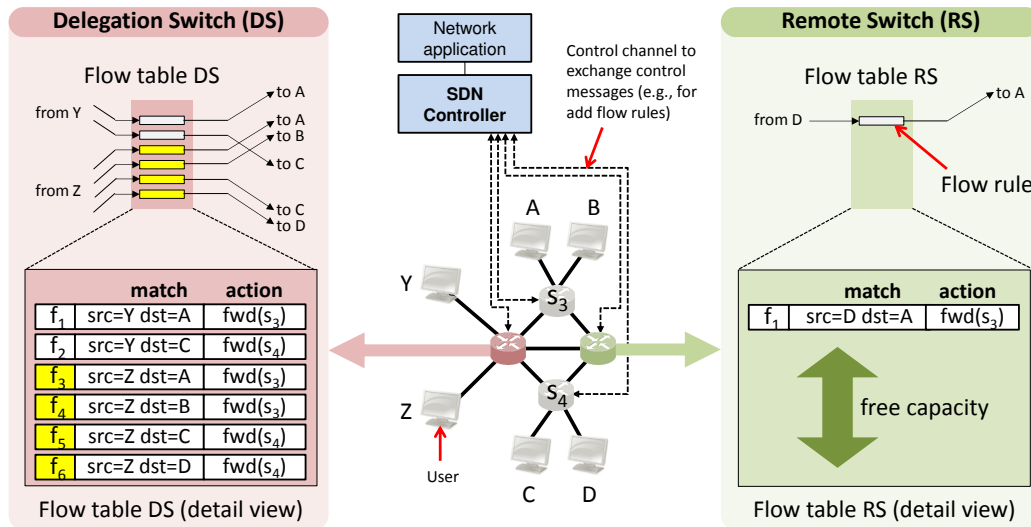


Figure 1.6: Example scenario

The match part of the rules selects all packets for a single connection and the action part determines the forwarding destination. The first rule  $f_1$ , for example, selects all packets sent from user  $Y$  to user  $A$  and the corresponding action forwards the packet to switch  $s_3$  (because user  $A$  is connected to switch  $s_3$ ). The second rule  $f_2$  selects packets from user  $Y$  to user  $C$  and forwards them to  $s_4$ . The other flow rules work the same way. Note that there is currently only a single flow rule in the remote switch.

Assume for demonstration purposes that the flow tables of all switches have a maximum capacity of six rules. This means the delegation switch is fully utilized and will run into a bottleneck if a new flow rule is required – for example if user  $Y$  starts sending packets to user  $B$ . Without countermeasures, the controller has only two options to deal with this situation: it can either deny installation of the new rule or remove one of the existing rules first. Both options lead to failing connectivity between one pair of users. Flow delegation avoids this by relocating some of the rules to the remote switch.

### 1.4.3 Core Mechanism

First recall that all flow rules in the delegation switch were determined by one of the network applications<sup>1</sup>. To make sure that flow delegation does not interfere with the logic of these network applications, it is required that all flow rules “remain as they are”, i.e., all packets are processed exactly as intended by the network applications.

<sup>1</sup>The example shows only one network application, but a real network will usually run multiple network applications

It is possible, however, to delegate rules to another switch without interfering with the network application logic. For this to work, a novel **detour procedure** – the core mechanism of flow delegation – is introduced that makes use of three classes of flow rules [BZ16]: aggregation rules, remote rules, and backflow rules.

**Aggregation rules** are installed in the delegation switch in order to forward traffic to the remote switch. To achieve this, they use a wildcard match that “covers“ a subset of the flow rules in the flow table (the so-called *cover set*). Every packet matched by a rule in the cover set is also matched by the aggregation rule. Consider the four flow rules  $f_3$  to  $f_6$  in the delegation switch of the above example (Fig. 1.6, the rules are highlighted in yellow). Now consider an aggregation rule that matches all packets sent from user Z regardless of the destination. This can be expressed with a wildcard for the destination part of the match, i.e.,  $src = Z, dst = *$ . It is easy to see that every packet that has to be processed by one of the yellow rules is also matched by this new wildcard rule.

**Remote rules** are installed in the remote switch. They represent the “delegated“ rules, i.e., the rules that were relocated from the delegation to the remote switch. They are basically a copy of the original rule from the delegation switch, with one small exception: The action part of the rule is modified so that i) a marker is attached to the packet that encodes the forwarding decision of the network application and ii) packets are sent back to the delegation switch after processing.

**Backflow rules** are installed in the delegation switch. They are used to forward the marked packets from a remote rule towards their correct destination. Because the maximum number of potential forwarding destinations is limited by the number of interfaces of the delegation switch, the number of backflow rules is also strictly limited to the same value.

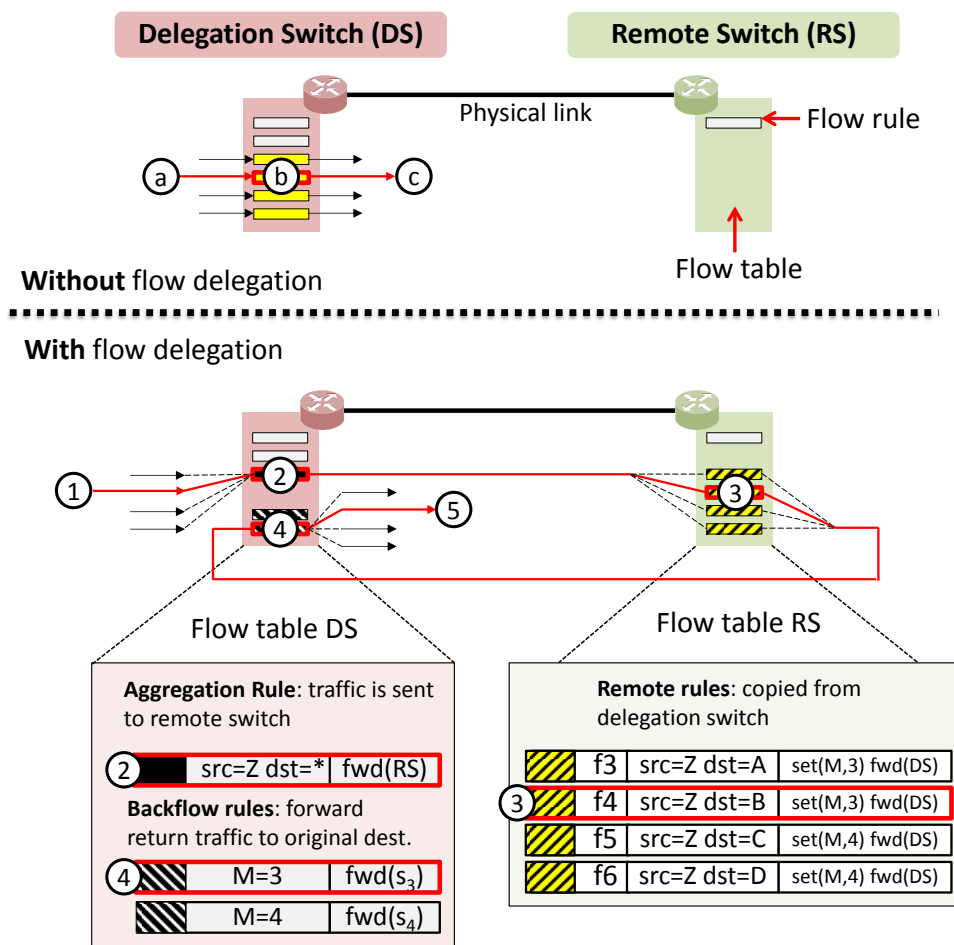
The detour procedure is then realized as follows: It first takes a set of rules from the delegation switch (e.g., the four yellow rules from Fig. 1.6) that can be covered by an aggregation rule. The rules from this so-called cover set are transformed into remote rules and installed in the remote switch. Next, the aggregation and backflow rules are installed in the delegation switch. Finally, the rules from the cover set are removed from the delegation switch which will decrease its flow table utilization<sup>2</sup>.

---

<sup>2</sup>Four rules are removed from the delegation switch. Because one aggregation rule and two backflow rules are required, the total utilization is only reduced by 1 in this trivial example. This is different in more realistic scenarios where the cover set can contain hundreds of rules while the number of aggregation and backflow rules stays small.

### 1.4.4 Workflow

Fig. 1.7 illustrates the flow delegation workflow. Consider a packet sent from user  $Z$  towards user  $B$  that is received at the delegation switch. The top of the figure shows how processing for this packet is done without flow delegation. Processing with flow delegation is shown at the bottom. In the first case (without flow delegation), the packet arrives at the delegation switch in step (a). It is matched by rule  $f_4$  (match:  $src = Z, dst = B$ ) in step (b). The action part of  $f_4$  is executed and the packet is forwarded to switch  $s_3$  in step (c). This is the normal processing workflow in SDN.



**Figure 1.7:** Flow delegation workflow example

In the second case (with flow delegation), the packet arrives at the delegation switch in step (1). Flow rule  $f_4$ , however, was relocated together with the other yellow flow rules to the remote switch. The four yellow rules were replaced with an aggregation rule

(match:  $\text{src} = Z, \text{dst} = *$ ). So this time, the packet is matched by the aggregation rule and sent towards the remote switch in step ②. After the packet arrived at the remote switch it is matched by the remote rule for  $f_4$  (match:  $\text{src} = Z, \text{dst} = B$ ) in step ③. The forwarding decision of the original rule is attached as a marker with  $\text{set}(M, 3)$  and the packet is sent back to the delegation switch. In step ④, one of the backflow rules (match:  $M = 3$ ) is matched and the packet is finally forwarded to the intended destination in step ⑤ (switch  $s_3$ ).

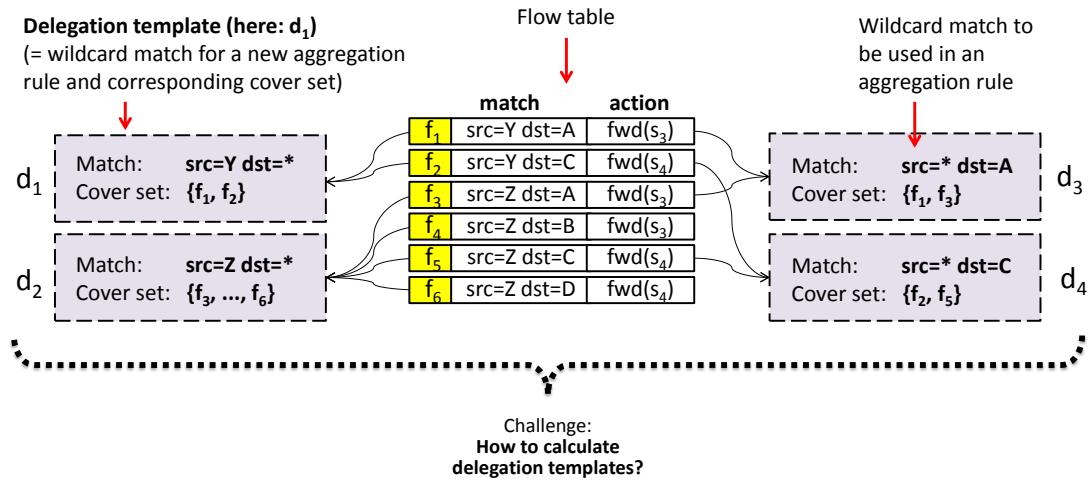
## 1.5 Structure and Challenges

The investigations carried out in the thesis are structured into three main parts: The *system design part* investigates practical and architectural challenges associated with the design of the flow delegation system. The *algorithms part* investigates the underlying optimization problem – which is mitigation of bottlenecks considering future knowledge and multiple objectives. And the *evaluation part* investigates feasibility, performance, overhead, and scalability of flow delegation covering different scenarios. The subsequent sections will briefly introduce these three parts and highlight the challenges involved.

### 1.5.1 System Design

The system design part investigates how flow delegation can be realized for software-based networks. This includes design of an architecture for flow delegation with building blocks that can address practical issues such as identifying the required monitoring information, proper prioritization of the involved flow rules, transportation of meta-information between delegation and remote switches, generation of control messages, or state management. One important challenge in this context is defining a suitable abstraction to describe subsets of flow rules to be delegated. This is addressed with the so-called *delegation template abstraction*. The general idea behind this abstraction is illustrated in Fig. 1.8.

The six flow rules in the middle represent the flow table of a bottlenecked switch. The four purple boxes are four different delegation templates  $d_1$  to  $d_4$ . Each delegation template consists of an aggregation rule and its corresponding cover set. Delegation template  $d_1$ , for example, consists of an aggregation rule with match  $\text{src} = Y, \text{dst} = *$  which covers rules  $f_1$  and  $f_2$ , i.e., the cover set is  $\{f_1, f_2\}$ . This represents one "option for delegation". It basically means the flow delegation system can install an aggregation rule with the specified match and relocate all rules in the cover set to a remote switch. This allows for an easy comparison between delegation options and also for an easy-to-understand interface between the building blocks of the architecture.



**Figure 1.8:** Delegation template abstraction

However, calculating such delegation templates is a challenging task. First, the number of possible aggregation rules for a realistic scenario can be large – it is easy to create examples with millions or billions of wildcard matches if the flow table contains 1000 or more flow rules. This is because SDN supports more than 30 different packet header fields and all combinations of these packet header fields can be used. And second, the cover set for each aggregation rule has to be independent of the remaining rules in the flow table (those that are not covered) to avoid conflicts that can lead to wrong forwarding behavior. These two challenges are addressed with a novel indirect rule aggregation scheme.

Another challenge in the system design context is control message interception. This is required because flow delegation should not interfere with network application logic. This can cause conflicts if the view of the network applications and the controller (without flow delegation) and the actual view (with flow delegation) differ from each other. As a consequence, the control messages sent from the controller have to be intercepted in order to detect and resolve such conflicts. Fig. 1.9 shows an example for such a conflict.

It is the same example that was presented above in Sec. 1.4.2 where the four yellow flow rules  $f_3$  to  $f_6$  are relocated to the remote switch. The network application, however, does not know that these four rules were relocated, i.e., it still assumes they are present in the flow table of the delegation switch. Now assume the network application decides in step ① that flow rule  $f_4$  has to be removed. The controller creates a new control message with a command to remove rule  $f_4$  ②. If this control message is sent to the delegation switch, an error occurs because no such flow rule exist ③. To address this, a new interception component is placed between the controller and the switches, shown here as the black box. This component intercepts the control message. Since it knows that



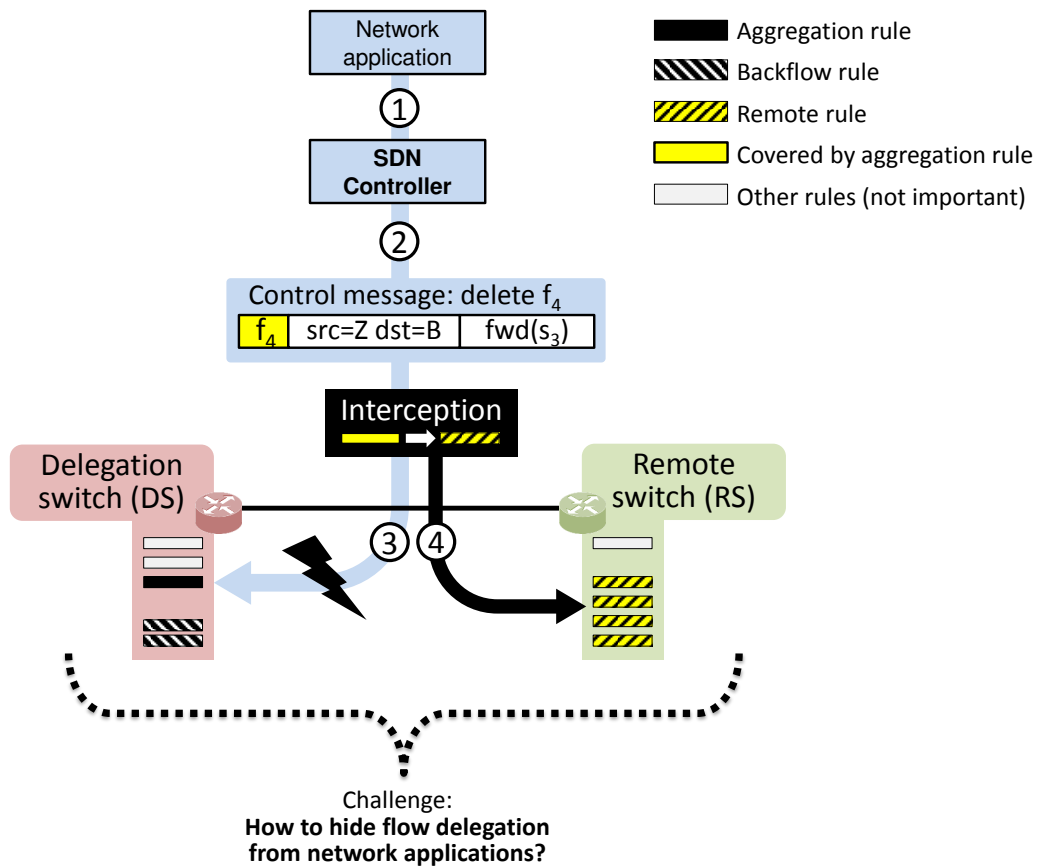


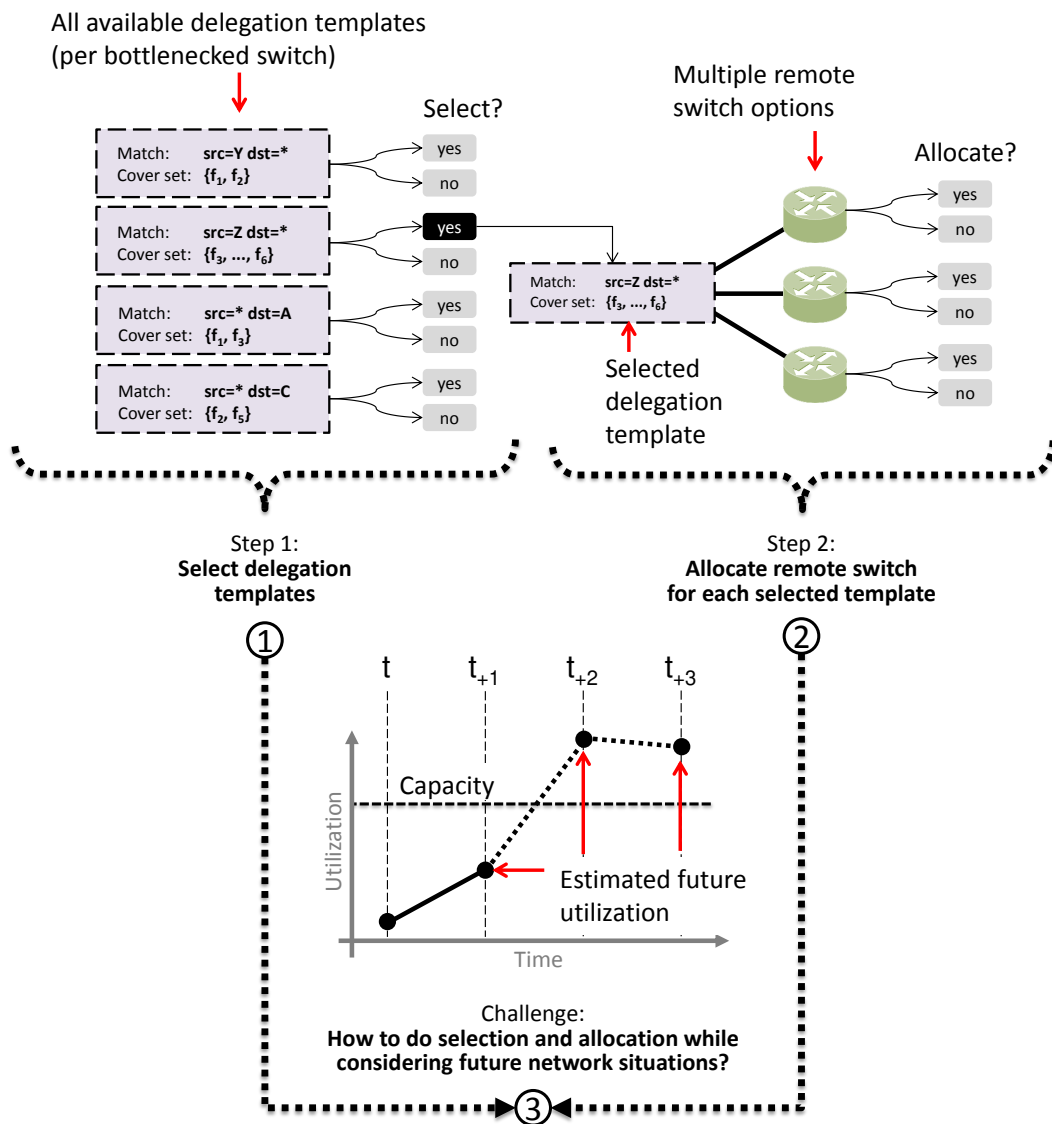
Figure 1.9: Control Message Interception

flow rule  $f_4$  was relocated to the remote switch, it can resolve the conflict by directing the control message to the correct switch in step ④ (which is the remote switch in the example).

### 1.5.2 Algorithms

The algorithms part investigates problem formulations and algorithms for flow table capacity bottleneck mitigation. The basic problem is modeled as a two-step combinatorial optimization problem as shown in Fig. 1.10: The first step is selection of delegation templates so that all bottlenecks are mitigated, i.e., the different delegation templates must be compared with each other. The second step is allocation of a suitable remote switch with enough spare capacity to each selected delegation template, i.e., the different remote switch options must be compared with each other.

Given that only the current situation is considered (single-period case), both steps can be modeled as simple linear problems: finding a subset of the delegation templates can be



**Figure 1.10:** Structure of the flow table capacity bottleneck mitigation problem

modeled as a knapsack problem and finding a suitable remote switch for each selected template can be modeled as a facility location problem. This is more challenging, however, if future network situations are taken into consideration which requires a multi-period problem formulation. For the first step (selecting delegation templates), this is difficult because the cover set of an aggregation rule changes over time if flow rules are added and removed and time-related dependencies lead to a non-linear problem. And for the second step (allocating suitable remote switches), quadratic objective terms are required because the allocated remote switch may change over time.

The challenge of including future knowledge is addressed with an assignment-based linearization approach that iterates over all possible selections (in step 1) and allocations (in step 2). The thesis further introduces a concept for periodic optimization to address the exponentially growing input length that comes with the linearization.

### 1.5.3 Evaluation

Practical feasibility of flow delegation is first investigated in the system design part with two proof-of-concept implementations. The two prototypes were created independently from each other with two different programming languages and focus primarily on the detour procedure and control message interception. Mininet is used here to emulate a realistic software-based network.

Feasibility, performance, overhead, and scalability of flow delegation with respect to a wide range of different scenarios is then investigated analytically<sup>3</sup> in the evaluation part. This requires, for example, new metrics to quantify flow delegation performance and overhead. Important challenges in this context are the evaluation methodology and reproducibility of the presented results. It has to be ensured that flow delegation is tested against different scenarios, i.e., without introducing bias by only selecting scenarios where the approach can work in its “comfort zone”. This is addressed with a four-step scenario generation process and creation of randomized sets of scenarios. In addition, all data sets and the code are made available to the public to support reproducibility.

## 1.6 Main Contributions

The overall contribution of this work is a flow delegation system that can mitigate flow table capacity bottlenecks in an on-demand and transparent fashion. Flow delegation differs from related work in four important ways. First, it has few requirements in terms of protocol features that must be supported by the infrastructure, i.e., can work with all actual versions of OpenFlow or similar protocols. No changes are necessary with respect to the controller or the network applications. The fact that a flow rule is delegated to another switch is hidden from the controller and the network applications, i.e., flow delegation is transparent to the control plane. Second, flow delegation makes use of existing spare resources in the infrastructure. Third, the approach can be enabled and disabled on-demand and is thus well suited for bottlenecks that only occur in parts of the network or within certain time frames – something that cannot be achieved with other solutions. And fourth, the used algorithms can consider future network situations and multiple objectives.

---

<sup>3</sup>Supported with simulations

This thesis includes the following main contributions:

- The **system design part** (chapters 3 to 8) introduces a *modular architecture for flow delegation* that consists of five independent building blocks: monitoring system, rule aggregation scheme, delegation algorithm, detour procedure, and control message interception. The interplay between the building blocks and the associated practical challenges are investigated in detail. Besides the architecture and the delegation template abstraction, a novel *detour procedure* is introduced that uses three classes of carefully designed flow rules to realize delegation: aggregation rules, remote rules, and backflow rules. This is complemented with a new *indirect rule aggregation scheme* that can efficiently calculate delegation templates independently of the flow rules installed in the flow table [BZ16]. In addition, a concept for *control message interception* is created to hide flow delegation from the controller and the network applications [BDZ19]. Two independent *proof-of-concept implementations* based on OpenFlow show that flow delegation is feasible in real networks. It is further shown that the overhead of the detour procedure and the rule aggregation scheme is acceptable in terms of CPU consumption and additional end-to-end delay for delegated traffic (which is below 0.1ms).
- The **algorithms part** (chapters 9 to 11) introduces a novel *algorithm for flow table capacity bottleneck mitigation* based on combinatorial optimization. The problem is first *decomposed* into two independent sub-problems [BD17]: a switch-local problem to select delegation templates calculated by the rule aggregation scheme (can be done individually for each switch) and a global problem to select an appropriate remote switch for each selected delegation template. Both problems are modeled as integer linear programming problems that are then *addressed with heuristics*. The problem formulations and heuristics take *predicted future network situations* into account and can proactively mitigate anticipated bottlenecks. They can work with *multiple and potentially conflicting objectives*, e.g., minimize the number of required aggregation rules, minimize the traffic that is detoured to remote switches, or minimize the number of required control messages. The designed heuristics operate in the millisecond range, i.e., they are *fast enough to be used in practice*.
- The **evaluation part** (chapters 12 to 16) evaluates *feasibility, performance, overhead, and scalability* of flow delegation based on a wide range of different scenarios. As one main result, it is shown that flow delegation can successfully mitigate bottleneck situations using only existing spare capacity from the infrastructure. The evaluation further includes a careful *analysis of the induced overhead*. It is shown that the amount of required aggregation and backflow rules is negligible ( $< 30$ ) and the number of additional control messages per second is also in an acceptable

region – well below 100 messages/s in most cases. The amount of detoured traffic (link overhead) depends on the scenario. Half of the investigated scenarios can be handled with a maximum link overhead of less than 13 Mbit/s per bottlenecked switch which may be distributed over multiple links if more than one remote switch is used. The evaluation also demonstrates that flow delegation *scales linear or sub-linear with respect to the following important parameters*: runtime of the required algorithms, number of considered delegation templates, and number of switches in the topology. The algorithms can be executed in less than 150ms using one core which means a single commodity server with 32 cores can handle a network with hundreds of switches.

## 1.7 Outline

Chapter 2 first introduces the necessary background and gives an overview of related work. The remainder is structured as follows.

The **system design part** consists of chapters 3 to 8 and investigates the practical challenges associated with flow delegation. Chapter 3 introduces the architecture and the central delegation template abstraction that is used in all subsequent chapters. Chapter 4 introduces the parameters to be monitored in order to realize flow delegation and defines the lambda notation, a time slot based description for monitored parameters. Chapter 5 presents indirect rule aggregation schemes to efficiently calculate delegation templates. Chapter 6 introduces the detour procedure to relocate rules and corresponding traffic from delegation to remote switch. Chapter 7 introduces a solution for control message interception to hide flow delegation from the controller and the network applications. And Chapter 8 finally discusses the existing prototype implementations of the flow delegation system.

The **algorithms part** consists of chapters 9 to 11 and introduces the delegation algorithm. Chapter 9 explains the two-step optimization approach and the decomposition into the DT-Select and RS-Alloc sub-problems: DT-Select (first step) is solved for each bottlenecked switch and selects a subset of the delegation templates calculated by the rule aggregation scheme. The global RS-Alloc problem (second step) allocates a suitable remote switch to each selected delegation template. Single and multi period formulations are introduced and analyzed for both sub-problems. Two two following chapters then introduce algorithms and heuristics for the multi period DT-Select (Chapter 10) and the multi period RS-Alloc problem (Chapter 11).

The **evaluation part** consists of chapters 12 to 16 and evaluates the developed flow delegation approach based on a broad set of different scenarios. Chapter 12 defines

assumptions, terminology and the evaluation methodology. Chapter 13 presents a case study with selected scenarios and shows how flow delegation works from a functional perspective. Chapter 14 investigates the performance for different scenarios, i.e., the amount of flow rules that can or cannot be relocated to another hardware switch. Chapter 15 investigates the overhead that is associated with flow delegation. Chapter 16 finally evaluates runtime and scalability of the designed algorithms.

Chapter 17 concludes the thesis.

# Background

---

Some general definitions are given in Sec. 2.1. Software-defined Networking and the most important underlying concepts for the thesis are defined in Sec. 2.2. Sec. 2.3 introduces the central term flow table capacity. Sec. 2.4 defines flow table capacity bottlenecks and discusses related work.

## 2.1 General Definitions

This section introduces two frequently used concepts: time slots and the terminology of delegation and remote switches.

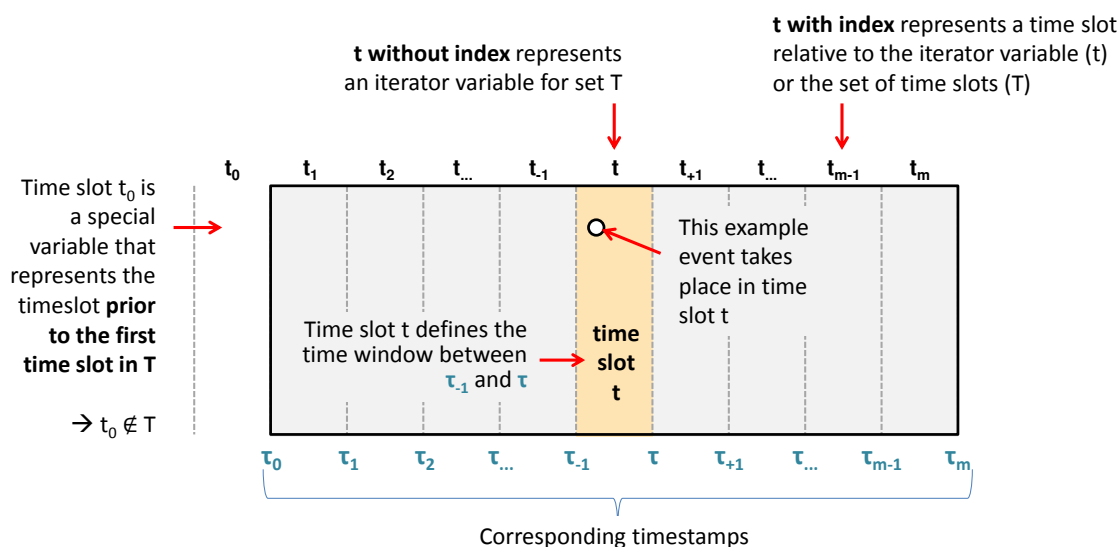
### 2.1.1 Time Slots

Time recorded in a computer system is usually stored as a real value derived from a system clock. We refer to such a value as a **timestamp**. A timestamp  $\tau$  represents a discrete point in time, e.g., the time a network application makes a decision, the time a control message is sent, the time a flow rule is installed into a switch and so on. However, because many processes in the context of flow delegation are executed periodically – such as gathering monitoring data or running the delegation algorithm – this thesis uses **time slots** instead of timestamps for simplicity.

**Definition 2.1: Time Slot**

A **time slot**  $t \in T$  identifies the time window between  $\tau_{-1}$  and  $\tau$ . Timestamp  $\tau_{-1}$  represents the end of time slot  $t_{-1}$  and the beginning of time slot  $t$ . Similarly, timestamp  $\tau$  represents the end of time slot  $t$  and the beginning of time slot  $t_{+1}$ . All time slots have the same duration, i.e.,  $(\tau_{-1} - \tau_{-2}) = (\tau - \tau_{-1}) = (\tau_{+1} - \tau) = \dots = 1$  second. Set  $T = \{t_1, \dots, t_m\}$  is called an ordered set of  $m$  consecutive time slots. Important time slots that come with a fixed meaning are summarized in Table 2.1.

According to the above definition, a time slot  $t$  simply represents a fixed time interval of one second between two timestamps. The indices are organized in such a way that timestamp  $\tau_i$  always represents the end of time slot  $t_i$ . Consider the following example: Let  $F_{s,t}$  be the set of flow rules active in the flow table of switch  $s$  in time slot  $t$ . This means the set contains all flow rules that are present in the flow table for at least some time between timestamp  $\tau_{-1}$  and timestamp  $\tau$  with  $\tau = \tau_{-1} + 1$  second. Another example: the statement “a bottleneck occurred in time slot  $t$ ” is equivalent to the statement “a bottleneck occurred between timestamp  $\tau_{-1}$  and  $\tau$ ”.



**Figure 2.1:** Illustration of the time slot concept

There is always only one single set  $T$  and this set will always have  $t_1$  as its first and  $t_m$  as its last element. The basic idea behind the indices in the time slot notation is summarized in Fig. 2.1. If  $t \in T$  is used without an index, it represents an arbitrary time slot in  $T$ , often used as an iterator variable on set  $T$ , as in the following example.

$$\bigcap_{t \in T} F_{s,t} = F_{s,t_1} \cap F_{s,t_2} \cap \dots \cap F_{s,t_m}$$



The above formula will return a set of all flow rules that are present in all time slots in  $T$ . If  $t$  is used with an index, the variable is interpreted relative to time slot  $t$  or  $T$ . Consider the following example:

$$\max_{t=t_1}^{t_{m-1}} (|F_{s,t} \cap F_{s,t_{+1}}|) = \max (|F_{s,t_1} \cap F_{s,t_2}|, |F_{s,t_2} \cap F_{s,t_3}|, \dots, |F_{s,t_{m-1}} \cap F_{s,t_m}|)$$

This will return the maximum amount of flow rules that are installed over a period of at least two consecutive time slots (the formula itself is not important, only the usage of the time slot variables). Note that set  $T$  is not specified in this formula (implicitly defined by context because there is only one  $T$ ).  $t_{m-1}$ , used here as the upper iterator bound, is interpreted relative to  $T$ , i.e., the second last element in  $T$ . Variable  $t_{+1}$  is interpreted relative to the current position of the iterator variable.

Variable	Meaning
$t_0$	Time slot prior to $t_1$ , i.e., $t_0 \notin T$
$t_1$	First time slot in set $T$
$t_m$	Last time slot in set $T$
$t_x$	$x$ -th time slot in $T$
$t_{-x}$	$x$ -th time slot prior to $t$ , i.e., $t_3$ if $t = t_4$ and $x = 1$
$t_{+x}$	$x$ -th time slot after $t$ , i.e., $t_5$ if $t = t_4$ and $x = 1$

**Table 2.1:** Time slot variables with fixed meaning

### 2.1.2 Delegation and Remote Switch

Another important concept is that all switches  $s \in S$  in the infrastructure are automatically assigned to one of two basic roles:

#### Definition 2.2: Delegation and Remote Switch

A switch with a flow table capacity bottleneck in time slot  $t$  is referred to as a **delegation switch** (usually assigned variable  $s$ ) for this time slot. All switches  $r \in S \setminus s$  with a direct connection to  $s$  and some free flow table capacity are referred to as **remote switches** (usually assigned variable  $r$ ) for this delegation switch.

The role of a switch can change between time slots if the flow table utilization changes. For a single time slot, however, a switch can either be a delegation switch or a remote switch but not both. Flow delegation is only necessary if at least one of the switches

suffers from a flow table capacity bottleneck, i.e., there is usually always at least one delegation switch in the context of this work.

## 2.2 Software-defined Networking (SDN)

Software-defined Networking (SDN) is a modern network control paradigm where control is done in software which provides network operators with a high degree of flexibility. With SDN, new functions such as routing, traffic engineering or access control can be added to the network easier and faster. This section will introduce SDN in detail and is structured as follows. Sec. 2.2.1 presents three fundamental definitions. Sec. 2.2.2 discusses the overall SDN architecture and introduces northbound and southbound interface. Sec. 2.2.3 briefly enumerates application areas for SDN. Sec. 2.2.4 covers the most important SDN concepts relevant in this work.

### 2.2.1 Definitions

Before the concepts behind SDN are explained in detail, this section will first introduce three fundamental definitions: packet header fields, flows and the term SDN itself.

#### 2.2.1.1 Packet Header Field

##### **Definition 2.3: Packet Header Field**

A **packet header field** is a protocol field carried in the header of a packet. It is interpreted based on the used protocol(s). Common header fields relevant for SDN are from layer 2 (e.g., MAC addresses), layer 3 (e.g., IP addresses) and layer 4 (e.g. TCP ports) of the OSI model.

Depended on the deployed hardware, SDN may utilize a broad range of packet header fields. The first generation of SDN switches was based on the OpenFlow switch specification [ONF09]. This generation only supported the 12 basic packet header fields listed in Table 2.2, some of which were overloaded in the beginning (the same field was used for TCP ports, UDP ports and ICMP type/code).

Note that the first entry in Table 2.2 (`in_port`) is not a real packet header field according to the above definition because it is not carried in the packet header. If a packet arrives at an SDN switch, `in_port` represents the physical ingress port of the packet. It was added to the table because it is a common strategy to match on physical ingress ports and matches are defined based on packet header fields.

Later generations added support for SCTP, ARP, IPv6 and MPLS. And starting with OpenFlow v1.3 [ONF12], the protocol uses extensible matches based on a type-length-value

Packet Header Field	Key	Size in bits
Ingress Port Indicator	in_port	depends
Ethernet source address	mac_src	48
Ethernet destination address	mac_dst	48
Ethernet type	ethertype	16
VLAN id	vlan_id	12
VLAN priority	vlan_pcp	3
IPv4 source address	ip_src	32
IPv4 destination address	ip_dst	32
IPv4 protocol	ip_proto	0
IPv4 ToS bits	ip_tos	6
TCP source port	tcp_src	16
TCP destination port	tcp_dst	16

**Table 2.2:** *Relevant packet header fields for the thesis (based on OpenFlow v1.0 [ONF09])*

definition to further increase flexibility. The use cases discussed in this thesis, however, do only require the basic packet header fields shown in Table 2.2. All of them are supported by current hardware switches. To support readability, the abstract field identifiers in the second column of Table 2.2 (denoted “Key”) are used to refer to the corresponding packet header field from now on. Due to space constraints, abbreviations of the identifiers are used in certain cases – like `src` instead of `ip_src`. In this case, the concrete meaning should be clear from context.

### 2.2.1.2 Flow

#### Definition 2.4: Flow

A **flow** is a sequence of packets where two packets  $z_i$  and  $z_j$  ( $i \neq j$ ) from the sequence share the same values for a pre-defined but otherwise arbitrary subset of packet header fields.

Flows are the main forwarding abstraction of SDN. Following the above definition, a flow can be described as a vector  $(\langle k_1, v_1 \rangle, \dots, \langle k_n, v_n \rangle)^T$  of  $n$  different key-value pairs. The key  $k_i$  of the  $i$ -th entry represents a concrete packet header field. The corresponding value  $v_i$  represents the value that is associated with this packet header field. Table 2.3 gives three examples.

Vector Notation	Description
$\langle \langle \text{mac\_dst}, \text{aa:aa:aa:aa:aa:01} \rangle \rangle$	A flow that consists of all packets with the same destination MAC address (uses only a single packet header field with an exact value)
$\left( \begin{array}{l} \langle \text{ip\_src}, 10.0.0.* \rangle \\ \langle \text{ip\_dst}, 20.0.0.* \rangle \end{array} \right)$	A flow that consists of all packets where the IP source and destination addresses are within a certain subnet (two packet header fields, values are wildcarded to model the subnet relationship)
$\left( \begin{array}{l} \langle \langle \text{mac\_src}, \text{aa:aa:aa:aa:aa:01} \rangle \\ \langle \text{mac\_dst}, \text{bb:bb:bb:bb:bb:01} \rangle \\ \langle \text{ip\_src}, 10.0.0.1 \rangle \\ \langle \text{ip\_dst}, 20.0.0.1 \rangle \\ \langle \text{ip\_proto}, \text{TCP} \rangle \end{array} \right)$	A flow that consists of all packets with the same five tuple of IP addresses, MAC addresses and protocol type (five packet header fields with five exact values).

**Table 2.3:** *Flow examples*

Note that the definition of the value part ( $v_i$ ) depends on the packet header field ( $k_i$ ). In general, there are three types of values that can be distinguished. Exact values such as the destination mac address `aa:aa:aa:aa:aa:01` in the first column of Table 2.3. Wildcarded values where a prefix of the selected packet header field is fixed and the remaining bits are unspecified – see the two IP prefixes in the second column where the unspecified bits are replaced with an asterisk (\*). And partially wildcarded values where an arbitrary bit mask can be specified (such as `ip_src, 10.*.*.1`, not included as an example in Table 2.3). In practice, the majority of packet header fields can usually only be used with exact values due to hardware restrictions. Only the address fields from layer 2 and 3 support wildcards and partial wildcards.

In the remainder of this document, wildcards as well as partial wildcards are both denoted with \*. A single wildcard symbol (i.e.,  $v_i = *$ ) means the packet header field can take an arbitrary value. Wildcards for IP source and destination addresses can also be given in CIDR notation (`10.0.0.0/24` is equivalent to `10.0.0.*`). In addition, a packet header field  $k_i$  that was not explicitly defined in the vector is interpreted as if it was defined with  $v_i = *$ . If required, shorthand notations like `src =00**` are used to save space.

### 2.2.1.3 Software-defined Networking

Several definitions for Software-defined Networking are available. One that has attracted attention from many researchers and fits well in the context of this thesis is the pillar-based definition:

#### Definition 2.5: Software-defined Networking

According to Kreutz et. al. [Kre+15], **Software-defined Networking** (SDN) is a network control architecture with four pillars:

- (1) Decoupling of control plane and data plane
- (2) Flow-based forwarding decisions
- (3) External entity where the control logic is executed
- (4) Software programmability via network applications

The **first pillar** (decoupling) says that the control logic is separated from the packet processing functionality. In a traditional network, each forwarding device (switch, router) has to provide support for both, control decisions and packet processing. With SDN, the devices solely focus on packet processing and the control part is handled outside of the forwarding devices.

The **second pillar** (flow-based forwarding) says that decisions in the network are flow-based. In other words: It is a key concept of SDN to describe flows in the network based on arbitrary packet header fields and define forwarding actions on top of them. This is a fundamental improvement over traditional approaches. Consider routing as an example. OSPF routes packets based solely on the destination IP address present in the IP packet header [Moy98]. The same is true for most other interior (and exterior) gateway protocols. Routing with SDN, on the other hand, is not by default restricted to a single packet header field<sup>1</sup>. It can also utilize the source address or additional information from the transport protocol header, if necessary.

The **third pillar** (external controller) says that the control logic is executed externally inside the so-called SDN controller. This controller is a software framework running on a commodity server. It provides an interface to network applications and establishes a logical view of the network. Control decisions made in the controller are communicated

---

<sup>1</sup>The general concept of SDN is not restricted to pre-existing packet header fields at all. Approaches like P4 or Protocol Oblivious Forwarding can also work with arbitrary / custom headers. Because this distinction is not important for flow delegation, it is not discussed here further.

towards the switch (in the form of control messages). Main benefits of this design: all relevant information is logically centralized and the hardware can be scaled up easily. Drawbacks are a potential single-point-of-failure that has to be addressed.

The **fourth pillar** (software programmability) says that the control logic is implemented in the form of network applications. Such applications are executed by the SDN controller and utilize the logically centralized view. This is the “main value proposition” of SDN according to [Kre+15].

### 2.2.2 SDN Architecture and Interfaces

SDN is often depicted as a three-layered architecture. Fig. 2.2 shows the well-known proposal by [Jar+14]. It consists of network applications, SDN controllers and SDN switches (all colored blue) as well as some legacy components (not part of SDN, shown here in beige). The three layers of the architecture are as follows. The lowest layer is the **data plane** that consists of the SDN switches. It is responsible for packet processing and forwarding. As already mentioned in the introduction, each SDN switch in the data plane is equipped with a flow table to store flow rules from the SDN controller (shown here as a match-action table).

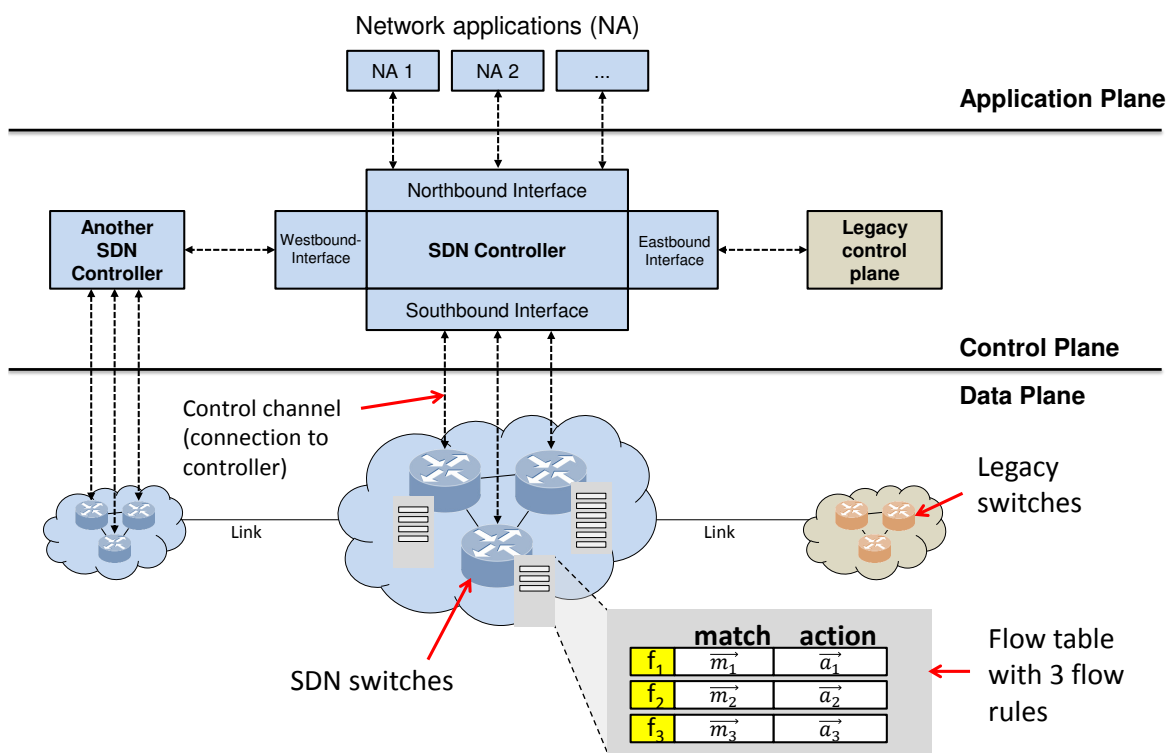


Figure 2.2: SDN Architecture, based on [Jar+14]

The layer in the middle is the **control plane** that consists of the SDN controller(s). It is responsible for managing control connections with SDN switches – which is required because the switches are programmed remotely. The control plane basically serves as an abstraction layer for the application plane. This can be compared to a “normal” operating system (like Linux, Windows) that also provides various useful abstractions to applications, e.g., memory management and support for I/O. The SDN controller does the same for the network. If there are multiple controllers, the synchronization between them is also done inside the control plane. The upper layer is the **application plane** that consists of network applications. These are responsible for the actual implementation of the required functionality like routing, load-balancing or any other form of control logic for the network.

The three layers are interconnected via two interfaces: the northbound interface placed between control plane and application plane and the southbound interface placed between control plane and data plane. There are two additional interfaces inside the control plane. The westbound interface is used for synchronization between different SDN controllers and the eastbound interface connects the SDN part of the network with the legacy part (if existing).

Various different protocols and solutions are available for all four interfaces (a good and recent overview is given by Latif et. al. in [Lat+19]). The west- and eastbound interfaces are not considered further in this thesis. The north- and southbound interfaces are characterized below.

### 2.2.2.1 Northbound Interface

The northbound interface is used by network applications to communicate decisions towards the control plane – for example, a routing decision for a pair of active users in the network. The northbound interface is not standardized, i.e., every SDN controller uses a proprietary solution (often a REST interface). The functionality offered via the northbound interface differs widely. Existing SDN controllers such as Ryu [Ryu19] or OpenDaylight [ODL19] give network application programmers access to low-level constructs, i.e., programmers can create flow rules for individual SDN switches directly.

The research community also works on high level programming languages for SDN that introduce an additional layer of abstraction. Frenetic [Fos+10], for examples, uses a declarative query language similar to SQL to describe control decisions. NetKAT [And+14] uses regular expressions to achieve similar results. Trident [GNY18] uses a special algebra and several advanced programming constructs. SDNSOC [Cho+19] follows a object-oriented approach for end-to-end policy composition. And there are many other examples [Tro+16].

One important aspect that all these approaches have in common is a (more or less) automatic translation process between higher level decisions and lower level flow rules for the individual switches. This means all decisions communicated via the northbound interface are ultimately also reflected as flow rules communicated via the southbound interface. Because of this, the focus in this thesis is solely on the southbound interface. If a northbound interface is required (e.g., for prototyping) it is assumed that flow rules are created directly by network applications.

#### 2.2.2.2 Southbound Interface

The southbound interface is used by the controller to remotely manage SDN switches. It basically serves as a hardware abstraction layer or “device driver” and handles all communication between the control plane and the data plane (in both directions). This includes the process of installing, removing and updating flow rules – which is one of the crucial features necessary to realize the SDN paradigm. It also includes configuration and management tasks such as port status or queue configuration as well as notifications from the switches towards the controller, e.g., in case of certain events (link up/down) or errors (link failure).

Unlike the northbound interface, different standardization efforts exist with respect to the southbound interface. OpenFlow [ONF15a] driven by the Open Networking Foundation is a well-known protocol in this area, sometimes described as the “lingua franca” of SDN [JRW14]. The OpenFlow specification describes the components and requirements for OpenFlow-capable switches and defines the OpenFlow pipeline, i.e., how packet processing is done within the switch. Other well known southbound interface protocols are NETCONF and OVSDB. NETCONF [Enn06] is based on XML and remote procedure calls. OVSDB [PD13] is a JSON-based protocol that allows programmatic access to the Open vSwitch (used in data centers and virtualized environments). In addition, some hardware vendors have provided their own solutions such as OpFlex. And more recently, the P4.org API Working Group proposed a new approach towards a unified and silicon-independent interface called P4Runtime [Car18; P4o19].

This thesis will focus on OpenFlow as southbound interface for several reasons: i) OpenFlow has attracted significant attention in industry and research and is supported by a wide range of software and hardware switches. ii) The protocol is easy to understand and can be used with all controllers including Ryu and Floodlight (which were used for test bed experiments). iii) Fully functional library implementations for OpenFlow are available for different programming languages which is a key requirement for control message interception (see Chapter 7). And several tools required for prototyping such as mininet or Wireshark also provide support for OpenFlow. The concepts discussed in this thesis, however, are not restricted to OpenFlow. Flow delegation could potentially also



be implemented with NETCONF or P4Runtime (or other southbound interfaces that are based on flow rules with a match-action paradigm) without conceptual changes.

### 2.2.3 Application Areas

Since SDN was introduced in early 2009 [FRZ14], the paradigm was applied in many different contexts. Well-known application areas are cloud and data center [Wan+15], wide area networks [MK17] and telecom transport networks [Alv+17]. And there are various other use cases, for example edge computing [BOE17], network function virtualization [LC15], 5G [ARS16], the Internet-of-Things [BK16] or even explicit support for big data [CYY16]. And while the individual areas are quite different from each other, all of them are driven (and limited) by the fundamental SDN concepts discussed in the next section. As a result, they may all suffer from flow table capacity bottlenecks (and could all benefit from flow delegation).

### 2.2.4 Important Concepts of SDN

This section introduces fundamental concepts of SDN such as matches, actions and flow rules. The two conceptually different modes of operation (proactive and reactive) are discussed. The rule installation workflow and the packet processing workflow are explained.

#### 2.2.4.1 Match

It was already mentioned that decisions in a software-defined network are flow-based. In order to realize a decision, the controller creates so-called “flow rules” to specify how certain packets – that is all packets that belong to one flow – are processed by an SDN switch. To achieve this, flow rules have two important components: a match part used to define a flow (discussed in this section) and an action part to describe the instructions to be executed for each matched packet (discussed in the next section).

#### Definition 2.6: Match

A **match**  $\vec{m}$  is a vector  $(\langle k_1, v_1 \rangle, \dots, \langle k_n, v_n \rangle)^T$ . The key  $k_i$  of the  $i$ -th entry represents a concrete packet header field (e.g., `ip_src`). The corresponding value  $v_i$  represents the value that is associated with this packet header field (e.g., `10.0.0.1` or any other valid IPv4 address/subnet).  $v_i = *$  indicates that all possible values can be taken.

This is the same vector notation introduced in Sec. 2.2.1.2 – which makes sense because it is the goal of the match to define a flow. An easy way to understand how this is used

within an SDN switch is the following function that takes a match  $\vec{m}$  and an arbitrary packet  $z$ :

$$\text{check\_match}(z, \vec{m}) := \begin{cases} 1, & \text{inspect\_header}(z, k) = v \quad \forall (k, v) \in \vec{m} \\ 0, & \text{otherwise} \end{cases} \quad (2.1)$$

The helper function  $\text{inspect\_header}(z, k)$  extracts the value of packet header field  $k$  within the header of packet  $z$ . The  $\text{check\_match}(z, \vec{m})$  function will now evaluate to 1 if all packet header fields specified in  $\vec{m}$  are present in packet  $z$  and the value of all fields match the values in  $\vec{m}$ . Note that the equality check of  $\text{inspect\_header}(z, k) = v$  cannot be realized as an exact match if  $v$  is a wildcarded value, but this is not considered here for simplicity (not important for the general concept). If the result of  $\text{check\_match}(z, \vec{m})$  is 1 (denoted as “packet is matched”), packet  $z$  belongs to the flow defined by match  $\vec{m}$ . If the result is 0 (denoted as “packet is not matched”), the packet does not belong to the flow defined by match  $\vec{m}$ . This is what an SDN switch essentially checks if a new packet  $z$  arrives and the switch has to determine whether this packet is matched by  $\vec{m}$  or not. A graphic representation of  $\text{check\_match}(z, \vec{m})$  is given in Fig. 2.9. Note that the statement “packet is matched by flow rule  $f$ ” is synonymous to “packet is matched by match  $\vec{m}$  of flow rule  $f$ ”.

#### 2.2.4.2 Action

The action part of a flow rule describes the instructions to be executed for each matched packet. An action is defined as follows:

##### Definition 2.7: Action

An **action**  $\vec{a}$  is vector  $(a_1(), \dots, a_m())^T$  where the  $i$ -th entry  $a_i()$  describes a packet processing instruction from Table 2.4. Instructions with smaller indices are executed first.

Note that this is a simplified definition. It does not take the existence of multiple flow tables into account and dependencies between certain types of actions are not considered (such as that a packet cannot be changed after it was forwarded). It is, however, sufficient to model the use cases discussed in this thesis. Table 2.4 is also not representative and contains only a brief list of actions relevant for this document.

#### 2.2.4.3 Flow Rule

Flow rules specify how packets of one flow are processed by an SDN switch. They are an essential construct used extensively in this thesis. Given the definitions in the two previous sections, a flow rule can be defined as follows:

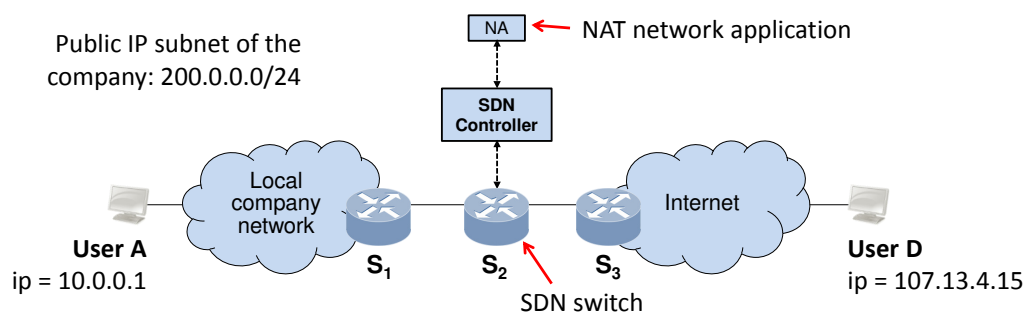
Action	Description
drop()	Packet is dropped at the switch
ctrl()	Packet is forwarded to controller
fwd( $\nu$ )	Packet is forwarded via interfaces $\nu$ OR towards target switch/host $\nu$
set( $k, \nu$ )	Overwrite packet header field $k$ with value $\nu$

**Table 2.4:** Actions considered in this thesis

### Definition 2.8: Flow Rule

A **flow rule**  $f$  is a triplet  $\langle \vec{m}, \vec{a}, \text{prio} \rangle$  where  $\vec{m}$  is a match according to Def. 2.6,  $\vec{a}$  is an action according to Def. 2.7 and  $\text{prio} \in \mathbb{N}$  is a priority value.

The logic behind this definition is simple. If a packet  $z$  belongs to the flow defined by match  $\vec{m}$ , all the instructions in  $\vec{a}$  are executed for packet  $z$ . The priority value  $\text{prio}$  is required because the same packet can be matched by multiple flow rules. Assume flow rule  $f_1$  has match  $\vec{m}_1 = (\langle \text{ip\_src}, 10.0.0.1 \rangle)$  and flow rule  $f_2$  has match  $\vec{m}_2 = (\langle \text{ip\_src}, 10.0.0.1 \rangle, \langle \text{ip\_dst}, 20.0.0.1 \rangle)$ . It is easy to see that a packet with  $\text{ip\_src} = 10.0.0.1$  and  $\text{ip\_dst} = 20.0.0.1$  will be matched by both  $\vec{m}_1$  and  $\vec{m}_2$ . The priority is used to ensure that there is always a single flow rule with highest priority for each possible packet  $z$ . It is assumed here that this priority is explicitly defined by network application programmers. Higher values for  $\text{prio}$  are considered a higher priority. Examples will also use paraphrases like “low” or “high” for the priority to keep things simple.



**Figure 2.3:** Network Address Translation (NAT) example

Flow rules provide a very intuitive abstraction to describe network control decisions. Take the scenario shown in Fig. 2.3 as an example. The local company network on the left uses private IP addresses from subnet 10.0.0.0/8. The core network that connects

the local company network with the Internet consists of three SDN switches  $s_1, \dots, s_3$ . A special NAT network application is executed inside the controller to translate private addresses into public addresses. Let us assume the NAT functionality is implemented in  $s_2$ . Further assume that user A with IP address 10.0.0.1 should be able to communicate with user D in the Internet (IP address 107.13.4.15). The required address translation functionality can be modeled as a flow rule  $\langle \vec{m}, \vec{a}, \text{prio} \rangle$  in the following way<sup>2</sup>:

$$\vec{m} = \left( \begin{array}{c} \langle \text{ip\_src}, 10.0.0.1 \rangle \\ \langle \text{ip\_dst}, 107.13.4.15 \rangle \end{array} \right)$$

$$\vec{a} = \left( \begin{array}{c} \text{set}(\text{ip\_src}, 200.0.0.17) \\ \text{fwd}(s_3) \end{array} \right)$$

$\text{prio} = \text{high}$

$\vec{m}$  defines a flow that matches on all IP packets sent from user A towards user D.  $\vec{a}$  consists of two consecutive packet processing instructions that will be executed for every packet  $z$  that arrives at  $s_2$  and matches on  $\vec{m}$ . The first instruction  $\text{set}(\text{ip\_src}, 200.0.0.17)$  overwrites the IP source address in the packet. The second instruction  $\text{fwd}(s_3)$  forwards the packet to the next switch which is  $s_3$  in this example. It is easy to see that this will realize the required functionality. After the flow rule  $\langle \vec{m}, \vec{a}, \text{prio} \rangle$  was installed (explained below in Sec. 2.2.4.5), the flow table of  $s_2$  will contain the following entry:

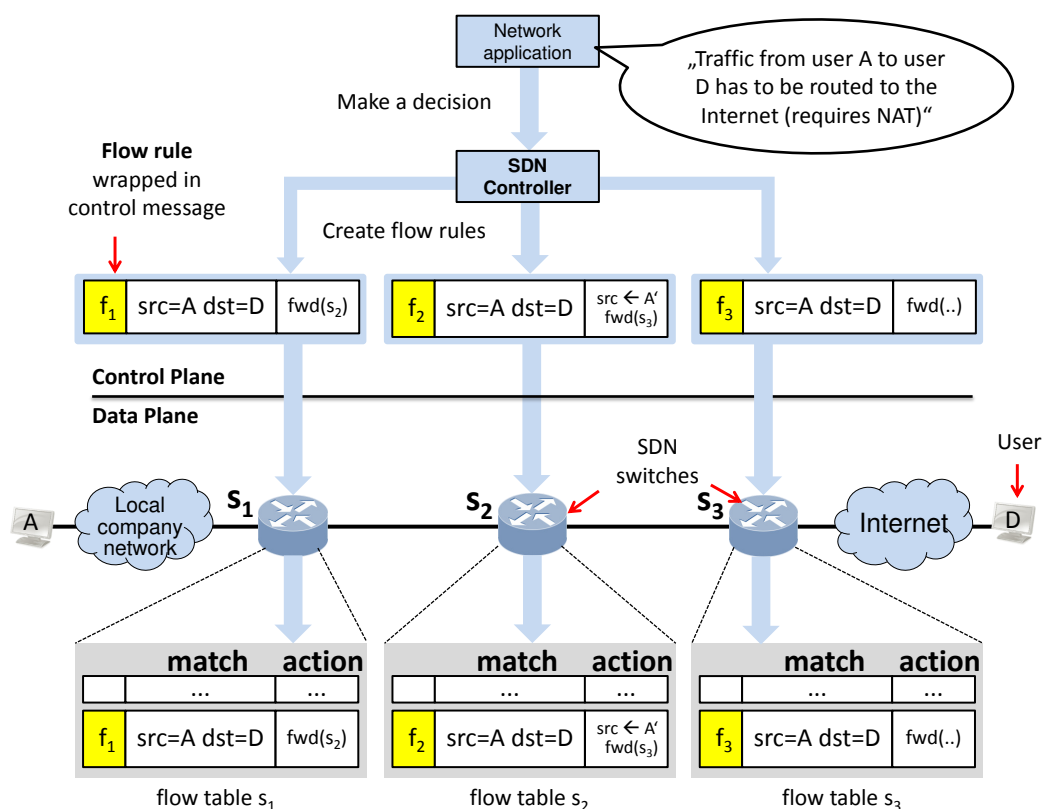
Match	Action	Priority
ip_src=10.0.0.1 ip_dst=107.13.4.15	set(ip_src, 200.0.0.17) fwd( $s_3$ )	high

The notation from Def. 2.8 and the table notation above can be used synonymous. Most examples will use the table notation, often in a reduced form to save space. A shortened version of the table above could look like this (the meaning of the variables should be clear from context; A' represents the public address 107.13.4.15):

Match	Action	Priority
src=A dst=D	src $\leftarrow$ A', fwd( $s_3$ )	high

<sup>2</sup>The example shows only one direction (from A to D). Also note that this is only one way to model this, there are many different alternative solutions.

It is important to mention that high level decisions usually result in multiple flow rules. Consider the NAT example from above. Here, at least three flow rules are required to get packets from the left side of the network to the right side (and three more for the return path which is not shown). Fig. 2.4 illustrates this.



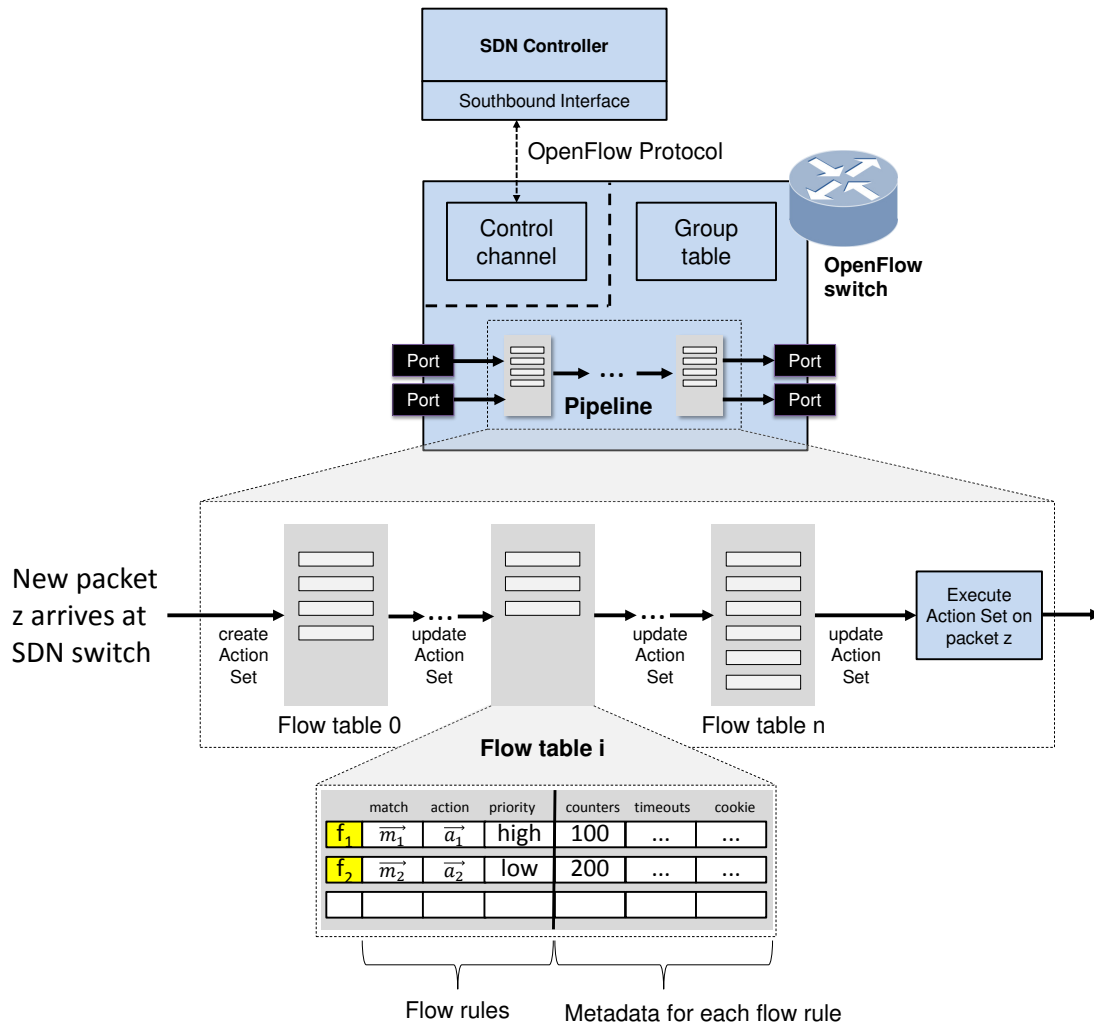
**Figure 2.4:** Extended Network Address Translation (NAT) example

Flow rule  $f_1$  forwards the packet from  $s_1$  to  $s_2$ . Flow rule  $f_2$  performs the address translation and forwards the packet to  $s_3$ . And flow rule  $f_3$  forwards the packet towards the Internet. So one high level decision resulted in multiple rules for three different switches.

#### 2.2.4.4 Flow Table

The core functionality of the flow table is storing flow rules. In general, its detailed structure and usage depends on the southbound interface protocol and the hardware in use. The reference model of an SDN switch for OpenFlow v1.3, for example, is shown in Fig. 2.5. It consists of ports, control channel (connection to controller), special tables such as the group table and a flow table pipeline that interconnects multiple flow tables.

A packet that arrives at one of the ports is processed by the pipeline (described in detail in the specification [ONF12]).

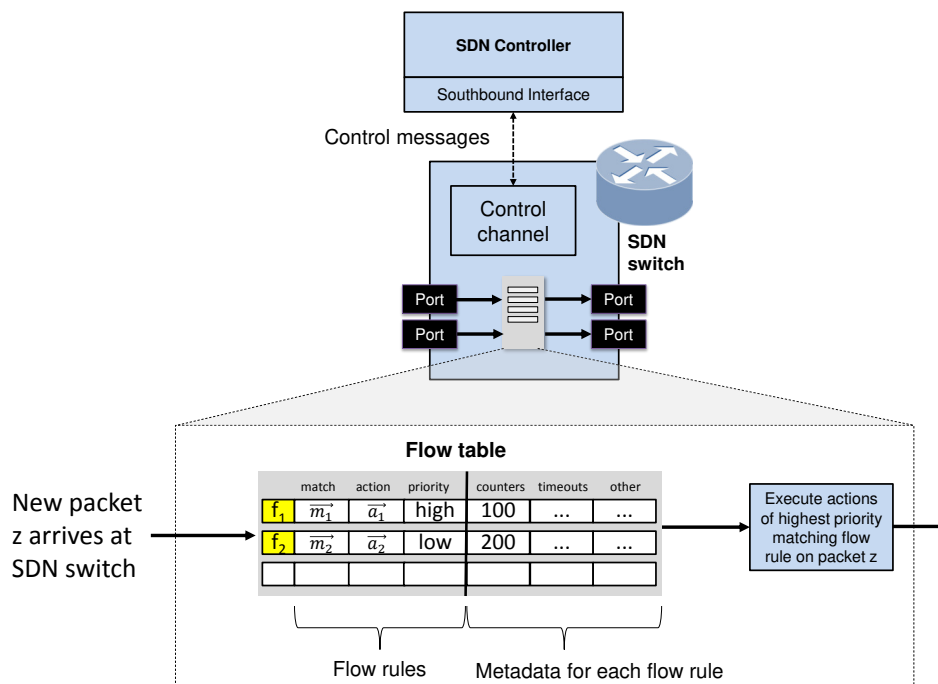


**Figure 2.5:** Switch and flow table model used in OpenFlow v1.3 (based on [ONF12])

For this thesis, however, the simpler model depicted in Fig. 2.6 is sufficient. The main difference between this model and the OpenFlow model shown above is that it uses a single flow table instead of a pipeline and does not consider special tables such as the group table. This simplification is possible because dependencies between flow rules stored in different flow tables – which may be relevant for flow delegation – can be simply modeled as a rule conflict (see Sec. 3.2.4). This way, the dependencies can be considered in the rule aggregation scheme without the need for a more complex flow table model. The same is true for special tables. Given the simplified model in Fig. 2.6, the term flow table can now be defined as follows:

**Definition 2.9: Flow table**

A **flow table** is a hardware component in an SDN switch that can store flow rules. Each entry of the flow table can hold one flow rule (regardless of the number of packet header fields used for the match). Packets that arrive at the switch are checked against all currently installed flow rules and processed based on the highest priority matching flow rule. Packets where no matching flow rule is found are discarded.



**Figure 2.6:** Simplified switch and flow table model used in this thesis

Note that – if not explicitly stated otherwise – the term flow table in this document always refers to the flow table inside a *hardware* SDN switch<sup>3</sup>. Such flow tables are realized with Ternary Content Addressable Memory (TCAM) which is an expensive and power hungry technology. Current TCAM chips can store only a few MByte per mm<sup>2</sup> which makes it difficult to equip hardware switches with large flow tables (some concrete numbers are given in Sec. 2.3.1). However, TCAMs allow it to compare an incoming packet against all flow rules in a single clock cycle which is required for high-speed packet processing. The flow table can store additional metadata (counters, timeouts) for each flow rule. Counters store monitoring information such as the number of packets and bytes processed

<sup>3</sup>Software switches which also have flow tables do not suffer from flow table capacity bottlenecks and can thus not benefit from flow delegation.

by the flow rule. Timeouts are used to automatically remove the flow rule from the flow table after a specified duration (hard timeout) or if no packet was matched for a certain duration (idle timeout).

#### 2.2.4.5 Remote Control

This section will now explain how SDN implements remote control, i.e., the ability to remotely control the SDN switches. Remote control is realized by the southbound interface protocol (here: based on OpenFlow) and consist of three basics steps: i) a control channel between controller and switch is established, ii) the command – such as installing a new flow rule – is wrapped in a control message which is sent to the switch via the established control channel, and iii) the switch extracts the command from the control message and executes it. Prior to a more detailed explanation of this process, it is first necessary to introduce the term “control message”:

#### Definition 2.10: Control Message

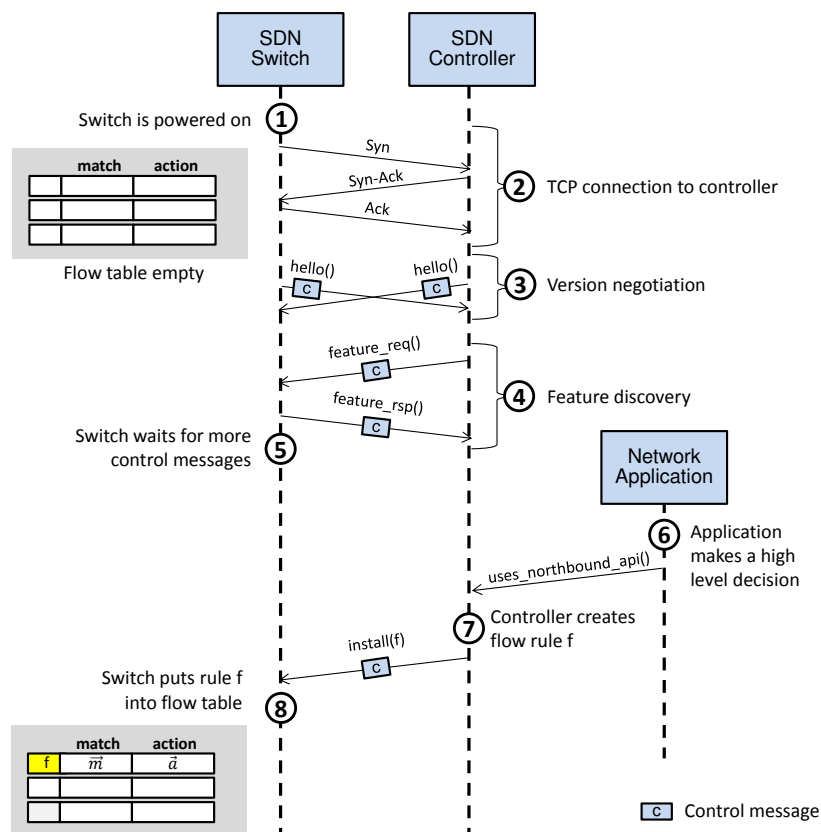
A **control message** is a packet that transmits protocol data units of the southbound interface protocol from SDN controller to SDN switches and vice versa. A list of relevant control messages is provided in Table 2.5. They are specified as functions and should be interpreted as a command the receiver of the control message has to execute, similar to a remote procedure call. Example: A control message with the `install(f)` protocol data unit sent from controller to switch means the switch (receiver of the message) has to install flow rule `f` in its flow table.

Protocol data unit	OpenFlow syntax [ONF12]	Direction	Description
<code>hello()</code>	OFPT_HELLO	S ↔ C	Version negotiation
<code>install(f)</code>	OFPT_FLOW_MOD	C → S	Install flow rule <code>f</code>
<code>delete(f)</code>	OFPT_FLOW_MOD	C → S	Delete flow rule <code>f</code>
<code>update(f, f')</code>	OFPT_FLOW_MOD	C → S	Update flow rule <code>f</code> with <code>f'</code>
<code>packet_in(z, p)</code>	OFPT_PACKET_IN	S → C	Unmatched packet <code>z</code> via port <code>p</code>
<code>packet_out(z, p)</code>	OFPT_PACKET_OUT	C → S	Forward packet <code>z</code> via port <code>p</code>
<code>feature_req(k)</code>	OFPT_FEATURES_REQUEST	C → S	Request switch capabilities
<code>feature_rsp(k)</code>	OFPT_FEATURES_REPLY	S → C	Report switch capabilities
<code>counters_req(f)</code>	OFPT_MULTIPART_REQUEST	C → S	Request flow rule counters
<code>counters_rsp(f)</code>	OFPT_MULTIPART_REPLY	S → C	Report flow rule counters

**Table 2.5:** Relevant control messages (*S* = Switch, *C* = Controller)



Workflow and the control messages listed in Table 2.5 are based on OpenFlow v1.3 [ONF12] which was used for all prototypes and test bed experiments. The core concept, however, can also be realized with other southbound interface protocols that use a match-action paradigm. Note that the notation is slightly different from the OpenFlow specification to be consistent with other chapters and to abstract from unnecessary details – see the “OpenFlow syntax” column in Table 2.5 for a mapping to the specification. Further note it is not the purpose of this section to explain the protocol behavior of OpenFlow in all details. However, a basic understanding of the mechanics and control messages is important for the control message interception building block in Chapter 7. The remote control workflow is depicted in Fig. 2.7. This example assumes a “new” software-defined network, i.e., the switch is not yet connected to the SDN controller. Control messages are shown as blue boxes (c).



**Figure 2.7:** Remote control workflow on the example of flow rule installation

- ① The switch is **powered on**. The flow table is empty and all incoming packets will be dropped. The controller does not yet know the switch.

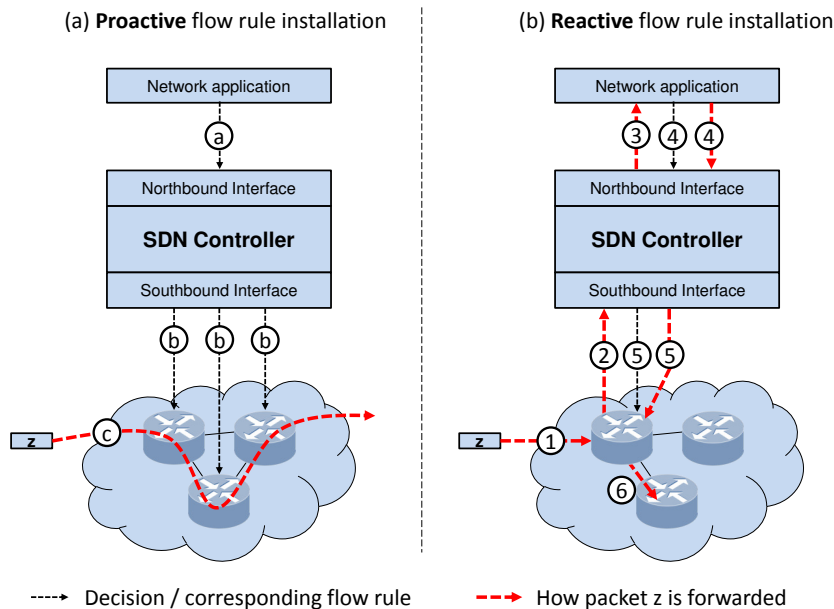
- ② It is assumed that the switch knows the IP address of the controller (statically configured). As soon as the switch OS is booted and the OpenFlow protocol daemon is started within the switch OS, it will **initiate a TLS or TCP connection** (control channel) towards the controller.
- ③ The control channel is established and can be used to exchange control messages. **Version negotiation** begins. Hello control messages are sent immediately by both sides (controller and switch). Both sides use the information inside the hello control message to determine the protocol version used for all subsequent control messages.
- ④ Version negotiation has succeeded and OpenFlow enters the **feature discovery** phase. Here, the controller uses the `feature_req` control message to get information about the capabilities supported by the switch. The switch answers with its capabilities using a `feature_rsp` control message.
- ⑤ Feature discovery phase has succeeded and the switch is **ready to receive any other control message**, e.g., to install flow rules.
- ⑥ A network applications uses the northbound interface to inform the controller about a **new high level control decision**.
- ⑦ The controller generates a **new flow rule  $f$  to implement the decision**, wraps it into a control message (using the install protocol data unit) and sends this message to the switch via the (still) active control channel.
- ⑧ The switch receives the control message, extracts the install protocol data unit and executes it. The **flow rule is added to the flow table**.

Note that the handshakes in the beginning (steps ② to ④) are only required once when the switch is first connected to the controller. The controller will manage separate active control channels (i.e., TLS/TCP connections) for all switches in the network.

#### 2.2.4.6 Proactive and Reactive Flow Rule Installation

Decision making – or flow rules installation – in SDN can be done proactively or reactively (or a combination of both). The difference is explained in Fig. 2.8. In the proactive case, the decision in step (a) is made before the first packet arrives which is packet  $z$  in the figure. This is basically the same concept used in traditional routing which can of course also be realized with SDN. Other common examples for a proactive decision are global access control or coarse granular load balancing. The flow rules that correspond to the proactive decision are installed regardless of whether there is any traffic that has to be processed by them or not (step (b)). If packet  $z$  arrives later on in step (c), the flow rules

are already present in the flow tables and the packet can be forwarded without involving the controller.



**Figure 2.8:** Proactive and reactive flow rule installation

In the reactive case, there are no pre-defined flow rules that match on packet  $z$ . Instead, the flow table has a proactively installed “default rule” that will send all unmatched traffic to the controller. The default rule has a special match  $\vec{m}$  with value  $v_i = *$  for every possible packet header field. This assures that  $\text{check\_match}(z, \vec{m}) = 1$  for all packets (every packet is matched). The action is set to `ctrl()`. `prio` is set to the lowest supported priority value so that any other flow rule has a higher priority than the default rule. The shorthand table notation for a default rule is as follows:

Match	Action	Priority
*	<code>ctrl()</code>	lowest

Now consider the workflow shown in the right hand of Fig. 2.8. If packet  $z$  arrives in step ①, there are no other rules except for the default rule and the packet is forwarded to the controller (step ②). The controller forwards the packet<sup>4</sup> to the network application in step ③. The network application can then make a fine granular decision based on the provided packet header information (step ④). This will result in one or more new

<sup>4</sup>Usually, only the first  $x$  bytes of the packet are sent to the controller where  $x$  is a configurable value.

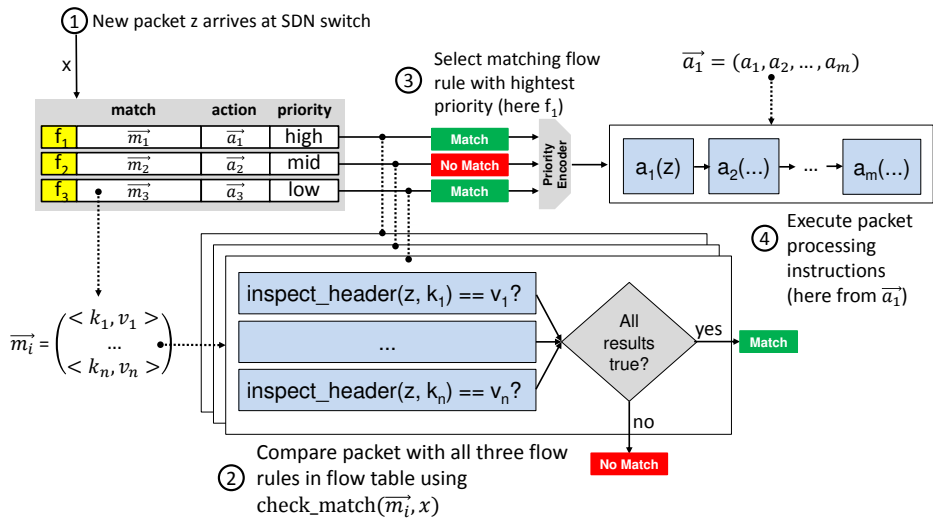
flow rules that matches on packet  $z$  (and all packets that belong to the same flow as  $z$ ) which is installed into the switch in step ⑤. Note that packet  $z$  itself is not automatically processed by the newly installed flow rule which is why Fig. 2.8 also has a red dashed arrow for step ④ and ⑤ (packet  $z$  is handled individually).

Reactive flow rule installation for arbitrary flows is one of the strongest SDN features (not possible with traditional routing!). It can be used to implement individual security policies for all active flows in the network, for fine granular monitoring or for dynamic load-balancing. On the other hand, it is also a common reason for flow table capacity bottlenecks (discussed in more detail below in Sec. 2.3.2.3).

Flow delegation does not distinguish between proactive and reactive flow rule installation (both concepts can cause bottlenecks). But it is especially useful if decisions are made reactively and some switches of the network suffer from high load due to a high number of active users or connections.

#### 2.2.4.7 Packet Processing inside the Switch

The focus so far was primarily on the control plane, i.e., how flow rules are defined and installed via control messages. This section briefly summarizes how flow rule processing in the data plane is done (from a conceptual point of view).



**Figure 2.9:** Packet Processing inside the Switch

Fig. 2.9 shows the flow table of a single switch with three flow rules. If a new packet  $z$  arrives at the switch (step ①), the packet header fields of this packet are checked against all installed flow rules (step ②). That is, for each flow rule  $f_i = \langle \vec{m}_i, \vec{a}_i, \text{prio}_i \rangle$  in the flow table, function `inspect_header(z, kj)` is executed for all tuples  $\langle k_j, v_j \rangle$  in  $\vec{m}_i$ . If this function returns 1 for all packet header fields ( $k_j$ ), rule  $f_i$  is considered a match.

Next, the flow rule with highest priority among all matching flow rules is selected (step ③). This determines the matching flow rule for packet  $z$ . Steps ① to ③ are realized with the help of Ternary Content Addressable Memory (TCAM) so that all checks for all flow rules can be done in a single clock cycle. After the matching flow rule is found, the corresponding packet processing instructions are executed for packet  $z$  (step ④).

## 2.3 Flow Table Capacity

Hardware flow table capacity is an important parameter for SDN scalability and flexibility. Cohen et. al. [Coh+14a], for example, showed that flow table capacity aware algorithms can improve performance in backbone and data center networks by up to 100%. And many other researchers have shown that limitations with respect to flow table capacity can have a significant negative impact on functionality and performance [YTG13; JMD14; Kat+14b]. The term *flow table capacity* ( $c_s^{\text{Table}}$ ) is defined here as follows:

### Definition 2.11: Flow Table Capacity

The **flow table capacity**  $c_s^{\text{Table}} \in \mathbb{N}$  denotes the maximum number of full-featured flow rules that can be installed into the flow table of switch  $s$ . Full-featured means potentially all supported packet header fields can be used in the match  $\vec{m}$  of a flow rule in parallel.

It is assumed that one flow rule always occupies exactly one entry in the flow table, i.e., aspects such as automatic rule expansion [RK10] inside the switch are not considered. In addition, it is assumed that the capacity refers to the maximum amount of “full-featured” flow rules, i.e., flow rules that can match on multiple fields in parallel including wildcards and partial wildcards for each packet header field (if wildcards are supported). This is important because such full-featured flow rules can i) only be efficiently realized with TCAM and ii) require a high amount of bits/entry. [KB13] reports 356bit/entry, which has increased for newer versions of OpenFlow. Non full-featured flow rules – such as rules that only match on the full destination MAC address – can be realized with DRAM/SRAM and do not suffer from capacity limitations to the same extent as full-featured flow rules.

### 2.3.1 Some Numbers for Hardware SDN Switches

Despite the fact that SDN is being around for more than one decade, most hardware SDN switches still only support between 1.000 and 10.000 full-featured flow rules. Information on the exact number of supported flow rules are difficult to obtain, at least without direct access to the physical hardware. Only some switch vendors like HP and Brocade provide exact TCAM numbers. Examples are given in Table 2.6.

Switch	Flow table capacity
HP 2920	500
HP 3500/5400/6600	1500-2000
HP 5900/5920	2048
Brocade ICX6610	1500

**Table 2.6:** *Flow table capacity of hardware SDN switches*

Other vendors like Pica8 or Juniper do not directly publish specific numbers. But various researchers from different institutions have published experiments with a wide range of available hardware switches. A recent study from Van Bemten et. al. [Van+19] reports numbers between 100 and 4094 for switches from Pica8, HP, Dell and NEC. The authors also observed that rules may be incorrectly added or suffer from unpredictable aging (size is reduced with each consecutive run). A study from Piotr et. al. [Ryg+17] reports between 460 and 1526 for a number of different HP switches. Costa et. al. [Cos+17] perform experiments with OpenFlow switches using different brands from Extreme, NetFPGA, Datacom, Pica8, Mikrotik, HP and LinkSys. They report similar numbers in the range from 100 to 4096. [KPK14] reports numbers between 750 and 2000 for switches from Pica8, Dell and HP. [CYP18] reports numbers between 2000 and 8000 for Pica8, Edgecore, Dell and HP.

### 2.3.2 High Demand for Flow Rules

From perspective of the network applications, the number of required flow rules can easily surpass the capacity provided by state-of-the-art SDN switches. Empirical data with respect to flow table capacity bottlenecks, however, is also difficult to find. A study from Yu et. al. [Yu+16] analyzed the number of required flow rules for a campus network and a nation wide research network based on real traffic traces. The campus network trace (captured 2010) resulted in 11.000 concurrent flow rules for one switch<sup>5</sup>. The research network trace (captured 2013) resulted in 100.000 concurrent flow rules. Another study from 2010 shows that the number of concurrent flows processed by data center edge switches is in the range of 1000 to 5000 [BAM10a]. It is an interesting observation that current hardware switches are not capable of dealing with such numbers (especially because the data used in the studies is 7-10 years old).

<sup>5</sup>Under the assumption that a flow rule is installed for each unique flow in the network (based on the default 5-tuple) and the rule is removed after 60s of idle time.

The remainder of this section discusses several common patterns that cause high flow rule demands. Certain types of networking applications conceptually deal with an effect that is known as the *flow table explosion problem* [ONF15b]. This problem occurs if independent actions have to be performed based on different packet header fields (discussed in Sec. 2.3.2.1). Multistage processing (Sec. 2.3.2.2) and fine-grained control and visibility (Sec. 2.3.2.3) are other reason why the demand for flow rules is constantly increasing.

### 2.3.2.1 Independent Actions

The well-known learning switch pattern relies on two independent actions that use two different packet header fields: Ethernet destination address (`mac_dst`) and Ethernet source address (`mac_src`). Assume a packet with `mac_src = MAC14` and `mac_dst = MAC6` enters a switch at port  $P_{14}$  and is then send to the controller because no matching flow rules are installed. The controller learns that address `MAC14` is behind port  $P_{14}$ . Assume further, the controller does already know that address `MAC6` is behind port  $P_6$ . To avoid that subsequent packets are also sent to the controller, the application has to install a new flow rule with  $\vec{m} = (\langle \text{mac\_src}, \text{MAC}_{14} \rangle, \langle \text{mac\_dst}, \text{MAC}_6 \rangle)$  and  $\vec{a} = (\text{fwd}(P_6))$ .

If the controller would only install a rule with  $\vec{m} = (\text{mac\_dst}, \text{MAC}_6)$ , other hosts behind port  $P_{14}$  are never learned because the packets are not send to the controller. This, however, results in  $N^2$  rules if  $N$  is the number of active hosts in the network (10.000 flow rules for 100 hosts). The problem can be avoided by using two flow tables (source address lookup is performed in the first table and destination address lookup in the second table), but many hardware switches do not support multiple tables in hardware.

### 2.3.2.2 Multistage Processing

Multistage processing is another pattern that often leads to high flow rule demands. One example is sliced policing, i.e., the traffic is separated into different slices and each slice is associated with a set of policies. This results in  $\#\text{slices} * \#\text{policies}$  flow rules because each packet has to be processed twice (selecting a slice and enforcing the policy). Service Function Chaining is another common example where different operations have to be executed after one another (e.g., identify the chain, select target network function within the chain, decrease index within the chain after processing). This results in an explosion of flow rules if the number of flow tables is lower than the number of processing stages (very similar to the learning switch problem discussed above).

### 2.3.2.3 Fine-grained Control and Visibility

Fine-grained control and visibility is another source for high flow rule demands. The authors in [Yu+10], for example, analyze different scenarios where up to 80 million access policies have to be considered. Various other applications have high demands as well, e.g., for realizing QoS [Kim+10], performing energy aware routing [Hel+10] or due to host mobility aspects [Yu+10]. Monitoring with full visibility (i.e., the global view inside the controller is aware of every single flow) can also require a large amount of flow rules [Cur+11].

### 2.3.3 Conclusion

Current SDN switches do not provide sufficient flow table capacity while the demand is constantly increasing due to more advanced use cases [Cur+11] (e.g., smart cities, Internet-of-Things, 5G, ...). It is not expected that this problem can be compensated with bigger or more powerful hardware. This holds true even under the assumption that Moore's law continues to be valid after 2020 [ITR15], because i) stronger hardware normally goes hand in hand with increased capital and operational expenditure and ii) the potential performance improvements are likely to be put into perspective by rising demands. In other words: flow table capacity bottlenecks are likely to become an even bigger problem in the future.

## 2.4 Flow Table Capacity Bottlenecks

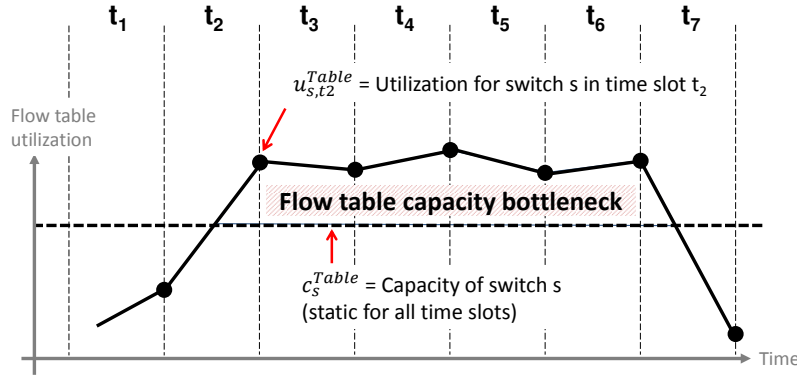
The following will first present a more detailed definition for the term flow table capacity bottleneck. Afterwards, existing research to cope with such bottlenecks is discussed.

### 2.4.1 Bottleneck Definition

Fig. 2.10 shows the situation for a switch with a flow table capacity bottleneck in time slot notation (see Sec. 2.1.1). The bottleneck is depicted as a red area between the curve of the utilization values and the capacity.

$c_s^{\text{Table}}$  denotes the flow table capacity of the considered switch (see Def. 2.11). The other variable  $u_{s,t}^{\text{Table}}$  denotes the flow table utilization measured for a specific time slot. Utilization means the current number of flow rules present in the flow table of the considered switch. More precisely: the current number of flow rules present at the end of the time slot (the exact number of installed flow rules, not the average or maximum





**Figure 2.10:** Flow table capacity bottleneck in time slot notation

utilization<sup>6</sup>). This value changes over time because network applications can install and remove flow rules in response to events from the network such as number of active users or active connections. Given  $u_{s,t}^{Table}$  and  $c_s^{Table}$ , a bottleneck can be defined as follows:

#### Definition 2.12: Flow Table Capacity Bottleneck

Switch  $s \in S$  suffers from a **flow table capacity bottleneck** in time slot  $t$  if  $u_{s,t}^{Table} > c_s^{Table}$  where  $c_s^{Table} \in \mathbb{N}$  denotes the maximum capacity of the flow table of switch  $s$  and  $u_{s,t}^{Table} \in \mathbb{N}$  denotes the current utilization of the flow table of switch  $s$  in time slot  $t$ .

**Important:**  $u_{s,t}^{Table}$  represents the utilization from the perspective of the *controller and the network applications*. Flow delegation is completely ignored for this value, i.e., flow rules relocated to a remote switch are included (would normally be installed in  $s$ ) and backflow / aggregation rules are not included (would normally not be installed in  $s$ ). This ensures that a switch is still considered a delegation switch even if the “real” utilization with flow delegation is below the capacity.

For simplicity, it is assumed that the capacity  $c_s^{Table}$  does not change over time while different switches can have different capacities (this is why  $c_s^{Table}$  has a switch index but no time slot index).

<sup>6</sup>Average and maximum values are valid alternatives here. The benefit of using the actual utilization is simpler verification of simulation results. Because flow delegation is primarily concerned with flow rules that last longer than a single time slot and the minimum lifetime of a flow rule in all experiments was set to 3 seconds, the measured difference between the three alternatives in the performed experiments was negligible. The raw data sets also include average and maximum values.

## 2.4.2 Existing Research on Flow Table Capacity Bottlenecks

Existing work on flow table capacity bottlenecks can be roughly divided into five large areas:

- (1) Resource-aware network applications: capacity limitations of the flow table are considered at application level (discussed in Sec. 2.4.2.1)
- (2) Flow rule eviction: a flow rule has to be selected for eviction if the capacity of the flow table is exceeded and a new flow rule is installed (discussed in Sec. 2.4.2.2)
- (3) Software offloading: the hardware flow table is used as a cache and low priority / low volume flows are offloaded to a software flow table (discussed in Sec. 2.4.2.3)
- (4) Flow rule distribution: flow rules are distributed over multiple hardware switches to utilize existing spare flow table capacity (discussed in Sec. 2.4.2.4)
- (5) Flow table compression: the set of all flow rules is compressed prior to installation into the hardware flow table (discussed in Sec. 2.4.2.5)

The following sections describe relevant related work for the above areas and briefly discuss the different approaches with respect to flow delegation. Sec. 2.4.3 contains pointers to relevant surveys. In conclusion, Sec. 2.4.4 summarizes the observations and presents a compressed overview in Table 2.8.

### 2.4.2.1 Resource-aware Network Applications

One approach to address flow table capacity bottlenecks are resource-aware network applications. These applications either try to minimize the number of flow rules required to achieve their functional goal (such as creating a traffic matrix for all flows in the network). Or they have detailed knowledge about the topology and the capabilities of the deployed switches and use this knowledge to avoid flow table capacity bottlenecks. There are literally hundreds of SDN-related publications where flow table capacity is considered at network application level. A few recent examples from 2018 and 2019 are listed in Table 2.7.

**Discussion:** The problem is that resource-aware network applications usually consider only a single use case such as routing, monitoring or security. The solutions are highly specialized towards this use case and cannot be generalized for other applications. Solving the bottleneck problem at application level might work well if the whole network is controlled by a single application. But this approach will fail if multiple network applications are executed in parallel. In addition, resource-aware network applications can only adapt to the maximum reported flow table capacity. This leads to reduced application

Category	Approach	Main task of the network application
Routing	Sway [SBM18]	QoS routing for the Internet-of-Things
	RA-RA [Pei+18]	Routing for service function chains
	Flow-Aware Routing [CYP18]	Routing and forwarding in wireless data centers
	Q-Data [Pha+19]	Machine-learning based flow rule management
Monitoring	FlowStat [BMJ19]	Gathering per-flow statistics
	TM Estimation Framework [TCL18]	TCAM efficient traffic matrix estimation
	FlowTracer [Wan+19a]	Effective flow trajectory detection
	DHHH Detection [WYW19]	Distributed detection of hierarchical heavy hitters
Security	RCMD [MTG18]	Security of in-band control paths
	FlowCloak [Bu+18]	Prevention of middlebox-bypass attacks
	Revive [HM18]	Data plane failure recovery

**Table 2.7:** *Network applications that consider flow table capacity bottlenecks*

performance if the flow table is highly utilized and the application adapts accordingly (e.g., by reducing the target monitoring accuracy in case of monitoring application). Flow delegation, on the other hand, works independently from individual use cases. The delegation process is hidden from the controller and the network applications. It can thus support any kind of network application, regardless of whether that application is resource-aware or not. And because additional capacity of neighboring switches can be utilized, network applications are not forced to reduce their flow rule demand without necessity.

#### 2.4.2.2 Flow Rule Eviction

The idea behind flow rule eviction is it to remove an existing (and potentially still active) flow rule from the flow table if a new flow rule has to be installed and there is no free flow table capacity available. This can be either done switch-driven or controller-driven.

**Switch-driven flow rule eviction:** Here, the switch itself has a local eviction strategy – such as least-recently-used – that is executed when a new flow rule is installed. In this case, eviction can be done without involving the controller. This requires hardware support from the SDN switch and is not standardized by currently existing southbound interfaces. Several non-trivial proposals for switch-driven eviction exist. The authors in [CLC16] utilize bloom filters stored in SRAM to track the importance of existing flow rules. FlowMaster [KB14] proposes a per flow prediction algorithm to assess the importance that can be implemented in TCAM/SRAM (demonstrated with an FPGA prototype). IRCR [Che+18] uses a new software module at the SDN switch that considers the arrival

probability of new traffic for existing flow rules to determine the best eviction candidate. Another notable mechanism for P4-based eviction was proposed in [He+18] where flow rules are removed after a TCP FIN or RST flag is detected.

**Controller-driven flow rule eviction:** Without explicit support from the switch, the controller has to decide which flow rules are kept and which are removed from the flow table in case of a bottleneck. This is usually done by setting timeouts for each flow rule. SmartTime [Vis+14], for example, uses a heuristic to determine TCAM-efficient idle timeouts. The idea is to proactively evict flow rules from the flow table before a bottleneck occurs. TimeoutX [Zha+15] calculates adaptive timeouts for new flow rules based on estimated flow rule lifetime and current flow table utilization. The authors in [YR18] and [Li+19b] use machine learning to tune timeouts for proactive flow rule eviction. And there are many other approaches that manipulate timeouts to prevent bottlenecks [Zha+14; Zhu+15; Yan+16; xu+18].

**Discussion:** All eviction-based approaches have the problem that the available flow table capacity of a bottlenecked switch is not actually increased. Eviction will just remove “less important” flow rules from the flow table. This will only help with scalability if such less important flow rules exist. It cannot deal with situations where a switch is highly utilized and all flow rules are actively required – i.e., situations where there are no good eviction candidates. Even worse, installing and evicting a large number of flow rules in short succession can lead to a phenomenon called “rule replacement storm” [CYP18] which has significant negative impact on scalability. Flow delegation, on the other hand, was explicitly designed to cope with bottlenecks that are caused by over-utilization (as long as some nearby switches have free capacity).

#### 2.4.2.3 Software Offloading

Another well-known technique for dealing with flow table capacity bottlenecks is software offloading. Software offloading is based on the observation that fast hardware and flexible software can complement each other. The basic idea is to distinguish between a “fast path” (hardware flow table) and a “slow path” (software flow table). Flow rules processing a large amount of packets are stored in the hardware flow table. The remaining flow rules are offloaded to a software flow table.

**Software offloading inside the switch:** Some approaches exist where software offloading is realized directly inside the switch. SSDN [Nar+12], for example, proposes a programmable software subsystem based on network co-processors placed at the switch. Packets that are not matched in the hardware flow table are processed in this subsystem. T-Flex [Maq+15] introduces a virtual memory mechanism for the hardware flow table. This is realized with an additional software flow table executed at the switch CPU (the authors implemented their idea in a data center switch equipped with a multi-core Intel

Xeon processor). The authors in [Hun+18] propose an approach that exploits other (non-TCAM) hardware tables – such as IP/MAC/ACL tables – to increase the amount of flow rules that are processed by the ASIC. However, these approaches are difficult to realize in practice because changes to the switch design are required.

**Software offloading with external memory:** A less invasive solution is software offloading with external memory. Here, the hardware flow table is complemented with an external software flow table (usually a software switch placed on the controller or some remote server). This allows for a trade off between data plane performance (forwarding speed) and flow table capacity. The hardware flow table provides high data plane performance but has limited capacity. The software table, on the other hand, has (almost) unlimited flow table capacity but the provided data plane performance is usually one or two orders of magnitude lower than that of a hardware switch.

A well-known approach in this category is CacheFlow [Kat+14b]. The CacheFlow architecture uses one or multiple software switches attached to a hardware switch. The latter is then used as a cache for the most popular flow rules in the network. A caching algorithm is responsible to decide which flow rules are placed in the cache and which are placed in the slower software switches. Similar to the flow delegation approach in this thesis, CacheFlow is transparent towards the control plane. This is achieved by a so called CacheMaster module that intercepts all control messages between controller and switches.

There are several approaches with similar focus as CacheFlow. AutoSlice [BP12] uses a concept called “auxiliary software datapath” to offload low-volume traffic flows to a software table. Shadow Switch [BM15] makes use of a software switching layer to speed up flow rule installation. The authors in [MDS17] propose a memory management system placed at the controller that supports a swapping function. If a hardware flow table is full, some flow rules are removed and stored in a “swap database” at the controller.

**Caching algorithms for software offloading:** A lot of work has been done in the area of caching algorithms for software offloading. The goal of these algorithms is it to select flow rules that should be offloaded to the slower software table while existing dependencies are not violated. CoverSet [Kat+14a; Kat+14b], for example, uses newly created dummy flow rules to reduce the number of dependencies prior to the caching decision. The authors in [SC16] extend this approach so it can work on set of flow rules instead of only individual flow rules. In addition, their algorithm exploits temporal and spatial localities. Temporal locality means a flow rule that was triggered by network traffic will be triggered again soon and spatial locality means that, at short time scales, traffic concentrates on flow rules with similar packet header fields. CAB [Yan+14; Yan+18a] chooses a different approach based on a geometric representation of the set of flow rules – hyper-rectangles where each dimension refers to one packet header field – to deal with

dependencies. Other researchers propose two-stage caching architecture that rely on multiple (smaller) TCAM chips in the hardware flow table to further optimize caching [XW17; Wan+17; Din+17]. And there is a range of other approaches [Li+15; Hua+16a; Yan+18b; Li+19c].

**Discussion:** Other than in the eviction case, software offloading will increase the maximum available flow table capacity of a bottlenecked switch. And because software tables can store a large amount of flow rules, this approach can theoretically increase flow table capacity by orders of magnitude (which is not possible with flow delegation<sup>7</sup>). In addition, software offloading is a generic approach and existing network applications can be used without modifications. However, the approach has two severe drawbacks: i) additional infrastructure in form of software tables has to be installed in the network and ii) flow rules placed in a slower software flow table suffer from a significant performance degradation [Bei+15; Sha+16; Emm+18]. This is especially critical if the software flow table is not placed in immediate vicinity of the hardware flow table which can result in unacceptable delay. In comparison, flow delegation requires no infrastructural changes. Performance impact on delegated flows is small (up to 0.1ms of additional delay) and unsusceptible to fluctuations because processing of the packets is still done in hardware. Furthermore, flow delegation can be easily combined with software offloading by using a software switch as extension switch. And all caching algorithms designed for software offloading can be used without no or minor modifications for the rule aggregation scheme (see Sec. 5.1.2).

#### 2.4.2.4 Flow Rule Distribution

Flow rule distribution will – as the name suggests – distribute the set of required flow rules among all available hardware switches in the network. The most notable work in this area is DIFANE [Yu+10]. DIFANE achieves scalability by distributing flow rules over a certain set of authority switches. The controller first distributes the space of all possible flow rules (flowspace) into several partitions and each authority switch is responsible for only one partition. All flow rules are then proactively installed at their respective authority switch. Ingress switches are equipped with coarse-grained, low priority partition rules that encapsulate and redirect incoming traffic to an authority switch where the appropriate packet processing is done. Palette [KHK13] follows a very similar approach and distributes large flow tables into equivalent sub-tables that are then distributed among the hardware switches. JumpFlow [Guo+15] proposes a new forwarding scheme that uses the `vlan_id` packet header field to carry routing information.

---

<sup>7</sup>The maximum number of extra flow rules that can be provided by flow delegation depend on the available free capacity in the network. Say the bottleneck switch has a flow table capacity of  $N$ . Flow delegation may be able to deal with a demand of  $2 * N$  or  $3 * N$  flows rules, but not with  $100 * N$ .

It tries to achieve load balancing between all flow tables based on a global optimization problem calculated in the controller. And there are many other approaches with similar ideas [Kan+13; Ngu+14; Len+15; Hua+16b; SLC18; OA18].

**Discussion:** Flow rule distribution is similar to flow delegation in the sense that it tries to efficiently utilize existing flow table capacity in the network. The main problem of all the above approaches is compatibility. Flow rule distribution usually performs a “re-design” of the network from a control plane perspective so that existing applications are restricted to the specific design of the solution. As a result, none of the above approaches can be used with arbitrary network applications. Most of them also require infrastructural changes or changes to the southbound interface. Flow delegation, on the other hand, is hidden from the controller and the network applications and can support arbitrary network applications.

#### 2.4.2.5 Flow Table Compression

Another technique to address flow table capacity bottlenecks is compression. Approaches based on compression try to minimize the number of required flow rules without violating the high level decisions from the network applications. Unlike flow rule distribution, compression usually focuses on a single switch / flow table.

TCAM Razor [LMT10], for example, takes a set of input flow rules and uses a multi-step process based on decision diagrams, dynamic programming and redundancy removal to create a smaller but semantically equivalent set of output flow rules. Bit Weaving [MLT12] is based on the idea that adjacent flow rules with the same action and a hamming distance of 1 and can be merged into a single flow rule. And there is many other work on generic TCAM compression independently of SDN [Don+06; MY09; BH12]<sup>8</sup>. Other work was done in the context of SDN. Compact TCAM [KB13] reduces the number of bits occupied by a single flow rule in the hardware flow table. It is based on the idea that flows in the network are classified based on static-sized, unique flow identifiers instead of arbitrary packet header fields (which can require hundreds of bits). The authors in [BM14] rely on wildcard aggregation based on the Espresso heuristic (usually used for logic minimization). Minnie [Rif+15] uses wildcard rules to compress existing routing tables based on source and destination addresses.

**Discussion:** Flow table compression can reduce the number of required flow rules but their efficiency depends heavily on the use case. Generic compression schemes such as TCAM Razor report average savings of 81.8% [LMT10]. This is possible because the evaluation focuses on range expansion which is not that relevant for SDN. Compression schemes that focus on SDN report savings between 17% [BM14] and 50% [Rif+15].

---

<sup>8</sup>Note that these approaches usually consider packet classification with range fields – such as matching on all ports between 1000 and 2000 – which is not considered here (not supported by OpenFlow).

Compression-based approaches, however, suffer from the same problem as flow rule distribution: the compression process will change the set of flow rules which is not transparent to the control plane. This hinders compatibility and the approach cannot be used with arbitrary network applications.

### 2.4.3 Further Reading

Several surveys are available that focus on flow table capacity bottlenecks. [Ngu+16] from 2016 deals with the rule placement problem in OpenFlow networks. The authors distinguish between three categories (eviction, distribution, compression) and discuss several interesting future research directions. Two surveys on energy efficient SDN from 2015 [AO15] and 2019 [AÖ19] cover many of the approaches mentioned above. [AMA19] from 2019 also studies flow table reduction mechanisms in great detail and covers some related areas such as predictive flow rule placement that are not discussed here. Please note that this thesis focuses solely on the data plane. Dedicated control plane issues such as controller placement, controller performance, horizontally and vertically scaling of the controller or consistency of the global view are not discussed. A survey on control plane scalability issues conducted by Karakas and Durresi [KD17] gives an excellent overview of this related research domain.

### 2.4.4 Summary

All approaches discussed so far have the same overall goal as flow delegation – that is avoid or mitigate flow table capacity bottlenecks –, but none of them fulfills all requirements considered in this thesis. The differentiation from related work is summarized in Table 2.8. The second column (scal) rates the expected scalability improvements that can be reached using the approach. A value of ++ means that more flow rules can be provided. The third column (ovhd) rates the overhead associated with the approach (e.g., in terms of additional latency). A value of ++ means less overhead. The last column (combined) states whether the approach can be combined with flow delegation or not. The columns OD to HC denote the following:

**[OD]** On-demand deployment is considered, i.e., it is possible to enable and disable the approach based on the current situation in the network (core requirement).

**[IC]** No infrastructural changes are required (such as new hardware or changes to existing interfaces)

**[SC]** Existing spare capacity available in the network is utilized to cope with the bottlenecks



[HC] The approach is hidden from the controller and the network applications, i.e., arbitrary network applications can benefit from it

**Table 2.8:** Comparison between flow delegation and related work

	scal	ovhd	OD	IC	SC	HC	cmb
<b>Resource-aware applications</b> (Sec. 2.4.2.1) e.g. RA-RA [Pei+18], FlowCloak [Bu+18], Revive [HM18], Q-Data [Pha+19], FlowTracer [Wan+19a], FlowStat [BMJ19]	++	++	✗	✓	✗	✗	✓
<b>Flow rule eviction</b> (Sec. 2.4.2.2) e.g. AHTM [Zha+14], FlowMaster [KB14], SmartTime [Vis+14], TimeoutX [Zha+15], IRCR [Che+18], HQ-Timer [Li+19b]	o	o	✗	✗	✗	✓	✓
<b>Software offloading</b> (Sec. 2.4.2.3) e.g. SSDN [Nar+12], AutoSlice [BP12], CacheFlow [Kat+14b], T-Flex [Maq+15], Memory Swapping [MDS17], CRAFT [XW17]	++	-	✗	✗	✗	✓	✓
<b>Flow rule distribution</b> (Sec. 2.4.2.4) e.g. DIFANE [Yu+10], Palette [KHK13], One Big Switch [Kan+13], JumpFlow [Guo+15], FTRS [Len+15], SA/SSP [SLC18]	+	+	✗	✓	✓	✗	✗
<b>Flow table compression</b> (Sec. 2.4.2.5) e.g. TCAM Razor [LMT10], Bit Weaving [MLT12], Compact TCAM [KB13], Wildcard Compression [BM14], Minnie [Rif+15]	o	++	✗	✓	✗	✗	✗
<b>Flow delegation</b> The approach proposed in this thesis	+	+	✓	✓	✓	✓	

scal = Expected scalability improvement

ovhd = Expected overhead / delay for individual flows

OD = On-demand deployment is considered

IC = No infrastructural changes are required

SC = Utilizes existing spare capacity

HC = Hidden from controller / network applications

cmb = Approach can be combined with flow delegation

**Resource-aware network applications** (Sec. 2.4.2.1) can provide very efficient solutions with respect to scalability (++) as well as overhead (++) because they can focus on a single use case perfectly tuned to the expected situation. The main problem with resource-aware applications is that bottleneck mitigation aspects are closely coupled to one use case and cannot be used by arbitrary (other) applications.

**Flow rule eviction** (Sec. 2.4.2.2) has the problem that real over-utilization scenarios cannot be mitigated because the capacity of the switch is a hard limit (existing spare capacity is not utilized). This limits the scalability potential (scalability: o). Eviction can also cause significant overhead if flow rules are evicted and installed in short succession (overhead: o).

**Software offloading** (Sec. 2.4.2.3) can theoretically increase flow table capacity by orders of magnitude (scalability: ++) but flow rules placed in a slower software flow table suffer from a significant performance degradation (overhead: -). In addition, the approach requires changes to the infrastructure.

**Flow rule distribution** (Sec. 2.4.2.4) can improve scalability by exploiting existing spare capacity in the network but the expected improvements are smaller than those of software offloading (scalability: +). But at the same time, the overhead is also smaller because packet processing is done in hardware and not in software (overhead: +). Major problem is compatibility, i.e., the distribution process is not hidden from the controller and the network applications.

**Flow table compression** (Sec. 2.4.2.5) is limited by the flow table capacity of a single switch (same as flow rule eviction, scalability: o) but the overhead is very low because packet processing remains in hardware (overhead: ++). It shares the same compatibility problem as flow rule distribution and cannot support arbitrary network applications.

In comparison, **flow delegation** (the approach proposed in this thesis) has similar expected scalability improvements and overhead as flow rule distribution while being transparent towards controller and network applications. It can thus support arbitrary network applications. And unlike all other discussed approaches, flow delegation explicitly supports on-demand deployment. It is also important to mention that three of the five existing approaches for flow table capacity bottleneck mitigation can be combined with flow delegation<sup>9</sup>.

---

<sup>9</sup>Flow rule distribution and flow table compression cannot be used together with flow delegation because both approaches usually require that network applications are aware of the distribution/compression scheme and changing the set of installed flow rules “a second time” (i.e., by flow delegation) is problematic.

Part II

System Design



# Architecture and Abstractions

---

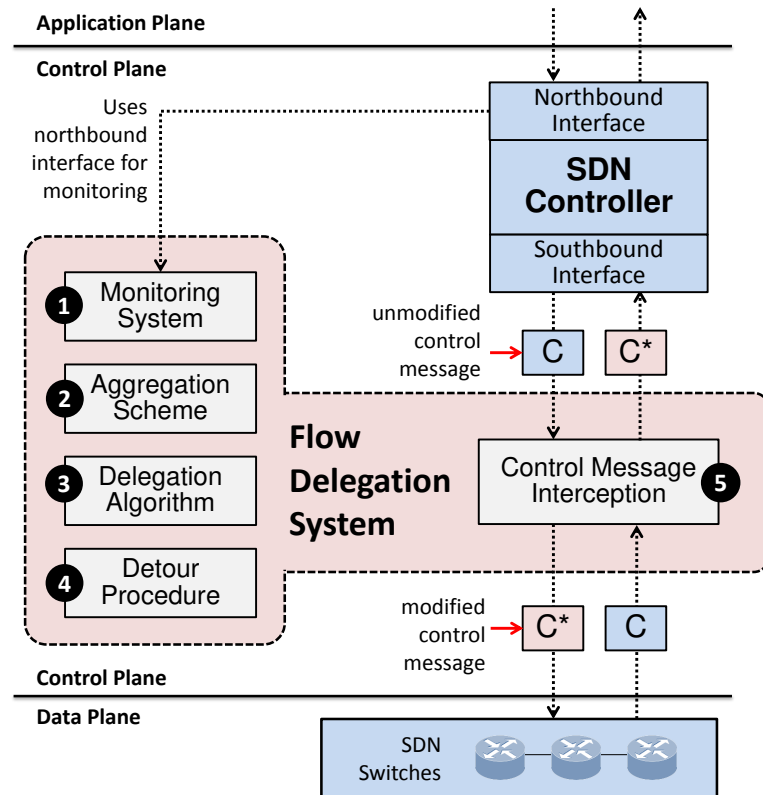
The core principle behind flow delegation and the general workflow are simple. Application of the flow delegation approach to real networks, however, is not. The system design part of the thesis – which consists of Chapters 3 to 8 – investigates numerous practical challenges associated with flow delegation which includes overall architecture, interfaces, monitoring, rule conflicts, rule aggregation, metadata transport between delegation and remote switch, generation of control messages, state management, control message interception, and other relevant aspects. The system design part makes the following contributions:

- (1) Chapter 3 introduces a **modular architecture for flow delegation** that consists of five independent building blocks. It also introduces the novel delegation template abstraction as an **interface** between the building blocks.
- (2) Chapter 4 deals with **monitoring**. It introduces the parameters to be monitored in order to realize flow delegation and defines the so-called lambda notation, a time slot based description for monitored parameters.
- (3) Chapter 5 presents the novel concept of **indirect rule aggregation schemes** to calculate delegation templates independently of the flow rules installed in the flow table.
- (4) Chapter 6 proposes a detailed design of the **detour procedure** to relocate rules and corresponding traffic from delegation to remote switch. This procedure translates delegation templates to concrete flow rules and control messages.
- (5) Chapter 7 introduces a solution for **control message interception** to hide flow delegation from the controller and the network applications.

- (6) Chapter 8 finally discusses the existing **prototype implementations** of the flow delegation approach and demonstrates that the designed architecture works in real (emulated) software-based networks.

### 3.1 Architecture

Fig. 3.1 shows the proposed architecture. Pre-existing components – the SDN controller and the SDN switches – are colored in **blue**. The flow delegation system is colored in **red**. It is located inside the control plane but outside of the SDN controller and could, for example, be executed as a separate computer program on the same host as the SDN controller.



**Figure 3.1:** Architecture of the flow delegation system

The flow delegation system consists of five building blocks shown as grey boxes in the figure. These building blocks represent the required core functionality of the system:

- ➊ A **monitoring system** to detect or anticipate flow table capacity bottlenecks
- ➋ A **rule aggregation scheme** to calculate (potential) aggregation rules

- ③ A **delegation algorithm** to select aggregation rules and allocate remote switches
- ④ A **detour procedure** to relocate rules/traffic from delegation to remote switch
- ⑤ **Control message interception** to hide flow delegation from network applications

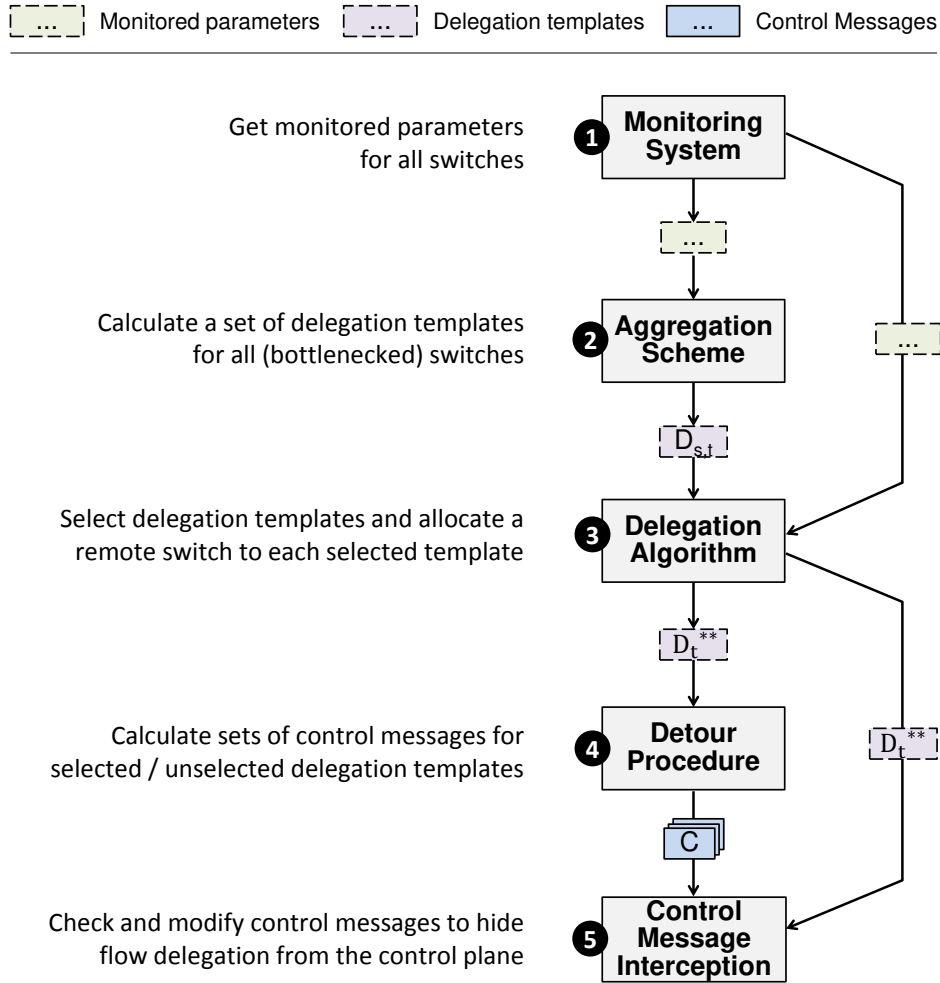
All building blocks will be explained in detail in this thesis. Monitoring system, rule aggregation scheme, detour procedure and control message interception are discussed below. The delegation algorithm is discussed in the algorithms part (chapters 9 to 11).

## 3.2 Core Concepts and Abstractions

This section defines several important concepts such as symbolic set, conflict-free cover set, and rule conflicts. Based on these definitions, the delegation template abstraction is defined. Main goal of this abstraction is it to describe “options” for delegation. Each delegation template represents one option. This allows for a clear and easy-to-understand separation of concerns between the building blocks (see Fig. 3.2):

- The rule aggregation scheme calculates a set of delegation templates for all (bottlenecked) switches
- The monitoring system provides information for each calculated delegation template
- The delegation algorithm takes the calculated delegation templates and the monitoring information to select a subset of the templates so that the bottlenecks are mitigated
- The detour procedure takes the selected delegation templates and creates the necessary control messages
- The control message interception component uses the currently selected delegation templates to make sure that flow delegation is hidden from the control plane

In this context, two important aspects have to be taken into account. First: in order to mitigate a flow table capacity bottleneck in switch  $s$  and time slot  $t$ , the flow delegation system has to relocate some of the flow rules in  $F_{s,t}$  to a remote switch. However, it is not possible to simply select a random subset  $F_{s,t}^{\text{subset}} \subset F_{s,t}$  and relocate all flow rules in this subset. To be able to relocate the flow rules in  $F_{s,t}^{\text{subset}}$ , an **aggregation rule** is required that can forward all traffic associated with the flow rules in the subset. For a random subset, such an aggregation rule usually does not exist. Secondly, even if a suitable aggregation rule is found, flow rules can have dependencies among each other. These dependencies can lead to the so-called **rule conflict** problem between rules in  $F_{s,t}^{\text{subset}}$  and



**Figure 3.2:** Idea of how the delegation templates are used

$F_{s,t}$  which results in wrong packet processing if the rules in  $F_{s,t}^{\text{subset}}$  are relocated to the remote switch.

The delegation template abstraction splits the flow rules in  $F_{s,t}$  into multiple disjoint sets  $F_{s,t}^{\text{subset}_1}, \dots, F_{s,t}^{\text{subset}_n}$  so that i) an aggregation rule exists for each of the subsets and ii) all subsets can be relocated without rule conflicts. A single subset  $F_{s,t}^{\text{subset}_i} \subset F_{s,t}$  together with instructions for creating a suitable aggregation rule for this subset is what is referred to here as **delegation template**.

The following sections will now introduce necessary concepts for the delegation template abstraction. Sec. 3.2.1 introduces two basic definitions (aggregation match and aggregation rule). Sec. 3.2.2 contains a comprehensive example of how aggregation is used with flow delegation. Sec. 3.2.3 provides additional definitions before the rule conflict



problem is explained in Sec. 3.2.4. Finally, the concept of the conflict-free cover set is introduced in Sec. 3.2.5 together with a generic algorithm to calculate it (Sec. 3.2.6). Sec. 3.2.7 finally defines the delegation template abstraction.

### 3.2.1 Aggregation Rules

Two important concepts for flow rule aggregation are aggregation matches and aggregation rules. An aggregation match is a special wildcard match  $\overrightarrow{m}_{\text{agg}}$  where some of the packet header fields are wildcarded (see Sec. 2.2.4.1) so that the new match “covers“ a set of already existing matches.

#### Definition 3.1: Aggregation Match

Let  $M_{s,t} := \{\overrightarrow{m}_i \mid \langle \overrightarrow{m}_i, \overrightarrow{a}_i, \text{prio}_i \rangle \in F_{s,t}\}$  be the set of matches for a set of flow rules  $F_{s,t}$ . Further let  $M_{\text{agg}} \subseteq M_{s,t}$  be a non-empty subset of these matches. An **aggregation match**  $\overrightarrow{m}_{\text{agg}} \notin M_{s,t}$  is then a match according to Def. 2.6 with at least one wildcarded value so that all packets matched by a match in  $M_{\text{agg}} \subseteq M_{s,t}$  will also be matched by  $\overrightarrow{m}_{\text{agg}}$ .

The set  $M_{s,t}$  is a helper construct that contains all matches from  $F_{s,t}$  (each flow rule in  $F_{s,t}$  has one match). The relationship between  $\overrightarrow{m}_{\text{agg}}$  and  $M_{\text{agg}}$  is also known as the **cover set** relationship. The aggregation match allows for a very simple and intuitive definition of aggregation rules:

#### Definition 3.2: Aggregation Rule

A flow rule  $f_{\text{agg}} = \langle \overrightarrow{m}_{\text{agg}}, \overrightarrow{a}_{\text{agg}}, \text{prio}_{\text{agg}} \rangle \notin F_{s,t}$  is referred to as an **aggregation rule** if  $\overrightarrow{m}_{\text{agg}}$  is an aggregation match according to Def. 3.1. The two other parts of the flow rule ( $\overrightarrow{a}_{\text{agg}}$  and  $\text{prio}_{\text{agg}}$ ) are referred to as aggregation action and aggregation priority.

Note that aggregation action and aggregation priority are not restricted in this definition because the cover set relationship is only defined on matches. This is special in the context of flow delegation where flow rules are relocated and not replaced (explained in the example in the next section).

### 3.2.2 Aggregation Example

The goal in most existing aggregation schemes is it to replace some of the rules in  $F_{s,t}$  with an equivalent aggregation rule so that all packets are processed as intended by the network applications but with a smaller number of rules (see related work in Sec.

2.4.2.5). In the proposed flow delegation system, however, aggregation is used somewhat differently. Because the rules are not replaced but only relocated to a remote switch, it is not necessary to find an equivalent rule. It is only required to find a suitable aggregation match – which is much easier to find. This is very similar to how aggregation is used in the context of software offloading (see Sec. 2.4.2.3).

To better understand how aggregation matches and aggregation rules are applied in this thesis, consider the example below. It consists of nine flow rules  $F_{s,t} = \{f_1, \dots, f_9\}$  in delegation switch  $s$  in time slot  $t$  with different actions and priorities (0 is the lowest priority).

Flow table of delegation switch $s$ (without flow delegation)				Flow table of delegation switch $s$ (with flow delegation)			
Rule	Match $\vec{m}_i$	Action $\vec{a}_i$	Priority	Rule	Match $\vec{m}_i$	Action $\vec{a}_i$	Priority
$f_1$	src=11 dst=*1	$\vec{a}_1$	100	$f_1$	src=11 dst=*1	$\vec{a}_1$	100
$f_2$	src=*0 dst=10	$\vec{a}_2$	90	$f_2$	src=*0 dst=10	$\vec{a}_2$	90
$f_3$	src=00 dst=10	$\vec{a}_3$	80	$f_3$	src=00 dst=10	$\vec{a}_3$	80
$f_4$	src=** dst=10	$\vec{a}_4$	70	$f_4$	src=** dst=10	$\vec{a}_4$	70
$f_5$	src=10 dst=0*	$\vec{a}_5$	60	$f_{5,6,7}$	src=10 dst=**	fwd( $r$ )	60
$f_6$	src=10 dst=10	$\vec{a}_6$	50	$f_8$	src=11 dst=00	$\vec{a}_8$	30
$f_7$	src=10 dst=11	$\vec{a}_7$	40	$f_9$	src=** dst=**	ctrl()	0
$f_8$	src=11 dst=00	$\vec{a}_8$	30				
$f_9$	src=** dst=**	ctrl()	0				

The table on the left side shows the flow table of delegation switch  $s$  without flow delegation. The table on the right side shows the same flow table with flow delegation. This means an aggregation rule was added ( $f_{5,6,7}$ ) and three rules ( $f_5, f_6, f_7$ ) were relocated to a remote switch  $r$ . The important rules are highlighted in yellow.

The corresponding flow table of the remote switch  $r$  is shown below (only for the case with flow delegation).  $\vec{m}_i^*$  and  $\vec{a}_i^*$  indicate that matches and actions of the relocated rules were translated prior to the relocation. However, it is assumed that the packet processing behavior is the same, except that the processing is happening in the remote switch and the packet is sent back afterwards (the translations are explained in the detour procedure building block in Chapter 6, not important here):

Flow table of remote switch  $r$ :

Rule	Match	Action	Priority
...	...	...	...
$f_5^*$	$\vec{m}_5^*$ , src=10 dst=0*	$\vec{a}_5^*$ , fwd(DS)	60
$f_6^*$	$\vec{m}_5^*$ , src=10 dst=10	$\vec{a}_6^*$ , fwd(DS)	50
$f_7^*$	$\vec{m}_7^*$ , src=10 dst=11	$\vec{a}_7^*$ , fwd(DS)	40
...	...	...	...

Given the special structure of this example –  $f_5$ ,  $f_6$  and  $f_7$  have the same source address and are placed next to each other with respect to priority – there is an easy to find **aggregation match**  $\vec{m}_{agg} = (\langle \text{src}, 10 \rangle, \langle \text{dst}, ** \rangle)$ . Because the example is small, it is possible to go through all possible packets that could arrive at switch  $s$ . Doing so, it can be seen that all packets matched by  $\vec{m}_5$ ,  $\vec{m}_6$  and  $\vec{m}_7$  will also be matched by  $\vec{m}_{agg}$ . So  $\vec{m}_{agg}$  is a valid aggregation match according to Def. 3.1 (in this example,  $M_{s,t}$  is  $\{\vec{m}_1, \dots, \vec{m}_9\}$  and  $M_{agg}$  is  $\{\vec{m}_5, \vec{m}_6, \vec{m}_7\}$ ).

In a next step, aggregation match  $\vec{m}_{agg}$  is used to create a new **aggregation rule**  $f_{5,6,7} = (\vec{m}_{agg}, \vec{a}_{agg}, 60)$ . The action part of the rule is set to  $\vec{a}_{agg} = (\text{fwd}(r))$ . This new aggregation rule  $f_{5,6,7}$  will forward all traffic for  $f_5$ ,  $f_6$  and  $f_7$  to the remote switch without interfering with any other rule in  $F_{s,t}$ . The easiest way to verify this statement is again going through all possible packets that could arrive at switch  $s$ . For the situation depicted above, there are only three different cases: i) a packet is matched by a rule from  $\{f_1, \dots, f_4\}$  which all have higher priority and will thus have precedence over the orange rules, ii) a packet is matched by one of the orange rules, so this will be either  $\{f_5, f_6, f_7\}$  in the case without flow delegation or  $f_{5,6,7}$  in the case with flow delegation and iii) a packet is matched by a rule from  $\{f_8, f_9\}$  with  $f_9$  matching all possible packets. So given that packets matching on  $f_{5,6,7}$  are processed in the remote switch in the exact same way as if they were processed by  $\{f_5, f_6, f_7\}$  (which is true in the case of flow delegation), the packet processing outcome of the two tables shown above is exactly the same for all possible packets.

Three important remarks regarding the above example:

- Note that  $f_5$ ,  $f_6$ , and  $f_7$  are not actually deleted but relocated to the remote switch. This is important, because the three actions  $\vec{a}_5$ ,  $\vec{a}_6$  and  $\vec{a}_7$  can be different. If the three actions are the same ( $\vec{a}_5 = \vec{a}_6 = \vec{a}_7$ ), it would be possible to actually replace the three rules with  $(\vec{m}_{agg}, \vec{a}_5, 60)$  (this is the idea behind schemes like bit weaving [MLT12]). But if the actions are different, this is not possible because different actions have to be realized as different flow rules (obviously).

- The process of “going through all possible packets” is a common way of checking whether two sets of flow rules lead to the same packet processing. This process will be formalized using the **symbolic set** concept in the next section.
- The above example does only work because  $f_5$ ,  $f_6$  and  $f_7$  form one block with respect to priorities and the aggregation rule is inserted with the same priority. This however, is not a realistic assumption. In a real flow table, the rules covered by an aggregation match are not necessarily aligned in one block with respect to priorities. This can result in **rule conflicts** which will lead to wrong forwarding behavior. The following sections explain this in more detail and show how a conflict-free cover set can be created (using the same example discussed here).

### 3.2.3 Symbolic Set and Cover Set

The example from the previous section introduced the process of “going through all possible packets” as a natural way of comparing different matches with each other (equivalent here to different flow rules because only the matches are relevant). To formalize this process, the so-called symbolic set of a match  $\vec{m}$  is introduced as follows:

#### Definition 3.3: Symbolic Set

The **symbolic set** for match  $\vec{m}$  is defined as  $Z := \{z \in \mathcal{Z} \mid \text{check\_match}(z, \vec{m}) = 1\}$  where  $\mathcal{Z}$  is the set of all possible packets (all combinations of all packet header fields with all possible values). The symbolic set of a flow rule  $f = \langle \vec{m}, \vec{a}, \text{prio} \rangle$  is equal to the symbolic set of  $\vec{m}$ .

Function  $\text{check\_match}(z, \vec{m})$  was defined in Eq. (2.1) and evaluates to 1 if all packet header fields specified in  $\vec{m}$  are present in packet  $z$  and the value of all fields match the values in  $\vec{m}$ . So the symbolic set of  $\vec{m}$  basically consists of all packets matched by  $\vec{m}$ .

Each packet in a symbolic set will be expressed as a k-bit value where k is the combined number of bits of all packet header fields. Assume there are only the two packet header fields from the example in the previous section (src and dst, each represented with 2 bits). In this case, the set of all possible packets will be given as the combination of all 4-bit values, i.e.,  $\mathcal{Z} = \{0000, 0001, \dots, 1110, 1111\}$ <sup>1</sup>.

Based on this notation, the symbolic set of  $\vec{m}_5 = (\langle \text{src}, 10 \rangle, \langle \text{dst}, 0^* \rangle)$  is denoted as  $Z_5 = \{1000, 1001\}$ . The first element represents a packet with src=10 and dst=00. The second element represents a packet with src=10 and dst=01. It is easy to see that these

<sup>1</sup>The set of all packets in a real network is very large ( $\gg 2^{100}$ ). However, it is only a helper construct that is not involved in actual calculations.

are the only two packets that will be matched by  $\vec{m}_5$ . With the symbolic set, the cover set relationship between a single match and a set of matches can be defined as follows:

**Definition 3.4: Cover Set**

The **cover set** for match  $\vec{m}_{agg}$  with respect to set of matches  $M = \{\vec{m}_1, \dots, \vec{m}_n\}$  is defined as  $M^C := \{\vec{m}_i \in M \mid Z_i \subseteq Z_{agg}, i = 1, \dots, n\}$ .  $Z_{agg}$  is the symbolic set of  $\vec{m}_{agg}$  and  $Z_i$  is the symbolic set of the  $i$ -th match in  $M$ .

This is an intuitive way to define the cover set relationship: if all packets matched by  $\vec{m}_i \in M$  are also matched by  $\vec{m}_{agg}$ ,  $\vec{m}_i$  is covered by  $\vec{m}_{agg}$ . Note that only the match is important here. Action and priority are not relevant for the cover set relationship (this is why the cover set is defined using matches and not flow rules). Consider the following example to clarify the idea. It shows the symbolic sets for the example from Sec. 3.2.2. The first rule represents the new aggregation rule (using aggregation match  $\vec{m}_{agg}$ ). The cover set for  $\vec{m}_{agg}$  in this example is  $M^C = \{\vec{m}_5, \vec{m}_6, \vec{m}_7\}$ . The important rules are highlighted in orange.

Rule	Match ( $\vec{m}_i$ )	Action	Priority	Symbolic set $Z_i$ for match $\vec{m}_i$	$Z_i \subseteq Z_{5,6,7}$
$f_{5,6,7}$	src=10 dst=**	fwd(r)	60	$Z_{5,6,7} := \{1000, 1001, 1010, 1011\}$	-
$f_1 \in F$	src=11 dst=*1	$\vec{a}_1$	100	$Z_1 := \{1101, 1111\}$	no
$f_2 \in F$	src=*0 dst=10	$\vec{a}_2$	90	$Z_2 := \{0010, 1010\}$	no
$f_3 \in F$	src=00 dst=10	$\vec{a}_3$	80	$Z_3 := \{0010\}$	no
$f_4 \in F$	src=** dst=10	$\vec{a}_4$	70	$Z_4 := \{0010, 0110, 1010, 1110\}$	no
$f_5 \in F$	src=10 dst=0*	$\vec{a}_5$	60	$Z_5 := \{1000, 1001\}$	yes
$f_6 \in F$	src=10 dst=10	$\vec{a}_6$	50	$Z_6 := \{1010\}$	yes
$f_7 \in F$	src=10 dst=11	$\vec{a}_7$	40	$Z_7 := \{1011\}$	yes
$f_8 \in F$	src=11 dst=00	$\vec{a}_8$	30	$Z_8 := \{1100\}$	no
$f_9 \in F$	src=** dst=**	ctrl()	0	$Z_9 := \{0000, \dots, 1111\}$	no

### 3.2.4 Rule Conflict Problem

In general, it is not possible to perform aggregation solely based on the cover set relationship because of the so-called rule conflict problem which can lead to incorrect packet processing. Simply put, a rule conflict occurs if packet processing with respect to a set of flow rules for some packet  $z$  leads to a different result with and without flow delegation, i.e., wrong actions are executed for some packets  $z \in \mathcal{Z}$ . The formal rule conflict definition is as follows:

**Definition 3.5: Rule Conflict**

$F_{s,t}$  is a set of flow rules ordered by priority.  $M_{s,t}$  is a set that contains the matches of all flow rules in  $F_{s,t}$ .  $\vec{m}_{\text{agg}} \notin M_{s,t}$  is an aggregation match and  $M_{s,t}^C$  is the cover set of  $\vec{m}_{\text{agg}}$  with respect to  $M_{s,t}$ . Finally,  $F_{s,t}^C := \{f_i \in F_{s,t} \mid m_i \in M_{s,t}^C\}$  is the set of flow rules associated with cover set  $M_{s,t}^C$ , also ordered by priority.  $\vec{m}_{\text{agg}}$  now leads to a **rule conflict** with respect to  $F_{s,t}$  if

- (1) there exists a packet  $z$  that should be processed by a rule in  $F_{s,t}^C$  but is processed by a rule in  $F_{s,t} \setminus F_{s,t}^C$  or
- (2) there exists a packet  $z$  that should be processed by a rule in  $F_{s,t} \setminus F_{s,t}^C$  but is processed by a rule in  $F_{s,t}^C$ .

Consider the following example to better understand the rule conflict problem. The left-hand table (①) shows the situation in delegation switch  $s$  without flow delegation. The two right-hand tables (Ⓐ and Ⓑ) also show the situation in the delegation switch, but with flow delegation, i.e., the three flow rules  $f_5, f_6, f_7$  are relocated to the remote switch and aggregation rule  $f_{5,6,7}$  is installed. There are two different tables on the right side because there are two different options that need to be considered. The flow table in the remote switch with  $f_5^*, f_6^*, f_7^*$  is not shown here again – is exactly the same as in the example in Sec. 3.2.2. Rules involved with flow delegation are highlighted in orange.

① Delegation switch  $s$  without flow delegation

Rule	Match	Action	Priority
$f_5$	src=10 dst=0*	$\vec{a}_5$	60
$f_x$	src=*0 dst=**	$\vec{a}_x$	55
$f_6$	src=10 dst=10	$\vec{a}_6$	50
$f_7$	src=10 dst=11	$\vec{a}_7$	40

 $s$  with flow delegation, case Ⓐ

Rule	Match $\vec{m}_i$	Action	Priority
$f_{5,6,7}$	src=10 dst=**	fwd( $r$ )	>55
$f_x$	src=*0 dst=**	$\vec{a}_x$	55

 $s$  with flow delegation, case Ⓑ

Rule	Match	Action	Priority
$f_x$	src=*0 dst=**	$\vec{a}_x$	55
$f_{5,6,7}$	src=10 dst=**	fwd( $r$ )	<55

The main difference to the example from Sec. 3.2.2 is that the three rules covered by aggregation rule  $f_{5,6,7}$  are not aligned in one block with respect to priorities because of rule  $f_x$ . In this case, there are only two different options for setting the priority of  $f_{5,6,7}$ : it can be set to a value higher than that of  $f_x$  (shown in case Ⓐ) or to value lower than

that of  $f_x$  (shown in case (b)). However, both options suffer from a rule conflict. To illustrate the conflict, consider the processing of two selected example packets  $z_1$  and  $z_2$ :

	$z_1 = 1000$	$z_2 = 1010$
Case (a): Processing for packet $z_i$ without flow delegation. This is the expected and correct processing as defined by the network application.	$\vec{a}_5$	$\vec{a}_x$
Case (b): Processing for packet $z_i$ if $f_{5,6,7}$ is installed with a higher priority than $f_x$	$\vec{a}_5$	$\vec{a}_6 \rightarrow \text{Wrong!}$
Case (b): Processing for packet $z_i$ if $f_{5,6,7}$ is installed with a lower priority than $f_x$	$\vec{a}_x \rightarrow \text{Wrong!}$	$\vec{a}_x$

The first row (case (a)) shows the expected behavior without flow delegation. Packet  $z_1$  is matched by the highest priority flow rule ( $f_5$ ) so that action  $\vec{a}_5$  is executed. Packet  $z_2$  is matched by the rule with second highest priority ( $f_x$ ) so that action  $\vec{a}_x$  is executed. This processing behavior must not change if flow delegation is used. But this is not the case as illustrated by the two other rows:

- In case (a), processing of packet  $z_2$  is incorrect. Because  $z_2$  matches on  $f_{5,6,7}$ , the packet is forwarded to the remote switch where it will match on rule  $f_6^*$ . This will execute action  $\vec{a}_6$ , because rule  $f_x$  (which also matches  $z_2$  and has a higher priority) is not present at the remote switch.
- In case (b), processing of packet  $z_1$  is incorrect. Because  $z_1$  matches on  $f_x$ , the packet is processed in the delegation switch so that action  $\vec{a}_x$  is executed. Rule  $f_5$  (which also matches  $z_1$  and has a higher priority) was relocated to the remote switch. But because the aggregation rule is installed with a lower priority as  $f_x$ , the lower priority rule  $f_x$  is incorrectly triggered first.

This problem occurs because there is a dependency between  $f_x$  and  $f_{5,6,7}$  that is not reflected in the cover set relationship in Def. 3.4. The conflict-free cover set relationship discussed in the next section will address this problem.

### 3.2.5 Conflict-free Cover Set

Rule conflicts as described in the previous section are caused by dependencies between flow rules. These dependencies are not yet considered in the cover set relationship. This section will extend the definition from Sec. 3.2.3 into a conflict-free cover set.

The first (and simpler) dependency is a direct dependency between two flow rules  $f_1$  and  $f_2$  where a packet can be matched by both rules. This can be modeled as an intersection between the symbolic sets of  $f_1$  and  $f_2$  as shown in the example below.

Rule	Match ( $\vec{m}_i$ )	Action	Priority	Symbolic set $Z_i$ for match $\vec{m}_i$	$Z_i \cap Z_{5,6,7}$
$f_{5,6,7}$	src=10 dst=**	fwd(RS)	60	$Z_{5,6,7} := \{1000, 1001, 1010, 1011\}$	-
$f_1 \in F$	src=11 dst=*1	$\vec{a}_1$	100	$Z_1 := \{1101, 1111\}$	$\emptyset$
$f_2 \in F$	src=*0 dst=10	$\vec{a}_2$	90	$Z_2 := \{0010, 1010\}$	1010
$f_3 \in F$	src=00 dst=10	$\vec{a}_3$	80	$Z_3 := \{0010\}$	$\emptyset$
$f_4 \in F$	src=** dst=10	$\vec{a}_4$	70	$Z_4 := \{0010, 0110, 1010, 1110\}$	{1010}
$f_5 \in F$	src=10 dst=0*	$\vec{a}_5$	60	$Z_5 := \{1000, 1001\}$	{1000, 1001}
$f_6 \in F$	src=10 dst=10	$\vec{a}_6$	50	$Z_6 := \{1010\}$	{1010}
$f_7 \in F$	src=10 dst=11	$\vec{a}_7$	40	$Z_7 := \{1011\}$	{1011}
$f_8 \in F$	src=11 dst=00	$\vec{a}_8$	30	$Z_8 := \{1100\}$	$\emptyset$
$f_9 \in F$	src=** dst=**	ctrl()	0	$Z_9 := \{0000, \dots, 1111\}$	$Z_{5,6,7}$

The three rules in grey ( $f_5, f_6, f_7$ ) are rules with  $Z_i \subseteq Z_{5,6,7}$ , i.e., every packet matched by  $Z_i$  is also matched by  $Z_{5,6,7}$  (the original cover set relationship from Def. 3.4). The two orange rules  $f_2$  and  $f_4$  do not fulfill this condition, but they intersect with the symbolic set of the aggregation rule ( $Z_{5,6,7}$ ). These intersections can cause rule conflicts. Consider flow rule  $f_2$  and packet  $z = 1010 \in Z_2 \cap Z_{5,6,7}$ . If the aggregation rule  $f_{5,6,7}$  is installed with a higher priority as  $f_2$ , packet  $z$  is incorrectly matched by the aggregation rule and detoured to the remote switch (where it is incorrectly processed by remote rule  $f_6^*$ , not shown here).

This problem can (only) be solved by adding all flow rules to the cover set that can cause rule conflicts (except for the default rule<sup>2</sup>). At first glance, this can be achieved by defining the cover set for match  $\vec{m}_{\text{agg}}$  with respect to a set of matches  $M$  as follows:

$$M^C := \left\{ \vec{m}_i \in M \mid Z_i \cap Z_{\text{agg}} \neq \emptyset, i = 1, \dots, n \right\} \quad (3.1)$$

This will include both, the gray and orange rules from the above example. However, it does not take all possible dependencies into account. In the example, flow rule  $f_3$  – which does not intersect with  $f_{5,6,7}$  – can also cause a rule conflict, because it is possible to craft a packet  $z = 0010$  that is matched by  $f_3$  and one of the rules in the set defined by Eq. 3.1 (here:  $f_2$ ). Assume the aggregation rule  $f_{5,6,7}$  is installed with a lower priority than  $f_2$ . In this case, packet  $z = 0010$  is incorrectly matched by rule  $f_3$  because the higher priority rule ( $f_2$ ) which also matches the packet was relocated to the remote switch. This is referred to as an indirect dependency.

<sup>2</sup>The default rule ( $f_9$  in the example) intersects with all rules by design and has to be explicitly excluded – which makes sense because using the default rule for aggregation will detour all traffic.



Based on this observation, the cover set relationship can be updated to a conflict-free version. Note that the new conflict-free cover set is not only defined based on an aggregation match and a set of matches but explicitly includes the set of flow rules  $F_{s,t}$ . This is important because efficient countermeasures for the rule conflict problem utilize the priority (flow rules do have a priority, matches do not).

### Definition 3.6: Conflict-free Cover Set

The **conflict-free cover set** for a match  $\vec{m}_{\text{agg}}$  with respect to a set of flow rules  $F_{s,t}$  is (recursively) defined as  $F_{s,t}^{\text{CS}} := \{ \langle \vec{m}_i, \vec{a}_i, \text{prio}_i \rangle \in F_{s,t} \}$  so that the following two conditions are fulfilled:

- (1) every rule with  $\vec{m}_i \in M_{s,t}^C$  is included in  $F_{s,t}^{\text{CS}}$
- (2) every rule  $f_i \in F_{s,t}$  that has a conflict with a rule  $f \in F_{s,t}^{\text{CS}}$  is included in  $F_{s,t}^{\text{CS}}$

$M_{s,t}^C$  is the cover set of  $\vec{m}_{\text{agg}}$  with respect to  $M_{s,t}$  where  $M_{s,t}$  consists of all matches in  $F_{s,t}$  (see Def. 3.4). The final set  $F_{s,t}^{\text{CS}}$  is sorted in descending order by  $\text{prio}_i$ .

### 3.2.6 Calculating the Conflict-free Cover Set

One option to calculate the conflict-free cover set for aggregation rules with arbitrary priority is shown in Alg. 1. This algorithm is based on the dependency graph algorithm presented in [Kat+14a]. It takes a set of flow rules  $F_{s,t}$ , an aggregation match  $\vec{m}_{\text{agg}}$  and a priority value  $\text{prio}_{\text{agg}}$  as input and returns the conflict-free cover set for  $\vec{m}_{\text{agg}}$  with respect to  $F_{s,t}$ . Note that this algorithm is only shown here to illustrate how the conflict-free cover set can be constructed from a conceptual point of view. The authors in [Kat+14a], for example, present several sophisticated algorithms to solve this problem more efficiently. The basic idea of the algorithm is as follows. It first sorts the rules in  $F_{s,t}$  according to priority to avoid a recursive formulation (line 3). It then goes through all flow rules  $f_i$  in the sorted set and checks whether the aggregation match and the match of  $\vec{m}_i$  intersect (i.e.,  $Z_{\text{intersect}}$  is not empty, see line 9). This check has to be executed for all rules if the aggregation priority is higher than the priority of  $f_i$ . If this is not the case,  $f_i$  will be "above" the newly created aggregation rule  $f_{\text{agg}}$  (from a priority perspective) so it does not have to be included in the conflict-free cover set. In case the two matches intersect, the current flow rule  $f_i$  is added to  $F_{s,t}^{\text{CS}}$ . Afterwards, the symbolic set of  $\vec{m}_{\text{agg}}$  is updated as follows (see line 11):

- The packets that are matched by both,  $\vec{m}_{\text{agg}}$  and  $\vec{m}_i$  (which is all packets in  $Z_{\text{intersect}}$ ) are removed before the next iteration, because every such packet will be processed

**Algorithm 1:** Calculation of conflict-free cover set (based on [Kat+14a])

---

**Data:** aggregation match  $\vec{m}_{agg}$ , aggregation priority  $prio_{agg}$ , set of flow rules  $F_{s,t}$   
**Result:** conflict-free cover set  $F_{s,t}^{CS}$  for  $\vec{m}_{agg}$  with respect to  $F_{s,t}$

```

1 Function get_conflict_free_cover_set( $\vec{m}_{agg}$ ,  $F_{s,t}$ ,  $prio_{agg} \leftarrow highest$ ):
2    $F_{s,t}^{CS} \leftarrow \{\}$ 
3   /* Remove default rule and sort flow rules by prio in descending order */
4    $F^{sorted} \leftarrow \text{sort}(F_{s,t} \setminus \{f_{default}\})$ 
5    $Z_{agg} \leftarrow$  all packets matched by  $\vec{m}_{agg}$  (symbolic set)
6   for  $f_i = \langle \vec{m}_i, \vec{a}_i, prio_i \rangle \in F^{sorted}$  do
7     if  $prio_{agg} \geq prio_i$  then
8        $Z_i \leftarrow$  all packets matched by  $\vec{m}_i$  (symbolic set)
9       /* Helper set: all packets matched by  $\vec{m}_{agg}$  and  $\vec{m}_i$  */
10       $Z_{intersect} \leftarrow Z_{agg} \cap Z_i$ 
11      if  $Z_{intersect} \neq \emptyset$  then
12        /* Add current flow rule to cover set */
13         $F_{s,t}^{CS} \leftarrow F_{s,t}^{CS} \cup \{f_i\}$ 
14        /* Packets in both sets can be removed because they are now
15         covered by a rule in  $F_{s,t}^{CS}$ . On the other hand, packets matched
16         by  $f_i$  that are not yet covered by  $F_{s,t}^{CS}$  have to be added. */
17         $Z_{agg} \leftarrow (Z_{agg} \setminus Z_{intersect}) \cup (Z_i \setminus Z_{intersect})$ 
18      end
19    end
20  end
21  return  $F_{s,t}^{CS}$ 
22 end

```

---

by  $f_i$  (any other rule that could be added later on will have a lower priority because the input set is sorted in descending priority order).

- To account for potential indirect dependencies, all packets matched by  $\vec{m}_i$  that are not yet covered by any rule in the conflict-free cover set (which is all packets in  $Z_i \setminus Z_{intersect}$ ) are added to the symbolic set of  $\vec{m}_{agg}$ .

The following table illustrates Alg. 1 executed on the example from Sec. 3.2.2. Shown are the first five iterations of the algorithm together with the two important sets  $Z_{intersect}$  and  $Z_i \setminus Z_{intersect}$ .

Step	Rule	Match $\vec{m}_i$	$Z_i$ (symbolic set of $\vec{m}_i$ )	$Z_{\text{intersect}}$	$Z_i \setminus Z_{\text{intersect}}$
①	$f_{\text{agg}}$	10**	$Z_{\text{agg}} := \{1000, 1001, 1010, 1011\}$	-	-
②	$f_1$	11*1	$Z_1 := \{1101, 1111\}$ → nothing to do	$\emptyset$	$\emptyset$
③	$f_2$	*010	$Z_2 := \{0010, 1010\}$ → $F_{s,t}^{\text{CS}} = \{f_2\}$ → remove 1010, add 0010 $Z_{\text{agg}} := \{1000, 1001, 1011, 0010\}$	$\{1010\}$	$\{0010\}$
④	$f_3$	0010	$Z_3 := \{0010\}$ → $F_{s,t}^{\text{CS}} = \{f_2, f_3\}$ → remove 0010 $Z_{\text{agg}} := \{1000, 1001, 1011\}$	$\{0010\}$	$\emptyset$
⑤	$f_4$	**10	$Z_4 := \{0010, 0110, 1010, 1110\}$ → nothing to do	$\emptyset$	$\emptyset$
⑥	$f_5$	100*	$Z_5 := \{1000, 1001\}$ → $F_{s,t}^{\text{CS}} = \{f_2, f_3, f_5\}$ → remove 1000, 1001 $Z_{\text{agg}} := \{1011\}$	$\{1000, 1001\}$	$\emptyset$
⑦	$f_6$	...	...	...	...

It can be seen that  $f_2$  is correctly added in step ③ because of the direct dependency with  $f_{\text{agg}}$ . And  $f_3$  is correctly added as well in step ④ because the conflicting packet  $z = 0010$  was added to the symbolic set of  $\vec{m}_{\text{agg}}$  in the end of step ③. Another interesting part is that  $f_4$  is not added which is different from the example before (note that  $\vec{m}_4$  and  $\vec{m}_{\text{agg}}$  in their original form have a non-empty intersection set because of packet  $z = 0010$ ). This optimization can be done here because the priorities are taken into account. The algorithm knows that packet  $z = 0010$  can never match rule  $f_4$  because it will be matched by  $f_3$  which has a higher priority. This packet was thus removed from the symbolic set of  $\vec{m}_{\text{agg}}$  in the end of step ④.

### 3.2.7 Delegation Templates

The last sections explained in detail why rule conflicts are a critical problem for flow delegation. This section will now introduce the term delegation template, the core abstractions for this work. In essence, a delegation template represents a set of flow rules that can be relocated to a remote switch without rule conflicts. More precisely, it is guaranteed that no rule conflicts occur if all flow rules associated with the delegation

template are kept together, i.e., all of them are either installed in the remote switch or in the delegation switch.

### Definition 3.7: Delegation Template

A **delegation template**  $d \in D_{s,t}$  for a set of flow rules  $F_{s,t}$  is a tuple  $\langle \overrightarrow{m}_{\text{agg}}, F_{s,t}^{\text{CS}} \rangle$  where  $\overrightarrow{m}_{\text{agg}}$  is an aggregation match according to Def. 3.1 and  $F_{s,t}^{\text{CS}}$  is the conflict-free cover set for  $\overrightarrow{m}_{\text{agg}}$  with respect to  $F_{s,t}$ . For brevity, the above tuple is also written as  $\langle \overrightarrow{m}_d, F_{d,t} \rangle$  when appropriate where index  $d$  represents the delegation template (see Fig. 3.3 for example).

So a delegation template is basically a single<sup>3</sup> aggregation match together with the corresponding conflict-free cover set. It is therefore said that a rule belongs to a delegation template or is covered by a delegation template.

A visual explanation of this definition – and its application – is given in Fig 3.3. Assume a flow table with capacity  $c_s^{\text{Table}} = 9$  as shown in the bottom left of the figure. Further assume that nine flow rules  $f_1$  to  $f_9$  are installed in this flow table in time slot  $t$ , i.e., it is fully utilized. To mitigate the bottleneck, flow delegation has to relocate some of the flow rules in  $F_{s,t}$  to a remote switch. The delegation templates now represent the **different available options how this can be achieved without rule conflicts**.

In this example, there are three different delegation templates given as set  $\overline{D}_{s,t}$  in the top of the figure. The first delegation template  $d_1 \in D_{s,t}$  consists of aggregation match  $\overrightarrow{m}_a$  and the conflict-free cover set  $F_a = \{f_1, f_2, f_3\}$ .  $F_a$  is a placeholder for  $F_{s,t}^{\text{CS}}$  and can be calculated by Alg. 1, for example. The flow table in the middle labeled as ① shows what happens if delegation template  $d_1$  is used for flow delegation. First, the flow rules  $f_1$ ,  $f_2$ , and  $f_3$  are relocated to the remote switch (not shown). Afterwards, a new aggregation rule with aggregation match  $\overrightarrow{m}_a$  is installed that forwards the traffic for  $f_1$ ,  $f_2$ , and  $f_3$  to the remote switch. This aggregation rule is shown here in black. At the same time, the three rules  $f_1$ ,  $f_2$ , and  $f_3$  are removed and the utilization of the flow table is reduced by  $|F_a| - 1 = 2$  (backflow rules are omitted for simplicity).

The other delegation templates  $d_2$  and  $d_3$  represent two more options. The flow table in the right labeled as ② depicts the situation if  $d_2$  and  $d_3$  are selected instead of  $d_1$ . In this case, there are two aggregation rules: one for  $d_2$  with aggregation match  $\overrightarrow{m}_b$  and another one for  $d_3$  with aggregation match  $\overrightarrow{m}_c$  (again shown in black). With this selection, utilization of the flow table is reduced by  $|F_b| + |F_c| - 2 = 4$  (the  $-2$  represents the two required aggregation rules).

<sup>3</sup>Dependent on the aggregation scheme, a delegation template could require multiple aggregation matches. This is not considered here because no such scheme is discussed in the thesis.

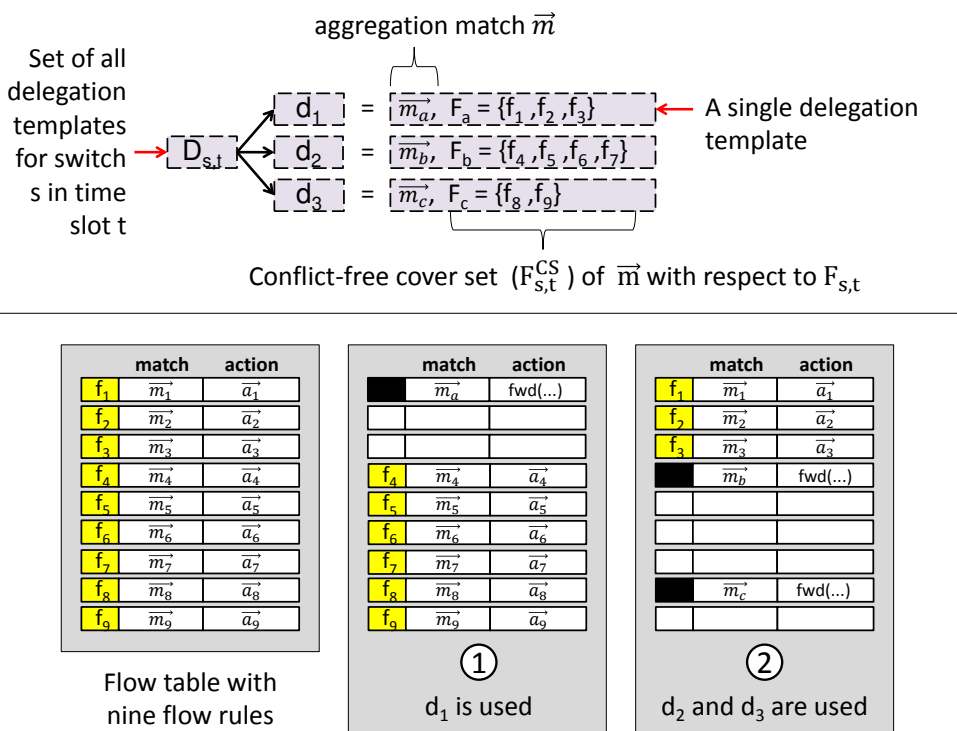


Figure 3.3: Delegation template example

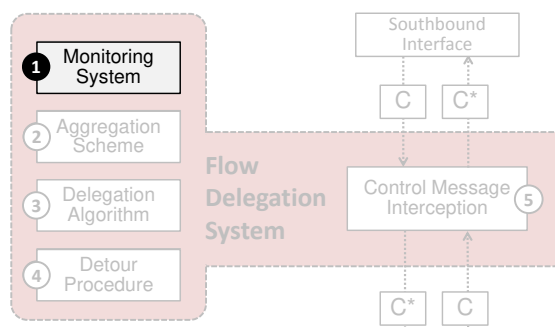
The example above shows only two valid possible selections (out of seven in this scenarios). The basic behavior, however, remains the same. If the delegation template is not selected, nothing happens. In case the delegation template is selected, an aggregation rule is created and installed into the delegation switch and the flow table utilization will be decreased by the cardinality of the cover set. The major benefit of the delegation template abstraction is separation of concerns. It allows it, for example, to develop the rule aggregation scheme completely independent from the delegation algorithm. This, in turn, makes it easy to benefit from existing algorithms for rule aggregation.

### 3.3 Conclusion

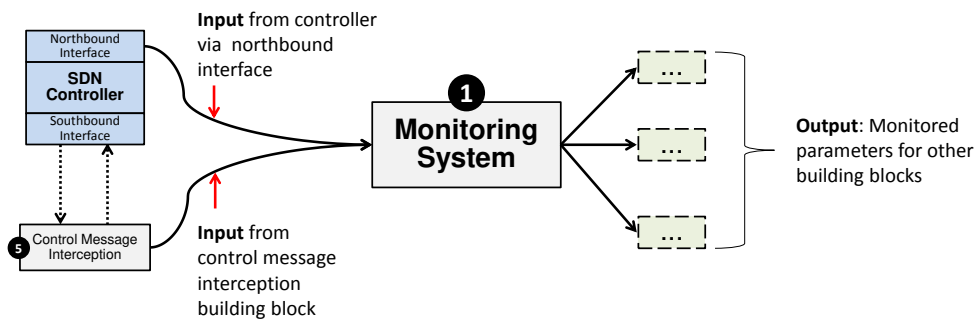
This chapter introduced the architecture of the flow delegation system that consists of five building blocks: monitoring system, rule aggregation scheme, delegation algorithm, detour procedure and control message interception. It is also shown how aggregation rules are used in the context of flow delegation and why rule conflicts lead to wrong packet processing. To deal with rule conflicts on a conceptual level, a new abstraction – the so called delegation template – is introduced. A delegation template  $d_i \in D_{s,t}$  consists of a match  $\vec{m}_i$  and a set of flow rules  $F_{i,t}$ . This can be used as a "template" to create

a new aggregation rule (by installing a rule with match  $\vec{m}_i$ ). Doing so will guarantee two important properties: The aggregation rule will match on every packet that would also be matched by one of the rules in  $F_{i,t}$  and all rules in  $F_{i,t}$  can be relocated to the remote switch without rule conflicts between a rule in  $F_{i,t}$  and other rules not in  $F_{i,t}$ . The delegation templates are used as an interface between the rule aggregation scheme, the delegation algorithm and the other building blocks so that all mechanisms are decoupled from each other.

# Monitoring System



The **monitoring system** continuously gathers information about the current state of the infrastructure and provides monitored parameters to other building blocks. The high level view is shown in Fig. 4.1.



**Figure 4.1:** Monitoring System Building Block

The “pieces of information” or data points acquired by the monitoring system are referred to as (monitored) **parameters** in the following. They are shown here as green boxes. Most parameters required for flow delegation can be collected passively by inspecting control messages exchanged between controller and switches (with the help of the control message interception building block which will be discussed in Chapter 7). Only

parameters related to individual flow rules require periodic monitoring. If the network already performs such periodic monitoring, this information can be utilized with no additional overhead. If no such monitoring is performed, the monitoring system has to carry out periodic monitoring on its own.

This chapter briefly elaborates what information is required for flow delegation and how it can be collected. Sec. 4.1 provides an overview of all relevant monitored parameters. Sec. 4.2 introduces the monitoring approach. This includes the translation from timestamp based monitoring data to time slots in Sec. 4.2.1. It also introduces the so-called lambda notation in Sec. 4.2.2 that is used as an abstraction for monitored flow rule parameters in subsequent chapters.

## 4.1 Overview

A comprehensive overview of all relevant monitored parameters is given in Fig. 4.2. It shows an example topology with five switches, two of which are presented with additional details (switch  $s$  on the left side and switch  $r$  on the right side of the figure).

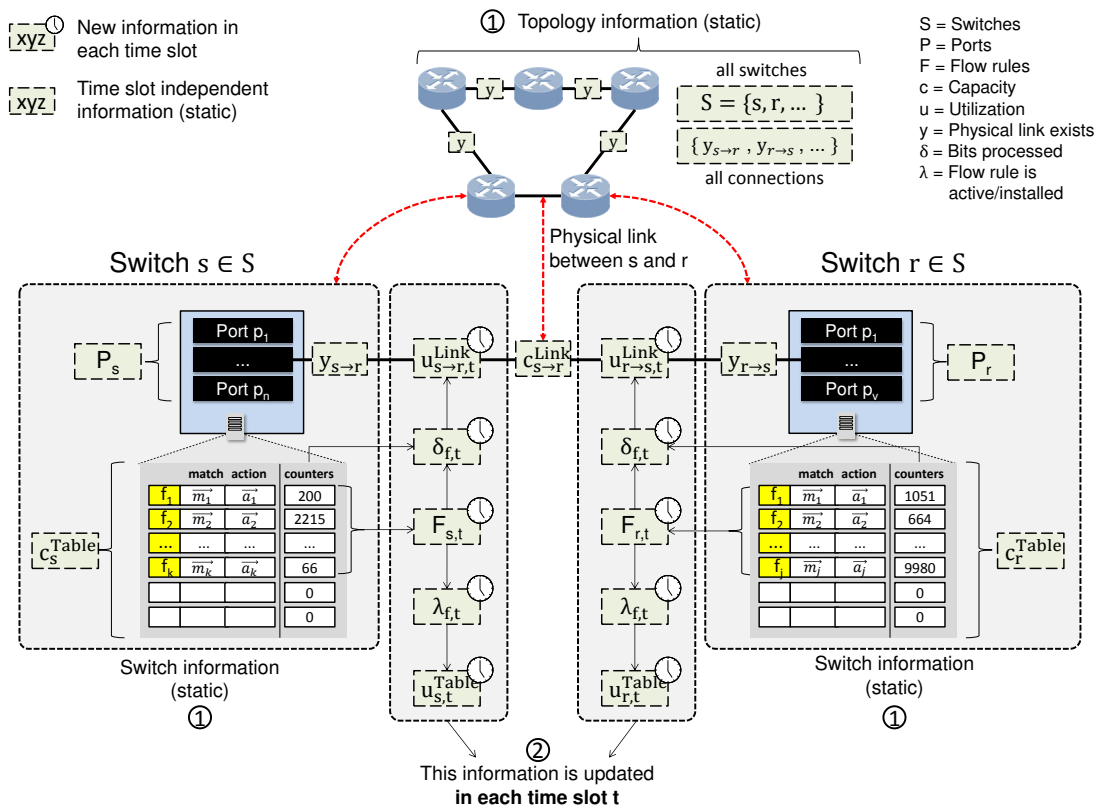


Figure 4.2: Overview of all monitored parameters



Each of the green boxes represent one parameter (or set of parameters) gathered or calculated by the monitoring system. Green boxes with a clock symbol represent parameters updated in every time slot. Green boxes without a clock symbol represent static parameters, i.e., they do not change over time.

It can be seen in the figure that the required information is organized in two categories. The first category ① consists of static infrastructure parameters. These parameters are given as a set of switches  $S$  and coefficients to indicate whether a switch is connected to one of the other switches in  $S$  ( $y_{s \rightarrow r} = 1$  means there is a direct physical link between switch  $s$  and switch  $r$ ). The second category ② consists of information that is updated in each time slot  $t$ . This is primarily the set  $F_{s,t}$  that contains all flow rules installed in the flow table of switch  $s$  in time slot  $t$  and parameters that further describe the flow rules in  $F_{s,t}$ . The individual parameters are listed in Table 4.1 and will be explained in the following two sub-sections.

Param	Range	Type	Description
<span style="border: 1px dashed green; padding: 2px;"><math>S</math></span>	$\{s_1, \dots, s_q\}$	①	Set of all switches in the infrastructure
<span style="border: 1px dashed green; padding: 2px;"><math>P_s</math></span>	$\{p_1, \dots, p_n\}$	①	Physical ports of switch $s$
<span style="border: 1px dashed green; padding: 2px;"><math>C_s^{\text{Table}}</math></span>	$\mathbb{N}$ (rules)	①	Flow table capacity of switch $s$
<span style="border: 1px dashed green; padding: 2px;"><math>C_{s \rightarrow r}^{\text{Link}}</math></span>	$\mathbb{N}$ (bits/s)	①	Link capacity of physical link between $s$ and $r$
<span style="border: 1px dashed green; padding: 2px;"><math>y_{s \rightarrow r}</math></span>	$\{0, 1\}$	①	1 if a direct physical link exists between $s$ and $r$
<span style="border: 1px dashed green; padding: 2px;"><math>F_{s,t}</math></span>	$\{f_1, \dots, f_k\}$	②	Set of flow rules installed in switch $s$ in time slot $t$
<span style="border: 1px dashed green; padding: 2px;"><math>\lambda_{f,t}^a</math></span>	$\{0, 1\}$	②	1 if flow rule $f$ is active in time slot $t$
<span style="border: 1px dashed green; padding: 2px;"><math>\lambda_{f,t}^i</math></span>	$\{0, 1\}$	②	1 if flow rule $f$ was installed in time slot $t$
<span style="border: 1px dashed green; padding: 2px;"><math>\delta_{f,t}</math></span>	$\mathbb{N}$ (bits/s)	②	Processed bits by flow rule $f$ in time slot $t$
<span style="border: 1px dashed green; padding: 2px;"><math>u_{s,t}^{\text{Table}}</math></span>	$\mathbb{N}$ (rules)	②	Flow table utilization of switch $s$ in time slot $t$
<span style="border: 1px dashed green; padding: 2px;"><math>u_{s \rightarrow r,t}^{\text{Link}}</math></span>	$\mathbb{N}$ (bits/s)	②	Link utilization between $s$ and $r$ in time slot $t$

Param = monitored parameter, ① = static parameter, ② = time slot dependent parameter

**Table 4.1:** Monitored parameters for flow delegation (see Fig. 4.2)

#### 4.1.1 Static Parameters

The static parameters are shown in the top of Table 4.1, labeled as type ①. The first four parameters are self-explanatory.

- $S$  represents the set of all active switches. This set can be acquired via the northbound interface or by intercepting control messages of type `feature_rsp` (explained in more detail below in Sec. 4.2).
- $P_s$  represents the set of physical ports of switch  $s$ . This can be acquired in the same way as set  $S$ . It could also be manually configured if the number of different switch models in the network is low.
- $C_s^{\text{Table}}$  represents the flow table capacity of switch  $s$ . Can be acquired in the same way as  $P_s$ .
- $C_{s \rightarrow r}^{\text{Link}}$  represents the link capacity of a physical link between switch  $s$  and switch  $r$ . Can be acquired in the same way as  $P_s$ .

It is assumed that these four parameters do not change over time, i.e., the topology is static and switches as well as links cannot fail. This is only a small simplification that does not affect the discussed concepts. Flow delegation can work with dynamic topologies and could also be extended to deal with link or switch failures.

The last static parameter ( $y_{s,r}$ ) is based on the assumption that delegation switch and remote switch must have a direct physical link to each other. This prevents flow delegation over more than one hop, i.e., restricts the set of potential remote switches to neighboring switches. Binary coefficients  $y_{s,r}$  are defined as follows:

$$\forall_{r \in S \setminus \{s\}} \mathcal{Y}_{s \rightarrow r} := \begin{cases} 1, & \text{if } s \in S \text{ has a direct physical link to } r \in S \\ 0, & \text{otherwise} \end{cases} \quad (4.1)$$

In total, there are  $|S|-1$  of the  $y_{s,r}$ -coefficients for each switch. The necessary information to create the coefficients can be acquired via the northbound interface or by intercepting control messages used for topology discovery. The latter is usually automatically initialized by the controller based on the link layer discovery protocol and a combination of `packet_in` and `packet_out` control messages.

In summary, all parameters of type ① in Table 4.1 can be easily acquired with either the northbound interface or the control message interception building block. And except for  $y_{s \rightarrow r}$  where a set of coefficients has to be calculated, no further processing is required from the monitoring system.

#### 4.1.2 Time Slot Dependent Parameters

The time slot dependent parameters are shown in the bottom of Table 4.1, labeled as type ②. From a monitoring perspective, these six parameters can be distinguished into three

sub-categories: structural parameters (match, action and priority), life cycle parameters (e.g., the amount of traffic processed so far or whether a rule is currently installed or not) and utilization parameters.

#### Structural Parameters:

- $F_{s,t}$  represents the set of all flow rules associated with switch  $s$  in time slot  $t$ . This includes all flow rules present in the flow table for any amount of time during the begin and the end of the time slot. For each flow rule, three structural parameters are recorded: match  $\vec{m}$ , action  $\vec{a}$  and priority value  $\text{prio}$ .

#### Life cycle Parameters:

- $\lambda_{f,t}^a$  is a binary coefficient that indicates whether flow rule  $f$  is active in time slot  $t$  or not. Active means the rule is present in the flow table for any amount of time during begin and end of the time slot (can be derived directly from  $F_{s,t}$ ).
- $\lambda_{f,t}^i$  is a binary coefficient that indicates whether flow rule  $f$  was installed in time slot  $t$  or not. This is different from  $\lambda_{f,t}^a$  because it includes knowledge about the previous time slot  $t_{-1}$  ( $f$  is installed in time slot  $t$  implies  $f$  is not active in time slot  $t_{-1}$ ). So this coefficient cannot be derived directly from  $F_{s,t}$ .
- $\delta_{f,t}$  is a non-negative integer coefficient that represents the average amount of bits/s processed by flow rule  $f$  in time slot  $t$ .

#### Utilization Parameters:

- $u_{s,t}^{\text{Table}}$  represents the flow table utilization of switch  $s$  in time slot  $t$  (number of rules stored in the flow table). Can be derived with the help of  $\lambda_{f,t}^a$ .
- $u_{s \rightarrow r,t}^{\text{Link}}$  represents the link utilization of the physical link between switch  $s$  and switch  $r$  in time slot  $t$  (given in bits/s). Can be derived with the help of  $\delta_{f,t}$ .

To acquire all six parameters, a hybrid monitoring approach is required, i.e. event-based and periodic techniques have to be combined. Pure periodic monitoring – e.g., at the end of a time slot – is not sufficient because short-lived flow rules that are created and deleted in-between the monitoring interval remain unrecognized. Pure event-based monitoring is also not sufficient because in this case, the number of processed bits is only available after the rule is removed and the lifetime of a flow rule can be much larger than the duration of a time slot. A suitable monitoring approach is presented in the next section.

## 4.2 Monitoring Approach

The monitoring approach presented here uses the control message interception building block to acquire relevant parameters in an event-based fashion (each control message is considered an event) and only uses additional polling if necessary. The idea is to intercept the following control messages exchanged between controller and switches to update relevant parameters. The control messages are defined in Sec. 2.2.4.5.

- **feature\_rsp()**: Switch  $s$  reports its capabilities to the controller. Static parameters  $S$ ,  $P_s$  and  $c_s^{\text{Table}}$  are updated accordingly.
- **install( $f$ )**: A new flow rule  $f$  is installed into the flow table of switch  $s$ .  $F_{s,t}$  is updated ( $\vec{m}$ ,  $\vec{a}$  and prio are encoded in the control message). A timestamp  $\tau^{\text{install}}$  is recorded as the install time of the flow rule.
- **delete( $f$ )**: Flow rule  $f$  is deleted.  $F_{s,t}$  is updated. A timestamp  $\tau^{\text{removed}}$  is recorded as the removal time of the flow rule.
- **counters\_rsp( $f$ )**: Number of processed bits of flow rule  $f$  are reported. A value  $\delta_{\tau_1, \tau_2}$  is recorded that stores the number of processed bits between a stored timestamp  $\tau_1$  (last event of this type) and a new timestamp  $\tau_2$  (this event).

Each of the above control messages is sent over a dedicated TCP or TLS connection. The reference to switch  $s$  can be derived from this connection. In addition to intercepting the above control messages, the monitoring system will also periodically send a monitoring request to all switches  $s \in S$  using the counters\_req control message. This serves two purposes: It ensures that the switches answer with a counters\_rsp( $f$ ) control message that is required to record the number of processed bits. And it ensures that the monitoring system can detect flow rules that were removed from the flow table because of a timeout (in order to update  $F_{s,t}$ ).

Following this simplified approach<sup>1</sup>, the monitoring system will gather the parameters listed in Table 4.2 for each flow rule. The first six parameters in the table are recorded only once per flow rule. The last parameter ( $\delta_{\tau_1, \tau_2}$ ) is recorded periodically. The following sections will now explain how these timestamp-based parameters can be converted into the parameters from Table 4.1, i.e., into their time slot based counterpart.

<sup>1</sup>There are several challenges that need to be addressed in a real system, e.g., gathering the initial state if the flow delegation system was not active from the beginning. The use of periodic monitoring to determine removed flow rules and processed bits is also not necessarily the best option with respect to performance. However, efficient monitoring in software-defined networks is a complex research area on its own (not considered here further).

Param	Description
$\vec{m}$	Match of the flow rule
$\vec{a}$	Action of the flow rule
$\text{prio} \in \mathbb{N}$	Priority of the flow rule
$s \in \mathcal{S}$	Switch associated with the flow rule
$\tau^{\text{install}} \in \mathbb{R}$	Recorded timestamp at which the flow rule is installed in the flow table of switch $s$
$\tau^{\text{removed}} \in \mathbb{R}$	Recorded timestamp at which the flow rule is removed from the flow table of switch $s$
$\delta_{\tau_1, \tau_2} \in \mathbb{N}$	Average number of bits processed by the flow rule between two consecutive timestamps $\tau_1$ and $\tau_2$

Param: monitored parameter, E = event-based, P = periodically

**Table 4.2:** Actual parameters recorded by the monitoring approach (per flow rule)



### 4.2.1 Translation from Timestamps to Time Slots

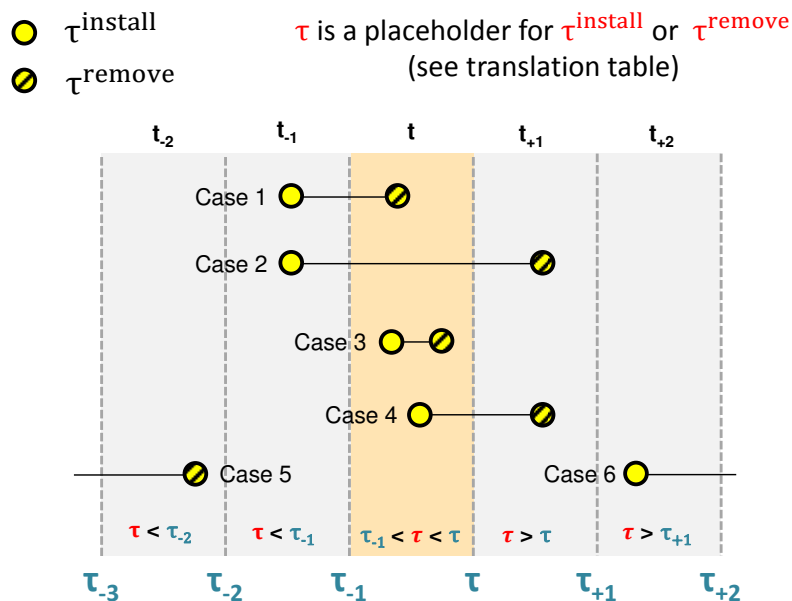
The flow rule specific parameters gathered in the previous section are based on timestamps derived from the clock of the monitoring system – which is the natural way how such data is acquired and stored. The parameters presented in Fig. 4.2 and Table 4.1, however, are based on time slots. The time slot based notation – introduced in Sec. 2.1.1 – is required for all problem formulations and algorithms that are discussed later in this document.

Timestamp based parameters can be easily “translated” to their time slot based counterpart. The crucial condition is whether the flow rule is active in a given time slot or not (for  $\lambda_{f,t}^a$ ) and whether the flow rule is installed in a given time slot or not (for  $\lambda_{f,t}^i$ ):

#### Definition 4.1: Active and Installed Flow Rules

A flow rule  $f \in F_{s,t}$  is **active** in time slot  $t$  if it is present in the flow table of switch  $s$  at least once in the time window between  $\tau_{-1}$  and  $\tau$  (the orange area in Fig. 4.3). The flow rule is **installed** in time slot  $t$ , if it is active in  $t$  and not active in  $t_{-1}$ .

Fig. 4.3 shows an example using a single flow rule  $f$ . Recall that time slot  $t$  is defined as the time window that begins at  $\tau_{-1}$  and ends at  $\tau$  which is shown here as the orange area in the middle. The four lines represent four flow rules that are installed at time  $\tau^{\text{install}}$   and removed at time  $\tau^{\text{removed}}$  .



**Figure 4.3:** Translation between timestamps and time slots

	Condition 1 <span style="color: yellow;">●</span>	Condition 2 <span style="color: yellow;">●</span>	$f$ is active in $t$	$f$ is installed in $t$
Case 1	$\tau^{\text{install}} < \tau_{-1}$	$\tau_{-1} < \tau^{\text{remove}} < \tau$	<b>Yes</b>	No
Case 2	$\tau^{\text{install}} < \tau_{-1}$	$\tau^{\text{remove}} > \tau$	<b>Yes</b>	No
Case 3	$\tau_{-1} < \tau^{\text{install}} < \tau$	$\tau_{-1} < \tau^{\text{remove}} < \tau$	<b>Yes</b>	<b>Yes</b>
Case 4	$\tau_{-1} < \tau^{\text{install}} < \tau$	$\tau^{\text{remove}} > \tau$	<b>Yes</b>	<b>Yes</b>
The flow rule is not active/installed in <b>all</b> other cases, for example:				
Case 5	$\tau^{\text{install}} < \tau_{-2}$	$\tau^{\text{remove}} < \tau_{-2}$	No	No
Case 6	$\tau^{\text{install}} > \tau_{+1}$	$\tau^{\text{remove}} > \tau_{+1}$	No	No

**Table 4.3:** Translation table (from timestamps to time slots)

To translate this into the form of  $\lambda_{f,t}^a$  and  $\lambda_{f,t}^i$ , the monitoring system has to decide whether flow rule  $f$  is **active** in a certain time slot and whether the flow rule was **installed** in a certain time slot. To decide this, the first four cases shown in Table 4.3 have to be considered. If one of the first four cases is fulfilled (which means condition 1 for ● and condition 2 for ● will both evaluate to true), the flow rule is active in time slot  $t$ . And in case 3 and case 4, the flow rule is also installed in time slot  $t$ . If none of the first four cases is fulfilled (e.g., case 5 with  $\tau^{\text{install}} < \tau^{\text{remove}} < \tau_{-2}$ ), the flow rule is neither active nor was is installed in time slot  $t$ .

Note that the time slot based notation is a theoretical construct to simplify problem formulations and algorithms. Timestamp level granularity is not required for these conceptual discussions and using time slots is far superior in terms of readability.

### 4.2.2 Lambda Notation

The translation approach from the previous section is now used to define parameters  $\lambda_{f,t}^a$  and  $\lambda_{f,t}^i$  (referred to as the lambda notation). All following sections will exclusively use this lambda notation<sup>2</sup>.

#### Definition 4.2: Lambda Notation

The **lambda notation** models the life cycle of a flow rule with time slots. It consists of binary coefficients  $\lambda$  with indices  $f$  and  $t$ .  $f$  represents a flow rule and  $t$  represents a time slot. There are two coefficient  $\lambda_{f,t}^a$  and  $\lambda_{f,t}^i$  per flow rule and time slot.  $\lambda_{f,t}^a$  and  $\lambda_{f,t}^i$  are defined as follows:

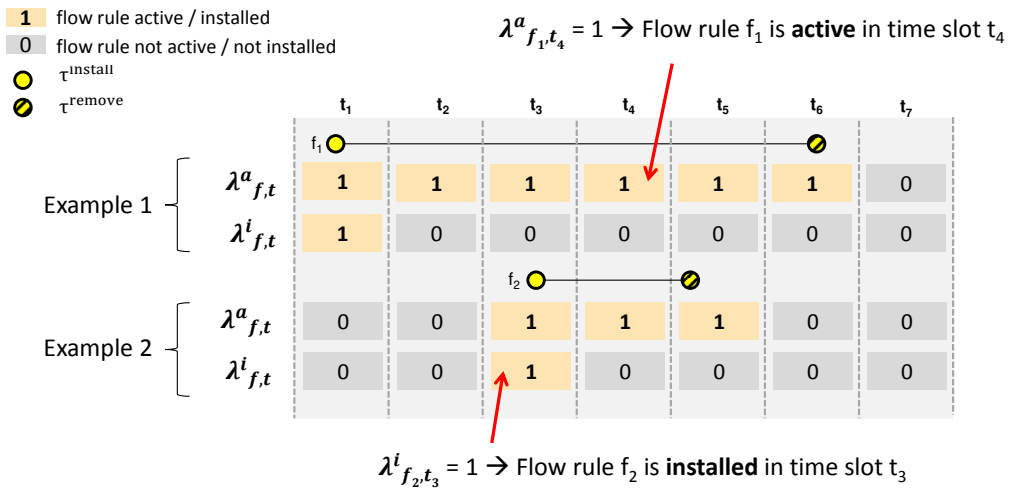
$$\lambda_{f,t}^a := \begin{cases} 1, & \text{flow rule } f \text{ is active in time slot } t \\ 0, & \text{otherwise} \end{cases} \quad (4.2)$$

$$\lambda_{f,t}^i := \begin{cases} 1, & \text{flow rule } f \text{ was installed in time slot } t \\ 0, & \text{otherwise} \end{cases} \quad (4.3)$$

Whether a flow rule is active or installed in time slot  $t$  is calculated based on Def. 4.1 and the conditions in Table 4.3. For  $t < t_1$ , both coefficients are defined as 0.

The two binary coefficients are intuitive to use.  $\lambda_{f,t}^a \in \{0, 1\}$  is set to 1 for all time slots where flow rule  $f$  is active in the flow table and  $\lambda_{f,t}^i \in \{0, 1\}$  is set to 1 only for the one time slot where the flow rule is installed.  $\lambda_{f,t}^i$  is required for many equations in subsequent chapters where it is important to differentiate between the case that a flow rule was already active in the previous time slot (i.e.,  $\lambda_{f,t}^i = 0$ ) or not ( $\lambda_{f,t}^i = 1$ ). This could be expressed as  $\lambda_{f,t}^i = (1 - \lambda_{f,t-1}^a) * \lambda_{f,t}^a$  (for example) but this would severely reduce readability in larger equations. Further note that the switch is not specified as an index here, because each flow rule  $f \in F_{s,t}$  is always implicitly associated with one switch.

<sup>2</sup>As a result, the timestamp variables ( $\tau^{\text{install}}$ ,  $\tau^{\text{removed}}$  and  $\delta_{\tau_1, \tau_2}$ ) are rarely used, only here and in parts of the evaluation. However, it is important to understand where the lambda notation comes from and how it can be derived from real monitoring data.



**Figure 4.4:** Example for the lambda notation

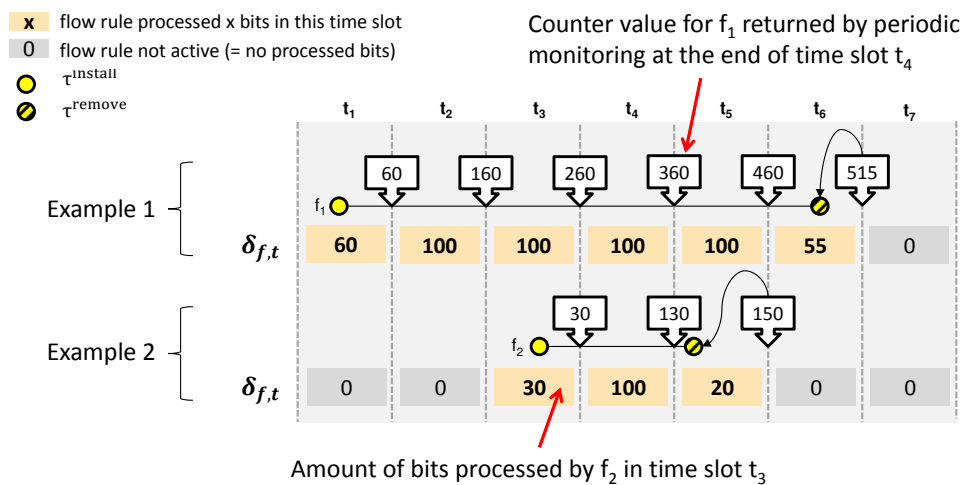
Because the lambda notation is used extensively, Fig. 4.4 shows two examples for two flow rules  $f_1$  and  $f_2$  to consolidate the idea. The two lines represent the exact time where  $f_1$  and  $f_2$  are present in the flow table of the switch (between timestamps  $\tau^{\text{install}}$  and  $\tau^{\text{removed}}$ ). In the lambda notation, this information is abstracted with two coefficients per time slot. Consider example 1 in Fig. 4.4. Given that there are seven time slots in the example, there will be 14 coefficients per flow rule. Seven for  $\lambda^a_{f_1, t}$  and seven for  $\lambda^i_{f_1, t}$ . The first  $\lambda^a_{f_1, t}$  coefficient for the first time slot is set to 1 because the flow rule is active (case 2 in Table 4.3). The second coefficient for the second time slot is also set to 1 (case 3 in Table 4.3). This continues for each time slot.

### 4.2.3 Further Translations

The number of processed bits ( $\delta_{\tau_1, \tau_2}$  in Table 4.2) is translated to time slot notation in a similar way. Here, it is assumed that the counters in the flow table are queried at the end of each time slot and at the time the flow is removed from the flow table ( $\tau^{\text{removed}}$ ). The former can be done by periodically sending a `counters_req()` control message to each switch as proposed above in the monitoring approach. The latter can be done by sending a status message from the switch to the controller if a flow rule is removed from the flow table (not discussed here further).

The actual translation is done based on the returned counters and a process very similar to the process described in Sec. 4.2.1 – not shown here in detail (trivial). The number of





**Figure 4.5:** Usage of  $\delta_{f,t}$

bits processed by a flow rule  $f$  in one time slot  $t$  is then defined analogous to the lambda notation:

$$\delta_{f,t} := \begin{cases} x \in \mathbb{N}, & \text{amount of bits processed by } f \text{ between } \tau_{-1} \text{ and } \tau \\ 0, & \text{flow rule } f \text{ is not active in time slot } t \text{ (i.e., } \lambda_{f,t}^a = 0) \end{cases} \quad (4.4)$$

Fig. 4.5 illustrates the usage of  $\delta_{f,t}$  using the same two example flow rules from above. There are seven non-negative integer coefficients  $\delta_{f,t}$  per flow rule (one per time slot) that represent the amount of processed bits.

Given  $\lambda_{f,t}^a$  and  $\delta_{f,t}$ , the utilization parameters can be defined as follows. The flow table utilization of switch  $s$  in time slot  $t$  is given as the number of flows that are active in the flow table in time slot  $t$  (which is  $|F_{s,t}|$ ). This can be expressed as:

$$u_{s,t}^{\text{Table}} := |F_{s,t}| = \sum_{f \in F_{s,t}} \lambda_{f,t}^a \quad (4.5)$$

The link utilization for a physical link between switch  $s$  and switch  $r$  is slightly more complex because it depends not only on  $\delta_{f,t}$  but also on the action of the flow rules (only

a subset of the flow rules will forward traffic towards switch  $r$ ). The following helper coefficient extracts the required information from existing parameters ( $F_{s,t}$  and  $S$ ):

$$\forall_{f \in F_{s,t}, r \in S \setminus \{s\}} : \phi_{f,r} := \begin{cases} 1, & \text{Action } \vec{a} \text{ of flow rule } f \text{ contains } \text{fwd}(r) \\ 0, & \text{otherwise} \end{cases} \quad (4.6)$$

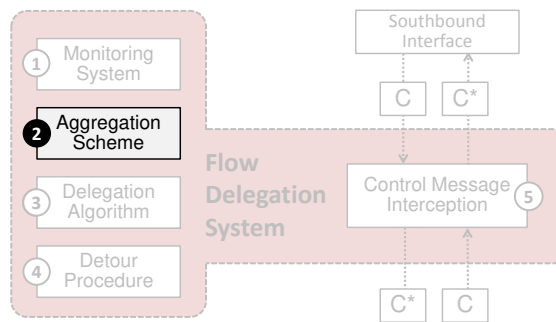
This will result in  $|S| - 1$  coefficients for each flow rule in  $f \in F_{s,t}$ . These coefficients indicate whether the bits processed by  $f$  are transmitted towards switch  $r \in S \setminus \{s\}$  or not (recall that match and action are considered static and this coefficient does not change). With these coefficients, the utilization for a physical link between switch  $s$  and switch  $r$  can be expressed as:

$$u_{s \rightarrow r,t}^{\text{Link}} := \sum_{f \in F_{s,t}} \delta_{f,t} * \phi_{f,r} \quad (4.7)$$

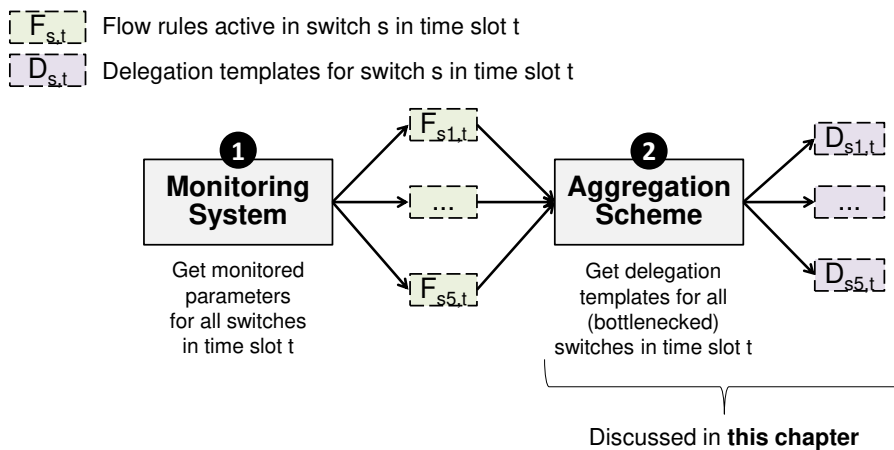
### 4.3 Conclusion

This section introduces relevant parameters to be monitored in order to realize flow delegation. It is furthermore explained how timestamp based monitoring data is translated into a time slot counterpart and how the important lambda notation is used. All concepts and algorithms in the following are based on the parameters in this chapter.

# Rule Aggregation Scheme



The **rule aggregation scheme** takes a set of flow rules  $F_{s,t}$  active in switch  $s$  in time slot  $t$  and calculates a set of delegation templates  $D_{s,t}$ . Each element in  $D_{s,t}$  represents one “delegation option” which may or may not be considered by the delegation algorithm in the next step (third building block). The high level view is shown in Fig. 5.1.



**Figure 5.1:** Rule aggregation scheme building block

Sec. 5.1 briefly discusses three conceptual approaches to realize the rule aggregation scheme. Sec. 5.2 then introduces the novel concept of indirect rule aggregation that is applied for flow delegation. The chapter concludes with a remark on aggregation priority in Sec. 5.3 and a short summary.

## 5.1 Different Approaches for the Rule Aggregation Scheme

This section briefly discusses different approaches to realize the rule delegation scheme: As a new functionality of the controller (Sec. 5.1.1), based on monitored parameters (Sec. 5.1.2) and indirectly with the help of additional context information (Sec. 5.1.3).

### 5.1.1 New Controller Functionality

One way to create delegation templates is based on explicit support from the controller. Potential rule conflicts are considered while flow rules are created and the controller exposes a list of delegation templates via the northbound interface. This approach works especially well if there are already conflict-free sets of flow rules in the network. Assume a system such as FlowVisor [She+09] or Covisor [Jin+15] that creates isolated slices for different controllers or network applications. Because of the inherent isolation requirement, delegation templates can be easily constructed based on the individual slices (rule conflicts across slices would violate the isolation requirement). This approach does not work with arbitrary network applications and requires changes in the controller. The general idea of exploiting existing conflict-free sets, however, is promising. It is picked up by the tagging scheme introduced later in Sec. 5.2.2.1.

### 5.1.2 Based on Monitored Parameters

Another (more general) approach is it to calculate delegation templates outside of the controller based on monitored parameters, e.g., for a set of flow rules  $F_{s,t}$  that was provided by the monitoring system. This can be performed on-demand (e.g., when a bottleneck is detected) and deterministically for arbitrary flow rule sets and does not require changes in the controller. A generic algorithm for delegation template calculation is shown in Alg. 2.

It consists of two steps. In the first step (line 3) a set of aggregation matches  $M^{\text{agg}}$  is calculated via `get_aggregation_matches()`. A naive approach, for example, can iterate over all possible wildcard matches to calculate candidates for  $M^{\text{agg}}$ . This can be improved by using concepts such as the  $H$ -distance from [LYL15] or schemes based on traffic entropy [Wan+17] to calculate the most relevant aggregation matches for a given set of flow rules. Most caching algorithms for software offloading from Sec. 2.4.2.3 can also be

**Algorithm 2:** Generic rule aggregation scheme algorithm

---

**Data:** set of flow rules  $\overline{F_{s,t}}$

**Result:** set of delegation templates  $\overline{D_{s,t}}$

```

1 Function rule_aggregation_scheme( $\overline{F_{s,t}}$ ):
2    $\overline{D_{s,t}} \leftarrow \{\}$ 
3   /* Part 1: calculate potential aggregation match */
4    $M^{agg} \leftarrow \text{get\_aggregation\_matches}(\overline{F_{s,t}})$ 
5   /* Part 2: calculate conflict-free cover set for each aggregation match */
6   for  $\overrightarrow{m_{agg}} \in M^{agg}$  do
7      $F_{s,t}^{CS} = \text{get\_conflict\_free\_cover\_set}(\overrightarrow{m_{agg}}, \overline{F_{s,t}})$ 
8      $d = \langle \overrightarrow{m_{agg}}, F_{s,t}^{CS} \rangle$ 
9      $\overline{D_{s,t}} \leftarrow \overline{D_{s,t}} \cup \{d\}$ 
10  end
11  return  $\overline{D_{s,t}}$ 

```

---

used. The dependency graph based algorithms in [Kat+14a], for example, can be used for `get_aggregation_matches()` without any modifications. This was successfully shown in the context of a master thesis. Other approaches such as [Yan+14; SC16; Wan+17; Din+17; Yan+18a] can also be used with small changes (from a conceptual point of view, this was not validated in practice).

To account for the rule conflict problem, a second step (lines 4 to 8) is required if `get_aggregation_matches()` does not already ensure that calculated matches form a conflict-free cover set with respect to  $F_{s,t}$  (the algorithms in [Kat+14a], for example, ensure this automatically and this step could be simplified). It consists of a for loop that iterates over all matches in  $M^{agg}$  and uses Alg. 1 to determine the conflict-free cover set.

### 5.1.3 Inferred from Context

A third approach is inferring delegation templates with the help of additional context information, e.g., based on the topology of the network or the history of flow rules that were installed in the past. This is different from the approach in the last section in the sense that the set of delegation templates  $D_{s,t}$  is not derived on-demand from the current set of installed flow rules  $F_{s,t}$ . Instead,  $D_{s,t}$  is built "indirectly" based on assumptions about the topology of the network (for example). These approaches are here referred to as **indirect rule aggregation schemes**.

The main benefit of such an approach is that – based on the design – delegation templates can be defined in a static way without complex calculations. To the best of my knowledge, the first indirect rule aggregation scheme proposed in the context of flow table capacity bottlenecks is the so-called flow-to-ingress-port mapping presented by the author of this thesis in [BZ16]. The ingress port scheme is explained further below. Prior to that, the general concept of indirect rule aggregation is introduced in the next section.

## 5.2 Indirect Rule Aggregation Schemes

An indirect rule aggregation scheme returns an independent set of  $n$  delegation templates that will automatically fulfill the delegation template requirements from Def. 3.7, i.e., there can be no rule conflicts by design.

### Definition 5.1: Indirect Rule Aggregation Scheme

An **indirect rule aggregation scheme** provides an independent set of  $n$  delegation templates  $D_{s,t} = \{d_1, \dots, d_n\}$  for a set of flow rules  $F_{s,t}$ . Independent means the delegation templates do not depend directly on  $F_{s,t}$ . The properties of a delegation template must be fulfilled, i.e., for each  $d_i = \langle \vec{m}_i, F_{i,t} \rangle \in D_{s,t}$ , set  $F_{i,t} = F_{s,t_i}^{\text{CS}}$  is a conflict-free cover set of aggregation match  $\vec{m}_i$ .

This is fundamentally different from existing approaches because there will be always  $n$  delegation templates regardless of the currently installed flow rules in the delegation switch ( $F_{s,t}$  can contain arbitrary flow rules). Note that  $n$  is usually static (but this is not a hard requirement).

Given that indirect rule aggregation is used, not only the number of delegation templates stays the same but the templates in  $D_{s,t}$  have the **exact same aggregation match** that will not change over time as long as the context information stays the same. Let  $d = \langle \vec{m}_d, F_{d,t_1} \rangle$  and  $d' = \langle \vec{m}'_d, F_{d',t_2} \rangle$  be two delegation templates calculated from the same context information (e.g., the same ingress port) for two consecutive time slots  $t_1$  and  $t_2$ . In this case, the aggregation matches  $\vec{m}_d$  and  $\vec{m}'_d$  will be identical and the associated aggregation rule is also identical if the same remote switch is used for both time slots ( $F_{d,t_1}$  and  $F_{d',t_2}$  can be different if flow rules are added or removed between  $t_1$  and  $t_2$ ). This means the maximum number of different aggregation rules – and the maximum overhead (aggregation rules are overhead) – is limited by  $n$ , which is desirable if  $n$  is a reasonable low number as it is in the ingress port case. This is the reason why the delegation algorithm discussed in the next chapter can simply iterate over the "same" set

of delegation templates for multiple time slots<sup>1</sup>. It is also very useful for the multi period case where a set of time slots  $t \in T$  has to be considered (can be modeled more easily).

Indirect rule aggregation schemes, however, can only be used if certain prerequisites are fulfilled. These prerequisites are explained in Sec. 5.2.1. Afterwards, three examples of different indirect rule aggregation schemes are proposed in Sec. 5.2.2. Finally, Sec. 5.2.3 presents a generic algorithm to implement such schemes.

### 5.2.1 Prerequisites

Indirect rule aggregation schemes as defined in Def. 5.1 only work conflict-free if certain prerequisites are fulfilled. These are generic in the sense that they can be applied to any indirect scheme that maps on Def. 5.1. They abstract from specific example schemes such as the ingress port scheme. The four key prerequisites are:

- (R1) A set of  $n$  delegation templates  $D_{s,t} := \{d_1, \dots, d_n\}$  can be defined independently of the set of installed flow rules  $F_{s,t}$
- (R2) Each packet  $z \in \mathcal{Z}$  can be linked to exactly one delegation template and the delegation template for all packets  $z \in \mathcal{Z}$  can be inferred from packet  $z$  using the same set of packet header fields
- (R3) Each rule in  $f \in F_{s,t}$  can be linked to exactly one delegation template and the delegation template for all rules  $f \in F_{s,t}$  can be inferred from rule  $f$
- (R4) A packet  $z \in \mathcal{Z}$  linked to delegation template  $d_i$  is always processed by a rule  $f$  that is also linked to delegation template  $d_i$

The following shows that no rule conflicts can occur if an aggregation scheme fulfills the above prerequisites. R1 first ensures there are  $n$  delegation templates. R2 ensures a packet can only belong to a single delegation template. This means the set of all packets  $\mathcal{Z}$  is divided into  $n$  disjoint sets  $Z_1, \dots, Z_n$  (one for each delegation template) and each packet can only belong to one of these sets. R2 further ensures that, for one set of packet header fields  $K = \{k_1, \dots, k_m\}$ , there are  $n$  different sets of values  $\{V_1, \dots, V_n\}$  with  $V_i = \{v_1, \dots, v_m\}$  which can unambiguously identify one of the  $n$  delegation templates. This means there is a match  $\vec{m}_i = \left( \langle k_j, v_j \rangle \mid j = 1, \dots, m, k_j \in K, v_j \in V_i \right)$  for each of the delegation templates and the symbolic set for  $\vec{m}_i$  is  $Z_i$  (follows directly from R2 and the definition of the symbolic set). Furthermore, R3 says the set of all flow rules  $F_{s,t}$

---

<sup>1</sup>It is also the reason why it is possible to use  $D_s$  instead of  $D_{s,t}$  to describe all available delegation templates in the ingress port scheme (the templates are time independent)

can be divided into  $n$  disjoint sets  $F_1, \dots, F_n$  where  $F_i$  represents all flow rules linked to delegation template  $d_i$ .

What is left, is to show  $F_i$  is actually the conflict-free cover set of match  $\vec{m}_i$  with respect to the set of flow rules  $F_{s,t}$ . According to Def. 3.7, a conflict-free cover set requires two properties:

- i)  $F_i$  is a cover set of match  $\vec{m}_i$  with respect to  $F_{s,t}$
- ii) every rule  $f \in F_{s,t}$  that has a rule conflict with a rule in  $F_i$  is included in  $F_i$

Say  $F_i$  consists of  $r$  flow rules  $f_1, \dots, f_r$ . Then define  $Z_{F_i} := \bigcup_{j=1, \dots, r} Z_j$  as the united symbolic set of  $F_i$  (here,  $Z_j$  is the symbolic set of rule  $f_j$ ). For the first property, recall from above that the symbolic set of match  $\vec{m}_i$  is  $Z_i$ . So it must be shown that  $Z_{F_i} \subseteq Z_i$ . This, however, follows directly from R4 which ensures that a packet  $z \in Z_i$  is always processed by a rule in  $F_i$  and a packet  $z \notin Z_i$  is not processed by a rule in  $F_i$ . And for the second property: It can be derived from Def. 3.5 that, for a conflict to occur, there has to be a flow rule  $f \notin F_i$  so that  $Z_f \cap Z_{F_i} \neq \emptyset$ . However, from  $f \notin F_i$  follows  $Z_f \subseteq Z \setminus Z_i$  and it was already shown that  $Z_{F_i} \subseteq Z_i$ . And because  $(Z \setminus Z_i) \cap Z_i = \emptyset$ , it can be followed that  $Z_f \cap Z_{F_i} = \emptyset$  (if the intersection of two super sets is empty, the intersection of two subsets has to be empty as well). This shows that no such flow rule  $f \notin F_i$  can exist and the second property is fulfilled.

## 5.2.2 Scheme Examples

The following proposes three different indirect rule aggregation schemes that use context information to infer delegation templates. Recall from the previous section that this can only work if certain prerequisites are met.

### 5.2.2.1 Tagging Scheme

This first proposed scheme for indirect rule aggregation can be applied if all packets in the network are tagged with an explicit identifier. Assume, for example, a network that manages only VLAN encapsulated traffic (IEEE 802.1Q). Or a network where user applications tag their own traffic with an application identifier (e.g., in the ToS bits). Under the assumption that all flow rules are tag-specific (matches use an exact value to match on a tag and not a wildcard value), a set of delegation templates can be defined where each delegation template represents one tag.

In this case, R1 (see prerequisites from Sec. 5.2.1) is fulfilled because the number of delegation templates does not depend on the set of installed flow rules. R2 is also fulfilled because the information is directly encoded in the packet. R3 is fulfilled as long as no wildcard matches on the tags are allowed. And R4 is fulfilled if there are no untagged packets in the network.



These are obviously strong assumptions, but they make up for a very easy aggregation scheme if they are fulfilled. Also note that this scheme can be extended to support networks where only a portion of the traffic is tagged or wildcard matching on tags is allowed. Given the number of tag-specific flow rules is large compared to the number of wildcard flow rules, for example, the rule conflict problem could be avoided by simply adding a copy of all wildcard flow rules into each delegation template (or by using Alg. 1). These and other possible optimizations, however, are not discussed here in detail.

### 5.2.2.2 Slicing Scheme

The second proposed scheme picks up the network slicing idea that was already briefly discussed in Sec. 5.1.1. If the network is virtualized into  $n$  different slices, the isolation mechanism of the slicing approach – such as FlowVisor [She+09] – ensures that traffic of slice  $i$  can only be processed by a set of flow rules  $F_i$  that are associated with slice  $i$ . In the original version of FlowVisor, for example, isolation is ensured by assigning each slice a dedicated portion of the so-called flowspace. This means a slice can only work on specific packets defined as set  $Z_i \subset \mathcal{Z}$  and this set is not allowed to overlap with any other set  $Z_j \subset \mathcal{Z}$  for all  $j \neq i$ . In other words: all flow rules in  $F_i$  are forced to have a match  $\vec{m}_j$  with symbolic set  $Z_{\vec{m}_j} \subseteq Z_i$ .

Now assume the aggregation scheme uses one delegation template to represent one slice. This already fulfills two of the four prerequisites from Sec. 5.2.1. R1 is given by the number of currently active slices which is not necessarily static over time but independent from the set of installed flow rules. And R4 follows directly from the isolation mechanism. R2 and R3 are not fulfilled directly. R3 demands that each rule  $f \in F_{s,t}$  can be linked to exactly one delegation template and the delegation template for all rules  $f \in F_{s,t}$  can be inferred from rule  $f$ . The first part is fulfilled because each rule is part of a slice  $i$  and the isolation mechanism ensures that  $\bigcup_{f \in F_i} Z_{f_i} \subseteq Z_i$  for each slice (same argument for R2). The second part, however, is not fulfilled. Because the rule aggregation scheme does not know how the individual slices are constructed by the slicing approach, it cannot infer the correct delegation template from flow rule  $f$  (or from a packet header of packet  $z$  in case of R2).

This can be solved in two ways: i) The slicing approach exposes the way how slices are defined (e.g., by providing a list of slices together with the packet header fields that are used for isolation). In this case, R2 and R4 are fulfilled and the rule aggregation scheme can be used without conflicts. ii) The behavior of the isolation mechanism is inferred automatically, e.g., with the help of machine learning (cluster analysis on  $\mathcal{Z}$  to identify the clusters  $Z_i$ ).

### 5.2.2.3 Ingress Port Scheme

The third proposed scheme uses the ingress port identifier for aggregation (this approach is used in the remainder of this document). The key idea is as follows: each physical port  $p_i \in P_s$  of switch  $s$  defines a delegation template  $d_i \in D_{s,t}$  and flow rules processing traffic that arrives at port  $p_i$  are associated with delegation template  $d_i$ . This is the flow-to-ingress-port mapping from [BZ16] mentioned above.

In this scheme, R1 is fulfilled because the delegation templates can be derived from the number of physical ingress ports of a switch. R2 is fulfilled because a single packet  $z$  can only arrive at one ingress port<sup>2</sup>. The two challenging prerequisites are R3 (each rule can be linked to exactly one delegation template) and R4 (a packet linked to delegation template  $d_i$  is actually processed by a rule that is linked to delegation template  $d_i$ ). Main problem: While the ingress port can be used in a match, the controller and the network applications are not forced to do so, i.e., the match for `in_port` can also be set to a wildcard (the two other schemes above suffered from a similar problem). Such rules will violate prerequisite R3 because it is not possible to determine the corresponding delegation template. It also violates R4 because a packet with ingress port  $p_i$  might be processed by a higher priority rule with a wildcard match for `in_port`.

This issue cannot be easily solved<sup>3</sup> but it is possible to mitigate the effects to a large extent using information from context – in this case, from the `packet_in(z, p_i)` control messages. The scheme distinguishes between three cases:

- (1) Rules that do match on a single `in_port`  $p_i$  are linked to delegation template  $d_i$
- (2) For reactively installed flow rules that do not match on `in_port` the delegation template is inferred from the ingress port specified in the `packet_in(z, p_i)` control message that was sent to the controller.
- (3) For proactive rules and reactive rules where the match on `in_port` is explicitly set to a wildcard, an additional delegation template  $d_0$  is introduced that represents rules that cannot be relocated.

Note that this cannot guarantee R4 and R5 because i) there can be rule conflicts between  $d_0$  and one of the other delegation templates  $d_1$  to  $d_n$  and ii) inferring the delegation template for reactively installed flow rules is not necessarily 100% accurate. So it is still required to make use of rule conflict detection (which would not be necessary if R4 and

<sup>2</sup>Note that multiple packets of the same flow can arrive at different ports but a single packet can not. The former case has to be covered by R4 (is not guarantee by R2)

<sup>3</sup>At least not without (minor) changes in the controller or the applications (which could be a viable approach in practice)

R5 are guaranteed). However, this will not change the main benefit of the approach – which is the static number of delegation templates that are independent from  $F_{s,t}$ .

#### 5.2.2.4 Summary

Table 5.1 summarizes the three indirect rule aggregation schemes proposed above with respect to the prerequisites R1 to R4 introduced in Sec. 5.2.1.

	Tagging scheme (Sec. 5.2.2.1)	Slicing scheme (Sec. 5.2.2.2)	Ingress port scheme (Sec. 5.2.2.3)
R1	one delegation template for each tag	one delegation template for each network slice	one delegation template for each physical ingress port
R2	fulfilled because the necessary information is encoded directly in the packet	fulfilled by the isolation mechanism of the slicing approach	fulfilled by the restriction that a single packet can only arrive at one ingress port
R3	fulfilled for flow rules with exact match on tag (no wild-cards)	can be provided by slicing mechanism or has to be inferred automatically	fulfilled for rules that match on <code>in_port</code> , can be inferred for reactive rules
R4	fulfilled if all packets are tagged	fulfilled by isolation mechanism / slicing approach	fulfilled because of R2 but only if R3 is also fulfilled

**Table 5.1:** *Indirect rule aggregation schemes proposed in this work*

Note that none of the three schemes can guarantee all four prerequisites without making assumptions about the flow rules and packets used in the network. While this may seem unfortunate at first glance, it does not diminish the main advantage of the indirect approach – which is delegation templates are calculated independently of  $F_{s,t}$ . It only means some additional countermeasures (such as using Alg. 1 on the inferred templates) are necessary to deal with potential rule conflicts that can arise if prerequisites are violated. A general algorithm that takes this into account is presented in the next section.

### 5.2.3 Algorithm for Indirect Rule Aggregation

Alg. 3 shows a generic algorithm for indirect rule aggregation schemes. It is highly simplified and uses helper functions that are not explained here in detail (the concrete implementation depends on the applied indirect aggregation scheme, the southbound protocol and potentially also context parameters such as topology or used switches). However, it shows the conceptual differences between the indirect approach and aggregation schemes that are calculated based on  $F_{s,t}$  (see Sec. 5.1.2).

The input and output is identical to Alg. 2. The first difference is that the number of delegation templates is calculated independently of  $F_{s,t}$  in line 3. It can depend on context

**Algorithm 3:** Generic indirect rule aggregation scheme algorithm

---

**Data:** set of flow rules  $\overline{F}_{s,t}$

**Result:** set of delegation templates  $\overline{D}_{s,t}$

```

1 Function indirect_rule_aggregation_scheme():
2    $\overline{D}_{s,t} \leftarrow \{\}$ 
3   /* Part 1: determine the number of delegation templates */
4    $n \leftarrow \text{get\_number\_of\_delegation\_templates}()$ 
5   for  $i \in \{1, \dots, n\}$  do
6     /* Part 2: build aggregation matches from context */
7      $K_i, V_i = \text{get\_packet\_header\_fields\_and\_values}(i)$ 
8      $\vec{m}_i \leftarrow \langle \langle k_j, v_j \rangle \mid k_j \in K_i, v_j \in V_i \rangle$ 
9     /* Part 3: deal with rule conflicts if necessary */
10    if  $\text{conflicts\_possible}(i, \overline{F}_{s,t})$  then
11       $F_{s,t}^{CS} = \text{get\_conflict\_free\_cover\_set}(\vec{m}_i, \overline{F}_{s,t})$ 
12    else
13       $F_{s,t}^{CS} = \text{get\_cover\_set}(\vec{m}_i, \overline{F}_{s,t})$ 
14    end
15     $d = \langle \vec{m}_i, F_{s,t}^{CS} \rangle$ 
16     $\overline{D}_{s,t} \leftarrow \overline{D}_{s,t} \cup \{d\}$ 
17  end
18 end

```

---

parameters, though (such as number of ports of a switch in the ingress port scheme). The second difference is the helper function in line 5. This function takes a reference to a delegation template (represented as index variable  $i \in \mathbb{N}$ ) and provides the aggregation scheme with two ordered sets  $K_i$  and  $V_i$ .  $K_i$  contains the packet header fields for the aggregation match of the  $i$ -th delegation template and  $V_i$  contains the corresponding values. The content of the sets depend on the scheme:

- The tagging scheme may return  $K_i = \{\text{vlan\_id}\}$  and  $V_i = \{207\}$  if tagging is realized with VLAN encapsulation (207 is one out of  $n$  used VLAN tags)
- The slicing scheme may return  $K_i = \{\text{ip\_src}, \text{ip\_dst}\}$  and  $V_i = \{10.0.0.0/8, 10.0.0.0/8\}$  if isolation is based on IP sub nets and the  $i$ -th out of  $n$  slices is only allowed to use subnet 10.0.0.0/8.
- The ingress port scheme may return  $K_i = \{\text{in\_port}\}$  and  $V_i = \{p_i\}$  where  $p_i$  is the  $i$ -th out of  $n$  physical ingress port of the switch

Based on  $K_i$  and  $V_i$ , the aggregation match for the delegation template is created in line 6, also independent from the rules in  $\overline{F}_{s,t}$ . The third and last difference is the additional

check whether rule conflicts can occur or not in lines 7 to 11. The first branch where the conflict-free cover set is constructed is only required if one or more of the four prerequisites from Sec. 5.2.1 are not fulfilled (and it is possible to optimize this step). If the aggregation scheme guarantees that no rule conflicts can occur, it is sufficient to calculate the normal cover set that consists of all flow rules in  $F_{s,t}$  that are matched by  $\vec{m}_i$  (which is much easier).

### 5.3 Remarks on Aggregation Priority

Delegation templates consist of an aggregation match and a conflict-free cover set (see Def. 3.7). Both aspects are discussed in detail above. Action and priority, however, are not intensively discussed in this chapter beside the fact that they are also required if a new aggregation rule is created. In case of the action, this is perfectly reasonable. The action of an aggregation rule primarily consists of a forwarding instruction that will redirect matched packets to the remote switch (which is selected by the delegation algorithm). So this decision is completely independent from the rule aggregation scheme. The priority of the aggregation rule, on the other hand, is indeed important here which was shown in the example in Sec. 3.2.2 and the discussions on the rule conflict problem. However, the priority is still not included in the delegation template.

The reason for this is simple: it is assumed that the aggregation priority is static. There are two options for the static priority: highest possible priority ( $\text{prio}_{\text{agg}}^{\text{highest}}$ ) or lowest possible priority ( $\text{prio}_{\text{agg}}^{\text{lowest}}$ , higher than the default rule but lower than all other rules). In the first case, all rules in the conflict-free cover set are relocated. In the second case, only new flow rules installed after the aggregation rule are relocated. Both options have benefits and drawbacks.

$\text{prio}_{\text{agg}}^{\text{highest}}$ : Easy to implement because the delegation template abstraction guarantees that no rule conflicts can occur. However, using the high priority option leads to massive control overhead if the conflict-free cover set of a delegation template is large because every single rule has to be relocated to the remote switch – which requires a large amount of control messages. This is especially problematic in the case of indirect rule aggregation schemes where the number of delegation templates is static (e.g., equal to the number of physical ingress ports in the case of the ingress port scheme) and the expected size of the conflict-free cover sets is large.

$\text{prio}_{\text{agg}}^{\text{lowest}}$ : More efficient compared to the high priority option because the number of relocated flow rules is small in the beginning and increases slowly over time when new flow rules are installed. The obvious drawback is that it is not longer

guaranteed that no rule conflicts can occur. Recall that the delegation template abstraction does only guarantee there are no rule conflicts with *other* delegation templates. So within a single delegation template, rules can still depend on each other. It is therefore not possible to relocate only a subset of the rules in one delegation template, at least not without additional rule conflict detection.

This thesis uses the second, low priority option. There are three reasons that lead to this decision. First, low priority aggregation rules are more efficient and not as disruptive as high priority delegation rules from the perspective of the packets in the network. Secondly, it is much easier to translate the algorithms for the low priority case into the high priority case as opposed to the other way round (the high priority case is simpler). The thesis will therefore only discuss the more complex case. And thirdly, the main drawback of the low priority option (rule conflicts) is not as critical as it seems. Recall from Sec. 5.2.2.4 that indirect rule aggregation schemes already require additional countermeasures for rule conflicts to deal with cases where some of the prerequisites R1 to R4 from Sec. 5.2.1 are violated. So in practice, there is no additional overhead (rule conflict detection is required anyway).

Flow rule delegation with an indirect aggregation scheme and low priority aggregation rules works as follows: for each flow rule installed after the aggregation rule, it is first checked whether this rule has a rule conflict with one of the rules in the associated delegation template (R3 ensures that it is possible to link each flow rule to one delegation template). If there is no conflict, the rule is relocated to the remote switch. In case of a conflict, all conflicted rules are also moved the remote switch or the rule is installed in the delegation switch. This can be easily decided individually for each flow rule (more details on this are given in Chapter 7).

## 5.4 Conclusion

Based on the delegation template abstraction, the concept of indirect rule aggregation is proposed. This is a novel way to calculate delegation templates independently of the flow rules currently installed in the flow table of a switch. To achieve this, the indirect scheme exploits knowledge about the context of the network, i.e., the fact that the network is organized in slices or that it is possible to create a mapping between flow rules and ingress ports. The benefit here is that an indirect scheme will always return  $n$  delegation templates regardless of the flow rules installed in the switches.  $n$  only depends on context parameters, e.g., number of slices or number of physical ingress ports per switch. This is a desirable property in the flow delegation context because the same  $n$  delegation templates are used over time (only the conflict-free cover set changes, the aggregation match stays the same).

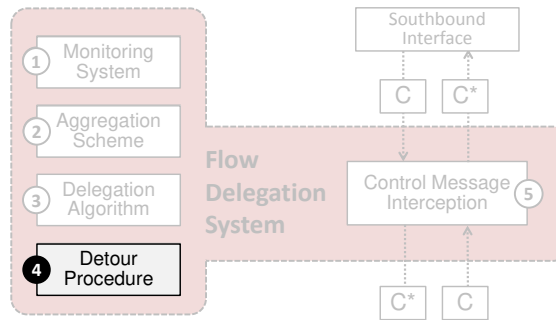
---

It is shown what kind of prerequisites have to be fulfilled in order to guarantee the delegation template properties for indirect rule aggregation. Furthermore, three different examples for indirect rule aggregation are presented based on tagging, slicing and ingress ports. Note that indirect rule aggregation is not only useful for flow delegation but could also be used in the context of software offloading (see Sec. 2.4.2.3) or flow rule distribution (Sec. 2.4.2.4).





## Detour Procedure



The **detour procedure** takes a set of selected and assigned delegation templates  $D_t^{**}$  from the delegation algorithm and translates this to sets of control messages to be installed in the switches. The high level view is shown in Fig. 6.1. It shows two sets of control messages calculated for one element  $\langle d, r \rangle \in D_t^{**}$  ( $d$  is the delegation template for delegation switch  $s_d$  and  $r$  is the remote switch allocated to this template).  $C_s$  on the left side contains the aggregation and backflow rules that are installed (or removed) from delegation switch  $s_d$ . And  $C_r$  contains the remote rules that are installed (or removed) from remote switch  $r$ .

This chapter is structured as follows. Sec. 6.1 first gives a brief summary of the different flow rule types. Sec. 6.2 discusses the problem of metadata transport between delegation and remote switch. Sec. 6.3 explains how the detour procedure works and how the different flow rule types interact with each other. Sec. 6.4 deals with control messages generation in more detail. Finally, the chapter is concluded in Sec. 6.5.

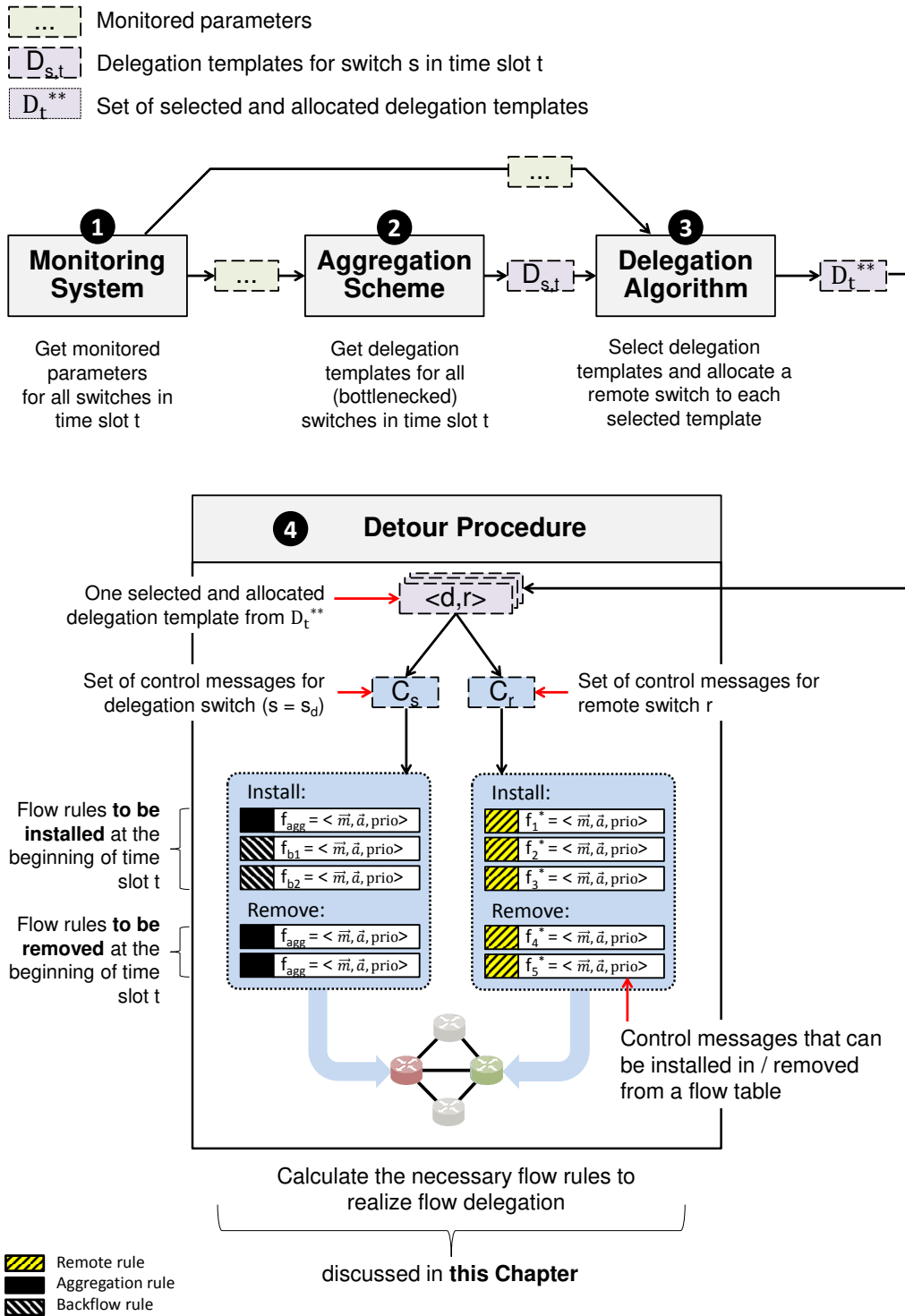






Figure 6.1: Detour procedure building block

## 6.1 Flow Rule Terminology

Table 6.1 briefly summarizes the (already established) terminology with respect to flow rules. The small boxes on the left show the color codes that are associated with the individual rule types (used in the examples).

Rule	Description
 Regular rule	A rule that was installed by the controller or the network applications, not the flow delegation system.
 Aggregation rule	An aggregation rule $f_d = \langle \vec{m}_d, \vec{a}_d, \text{prio}_d \rangle$ is used to relocate the traffic for remote rules to the remote switch. $d$ represents the delegation template. Explained in detail in Sec. 3.2.1.
 Remote rule	A rule that was relocated from the delegation switch to the remote switch. Is given as $f_i^* = \langle \vec{m}_i^*, \vec{a}_i^*, \text{prio}_i \rangle$ . The asterisk used with match and action indicate that those two parts are changed by the detour procedure before the rule is installed in the remote switch.
 Backflow rule	A rule that is (proactively) installed in the delegation switch to handle the return traffic sent back from the relocated rule. In scenarios without multicast, each port of the delegation switch has its own backflow rule.

**Table 6.1:** Overview of flow rules and color codes

## 6.2 Packet-level Metadata

One important aspect of the detour procedure is transport of packet-level metadata between delegation switch and remote switch. Packet-level metadata means some additional information is added to each packet  $z$  that is detoured from delegation switch to remote switch or vice versa. This is required to ensure correct packet processing. More precisely, packet-level metadata is required because:

- It is necessary to distinguish between remote rules of different delegation switches and between regular rules and remote rules in the remote switch

- It is necessary to attach the `in_port` information to packets detoured to the remote switch
- It is necessary to attach the forwarding decision of regular rules to packets sent back from the remote switch to the delegation switch

### 6.2.1 Flow Delegation Indicators

This section introduces the concept of so-called indicators that are used by the detour procedure to transport metadata between delegation and remote switch.

#### Definition 6.1: Flow Delegation Indicators

A **flow delegation indicator** is an integer value  $v \in \mathbb{N}$  representing the metadata information to be transported in packet  $z$  between delegation and remote switch or vice versa. In the context of flow delegation, there are three different indicators used to encode information about packet  $z$  which are not available otherwise:

- **remote\_rule\_indicator** ( `rri` ): A value of  $v$  means that packet  $z$  is a relocated packet from delegation switch  $v$  and has to be processed by a remote rule.
- **inport\_indicator** ( `ipi` ): A value of  $v$  means packet  $z$  was received at ingress port  $v$  at the delegation switch before it was relocated to the remote switch.
- **backflow\_indicator** ( `bfi` ): A value of  $v$  means packet  $z$  was sent back from the remote switch and has to be forwarded via port  $v$

Before transport of indicators and the overall workflow is explained, the following subsections briefly discuss the three indicators from Def. 6.1.






#### 6.2.1.1 Remote Rule Indicator

This indicator is required for two reasons: First, distinction between regular rules and remote rules is required to avoid rule conflicts. And second, it is possible that delegation templates from multiple different delegation switches are allocated to the same remote switch. And while no distinction is required for all remote rules relocated from one delegation switch, rules from other delegation switches can obviously lead to rule conflicts if not taken into account.

To do so, the detour procedure has to ensure that the symbolic sets (see Def. 3.3) of the regular rules in the remote switch and the remote rules of different delegation switches do not intersect (intersections will lead to incorrect packet processing). Assume a remote

rule  $f_x^*$  installed in the remote switch (symbolic set  $Z_{x^*}$ ). Now think of a regular rule  $f_y$  (or a rule that was relocated from another delegation switch) also installed in the remote switch with a symbolic set  $Z_y$  that intersects with  $Z_{x^*}$ . This will lead to incorrect packet processing for a packet  $z \in Z_{x^*} \cap Z_y$ . The following example illustrates the problem if no remote rule indicator is used:

Flow table remote switch

Rule	Match	Action	Priority	Comment
...	...	...	...	
 $f_{90}$	src=10 dst=0*	$\vec{a}_{90}$	90	
...	...	...	...	
 $f_5^*$	src=10 dst=0*	$\vec{a}_5^*$ , fwd( $s_1$ )	60	relocated from $s_1$
 $f_6^*$	src=10 dst=10	$\vec{a}_6^*$ , fwd( $s_1$ )	50	relocated from $s_1$
 $f_2^*$	src=10 dst=10	$\vec{a}_2^*$ , fwd( $s_2$ )	40	relocated from $s_2$
...	...	...	...	
 $f_{30}$	src=10 dst=10	$\vec{a}_{30}$	30	
...	...	...	...	

$f_5^*$ ,  $f_6^*$ , and  $f_2^*$  are remote rules.  $f_5^*$  and  $f_6^*$  are associated with delegation switch  $s_1$  and  $f_2^*$  is associated with delegation switch  $s_2$ . Note that priorities 60, 50, and 40 are not defined by the flow delegation system. They are derived from the relocated regular rules in the associated delegation switch ( $f_5, f_6, f_2$ ) and cannot be changed.

Given that  $f_2^*$  and  $f_6^*$  have the exact same match, it is obvious that a packet relocated from  $s_2$  (should be processed by  $f_2^*$ ) is processed by  $f_6^*$  here, which is wrong. So the rules from different delegation switches have to be distinguished. And a distinction is also required between the relocated rules and the regular rules installed in the remote switch (for the same reason). Look at regular rules  $f_{90}$  and  $f_{30}$  in this example. Now consider packet  $z_1 = 1000 \in Z_{90} \cap Z_{5^*}$  that was relocated from delegation switch  $s_1$  (should be processed by  $f_5^*$ ). This packet will be incorrectly processed by  $f_{90}$ . Another packet  $z_2 = 1010 \in Z_{30} \cap Z_{6^*} \cap Z_{2^*}$  that is not detoured from the delegation switch (should be processed by regular rule  $f_{30}$ ) is incorrectly processed by  $f_6^*$ . In addition, it could be required that two identical flow rules have to be stored in the flow table of the remote switch with different priorities (e.g., if  $\vec{a}_2^*$  and  $\vec{a}_{30}$  are identical and  $f_{30}$  also happens to have  $s_2$  as a forwarding target) which is not possible.

The remote rule indicator `rri` solves all the above problems because an identifier for the delegation switch is encoded in every relocated packet. Given that only relocated packets – i.e., packets forwarded by an aggregation rule – will have the `rri` bits in their

packet headers set to a value different from 0 and none of the regular rules is allowed to match on  $rri$ , this simple modification will solve the problem (the match of  $f_5^*$ ,  $f_6^*$ , and  $f_2^*$  is extended to also include a match on  $rri$ ). This requires  $n$  bits so that  $2^n \geq |S - 1|$  (worst case if  $n - 1$  delegation switches use the same remote switch). If flow delegation is restricted to one hop, this is reduced to  $2^n \geq |P_r|$  (the remote switch can only be connected to one delegation switch per port). In practice, however, it is assumed that the number of delegation switches that use the same remote switch is much smaller ( $< 10$ ) which can be realized with 4 bits.

### 6.2.1.2 Ingress Port Indicator

A match is usually based on information extracted from one or multiple packet header fields, i.e., on information that comes directly from the packet. There is, however, one notable exception which is the ingress port information ( $in\_port$  in Table 2.2).  $in\_port$  is the switch ingress port where the packet was received. This information is lost if a packet is relocated to a remote switch because the ingress port at the remote switch is the port connected to the delegation switch (and not the port where the packet originally arrived). If relocated flow rules match on  $in\_port$ , information about the ingress port has to be attached to all relocated packets. This is done with the  $ipi$  indicator. Note this is only required if at least one relocated flow rule matches on the ingress port. If this is not required, the  $ipi$  indicator can be omitted. This indicator requires  $n$  bits so that  $2^n \geq |P_s|$  where  $P_s$  is the set of ports of the delegation switch.

### 6.2.1.3 Backflow Indicator

The backflow indicator ( $bfi$ ) is required because the forwarding decision of remote rules would be lost otherwise. Assume flow rule  $f_5^* := \langle \vec{m}_5^*, \vec{a}_5^*, prio_5 \rangle$  from the above example is defined with action  $fwd(p_{15})$  with  $p_{15} \in P_{s_1}$  (the network application has decided the traffic should be forwarded via port  $p_{15}$  at switch  $s_1$ ). Recall that this rule was relocated from switch  $s_1$  and  $p_{15}$  identifies a port of switch  $s_1$  and not switch  $r$ . So just using the original action ( $\vec{a}_5^*$ ) is not possible, because  $p_{15}$  on switch  $r$  will forward the packet to a wrong destination (if the port does even exist). Instead, the forwarding action has to be replaced with  $fwd(s_1)$ . To save the information that a packet processed by rule  $f_5^*$  is forwarded via port  $p_{15}$  after it was sent back to the delegation switch, the backflow indicator with value  $v = p_{15}$  is attached to the packet (this is why the above table uses  $\vec{a}_5^*$  and not only  $\vec{a}_5^*$ ). The backflow rule for  $v = p_{15}$  installed in the delegation switch will then forward the packet to its correct destination which is  $p_{15}$  of switch  $s_1$ .

There are two basic cases. In the simple case, all flow rules have only a single forwarding action (no multicast) which results in a maximum of  $|P_s|$  backflow rules for delegation

switch  $s$ , one for each port. Here,  $v$  can be represented as the index of set  $P_s$ , i.e., as an integer value between 1 and  $|P_s|$ . This requires  $n$  bits for the indicator so that  $2^n \geq |P_s| + x$  where  $x$  represents special forwarding actions such as flooding ( $x < 10$  for OpenFlow-based networks). In the more complex case where flow rules can have multiple forwarding actions (multicast), the number of required backflow rules can be higher because each unique combination of forwarding targets has to be represented with a new backflow rule (if done naively). Efficient multicast in software-based network with flow delegation, however, is not in the scope of this work.

### 6.2.2 Transport Packet Header Fields

Given that metadata transport has to work with existing southbound interface protocols such as OpenFlow, an existing packet header field is required to store indicators in packet  $z$ . These are called transport packet header fields in the following. An indicator is “written” into a transport packet header field by one rule (with the set  $(k, v)$  action where  $k$  is the transport packet header field and  $v$  is the value of the indicator) and “read” by another rule by matching on the transport header field. This is a common strategy not only used here but also in various related approaches [Kat+14b; Aga+14].

#### Definition 6.2: Transport Packet Header Field

A **transport packet header field** is a packet header field according to Def. see 9.16 that can be overwritten by the flow delegation system without affecting regular packet processing. That is, none of the existing flow rules in  $F_{s,t}$  and  $F_{r,t}$  matches on the transport packet header field if it is used to transport metadata between switch  $s$  and switch  $r$  (or vice versa).

Transport packet header fields are shown as white rectangles in the following: `rri` is the transport packet header field used for `remote_rule_indicator`. `ipi` is the transport packet header field used for `inport_indicator`. And `bfi` is the transport packet header field used for `backflow_indicator`.

#### 6.2.2.1 Index function

One important helper construct that is required in the following is a function to encode the information required for the indicators so that they can be used in a transport packet header field. This encoding is done by mapping information from a delegation template (for example) to an integer value:

**Definition 6.3: index()**

**index()** realizes one of the following mappings to an integer value  $v \in \mathbb{N}$  that is used inside a transport packet header field:

- $\text{index}(d)$  : maps a delegation template  $d \in D_s$  to the port number of the ingress port used in aggregation match  $\vec{m}_d \in d$  (ingress port scheme).
- $\text{index}(p_i)$  : maps a single port  $p_i \in P_s$  to its port number (its index  $i$ ).
- $\text{index}(s)$  maps a switch  $s \in S$  to a unique identifier, e.g., by iterating over all switches in the infrastructure from 1 to  $|S|$ .

This function is used here to be consistent with already established definitions. It would not be clear, for example, what is meant if a delegation template (say  $d_4 \in D_s$ ) is added to a packet header field with the set packet processing instruction, i.e.,  $\text{set}(\boxed{\text{ipi}}, d_4)$ . Given the above definition, this can be defined as  $\text{set}(\boxed{\text{ipi}}, \text{index}(d_4))$  where  $\text{index}(d_4)$  is a well-defined integer that can be easily encoded into an existing packet header field (as long as this field provides a sufficient amount of bits, of course). Because it is always clear from context which mapping is required, the same function name is used.

### 6.2.2.2 Transport Options

There are different options available for the specific packet header fields that can be used for metadata transport. Table 6.2 provides a brief summary of the most relevant options. It is assumed that few software-defined networks will utilize all packet header fields in Table 6.2 in parallel. If this is the case, however, changes to the infrastructure are required in order to use flow delegation, e.g., introduce an additional layer of encapsulation for metadata support or utilize more advanced packet processing capabilities of programmable switches.

The first prototype of the flow delegation system presented in [BZ16] used the ToS bits for metadata transport. The second prototype presented in [BDZ19] was also successfully tested with VLAN encapsulation. Further note that multiple indicators might be encoded in the same transport packet header field which is not discussed here in detail but was investigated in a master thesis from David Körver entitled “Generic and Transparent Attribute Tagging for Software-defined Networks”. This work also presents an advanced approach to automatically utilize packet header fields that are not actively used in the network. In this context, it is further shown that source and destination addresses can be used for metadata transport under certain conditions (if the addresses are restored after relocation). This can be realized with control message interception, i.e., without changing the network applications.



Packet Header Field	Available bits	Comment
Ethernet addresses	up to 96	Suitable if the network only works on layer 3 and above. MAC filtering on end systems has to be considered.
IPv4 / IPv6 addresses	up to 64 / 256	Due to support of partial wildcards, parts of the address fields could be used if the original addresses are restored after relocation
ToS bits (IPv4) / Flow Label (IPv6)	up to 6 / 20	Suitable if the ToS bits or the flow label bits are not used otherwise. Note that even the 6 bits from the ToS field alone can be sufficient.
MPLS / VLAN encapsulation	up to 20 / 12	Encapsulation can be used if no other alternative is available. This will induce additional overhead and is not available in all networks (depends on hardware support)

**Table 6.2:** *Transport packet header field options*

### 6.3 Detour Procedure

The high level interaction between aggregation, remote and backflow rules was already explained in the introduction (Sec. see 9.16). This section presents the workflow of the detour procedure in more detail based on the terminology introduced above. It is important to mention that this workflow was specifically designed for the ingress port rule aggregation scheme presented in Sec. 5.2.2.3. In this aggregation scheme, each ingress port  $p_i \in P_s$  defines exactly one delegation template  $d_i \in D_s$ . This means the ingress port of a packet  $z$  that is matched by an aggregation rule with aggregation match  $\vec{m}_i \in d_i$  is – by design – given as  $p_i$ . This is different for other rule aggregation schemes.

Fig. 6.2 now illustrates the workflow of the detour procedure (same example that was used in the introduction). The red switch in the left is a delegation switch ( $s$ ), the green switch in the right is a remote switch ( $r$ ). The circled numbers and the path in grey illustrate a single packet  $z$  detoured from  $s$  to  $r$  and back to  $s$ . The part in the bottom shows  $z$  “on the wire” after it was sent back from  $r$  to  $s$ . Further consider the following two flow tables that show the flow rules from Fig. 6.2 in more detail (the circled numbers correspond to the steps in the figure):

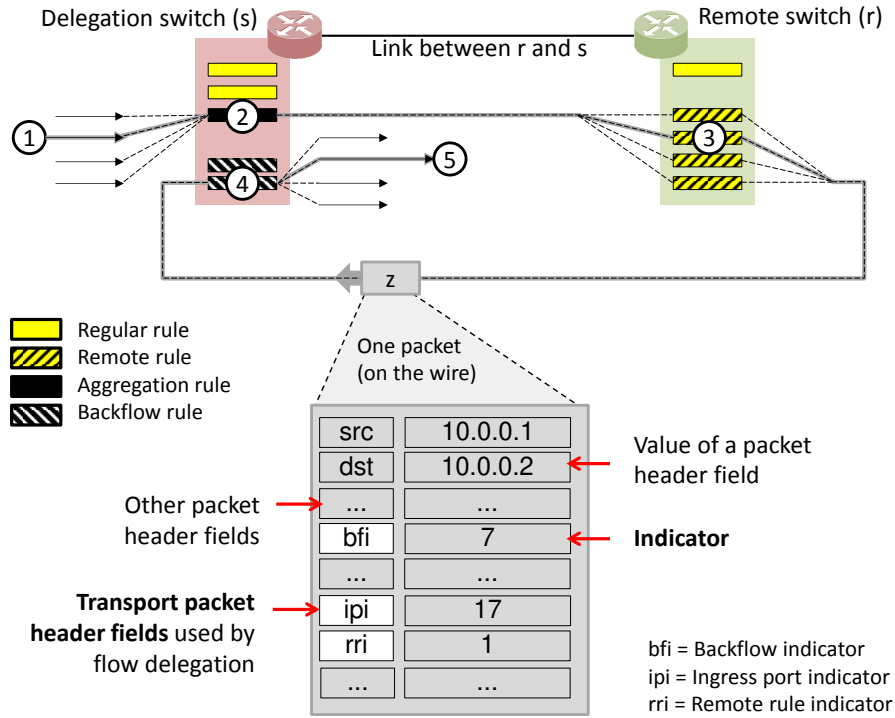







Figure 6.2: Metadata transport workflow

Flow table of delegation switch s

Rule	Match $\vec{m}_i$	Action $\vec{a}_i$	Priority
④  $f_{bf_1}$	$\boxed{\text{bfi}} = v_1$	$\text{set}(\boxed{\text{bfi}}, 0), \text{set}(\boxed{\text{rri}}, 0), \text{set}(\boxed{\text{ipi}}, 0), \text{fwd}(p_7)$	highest
④  $f_{bf_2}$	$\boxed{\text{bfi}} = v_2$	$\text{set}(\boxed{\text{bfi}}, 0), \text{set}(\boxed{\text{rri}}, 0), \text{set}(\boxed{\text{ipi}}, 0), \text{fwd}(p_4)$	highest
...	...	...	...
②  $f_d$	$\vec{m}_d$	$\text{set}(\boxed{\text{rri}}, \text{index}(s)), \text{set}(\boxed{\text{ipi}}, \text{index}(d)), \text{fwd}(r)$	1

Flow table of remote switch r

Rule	Match $\vec{m}_i$	Action $\vec{a}_i$	Priority
...	...	...	...
③  $f_1^*$	$\boxed{\text{rri}} = \text{index}(s) \vec{m}_1^*$	$\vec{a}_1^*, \text{set}(\boxed{\text{bfi}}, v_1), \text{fwd}(s)$	$\text{prio}_1$
...	...	...	...
③  $f_2^*$	$\boxed{\text{rri}} = \text{index}(s) \boxed{\text{ipi}} = 17 \vec{m}_2^*$	$\vec{a}_2^*, \text{set}(\boxed{\text{bfi}}, v_2), \text{fwd}(s)$	$\text{prio}_2$
...	...	...	...

The upper table shows the relevant flow rules installed in delegation switch  $s$ . The lower table shows the relevant flow rules installed in remote switch  $r$ . The transport packet header fields are given as  $\boxed{\text{rri}}$ ,  $\boxed{\text{bfi}}$ , and  $\boxed{\text{ipi}}$ . The matches are defined in such a way that packet  $z$  is matched by the aggregation rule  $f_d$  as well as one of the remote rules  $f_1^*$  or  $f_2^*$  (there are two rules to explain two different cases). The forwarding decision of rule  $f_1$  (before relocation) was  $\text{fwd}(p_7)$ . The forwarding decision of rule  $f_2$  was  $\text{fwd}(p_4)$ . The individual steps and interactions between the indicators and transport packet header fields is now explained step by step.

- ① Packet  $z$  arrives at the delegation switch where it is matched against all flow rules in the flow table.
- ② Packet  $z$  is matched by aggregation rule  $f_d$  and has to be relocated to remote switch  $r$ . Assume this aggregation rule is created from delegation template  $d_{17}$  that contains aggregation match  $\overrightarrow{m}_{17} = (\langle \text{in\_port}, 17 \rangle)^T$ , i.e., matches an all packets that arrive at ingress port number 17. This information is added to  $z$  by setting transport packet header field  $\boxed{\text{ipi}}$  to  $\text{index}(d) = 17$ . In addition, the  $\boxed{\text{rri}}$  transport packet header field is set to  $\text{index}(s)$  so that the remote switch knows the packet was relocated from  $s$ .
- ③ Packet  $z$  arrives at the remote switch. The additional match on  $\boxed{\text{rri}}$  ensures that no rule conflicts can occur with any of the regular rules in  $r$ . Remote rule  $f_1^*$  does not match on  $\text{in\_port}$  and no further changes were made to the match part of this rule by the detour procedure. Remote rule  $f_2^*$ , on the other hand, does match on  $\text{in\_port}$ . In this case, the match of the remote rule was changed accordingly: the original match for port 17 was replaced with a match on 17 using  $\boxed{\text{ipi}}$  instead of  $\text{in\_port}$ . The action part of both rules was changed as well to make sure the forwarding decision is not lost. The forwarding decision is encoded in the  $\boxed{\text{bfi}}$  transport packet header field. And the original forwarding action is replaced with  $\text{fwd}(s)$ . Any other actions (like changes to a packet header field) are not changed. If  $z$  is matched by  $f_1^*$ ,  $\boxed{\text{bfi}}$  is set to  $v_1 = \text{index}(p_7)$ . If  $z$  is matched by  $f_2^*$ , it is set to  $v_2 = \text{index}(p_4)$ .
- ④ Packet  $z$  – successfully processed by the remote rule – arrives at the delegation switch a second time. Using a match on the  $\boxed{\text{bfi}}$  transport packet header field, the corresponding backflow rule is identified. In case of  $f_1^*$  ( $f_2^*$ ), this is rule  $f_{bf_1}$  ( $f_{bf_2}$ ). It is easy to see that the packet is forwarded correctly. The three additional actions are required to delete the indicators. Note that  $\boxed{\text{ipi}}$  and  $\boxed{\text{rri}}$  (set in step ②) are still present in packet  $z$  and have to be deleted as well. In this design, the priority of the backflow rules has to be set to the maximum priority, above the aggregation rules. Otherwise, any wildcard rules matching on  $z$  with higher

priority would interfere with the process. It can help to also match on  $\text{in\_port} = r$  in the backflow rules to make sure that only (!) rules sent back from the remote switch are processed by these rules (e.g., if end systems or switches overwrite some of the packet header fields used for flow delegation by mistake). This is omitted in the table because of space constraints.

- ⑤ The packet is forwarded to its original destination. At this time, packet  $z$  has taken a detour via the remote switch and flow rule processing was done in the remote switch instead of the delegation switch.

## 6.4 Control Message Generation

The detour procedure is executed periodically once per time slot. In each time slot  $t$ , it receives a set of selected and allocated delegation templates  $D_t^{**}$  from the delegation algorithm and generates the necessary control messages to “realize” the decisions in  $D_t^{**}$  (the decisions have to be installed as flow rules in the network which requires control messages). The problem of generating a single control message always consists of three basic steps.

- (1) The switch where the control message is executed is determined. This is either the delegation switch  $s$  associated with the delegation template or the remote switch  $r$  that was allocated to the delegation template.
- (2) The flow rule to be used in the control message is determined, i.e., a fully parameterized flow rule  $f = \langle \vec{m}, \vec{a}, \text{prio} \rangle$  with match  $\vec{m}$ , action  $\vec{a}$  and priority  $\text{prio}$  (see Def. 2.8). There are three types of flow rules to consider: backflow rules, aggregation rules and remote rules.
- (3) The determined flow rule is wrapped in a control message of type `install()` (if the rule is added to the flow table), `delete()` (if the rule is removed from the flow table), or `update()` (if the rule is updated).

The generated control messages – the output of the detour procedure – can be distinguished into two classes:  $C_s$  (control messages for delegation switches) and  $C_r$  (control messages for remote switches). First recall from Sec. 2.1.2 that a switch can only take one of two roles at a time (delegation switch or remote switch). As a result,  $C_s$  can only consist of backflow and aggregation rules and  $C_r$  can only consist of remote rules. The following will now explain how these two sets are calculated. First, the necessary state kept by the detour procedure between time slots is discussed in Sec. 6.4.1. After that, the discussion is split into three blocks based on the three types of rules used by the flow delegation system: handling of backflow rules (Sec. 6.4.2), handling of aggregation rules (Sec. 6.4.3) and handling of remote rules (Sec. 6.4.4).

### 6.4.1 State Management

State management is required because control message generation depends on the status of the previous time slot. One obvious example: aggregation rules only have to be installed if the associated delegation template was not selected in the previous time slot (in the other case, the aggregation rule is already installed). This is very similar to the history definition in the context of periodic optimization discussed in Sec. see 9.16 (DT-Select) and in Sec. see 9.16 (RS-Alloc) – which is why the state variables are named accordingly. The detour procedure has to manage the following four state variables:

- $H_d^X$ : Binary variable that indicates whether delegation template  $d$  was selected in the previous time slot or not. This is also used in Def. 10.2 with the exact same meaning.
- $H_d^r$ : Represents the remote switch  $r \in S$  that is allocated to delegation template  $d$  in the previous time slot. Only defined if  $H_d^X = 1$ . This is very similar to  $H_j^r$  in Def. 11.4 except that the detour procedure does not work with allocation jobs (however, there is a 1:1 mapping between allocation job  $j$  and delegation template  $d$ ).
- $H_D$ : Set of selected and allocated delegation templates from the previous time slot. This set is required to detect delegation templates that are not selected any more.
- $H_s^{\text{bf}}$ : Set of all backflow rules installed in delegation switch  $s$  in the previous time slot. This set is required to determine whether a new backflow rule is required which cannot be derived from  $H_d^X$  (as it is done for aggregation rules).

### 6.4.2 Handling of Backflow Rules

If backflow rules are installed proactively, a static set of all possible backflow rules is installed in the delegation switch once (one rule for each port in  $P_s$ ). This simple case is not shown here. Instead, Alg. 4 shows the more interesting case when backflow rules can also be added reactively, i.e., only if they are really required by one of the remote rules.

Alg. 4 returns a set  $C_s$  with control messages for all backflow rules that are currently required and not already installed (for each delegation switch). It first determines all forwarding actions used by any of the remote rules. This is represented here as set  $V_s^{\text{fwd}}$  which is built in lines 3-11, separately for each delegation switch. Note that  $D_t^{**}$  can contain multiple delegation templates associated with delegation switch  $s$ .  $s_d$  represents the delegation switch associated with template  $d$ .  $F_d$  is the conflict free cover set from template  $d$ . The algorithm iterates over all flow rules in  $F_d$  and adds all forwarding actions to  $V_s^{\text{fwd}}$ . And in the second step (line 13-24), the collected forwarding targets in  $V_s^{\text{fwd}}$  are used to install new backflow rules for each delegation switch if they are not

**Algorithm 4:** Handling of backflow rules

---

```

1 Function backflow_rule_handling():
2    $V^{\text{fwd}} \leftarrow \{\}$ 
3   for  $\langle d, r \rangle \in D_t^{**}$  do
4     /* Step 1: identify all forwarding targets */
5      $s \leftarrow s_d$ 
6     for  $f_i = \langle \vec{m}_i, \vec{a}_i, \text{prio}_i \rangle \in F_d$  do
7       for  $a_j \in \vec{a}_i$  do
8         if  $a_j(v) = \text{fwd}(v)$  then
9           /* v is the forwarding target (output port of the fwd packet
10          processing instruction) */
11           $V_s^{\text{fwd}} \leftarrow V_s^{\text{fwd}} \cup \{v\}$ 
12        end
13      end
14    end
15  end
16  /* Step 2: install backflow rules */
17  for  $s \in S$  do
18     $C_s \leftarrow \{\}$ 
19    for  $v \in V_s^{\text{fwd}}$  do
20       $\vec{m}_{\text{bf}} \leftarrow (\langle \text{bfi}, \text{index}(v) \rangle)$ 
21       $\vec{a}_{\text{bf}} \leftarrow \begin{pmatrix} \text{set}(\text{bfi}, 0) \\ \text{set}(\text{rri}, 0) \\ \text{set}(\text{ipi}, 0) \\ \text{fwd}(v) \end{pmatrix}$ 
22       $f_{\text{bf}} \leftarrow \langle \vec{m}_{\text{bf}}, \vec{a}_{\text{bf}}, \text{highest} \rangle$ 
23      if  $f_{\text{bf}} \notin H_s^{\text{bf}}$  then
24         $H_s^{\text{bf}} \leftarrow H_s^{\text{bf}} \cup \{f_{\text{bf}}\}$ 
25         $C_s \leftarrow C_s \cup \{\text{install}(f_{\text{bf}})\}$ 
26      end
27    end
28  end
29 end

```

---

already present in  $H_s^{\text{bf}}$  (the already installed backflow rules from previous time slots). The backflow rules are created according to Sec. 6.3, i.e., use the  $\text{index}(v)$  function to match on the transport packet header field with the backflow indicator (line 16) and reset all indicators before the packet is forwarded to port  $v$  (line 17).

Note that Alg. 4 is only shown here to discuss the basic idea of backflow rule handling. It does not consider all corner cases. First, the algorithm does not take into account that flow rules with multiple forwarding actions require special treatment (a separate backflow rule per unique combination of forwarding targets). And second, it considers all flow rules from the conflict free cover set of all selected delegation templates, regardless of whether these rules are actually relocated or not. This could be optimized further, e.g., if flow delegation is used with low aggregation priority and there are flow rules installed before the template was selected (can be excluded).

### 6.4.3 Handling of Aggregation Rules

Aggregation rule handling is shown in Alg. 5. There are three different cases to consider. If the aggregation rule is associated with a delegation template that was not selected in the previous time slot ( $H_d^X = 0$ ), a new aggregation rule has to be installed. This is shown here as case 1 in lines 7-10. The aggregation rule is created according to Sec. 6.3, i.e., the forwarding action is set to  $\text{fwd}(r)$  to relocate packets to the remote switch. The `rri` and the `ipi` transport header fields are set to  $\text{index}(s)$  and  $\text{index}(d)$ . The aggregation priority is a global constant set to  $\text{prio}_{\text{agg}}^{\text{lowest}}$  in line 2 and the aggregation match  $\vec{m}_d$  (which is part of the delegation template) will match on all packets that arrive at a certain ingress port because the ingress port scheme from Sec. 5.2.2.3 is used here. This newly created rule is then used with the `install()` control message in line 10.

If the aggregation rule was selected in the previous time slot ( $H_d^X = 1$ ), the detour procedure only needs to perform an update on this rule if the allocated remote switch has changed (all other parts of the rule are statically defined by the delegation template). This is shown here as case 2 in lines 12-17. The remote switch allocated in the last time slot is given as  $H_d^r$ . If this value is different from the current remote switch ( $r$ ), the old aggregation rule is replaced with a new aggregation rule using the new remote switch as forwarding target. The three dots are used in line 13 to indicate that the first two instructions (set `rri` and `ipi`) are identical to line 8. Only the forwarding target is different ( $H_d^r$  is replaced by  $r$ ). In this case, the `update()` control message is used (line 16).

The third case in Alg. 5 deals with delegation templates that were selected in the previous time slot ( $H_d^X = 1$ ) but are not selected any more in the current time slot. This is shown in lines 20-26. In this case, the aggregation rule is removed from the flow table with the `delete()` control message.

**Algorithm 5:** Handling of aggregation rules

---

```

1 Function aggregation_rule_handling():
2   prio_agg ← prio_agglowest
3   ∀s∈S: Cs ← {}
4   for ⟨d, r⟩ ∈ Dt** do
5     s ← sd
6      $\vec{m}_{agg} = \vec{m}_d \leftarrow (\langle \text{in\_port}, \text{index}(d) \rangle)^T$ 
7     /* Case 1: add new aggregation rules */
8     if HdX = 0 then
9        $\vec{a}_{agg} \leftarrow \begin{pmatrix} \text{set}(\text{rri}, \text{index}(s)) \\ \text{set}(\text{ipi}, \text{index}(d)) \\ \text{fwd}(r) \end{pmatrix}$ 
10      fagg ← ⟨ $\vec{m}_{agg}$ ,  $\vec{a}_{agg}$ , prio_agg⟩
11      Cs ← Cs ∪ {install(fagg)}
12    else
13      /* Case 2: update existing aggregation rules */
14      if Hdr ≠ r then
15         $\vec{a}_{old} \leftarrow \begin{pmatrix} \dots \\ \dots \\ \text{fwd}(H_d^r) \end{pmatrix}$     $\vec{a}_{new} \leftarrow \begin{pmatrix} \dots \\ \dots \\ \text{fwd}(r) \end{pmatrix}$ 
16        fagg_old ← ⟨ $\vec{m}_{agg}$ ,  $\vec{a}_{old}$ , prio_agg⟩
17        fagg_new ← ⟨ $\vec{m}_{agg}$ ,  $\vec{a}_{new}$ , prio_agg⟩
18        Cs ← Cs ∪ {update(fagg_old, fagg_new)}
19      end
20    end
21  end
22  /* Case 3: remove obsolete aggregation rules */
23  for ⟨d, r⟩ ∈ HD \ D** do
24    s ← sd
25     $\vec{m}_{agg} = \vec{m}_d \leftarrow (\langle \text{in\_port}, \text{index}(d) \rangle)^T$ 
26     $\vec{a}_{old} \leftarrow (\dots, \dots, \text{fwd}(H_d^r))^T$ 
27    fagg_old ← ⟨ $\vec{m}_{agg}$ ,  $\vec{a}_{old}$ , prio_agg⟩
28    Cs ← Cs ∪ {delete(fagg_old)}
29  end

```

---



#### 6.4.4 Handling of Remote Rules

Handling of remote rules depend on the aggregation priority (see discussion in Sec. 5.3). If all aggregation rules are installed with the highest possible priority ( $\text{prio}_{\text{agg}}^{\text{highest}}$ ), all flow rules in the conflict-free cover set are installed as remote rules (if not already installed in previous time slots). In addition, the detour procedure has to make sure that all future flow rules – those installed after the aggregation rule – are relocated to the remote switch as well. If the lowest priority is used ( $\text{prio}_{\text{agg}}^{\text{lowest}}$ ), only new flow rules are installed as remote rules, i.e., the first part is not required. In total, there are three cases to consider:

- Relocate flow rules in the conflict free cover set of newly selected delegation templates if high aggregation priority is used
- For all selected delegation templates, relocate all new flow rules that are installed after the aggregation rule
- Relocate flow rules to a new remote switch if remote switch allocation is changed

Because the cases are very similar, the following illustrates the basic idea behind remote rule handling using the last case (remote switch allocation is changed). This case is handled in Alg. 6. The main loop over all selected and allocated delegation templates in line 2 has the same structure as above in Alg. 5 (the two algorithms are usually executed together and can be merged easily). If the delegation template was selected in the previous time slot ( $H_d^X = 1$ ) and the remote switch was changed from  $H_d^r$  to  $r$ , the relocated rules in the conflict-free cover set are moved to the new remote switch. This is done by the inner for loop in lines 4-8. If low aggregation priority is used (i.e., not all rules in the conflict-free cover set  $F_d$  are necessarily relocated), an additional check ( $f_i \in H_d^F$ ) is required inside the inner for loop to ensure that only relocated flow rules are selected which is omitted here due to space constraints. Further note that the rules from the inner for loop have to be removed from the old remote switch per `delete()` control message (also not shown, an example is given in line 25 of Alg. 5).

What is more important in this context are the two helper functions shown in the bottom of Alg. 6. These functions are required because match and action of a rule from  $F_d$  are changed by the detour procedure (see line 5 and 6). `create_remote_action( $\vec{a}_i$ ,  $s$ )` takes the original action  $\vec{a}_i$  from a rule in  $F_d$  and the delegation switch  $s$  ( $s_d$  is the delegation switch associated with delegation template  $d$ ). If action  $\vec{a}_i$  contains a forwarding packet processing instruction (`fwd`), this instruction is replaced with `fwd( $s$ )` to sent traffic back to the delegation switch. In addition, the original forwarding target (variable  $v$  in line 15 and 16) is added to a helper set  $v^{\text{fwd}}$ . This is required to create the proper index to be

**Algorithm 6:** Handling of remote rules

---

```

1 Function remote_rule_handling():
2   for  $\langle d, r \rangle \in D_t^{**}$  do
3     if  $H_d^X = 1$  and  $H_d^r \neq r$  then
4       for  $f_i = \langle \vec{m}_i, \vec{a}_i, prio_i \rangle \in F_d$  do
5          $\vec{m}_i^* \leftarrow \text{create\_remote\_match}(\vec{m}_i, s_d)$  // see step 1 below
6          $\vec{a}_i^* \leftarrow \text{create\_remote\_action}(\vec{a}_i, s_d)$  // see step 2 below
7          $C_r \leftarrow C_r \cup \{\text{install}(\langle \vec{m}_i^*, \vec{a}_i^*, prio_i \rangle)\}$ 
8       end
9     end
10  end
11 end
    /* Step 1: Forwarding action has to be replaced */
12 Function create_remote_action( $\vec{a}_i, s$ ):
13    $\vec{a}_i^* \leftarrow ()$     $v^{\text{fwd}} \leftarrow \{\}$ 
14   for  $a_j \in \vec{a}_i$  do
15     if  $a_j(v) = \text{fwd}$  then
16        $v^{\text{fwd}} \leftarrow v^{\text{fwd}} \cup \{v\}$  //  $v$  is the forwarding target
17     else
18        $\vec{a}_i^* \leftarrow \vec{a}_i^* \cup a_j$  // no modification required
19     end
20   end
21   return  $\vec{a}_i^* \leftarrow \vec{a}_i^* \cup (\text{set}(\boxed{\text{bfi}}, \text{index}(v^{\text{fwd}})), \text{fwd}(s))$ 
22 end
    /* Step 2: match on in_port has to be replaced */
23 Function create_remote_match( $\vec{m}_i, s$ ):
24    $\vec{m}_i^* \leftarrow ()$ 
25   for  $\langle k, v \rangle \in \vec{m}_i$  do
26     if  $k = \text{in\_port}$  then
27        $\vec{m}_i^* \leftarrow \vec{m}_i^* \cup (\langle \boxed{\text{ipi}}, \text{index}(v) \rangle)$  //  $v$  is a matched port number
28     else
29        $\vec{m}_i^* \leftarrow \vec{m}_i^* \cup \langle k, v \rangle$  // no modification required
30     end
31   end
32   return  $\vec{m}_i^* \cup (\langle \boxed{\text{rri}}, \text{index}(s) \rangle)$ 
33 end

```

---

used in the `bfi` transport packet header field in line 21 (to select the correct backflow rule).

The second helper function `create_remote_match( $\vec{m}_i$ ,  $s$ )` takes the original match  $\vec{m}_i$  from a rule in  $F_d$  and the delegation switch  $s$ . If this match contains the packet header field  $k = \text{in\_port}$  (line 26), the associated value  $v$  (the port the flow rule wants to match on) cannot be used with `in_port` (because this field now points to the port that connects the remote switch with the delegation switch). Instead, the information from the `ipi` transport packet header field has to be used where the original ingress port is encoded. This important transformation is done in line 27. The second transformation in the match is done in line 32. This is required because the remote rules have to be distinguished from remote rules relocated from other delegation switches which is done by adding a new match on the `rri` transport packet header field.

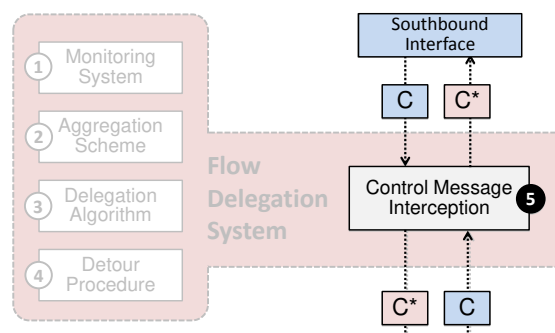
## 6.5 Conclusion

This chapter introduces the detour procedure that translates allocated delegation templates to concrete flow rules and control messages. It is explained how aggregation, backflow and remote rules interact with each other and how metadata transport between delegation and remote switch is handled. For the latter, three flow delegation indicators are defined to distinguish between remote rules of different delegation switches (remote rule indicator) and to make sure that the ingress port information from the delegation switch as well as the forwarding decision from the remote switch are not lost if packets are relocated (ingress port and backflow indicator).

The detour procedure, however, does not install the necessary flow rules directly. Instead, sets of control messages  $C_s$  and  $C_r$  are created and passed over to the control message interception building block. This is because flow delegation has to be hidden from the controller.



# Control Message Interception



Aggregation, remote and backflow rules are “hidden” from the network applications, i.e., a network application is not aware that some of the flow rules in the delegation switch may be relocated to a remote switch. This is necessary to not interfere with the network application’s internal logic (functionality) which in turn allows it to use existing network applications without modifications.

**Control message interception** is responsible to detect and resolve conflicts with respect to relocated flow rules and consists of two parts (Fig. 7.1). The periodic part is responsible for handling control messages calculated by the detour procedure. This happens periodically every time the delegation algorithm and the detour procedure generate new input, i.e., once per time slot. And the asynchronous part is responsible for intercepting control messages exchanged between controller and switches which can lead to conflicts if the view of the controller and the actual view with flow delegation differ from each other. Both parts are realized with a TCP/TLS proxy layer that intercepts and modifies control messages exchanged between controller and switches.

This chapter is structured as follows. The proxy layer design is explained in Sec. 7.1 together with the basic functionality of the periodic and the asynchronous part. The remainder of the section will then discuss the concrete interception logic required for flow delegation (Sec. 7.3).

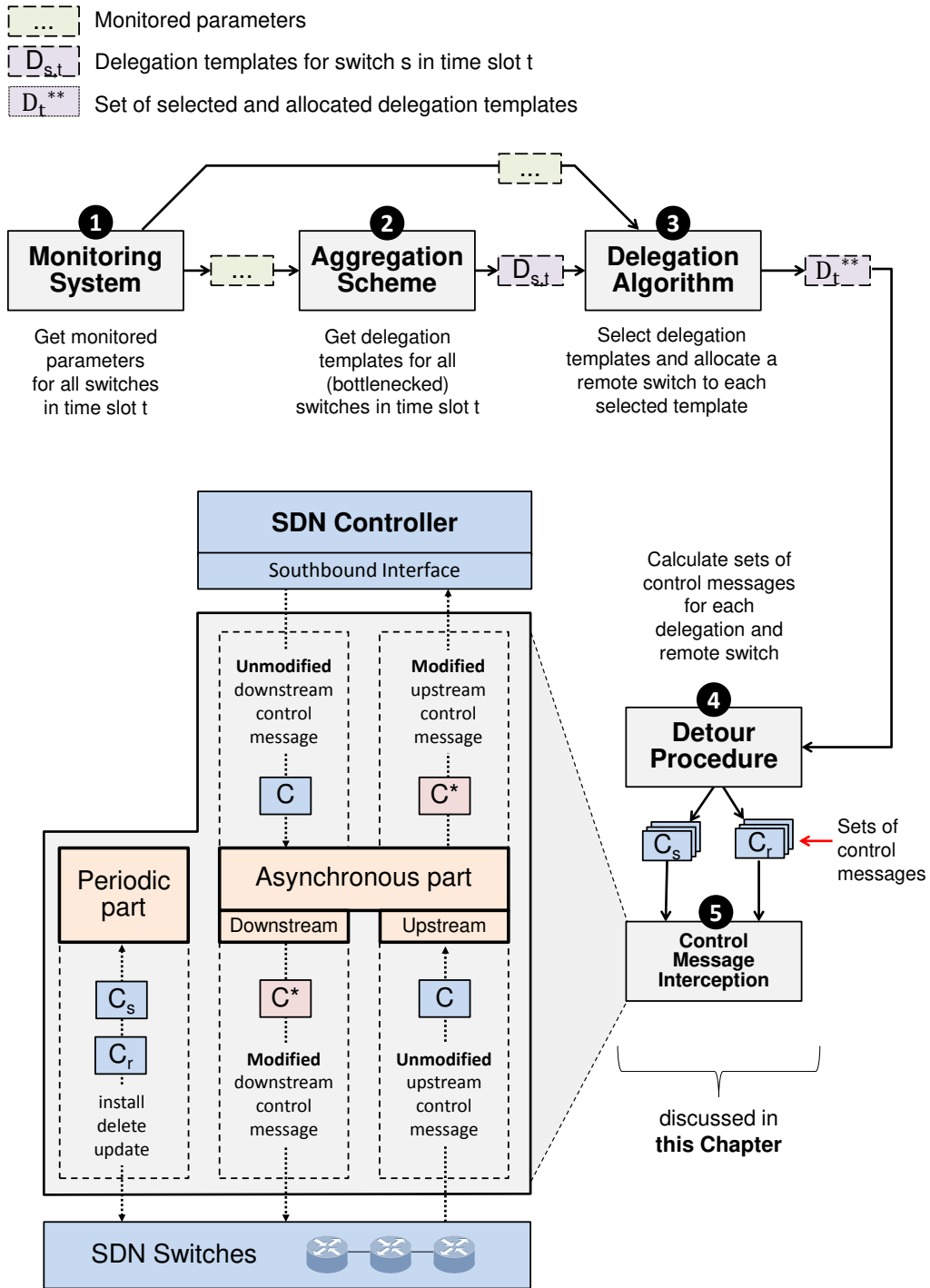


Figure 7.1: Control message interception building block

## 7.1 Proxy Layer

Control message interception is realized as a TCP/TLS proxy between the controller and the switches. Instead of a direct connection to the controller, the switches are connected to the lower layer endpoint of the proxy which is a TCP or TLS socket created by the control message building block. This can be achieved by changing the configuration of the switches<sup>1</sup>. And the controller is connected to the upper layer endpoint of the proxy. This connection is initiated by the proxy component as soon as a new switch is connected to the lower layer endpoint. With such a design, control messages sent from the controller and the switches can be intercepted and processed inside the proxy layer without changing the controller or the network applications.

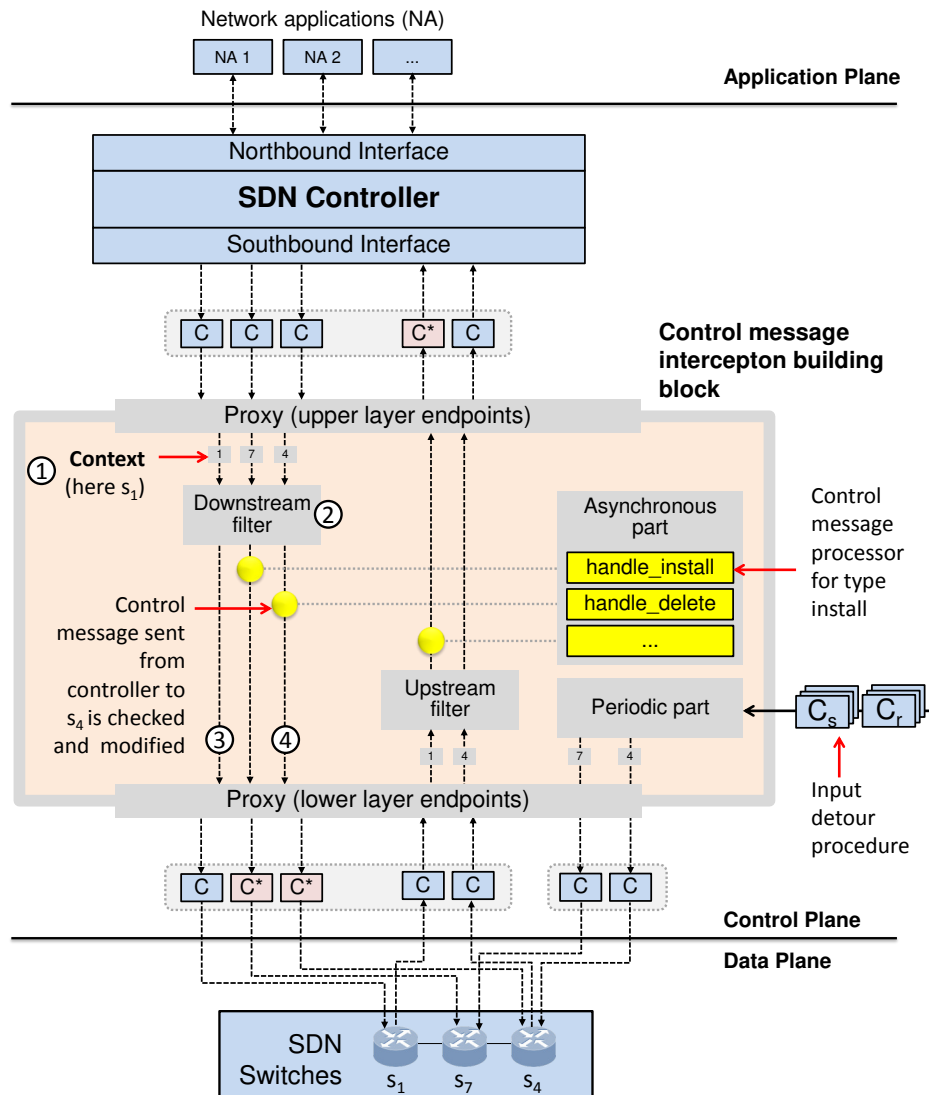
Fig. 7.2 shows an example with three switches  $s_1$ ,  $s_4$ , and  $s_7$  connected to the proxy layer. There are three TCP connections between the switches and lower layer endpoints of the proxy. And there are also three TCP connections between upper layer endpoints and the controller. The connections are not explicitly shown in the figure, only the control messages that are exchanged via the connections.

If a new control message is received at an upper layer endpoint (controller to switch, downstream direction), it is first **associated with a context** identified by the TCP connection in step ①. This context represents the switch  $s \in S$  the controller wants to communicate with. In step ②, the control message is examined by the downstream filter component. The filter component determines the type of the control message (install, delete, etc.) and **executes the corresponding control message processor**. Control messages of type `install`, for example, are processed with the `handle_install` processor. Control messages of type `delete` are processed with the `handle_delete` processor. And so on.

The control message processors are shown as yellow boxes in Fig. 7.2. They check the content of the control messages and update them if necessary. After that, the – potentially updated – control messages are forwarded according to the selected context. In the example, the control message with context 1 (sent to  $s_1$ ) is forwarded without modifications ③. The control messages with context 7 and 4 (sent to  $s_7$  and  $s_4$ ), on the other hand, are modified ④. It is also possible to select a different context inside the proxy layer, i.e., a control message can be sent to a different switch as originally intended. This is crucial for flow delegation where flow rules have to be relocated from delegation to remote switch. The unmodified control messages are shown here as c. The term unmodified expresses that a control message was not yet checked or modified by the

---

<sup>1</sup>SDN switches are configured with one or multiple controller IP addresses. These can be replaced with the IP address of the lower layer proxy.



**Figure 7.2:** Proxy layer design of the control message interception building block

proxy. Modified control messages are shown as  $C^*$ . The upstream direction is handled in the exact same way using the upstream filter component and a different set of processors.

### 7.1.1 Periodic Part

The periodic part processes the input of the detour procedure. It basically takes the control messages calculated by the detour procedure ( $C_s$ ,  $C_r$ ) and “executes” them. Execution means the control messages are sent to the switches using the lower layer endpoints of the proxy. This is required because all aspects of flow delegation should be hidden from the controller and the network applications. So the flow rules associated with the control



messages in  $C_s$  and  $C_r$  cannot be installed or updated directly by the SDN controller<sup>2</sup>. Instead, these rules are managed by the periodic part of the control message interception component.

The periodic part is rather simple. It is important to mention, however, that the control message interception building block has to keep track of the rules in  $C_s$  or  $C_r$  to make sure that no “unexpected” events are forwarded to the controller. Assume a flow rule from  $C_s$  or  $C_r$  installed by the proxy is removed because of a timeout and triggers a removed control message. This control message has to be processed inside the proxy layer and should not be forwarded to the controller. Such cases are covered by the asynchronous part below.

### 7.1.2 Asynchronous Part

The asynchronous part deals with events from switches that are associated with a flow rule installed by the periodic part. It also deals with control messages sent from a network application while the flow delegation system is active, i.e. when at least one flow rule is relocated. It is called “asynchronous” because events from a switch or control messages sent from the network applications can arrive at random times. Both aforementioned cases can cause conflicts because the view without flow delegation and the actual view with flow delegation differ from each other.

Assume a flow rule  $f$  is currently relocated from switch  $s$  to switch  $r$ . Further assume the network application has decided that flow rule  $f$  should be removed and communicates this decision to the controller. In response, the controller will eventually send a `delete(f)` control message. However, the controller does not know that  $f$  was relocated to switch  $r$ , i.e., it still assumes the rule is present in switch  $s$ . So the control message is sent to switch  $s$ . This, however, will lead to an error if the flow rule does not exist. The asynchronous part is responsible to prevent such errors. It will therefore intercept all control messages and resolve potential conflicts. To achieve this, control messages are changed in such a way that the delegation process remains hidden from the controller. The translation is realized with control message processors which are introduced in the next section.

## 7.2 Control Message Processors

The required logic for control message interception is defined in so-called processors – the yellow boxes inside the asynchronous part in Fig. 7.2. Such a processor is executed if a control message with specific type (install, update, ...) is intercepted by the proxy. This

---

<sup>2</sup>If the controller is used to install an aggregation rule, for example, this rule is obviously visible to the controller

section will briefly introduce the interface used inside the control message processors and explain how transaction identifiers are managed. The logic for flow delegation is discussed afterwards in the next section.

### 7.2.1 Interface

Control message processors use so-called context object to access, manipulate and forward control messages. These objects are the main interface to controller and switches and are injected into the selected processor together with control messages. They abstract from specific implementation details of the southbound interface protocol.

Each time a control message is intercepted, the proxy layer will automatically create a context object with pointers to the downstream and upstream connection of the unmodified control message.

- `context.s` is a pointer to the downstream connection and identifies the switch `s` to which the unmodified control message is to be forwarded as decided by the controller. If the control message was created by the detour procedure (periodic part), `s` was determined by the delegation algorithm.
- `context.controller` is a pointer to the upstream connection and identifies the controller to which the unmodified control message is to be forwarded as decided by the switch. It is assumed here that a single controller is used, i.e., this pointer is the same for all control messages received at a lower layer endpoint.

Besides the two pointers to upstream and downstream connections, a context object also contains two important primitives:

- `context.fwd_to_switch(s, c)` takes a control message `c` and forwards it via the downstream connection towards the switch identified by `s`. Note that `s` and `context.s` are not necessarily identical if the proxy decides to forward the message to a different switch.
- `context.fwd_to_controller(s, c)` takes a control message `c` and forwards it via an upstream connection towards the controller. Because we assume a network with a single controller, the pointer to the controller is not explicitly included. Instead, switch `s` denotes the upstream connection to be used. This is important, for example, if an asynchronous event from a remote switch has to be forwarded using the upstream connection of a delegation switch (because the controller expects that this control message is received via the TCP connection established with the delegation switch).

### 7.2.2 XID Management

Transaction identifiers (XIDs) are used for the mapping between control messages (requests and replies). Assume a series of control messages ( $c_1, c_2, \dots$ ) from the controller that are intercepted in the proxy. Further assume that ( $x_1, x_2, \dots$ ) are the XIDs used in the OpenFlow header for these control messages, assigned by the controller. Because the periodic part will create new control messages independently from the controller – i.e., messages ( $k_1, k_2, \dots$ ) with XIDs ( $y_1, y_2, \dots$ ) – a mechanism is required to make sure that the two XID spaces do not overlap ( $x_i \neq y_j \forall i, j$ ). Special caution is also required to assure that no control messages are forwarded to the controller with an XID unknown to the controller which would result in a protocol error. To achieve this, three separate XID name spaces are used: All messages created within the proxy are automatically associated with an internal name space  $NS_I$ . Similarly, all messages exchanged with the controller (upstream direction) are associated with  $NS_U$  and all messages exchanged with the switches (downstream) are associated with  $NS_D$ . Every time a message has to switch into another name space (e.g., from  $NS_U \rightarrow NS_I \rightarrow NS_D$ ), the proxy will create a new XID in the target name space and remember the mapping towards the old XID.

Example: The controller sends a control message with  $x_U = 50$  that is intercepted by the proxy. It creates a new internal XID  $x_I = 100$  and overwrites  $x_U$ . It also stores the tuple  $(x_U, x_I)$  in a database. The control message provided to a control message processor thus show 100 as XID. After processing, another XID  $x_D = 200$  is created together with a new mapping  $(x_I, x_D)$ . The message forwarded downstream to the switches will then contain  $x_D$  instead of  $x_I$ . If a reply with  $x_D$  is intercepted in upstream direction, the proxy can use the stored mapping  $(x_I, x_D)$  to restore the internal XID ( $x_I = 100$ ). Finally, if the message is forwarded back to the controller, the first mapping  $(x_U, x_I)$  is used to restore the original XID. This assures that no XID collisions can occur.

Note that this process is completely hidden from the control message processor. The processors use the primitives provided in the context object, they do not manually interact with the XIDs.

## 7.3 Processor Logic for Flow Delegation

This section introduces the logic for three important and conceptually different tasks in the flow delegation context:

- A network application **adds a new flow rule** to delegation switch  $s$ . If the match field of the new flow rule is covered by a currently used aggregation rule, the new flow rule must be installed into the remote switch instead of the delegation switch.

This has to be resolved by creating a new remote rule that is then installed into the remote switch. This case is discussed in Sec. 7.3.3.

- A network application **changes an existing flow rule** in the delegation switch that was relocated to the remote switch. Because neither the network application nor the controller are aware of the fact that the flow rule was relocated, the corresponding control message for updating the flow rule is directed to the delegation switch. This has to be resolved by directing a modified version of the control message to the remote switch. Discussed in Sec. 7.3.4.
- A remote rule installed by the periodic part is automatically **removed from the flow table after a timeout**. The flow rule was configured to send a control message with a “rule was removed event” towards the controller. However, the removed flow rule was installed in the wrong switch from the perspective of the controller. This has to be resolved by changing the control message so that the expectations of the controller are met. Discussed in Sec. 7.3.5.

There are other tasks – e.g., related to monitoring requests – but they all use the exact same mechanisms that are required for the three tasks above. The following explains how potential conflicts are resolved in general, discusses the required state and presents the logic for the three above tasks.

### 7.3.1 Conflict Resolution

Recall that the delegation templates only guarantee that no rule conflicts occur if all flow rules associated with the delegation template are kept together, i.e., all of them are either installed in the remote switch or in the delegation switch. So within a single delegation template, rules can still depend on each other. It was already explained in Sec. 5.3 that this requires continuous rule conflict detection because flow delegation will not just relocate all flow rules in the conflict-free cover set. Instead, the approach only relocates new flow rules, i.e., those installed after the aggregation rule.

In general, rule conflict detection works as follows: for each flow rule installed after the aggregation rule, it is checked whether this rule has a rule conflict with one of the rules in the associated delegation template. This can be done with Alg. 1 except that the conflict-free cover set of the delegation template is used as the set of input flow rules together with the match from the to be checked flow rule. The output of this step is called **conflict set** in the following. Note that this operation is cheap because the conflict-free cover set is small compared to a whole flow table. There is a conflict if not all rules in the calculated conflict set are either stored in the delegation or the remote switch. If there is no conflict, the rule is relocated to the remote switch. And in case of a conflict, there are two different options how the flow delegation system can continue:

- (1) In the first case, all flow rules in the conflict set are moved to the delegation switch. If this affects a relocated rule, this rule is moved back to the delegation switch.
- (2) In the second case, all flow rules in the conflict set are moved to the remote switch. If this affects a rule in the delegation switch, this rule is also relocated to the remote switch.

The flow delegation system will select the cheaper option where less flow rules have to be moved. For the following algorithms, however, we assume that always the second approach is chosen where the conflict set is moved to the remote switch (for simplicity).

### 7.3.2 Required State

The control message interception part has to manage some state, similar to the detour procedure. The following state variables are used in the remainder of this section:

- $H_d^X$ : Binary variable that indicates whether delegation template  $d$  was selected in the previous time slot or not. This is also used in Def. 10.2 with the exact same meaning.
- $H_d^r$ : Represents the remote switch  $r \in S$  that is allocated to delegation template  $d$  in the previous time slot. Only defined if  $H_d^X = 1$ .
- $H_d^F$ : Represents the set of flow rules currently relocated to switch  $H_d^r$

### 7.3.3 Installation of a New Flow Rule

The control message processor discussed in this section is used for all control messages of type `install` that are sent from the controller. This happens if a network application makes a new decision such as updating a route. It is assumed the flow rule to be installed is given as  $f_c = \langle \vec{m}_c, \vec{a}_c, \text{prio}_c \rangle$ .

Alg. 7 shows how the control message processor for this case is realized. The first step in line 2 extracts flow rule  $f_c$  from the control message. The second step in line 3 calculates the delegation template<sup>3</sup> associated with  $f_c$ . It depends on the used aggregation scheme how this is done. The procedure for the ingress port scheme used in this work is specified in Sec. 5.2.2.3. Next step is the calculation of the conflict set  $F_d^{\text{Conflicts}}$  in line 4. After that, there are two basic cases to consider:

---

<sup>3</sup>Recall that requirement (R3) for indirect rule aggregation in Sec. 5.2.1 ensures that each rule can be linked to exactly one delegation template and the delegation template can be inferred from the rule.

**Algorithm 7:** Asynchronous installation of a new flow rule

---

```

1 Function handle_install( $c$ , context):
2    $f_c = \langle \vec{m}_c, \vec{a}_c, prio_c \rangle \leftarrow$  get_flow_rule_from_control_message( $c$ )
3    $d \leftarrow$  get_delegation_template( $f_c$ )
4    $F_d^{Conflicts} \leftarrow$  detect_conflict( $d, f_c$ )
5   if  $H_d^X = 1$  then
6     /* Template  $d$  is selected */
7     if  $F_d^{Conflicts} \setminus \{f_c\} \not\subseteq H_d^F$  then
8       | resolve_conflict( $F_d^{Conflicts}$ )
9     end
10    install_in_remote_switch( $\vec{m}_c, \vec{a}_c, prio_c, context$ )
11  else
12    /* Template  $d$  is not selected */
13    if  $F_d^{Conflicts} \cap H_d^F = \emptyset$  then
14      | context.fwd_to_switch(context.s,  $c$ )
15    else
16      | resolve_conflict( $F_d^{Conflicts}$ )
17      | install_in_remote_switch( $\vec{m}_c, \vec{a}_c, prio_c, context$ )
18    end
19  end
20
21  /* Helper function to install a rule in the remote switch */
22  Function install_in_remote_switch( $\vec{m}, \vec{a}, prio, context$ ):
23     $\vec{m}^* \leftarrow$  create_remote_match( $\vec{m}, context.s$ ) // see Alg. 6
24     $\vec{a}^* \leftarrow$  create_remote_action( $\vec{a}, context.s$ ) // see Alg. 6
25     $c^* \leftarrow$  install( $\langle \vec{m}^*, \vec{a}^*, prio \rangle$ )
26    context.fwd_to_switch( $H_d^r, c^*$ )
27  end
28
29  /* Helper function to resolve conflicts */
30  Function resolve_conflict( $F_d^{Conflicts}$ ):
31    for  $f_i = \langle \vec{m}_i, \vec{a}_i, prio_i \rangle \in F_d^{Conflicts}$  do
32      | if  $f_i \notin H_d^F$  then
33        | install_in_remote_switch( $\vec{m}_i, \vec{a}_i, prio_i, context$ )
34      | end
35    end
36  end

```

---

- (1) Delegation template  $d$  is selected ( $H_d^X = 1$ ). This means there is an aggregation rule in the delegation switch that covers the match of the new flow rule  $f_c$ . In order to install  $f_c$  in the remote switch, it has to be checked whether there is at least one rule in the conflict set that is not installed in the remote switch. Note that  $f_c$  is new and has to be excluded for this check, see line 6. If this is the case, the conflict resolution mechanism is triggered before the new rule is installed – shown here as helper function `resolve_conflict` in line 25. This will relocate all flow rules in the conflict set to the remote switch.
- (2) Delegation template  $d$  is not selected ( $H_d^X = 0$ ). This means there is no aggregation rule and the new flow rule should be installed in the delegation switch. However, the new rule could still introduce conflicts with rules that are currently relocated. It is therefore required to ensure that all rules in the conflict set are stored in the delegation switch. This check is performed in line 11. If all rules in  $F_d^{\text{Conflicts}}$  are in the delegation switch, it is safe to install the rule (line 12). If not, conflict resolution is required again before the new rule is installed. This happens in lines 14 and 15.

It is further important to notice that relocated flow rules are modified before installation. This is done in lines 20 and 21. It is explained in detail in Sec. 6.4.4 why this is necessary.

### 7.3.4 Update of an Existing Flow Rule

The control message processor discussed in this section is used for all control messages of type update that are sent from the controller. It is assumed the old flow rule (the one to be changed) is given as  $f_{\text{old}}$  and the updated flow rule is given as  $f_{\text{new}}$ . The helper functions are the same as in Alg. 7.

The overall process is similar to the case discussed in Sec. 7.3.3. The first step extracts the old and the new flow rule from the control message and the second step calculates the associated delegation template. It is important to mention that the two delegation templates  $d$  and  $d_{\text{new}}$  are always identical if the ingress port scheme is used because the network application can not change the ingress port. This, however, could be different for other indirect schemes which is not discussed here further. The major difference to Alg. 7 is that the conflict set does not only include the updated flow rule but also the old flow rule, see line 7. This is done to ensure a proper transition from the old to the new state in case the update is not performed as one atomic operation. The rest of the algorithm in lines 8-20 is identical to the two basic cases discussed above.

**Algorithm 8:** Asynchronous update of an existing flow rule

---

```

1 Function handle_update( $c$ , context):
2    $f_{old} = \langle \vec{m}_{old}, \vec{a}_{old}, prio_{old} \rangle \leftarrow \text{get\_from\_control\_message}(c)$ 
3    $f_{new} = \langle \vec{m}_{new}, \vec{a}_{new}, prio_{new} \rangle \leftarrow \text{get\_from\_control\_message}(c)$ 
4    $d \leftarrow \text{get\_delegation\_template}(f_{old})$ 
5    $d_{new} \leftarrow \text{get\_delegation\_template}(f_{new})$ 
6   assert( $d = d_{new}$ )
7    $F_d^{Conflicts} \leftarrow \text{detect\_conflict}(d, f_{old}) \cup \text{detect\_conflict}(d, f_{new})$ 
8   if  $H_d^X = 1$  then
9     if  $F_d^{Conflicts} \not\subseteq H_d^F$  then
10      |  $\text{resolve\_conflict}(F_d^{Conflicts})$ 
11    end
12     $\text{install\_in\_remote\_switch}(\vec{m}_{new}, \vec{a}_{new}, prio_{new}, \text{context})$ 
13  else
14    if  $F_d^{Conflicts} \cap H_d^F = \emptyset$  then
15      |  $\text{context.fwd\_to\_switch}(\text{context.s}, c)$ 
16    else
17      |  $\text{resolve\_conflict}(F_d^{Conflicts})$ 
18      |  $\text{install\_in\_remote\_switch}(\vec{m}_c, \vec{a}_c, prio_c, \text{context})$ 
19    end
20  end
21 end

```

---

### 7.3.5 Asynchronous Event from a Switch

The control message processor discussed in this section deals with so-called “flow removed” events. A flow removed event is a control message of type removed used to signal that a flow rule  $f_{removed}$  was deleted from the flow table. Note that such events can also be triggered for aggregation and backflow rules which is not considered here because these cases are simple (the control message is discarded in the proxy layer). In case of a remote rule, however, the control message must be forwarded to the controller. Problem is that the event is triggered at the “wrong” switch. The controller expects the removed messages from the delegation switch and not the remote switch.

Alg. 9 shows the processor logic. The first two steps are identical to above: extraction of the removed flow rule from the control message and determination of the delegation template. The rest of the algorithm, however, is different. The major difference is that the original version of the control message has to be reconstructed so that the reconstructed message matches the expectations of the controller. This reconstruction consists of two steps:



**Algorithm 9:** Asynchronous event from a switch

---

```

1 Function handle_update( $\boxed{c}$ , context):
2    $f_{\text{removed}} = \langle \vec{m}, \vec{a}, \text{prio} \rangle \leftarrow \text{get\_from\_control\_message}(\boxed{c})$ 
3    $d \leftarrow \text{get\_delegation\_template}(f_{\text{removed}})$ 
4   if  $f_{\text{removed}} \notin H_d^F$  then
5     /* Case 1: flow rule was not relocated, no changes required */
6     context.fwd_to_controller(context.s,  $\boxed{c}$ )
7   else
8     /* Case 2: flow rule was relocated and the "original" control message
9     (as expected by the controller) has to be restored */
10     $\vec{m}^* \leftarrow ()$     $\vec{a}^* \leftarrow ()$ 
11    for  $\langle k, v \rangle \in \vec{m}$  do
12      if  $k = \boxed{ipi}$  then
13        do nothing // this match was added by flow delegation system
14      else if  $k = \boxed{rri}$  then
15        do nothing // this match was added by flow delegation system
16      else
17         $\vec{m}_i^* \leftarrow \vec{m}_i^* \cup \langle k, v \rangle$  // no modification required
18      end
19    end
20    for  $a_i \in \vec{a}$  do
21      if  $a_i = \text{fwd}(v)$  then
22        do nothing // was overwritten by flow delegation system
23      else if  $a_i = \text{set}(\boxed{bfi}, v)$  then
24         $\vec{a}^* \leftarrow \vec{a}^* \cup \{\text{fwd}(v)\}$  // restored from backflow indicator
25      else
26         $\vec{a}_i^* \leftarrow \vec{a}_i^* \cup a_i$  // no modification required
27      end
28    end
29     $f_{\text{removed}} = \langle \vec{m}^*, \vec{a}^*, \text{prio} \rangle$  // restored flow rule
30     $s_{\text{removed}} \leftarrow \text{index}(d)$  // delegation switch (for upstream connection)
31     $\boxed{c}^* \leftarrow \text{removed}(f_{\text{removed}})$  // restored control message
32    context.fwd_to_controller( $s_{\text{removed}}$ ,  $\boxed{c}^*$ )
33  end

```

---

- (1) The changes made to the rule by the detour procedure are reverted. That is, the matches for the different indicators (ingress port indicator and remote rule indicator) are removed and the original forwarding action is restored.
- (2) The control message is forwarded to the controller via the upstream connection identified by the delegation switch and not the remote switch.

The first step is realized in lines 7-25. The first for-loop iterates over all matches and removes the indicators. The second for-loop iterates over all actions and replaces the forwarding action. The second part in lines 26-28 creates the new flow rule that is sent to the controller. Note that the delegation switch can be determined from the delegation template. Further note that the whole process could also be realized with a database that stores the original version of a relocated rule.

## 7.4 Conclusion

This section introduced control message interception based on a proxy layer placed between switches and controller. Control message interception intercepts all exchanged control messages and runs a pre-defined control message processor for each intercepted message. This processor implements the logic necessary to hide flow delegation from the control plane without changing the controller or the network applications – one of the key design goals of the approach. It is explained how rule conflicts are resolved and how issues such as XID management are handled. The section further explains the required logic on the example of three concrete tasks (controller installs new rule, controller updates existing rule, switch sends a flow removed event).

# Prototype Implementations

---

The previous chapters showed that – despite the simple core idea – flow delegation is associated with a range of practical challenges, especially with respect to metadata transport in the detour procedure and transparency towards the control plane. This chapter demonstrates that the designed architecture and the five involved building blocks do actually work when used in a real software-based network, i.e., bottlenecks are detected and mitigated automatically without disturbance of network application logic.

For this purpose, two **prototype implementations** of the flow delegation system were developed and functionally evaluated in an emulated OpenFlow network (using mininet [LHM10]). The functional evaluation covers all five building blocks: monitoring system, rule aggregation scheme, delegation algorithm, detour procedure and (in case of the second prototype) control message interception. And there is more prototypical work created in the context of student theses that also shows the practical feasibility of the flow delegation approach.

Note that this chapter focuses primarily on functionality and practical aspects. A comprehensive investigation on performance, overhead, and scalability is conducted later in the evaluation part in chapters 12 to 16. Sec. 8.1 introduces the two prototypes mentioned above and other prototypical work. Sec. 8.2 shows that the prototype implementations work as expected and can mitigate flow table capacity bottlenecks. Sec. 8.3 investigates the overhead for monitoring in terms of required CPU resources and control channel bandwidth. And Sec. 8.4 shows that relocated packets experience an additional end-to-end delay of only 0.1ms on average which is considered acceptable for many use cases.

## 8.1 Existing Prototypes

The following two prototypes for flow delegation are investigated in this chapter: an early implementation of the flow delegation system that was created in 2016 (written in Python) and a complete re-implementation from 2019 (written in Java). The early version already implements monitoring, detour procedure, rule aggregation scheme, and a simple version of the delegation algorithm. However, it is directly integrated into the controller. This requires (small) changes in the network applications which is not fully transparent. The second prototype from 2019 is fully transparent. It is realized inside a proxy layer and uses control message interception to hide flow delegation from the controller and the network applications.

Because the version from 2016 works like a middleware inside the controller, this prototype is referred to here as **middleware prototype**. The version with proxy-based control message interception from 2019 is called **proxy prototype**. Note that it may be perfectly reasonable to integrate flow delegation into the controller – as it is done in the middleware prototype – under certain circumstances<sup>1</sup>. For example, if full transparency is not required in a very simple network that runs only a small set of network applications (which can be changed easily).

A fully transparent approach, however, has the benefit that the flow delegation system is completely independent from the network applications and the controller. This allows it to update the controller independently from the flow delegation system (which happens frequently with respect to security updates, for example). It also allows it to completely replace the controller if the new controller uses the same southbound interface.

An overview of the two prototypes is given in Table 8.1. Some more details are introduced in the two subsequent sections, followed by a brief summary of other prototypical work.

	Middleware Prototype (from 2016)	Proxy Prototype (from 2019)
More details in	[BZ16]	[BDZ19]
Language	Python 2.7	Java 8
OpenFlow	v1.3	v1.3
SLOC <sup>2</sup>	9.463	13.342

<sup>1</sup>Note that conflict resolution is still required, it is just implemented in the middleware and not in a separate proxy layer.

<sup>2</sup>Source lines of code in the prototype repository as reported by sloccount, see [http://manpages.ubuntu.com/manpages/precise/man1/compute\\_all.1.html](http://manpages.ubuntu.com/manpages/precise/man1/compute_all.1.html) (last accessed 2020-03-17)

Transparency	No transparency: this early prototype was directly integrated into the controller	Full transparency: realized with control message interception, i.e., inside a proxy layer
Contributors	Robert Bauer, Ioannis Papamanoglou	Robert Bauer, Simon Herter, Addis Dittebrandt
Code access	(1)	(2), (3)

(1) <https://git.scc.kit.edu/work/sdn-pbce>  
(2) <https://github.com/kit-tm/gcmi>  
(3) <https://github.com/kit-tm/gcmi-exp>

**Table 8.1:** *Key characteristics of the two investigated prototypes*

### 8.1.1 Middleware Prototype

The middleware prototype was created in the context of [BZ16]<sup>3</sup>. It implements the monitoring system as described in Chapter 4, the ingress-port based aggregation scheme as described in Sec. 5.2.2.3 and the detour procedure as described in Chapter 6. Monitoring is done via the REST interface of the controller. Metadata transport between delegation and remote switch uses the differentiated services codepoint of the differentiated services field inside the IPv4 header which is sufficient for small networks with less than 64 switches.

The delegation algorithm is realized as a simplified version of the greedy strategy described in Sec. 10.4. Simplified means only a single delegation template can be selected or unselected per time slot and the algorithm does not include knowledge about the future. Whether a template is selected or unselected is controlled with two simple thresholds: Threshold 1 is set to a value slightly below the capacity of the switch, i.e., a new delegation template is selected if the current utilization exceeds 90% of the capacity (for example). And a selected delegation template is unselected if the utilization falls below threshold 2 which is set to a lower value than threshold 1 (explained in more detail in [BZ16]).

Control message interception is not included in this version. The whole system is instead integrated into the application execution environment of the Ryu controller [Ryu19]. This is a middleware approach where all internal communication between controller and network applications is supervised and possibly altered directly inside the controller. More precisely, a small wrapper class around `app_manager.RyuApp` intercepts important

<sup>3</sup>Note that the notation in the original paper and this document are different. The core concept, however, is the same. Selection of EvPorts in [BZ16] represent delegation template selection. Selection of DPorts represents remote switch allocation.

callbacks like the `packet_in` handler function. This requires alterations to the core functionality of the controller and individual network applications have to change their super class to the middleware wrapper. It is therefore not transparent to the control plane which is addressed by the proxy prototype in the next section.

### 8.1.2 Proxy Prototype

The proxy prototype is a different implementation of the flow delegation system. It uses the same concepts as the middleware prototype but not the same code. It is implemented in Java<sup>4</sup> while the middleware prototype was implemented in Python. This was done to demonstrate that flow delegation can be realized with different programming languages.

Because the same concepts are applied, the majority of the functionality is identical to the middleware prototype which includes monitoring, rule aggregation, delegation algorithm, and detour procedure. The major difference is that this version is not integrated into the controller. Instead, the control message interception approach described in Chapter 7 is used to hide the flow delegation system from the control plane. The used framework for control message interception was developed by Simon Herter in his master thesis entitled “Transparent Intermediate Layers for Software Defined Networks” in 2017. The proxy prototype was then integrated into this framework.

### 8.1.3 Other Prototypical Work

Other prototypical work in the context of flow delegation conducted by several students is briefly summarized below. All mentioned student theses were supervised by the author of this thesis.

A very early and limited prototype for flow delegation was created in the bachelor thesis from Florian Schweizer in 2015 entitled “Implementierung und Evaluierung eines Ansatzes für Flowtable-Skalierbarkeit im SDN-Kontext”. This version was integrated into the ONOS controller<sup>5</sup> and did only support a single switch and static delegation, i.e., relocation of flow rules was triggered manually. Another more flexible version was developed by Ioannis Papamanoglou in his bachelor thesis entitled “Flowtable-Delegation in Multi-Switch-Topologien” in 2016. This work focused primarily on support for multiple switches and first simple heuristics for the delegation algorithm.

---

<sup>4</sup>The core functionality is implemented in <https://github.com/kit-tm/gcml/blob/master/composer/src/main/java/com/github/sherter/jcon/composer/Pbce2Layer.java> (last accessed 2020-03-16)

<sup>5</sup>Open Network Operating System, see <https://www.opennetworking.org/onos/>, last accessed 2020-03-16

There are also two simulator-based prototypes. The first one was created by Addis Dittebrandt in his bachelor thesis entitled “Konzeption und Evaluation einer Heuristik für Flow Table Delegation”. This work introduced important metrics such as over- and underutilization that are also used in this thesis. It evaluated the approach with a custom discrete event simulator written in Python using a three-tier data center topology with realistic traffic workloads based on [Kan+09] and [BAM10b]. It was shown here that flow delegation does also work in structured topologies with data center traffic characteristics. The second simulator-based prototype (using ns-3) was created by Eric Sallermann in his master thesis “Implementation and Evaluation of a Flow Delegation Algorithm”. This work primarily showed that existing algorithms for software offloading such as [Kat+14a] can be used in the rule aggregation scheme.

The ideas presented in [BZ16] were further successfully reproduced in the context of a practical course called “Projektpraktikum: Softwarebasierte Netze“ held at Karlsruhe Institute of Technology<sup>6</sup>. As part of this course, small groups of students re-implement and re-evaluate existing SDN concepts. It was shown in this context by one of the groups in 2019 that metadata transport can be successfully realized with VLAN tags<sup>7</sup>.

## 8.2 Functional Evaluation of the Prototypes

This section shows that flow delegation can mitigate flow table capacity bottlenecks in a real (emulated) network. The results are taken from [BZ16] and [BDZ19]. They were obtained with mininet [LHM10], a widely used network emulator in the SDN context. Sec. 8.2.1 introduces the setup of the experiment. Sec. 8.2.2 discusses the results.

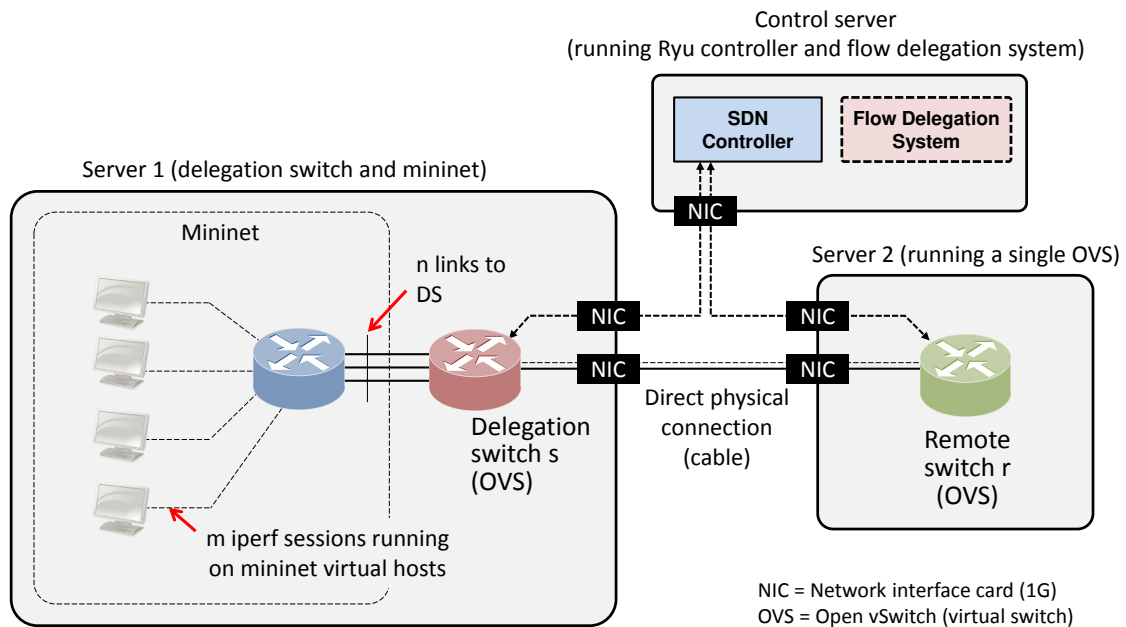
### 8.2.1 Setup

The setup for the functional evaluation is shown in Fig. 8.1. It consists of three physical servers, each equipped with an Intel(R) Xeon(R) L5420 processor (2.50 GHz, 2 sockets, 4 cores/socket) and separate physical 1Gbit/s networks for control and data plane traffic. Server 1 on the left runs a single delegation switch (Open vSwitch v2.4.0) shown in red. It also runs a set of virtual hosts inside mininet that are attached to an auxiliary switch shown here in blue. This auxiliary switch is connected to the delegation switch with  $n$  links and forwards TCP traffic from one virtual host to the delegation switch using one of these  $n$  links. The used link is determined by the networking application running inside the controller on server 3 (based on TCP ports). The remote switch – with unlimited

<sup>6</sup>see <http://telematics.tm.kit.edu/english/ppsdn.php>, last accessed 2020-03-16

<sup>7</sup>Results are stored in a private repository and can be provided upon request, see <https://git.scc.kit.edu/tm-praktika/ppsdn-2019/g2>, last accessed 2020-03-16

flow table capacity – is represented by another Open vSwitch placed on server 2. The controller and the flow delegation system are placed on a dedicated control server.



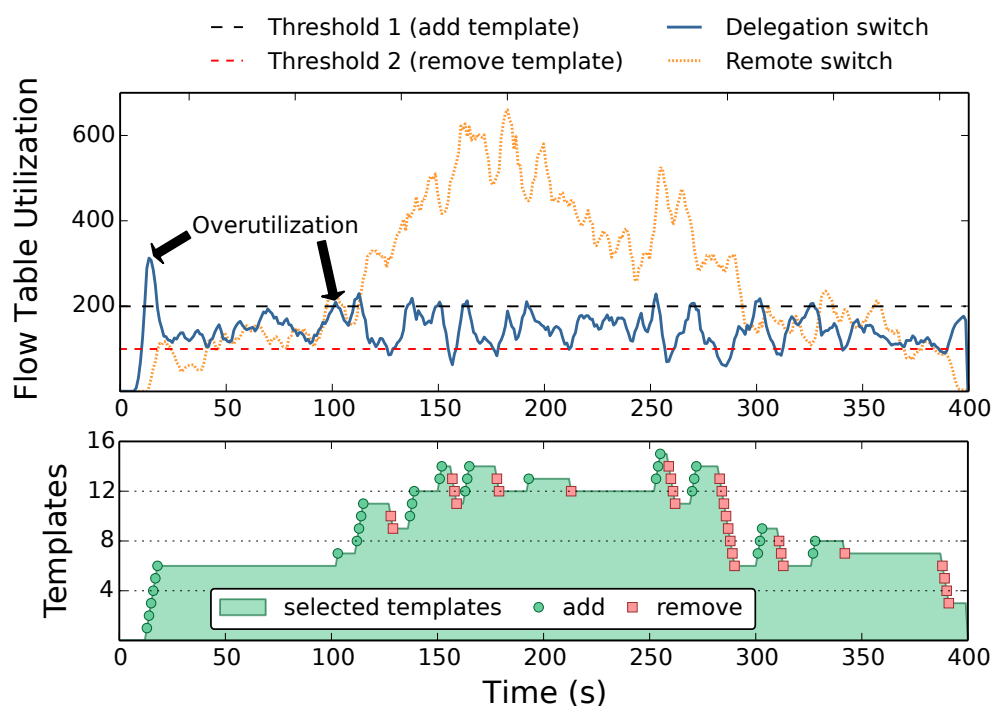
**Figure 8.1:** Setup for the functional evaluation

This simplified setup was chosen because i) we are primarily interested in overall feasibility of the flow delegation approach and ii) the setup allows it to scale the number of iperf sessions without changing the port count of the delegation switch. The latter means that traffic of multiple virtual hosts can be sent over the same link to the delegation switch. This is required because a high number of iperf sessions is needed to create high flow table utilization. However, if 300+ virtual hosts are directly connected to the delegation switch, this switch has 300+ ports which is not realistic – and also problematic because the number of ports is equal to the number of delegation templates in the ingress port based aggregation scheme. Using the setup described in Fig. 8.1, the 300+ virtual hosts are attached to the blue auxiliary switch while flow delegation does only consider the red delegation switch and the green remote switch.

### 8.2.2 Result

**Findings:** Both implementations of the flow delegation system – the middleware and the proxy prototype – work as expected when applied to an emulated software-based network, i.e., flow table capacity bottlenecks are mitigated.





**Figure 8.2:** Mitigation of a severe bottleneck situation with the middleware prototype

Fig. 8.2 shows a single experiment with the middleware prototype where the flow table capacity of the delegation switch is set to 200 flow rules. The upper part of the figure shows the amount of flow rules currently stored in the flow table of the delegation (blue curve) and the remote switch (yellow curve). The lower part shows the amount of selected delegation templates. At  $t=12$  seconds, the flow table of the delegation switch contains more than 200 rules, i.e., threshold 1 is exceeded<sup>8</sup>. The first delegation template  $d_1$  is selected and the associated aggregation rule is installed. As a result, all flow rules associated with  $d_1$  installed after the aggregation rule are automatically relocated to the remote switch.

Because the delegation algorithm only selects one delegation template per time slot (see Sec. 8.1.1), it takes six consecutive time slots  $t=6$  to  $t=11$  to select enough delegation templates in order to fully mitigate the bottleneck. As soon as enough delegation templates are selected at  $t=12$ , flow delegation is able to keep the utilization below the capacity without selecting additional delegation templates – until the utilization increases again at  $t=102$  and a new delegation template is selected. It can further be seen in the rest of the experiment that delegation templates are selected and unselected in such a way that the bottleneck is mitigated. It was further confirmed that all packets were forwarded correctly

<sup>8</sup>In this experiment, threshold 1 is set to 100% of the capacity and threshold 2 is set to 50% of the capacity.

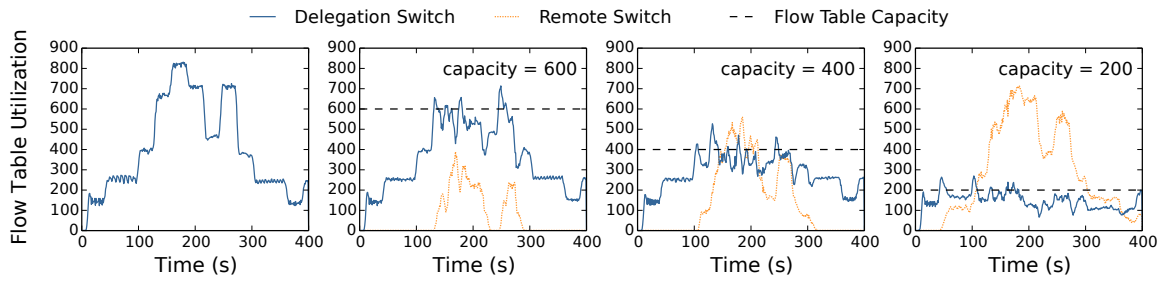


Figure 8.3: Mitigation examples with the middleware prototype

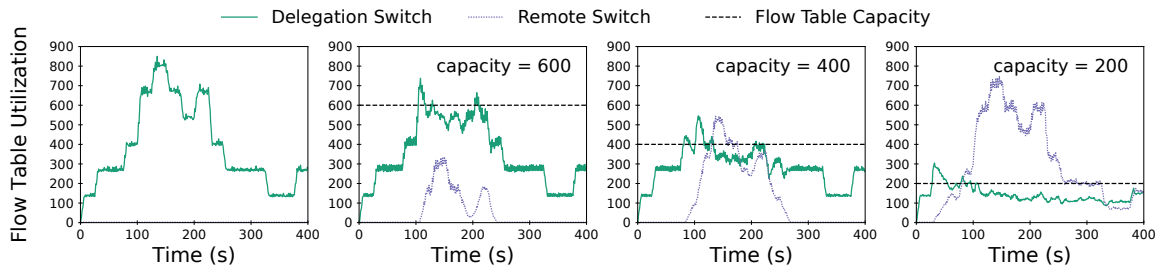


Figure 8.4: Mitigation examples with the proxy prototype

(no packet loss). It can be concluded that the interplay between the four<sup>9</sup> involved building blocks – monitoring system, aggregation scheme, delegation algorithm and detour procedure – **works as expected when applied to a real (emulated) software-based network**.

It can further be seen the simple threshold-based delegation algorithm does not perform particularly well here. The delayed selection of new delegation templates between  $t=6$  and  $t=12$  leads to a period with overutilization i.e., the amount of flow rules stored in the flow table of the delegation switch exceed the capacity. This shows **a sophisticated delegation algorithm is required** that can select multiple delegation templates at a time and include anticipated future knowledge to proactively deal with bottlenecks – which is the main motivation for the designed algorithms in chapter 9 to 11.

Fig. 8.3 shows the approach can deal with bottlenecks of different severity. The leftmost plot shows the flow table utilization with enough capacity and without flow delegation. The three other plots show the same scenario, except that the capacity of the flow table is reduced to 600, 400 or 200 flow rules and flow delegation is used. It can be seen that, in all cases, the bottlenecks are mitigated in a similar way as discussed above. Fig. 8.4 finally shows the exact same experiments with the proxy prototype, taken from [BDZ19]. Recall that the proxy prototype is not integrated into the controller but uses a TCP proxy

<sup>9</sup>The middleware proxy does not use control message interception

to intercept control messages while the other four building blocks are realized in the same way as before. It can be seen that the outcome is identical from a conceptual perspective which is the desired result. The small differences can be explained with randomization and timing effects in the conducted experiments.

The remainder of this chapter will now investigate two kinds of overhead associated with the flow delegation system that are important in practice: monitoring overhead and detour overhead. Overhead and runtime characteristics of the delegation algorithm are also important but this is discussed in detail in Chapter 15 and Chapter 16. All presented overhead values were obtained with the middleware prototype.

## 8.3 Monitoring Overhead

**Findings:** Monitoring – if not already done by the controller – can increase the amount of required CPU resources on the control sever by 5% to 28% and the traffic on the control channel between controller and switch by up to 87% (measured for a scenario with one bottlenecked switch).

The different building blocks of the flow delegation system induce overhead. In case of the monitoring system, it is required to continuously collect parameters for bottleneck detection and for comparison of delegation templates. The most expensive part in this context is gathering the average amount of bits/s processed by a flow rule  $f$  in time slot  $t$  ( $\delta_{f,t}$ ). This information requires periodic polling of all bottlenecked switches which causes two kinds of overhead:

- **Control message overhead:** additional control messages of type `counters_req(f)` and `counters_rsp(f)` are exchanged between controller and switch to perform monitoring, i.e., more bandwidth is required for the control channel
- **Computational overhead:** processing of these additional control messages requires CPU resources in the flow delegation system and in the switch CPU

Note that such additional monitoring is only required if the information for  $\delta_{f,t}$  is not already available. It is assumed that most software-defined networks will perform some kind of monitoring – e.g., for QoS, load balancing or other purposes – so that no or little additional monitoring is required by the flow delegation system. However, if this is not the case, it is important to keep in mind that gathering the required information can be expensive.

The two kinds of overhead mentioned above (control message and computational overhead) are investigated in [BZ16] with a series of mininet experiments under worst case

assumptions, i.e., all monitoring has to be done by the flow delegation system. It is shown in [BZ16] that the computational overhead is below 5% for "moderate" bottleneck scenarios. Moderate means the maximum flow table utilization does not exceed 110% of the capacity ( $u_{\max}^{\text{Table}} < 1.1 * c_s^{\text{Table}}$ ). 5% overhead means that a baseline setup running only the controller without monitoring required 5% less CPU time compared to a setup with flow delegation enabled and polling of all flow rules every 3 seconds. In extreme cases with  $u_{\max}^{\text{Table}} > 5 * c_s^{\text{Table}}$ , the overhead is below 18.75% in 99% of the cases if the switch is polled every six seconds and below 27.85% if the switch is polled every three seconds. Control message overhead is at 86.91% in extreme cases, i.e., the amount of traffic on the control channel is almost doubled compared to a baseline setup without monitoring. The values represent a scenario with a single bottlenecked switch, i.e., there is more overhead if multiple switches experience a bottleneck at the same time.

## 8.4 Detour Overhead

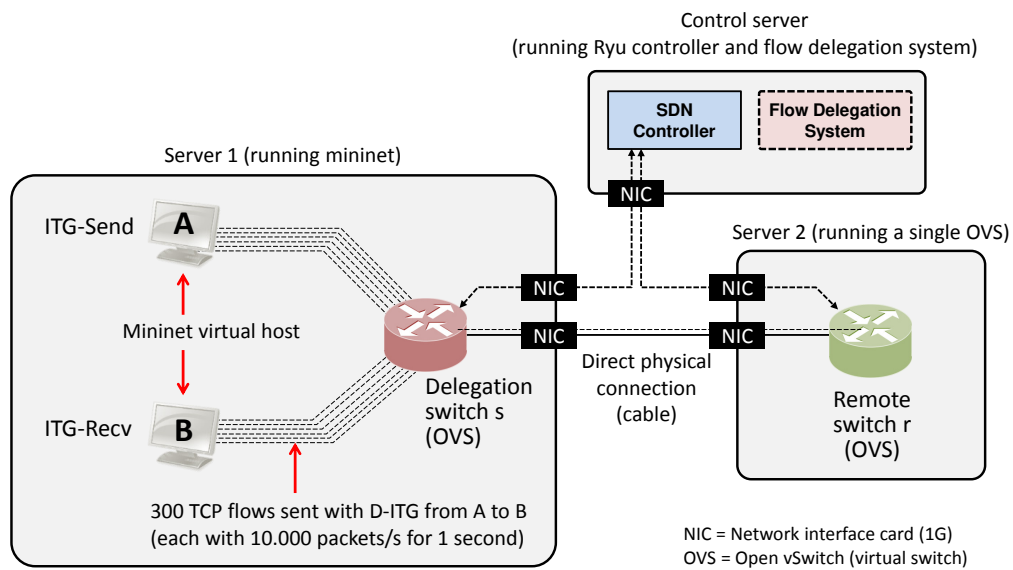
An important overhead parameter with respect to the detour procedure is additional delay of relocated packets. Packets that are matched by an aggregation rule are sent to the remote switch, processed by the remote switch, and sent back to the delegation switch. This takes additional time compared to a packet that is not relocated. The presented results are taken from [BZ16]. The setup for the experiment is explained in Sec. see 9.16. The results are discussed in Sec. see 9.16.

### 8.4.1 Setup

The setup for the delay experiment is shown in Fig. 8.5. It consists of three physical servers, each equipped with an Intel(R) Xeon(R) L5420 processor (2.50 GHz, 2 sockets, 4 cores/socket) and separate physical 1Gbit/s networks for control and data plane traffic. The sever on the left runs the mininet tool [LHM10] with delegation switch (OVS, Open vSwitch v2.4.0). The server on the right runs a stand-alone OVS used as a remote switch. The Distributed Internet Traffic Generator (D-ITG, [AP12]) is used in this experiment to provide accurate delay statistics on packet level granularity. The tool is configured as follows: 300 consecutive D-ITG flows (using the TCP profile with a constant packet rate of 30Mbit/s) are created between two virtual hosts A (executes ITG-Send) and B (executes ITG-Recv) where every D-ITG flow lasts for one second before the next one is started.

The Ryu controller [Ryu19] in the control server runs a reactive network application that installs two new flow rules  $f_1$  and  $f_2$  for each generated D-ITG flow<sup>10</sup>.  $f_1$  forwards

<sup>10</sup>In addition, this network application also installs the necessary flow rules to handle the D-ITG signalling traffic between ITG-Send and ITG-Recv (done proactively before the experiment is started)



**Figure 8.5:** Setup for the delay experiment

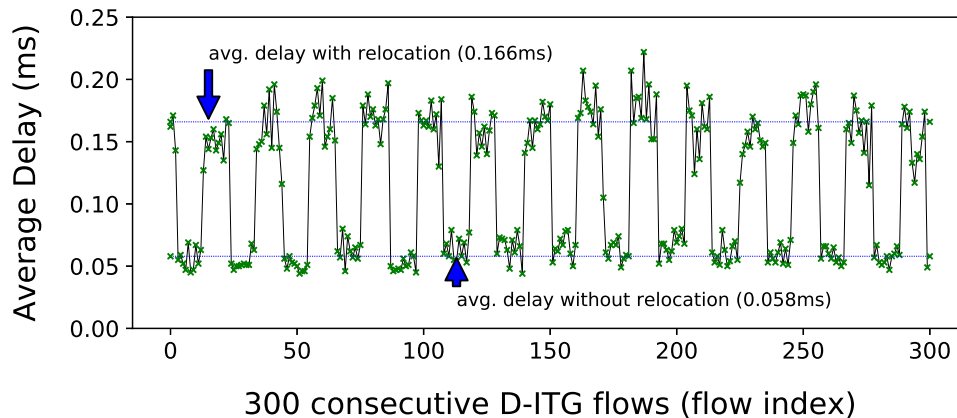
packets sent from A to B and  $f_2$  forwards packets sent from B to A. The rule aggregation scheme returns a single static delegation template  $d$  and all flow rules of type  $f_1$  are associated with this delegation template. The delegation algorithm is executed once per second. It is statically configured to “toggle” the selection status of delegation template  $d$  every 10 seconds, i.e., the template is selected if it was not selected in the last 10 seconds and vice versa. This is done to compare the measured delay based on whether the delegation template is selected (i.e., packets are relocated) or not.

## 8.4.2 Result

**Findings:** The average end-to-end delay for relocated packets is increased by approx. 0.1ms compared to the end-to-end delay of packets that are not relocated.

Fig. 8.6 shows the average delay measured by D-ITG for each of the 300 generated flows. Each flow is represented here as an index on the x-axis:  $x = 0$  is the first flow started after 0 seconds,  $x = 1$  is the second flow started after 1 second, and so on.

If the delegation template is not selected (no relocation), the average end-to-end delay is measured as 0.058 milliseconds, which is reasonable, because the forwarding is handled locally inside the OVS. This delay is increased to a value between 0.15 and 0.2 milliseconds if the delegation template is selected. Note that, in the latter case, packets are transmitted via a physical link to the OVS on server 2 and then back via the same link to the delegation switch (on server 1). The same experiment was also conducted with a physical SDN



**Figure 8.6:** Average delay measured for 300 flows

switch (Brocade ICX 6610) replacing server 2. In this case, the delay is even smaller (less than 0.13ms of *total* end-to-end delay for all relocated packets) because the forwarding in the remote switch is done in hardware and not in software.

0.1ms of *additional* delay is acceptable for many scenarios given the alternative of an unresolved capacity bottleneck. However, this might not be applicable in scenarios where ultra low delays are required. In this case, the delay-sensitive flows have to be excluded from flow delegation which is not discussed here further.

## 8.5 Conclusion

This chapter introduced two different prototypes of the flow delegation system: a middleware prototype written in Python which is integrated directly into the Ryu controller and a proxy prototype written in Java which uses control message interception to hide flow delegation from the control plane. It is shown that both prototypes can mitigate flow table capacity bottlenecks in an emulated software-based network. It is further shown that monitoring can be expensive in terms of required CPU resources as well as control channel bandwidth and that packets of relocated flows have a slightly increased end-to-end delay (+0.1ms). Finally, the prototypical results show that the simple threshold-based delegation algorithm introduced in [BZ16] does not perform particularly well with respect to overutilization which motivates the more sophisticated design that is discussed in the next three chapters.

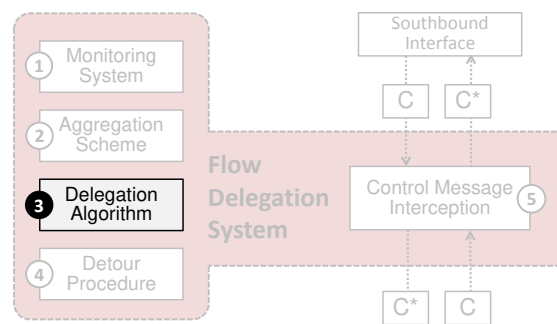
Part III  
Algorithms





# Decomposition and Problem Formulations

---



The algorithms part of the thesis – which consists of chapters 9 to 11 – investigates the so-called delegation algorithm. The **delegation algorithm** decides how flow table capacity bottlenecks are resolved based on discrete combinatorial optimization. The algorithms part makes the following contributions:

- (1) Chapter 9 models flow table capacity bottleneck mitigation as a **two-step optimization approach** with two independent sub-problems: a switch-local problem to select delegation templates (DT-Select) and a global problem to select an appropriate remote switch for each selected delegation template (RS-Alloc). Single-period and multi-period problem formulations are introduced for both sub-problems.
- (2) Chapter 10 presents three **algorithms for the multi period DT-Select problem** that take predicted future network situations into account and proactively mitigate anticipated bottlenecks. Select-Opt and Select-CopyFirst use integer programming. The third algorithm (Select-Greedy) uses a simple greedy strategy.
- (3) Chapter 11 presents an **algorithm for the multi period RS-Alloc problem**. The algorithm uses integer programming and reduce the search space with a pre-processing step that removes unfavorable allocation options (for better scalability).

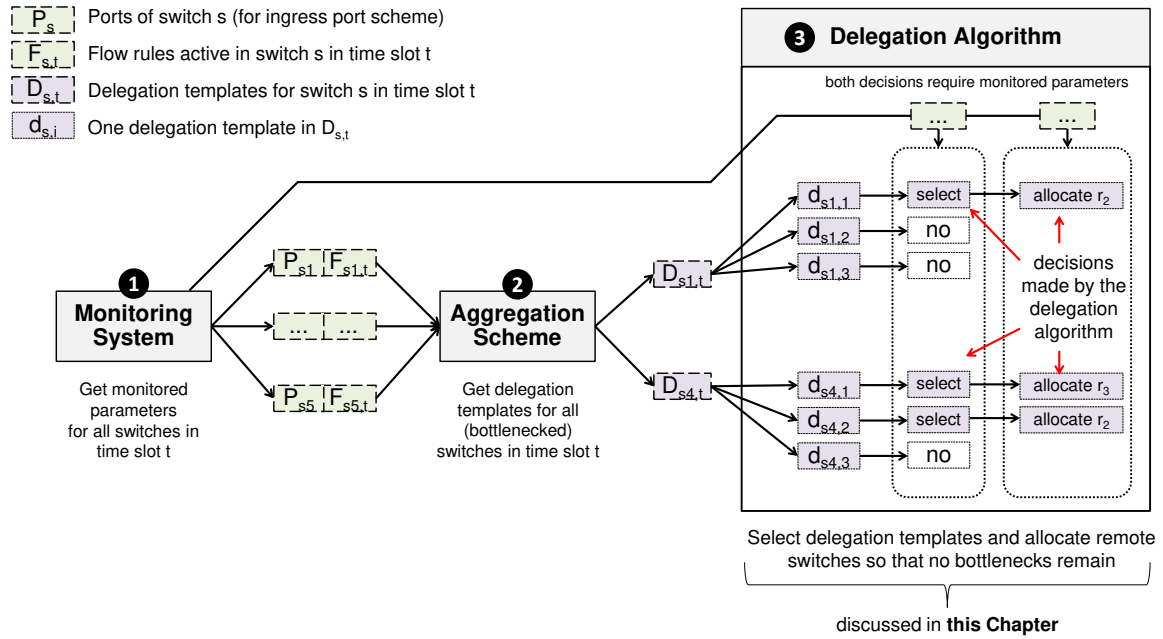


Figure 9.1: Delegation algorithm building block

The high level idea of the delegation algorithm is shown in Fig. 9.1. It takes two types of inputs: a set of delegation templates  $D_{s,t}$  calculated by the aggregation scheme for each switch with a flow table capacity bottleneck, shown in purple boxes. And several monitored parameters from the monitoring system with information about relevant flow rules (such as processed bits), shown in green boxes. Based on these inputs, the algorithm has to make two decisions:

- Given switch  $s$  suffers from a flow table capacity bottleneck in time slot  $t$ , which of the delegation templates  $d \in D_{s,t}$  are **selected** in order to mitigate the bottleneck? This is called delegation template selection in the following (DT-Select).
- For each of the selected delegation templates in time slot  $t$ , which remote switch  $r$  is **allocated** to the delegation template so that the available free resources (flow table capacity, link bandwidth) are utilized best? This is called remote switch allocation in the following (RS-Alloc).

In the example in Fig. 9.1, there are five switches  $s_1$  to  $s_5$  but only two of them –  $s_1$  and  $s_4$  are suffering from a flow table capacity bottleneck in time slot  $t$  (the figure depicts only a single time slot). Based on the information from the monitoring system (building block 1), the rule aggregation scheme (building block 2) has calculated two sets of delegation templates  $D_{s_1,t}$  and  $D_{s_4,t}$ . Here, both sets consists of three delegation templates. These are  $d_{s_1,1}$  to  $d_{s_1,3}$  in case of switch  $s_1$  and  $d_{s_4,1}$  to  $d_{s_4,3}$  in case of switch  $s_4$ . The delegation algorithm uses the conflict-free cover sets in the delegation templates and

parameters provided by the monitoring system – shown here as [...] – to make its decision. A possible output of the delegation algorithm is shown in the right of Fig. 9.1:

- It has decided that one delegation template for  $s_1$  is sufficient to mitigate the bottleneck (**selection** of  $d_{s_1,1}$ ). The flow rules in the conflict-free cover set of  $d_{s_1,1}$  are relocated to remote switch  $r_2$  ( $r_2$  is **allocated** to delegation template  $d_{s_1,1}$ ).
- It has decided that two delegation templates are required for  $s_4$  (**selection** of  $d_{s_4,1}$  and  $d_{s_4,2}$ ). The flow rules in the conflict-free cover set of  $d_{s_4,1}$  are relocated to remote switch  $r_3$  ( $r_3$  is **allocated** to  $d_{s_4,1}$ ) and the flow rules in the conflict-free cover set of  $d_{s_4,2}$  are relocated to remote switch  $r_2$  ( $r_2$  is **allocated** to  $d_{s_4,2}$ ).

The remainder of this chapter explains how the two above decisions – selection and allocation – can be modeled as a two-step optimization approach using a 0-1 knapsack problem for the selection part (DT-Select) and a capacitated facility location problem for the allocation part (RS-Alloc). Sec. 9.1 summarizes the requirements for the delegation algorithm. Sec. 9.2 repeats important concepts and variables from previous chapters that are used here. Sec. 9.3 explains the decomposition and interplay of the two sub-problems DT-Select and RS-Alloc. The two following sections then discuss the sub-problems in detail: Sec. 9.4 discusses the DT-Select sub-problem and Sec. 9.5 discusses the RS-Alloc sub-problem. Both sections explain the modeling approach, the used decision variables, the coefficients and introduce a single period (considering only a single time slot) as well as a multi period problem formulation (multiple time slots are considered). In addition, the main challenge of both sub-problems is analyzed which will then be addressed in the subsequent chapters where algorithms are presented for the multi period case. The chapter concludes with a short summary in Sec. 9.6.

## 9.1 Requirements

This section briefly summarizes the most important requirements for the delegation algorithm.

- (1) Prediction of future network situations has to be taken into account. To achieve this, the delegation algorithm must work on multiple time slots which requires a multi period problem formulation.
- (2) Support for different objectives. The goal to not exceed the flow table capacity is only one (key) objective. There are several other objectives with respect to utilization of available resources (flow table capacity, link bandwidth, number of required control messages). This leads to a multi-objective optimization problem.

- (3) High performance and scalability. The delegation algorithm has to be executed frequently, for a potentially large number of SDN switches. It is therefore essential to develop a fast and scalable optimization approach that can be solved in a few milliseconds. This led to the decision to design a two-step optimization approach that consists of two sub-problems (instead of using a single more complex optimization problem).

## 9.2 Preliminaries

The problems introduced below depend on concepts from previous chapters. The most important concepts and the associated variables are briefly summarized here (contains no new information, can be skipped if the concepts are already known).

- Time is represented in **time slots**. Time slot  $t \in T$  is defined as the time window that begins at  $\tau_{-1}$  and ends at  $\tau$ . Multiple time slots are given as a set  $T := \{t_1, t_2, \dots, t_{-1}, t, t_{+1}, \dots, t_{m-1}, t_m\}$ .  $t_{-1}$  represents the time slot before  $t$ ,  $t_{-2}$  the time slot before  $t_{-1}$  and so on.  $t_1$  is defined as the first time slot in  $T$  and  $t_m$  is defined as the last time slot. Details can be found in Sec. 2.1.1 (definition) and Sec. 4.2.1 (translation from timestamps to time slots).
- A **delegation switch** is represented with variable  $s$  (and  $s_j$ ), a **remote switch** is represented with variable  $r$ . Details can be found in Def. 2.2.
- $u_{s,t}^{\text{Table}}$  denotes the current **flow table utilization** from the perspective of the controller and the network applications in switch  $s$  in time slot  $t$  *without* flow delegation (see Sec. 2.12).
- Coefficients are represented in **lambda notation**. This notation models the life cycle of a flow rule with time slots.  $\lambda_{f,t}^a \in \{0, 1\}$  is set to 1 for all time slots where the flow rule is present in the flow table.  $\lambda_{f,t}^i \in \{0, 1\}$  is set to 1 only for the one time slots where the flow rule was installed. Details can be found in Sec. 4.2.2.
- Options to mitigate a bottleneck are modeled as **delegation templates**. A delegation template  $d \in D_{s,t}$  is a tuple  $\langle \vec{m}_d, F_{d,t} \rangle$  where  $\vec{m}_d$  is an aggregation match and  $F_{d,t}$  is the conflict-free cover set of  $\vec{m}_d$ . Details can be found in Sec. 3.2.7.
  - An **aggregation match** is a special wildcard match that covers a set of other matches  $M$ . If a packet is matched by a match in  $M$ , it is also matched by the aggregation match (see Sec. 3.2.1).
  - The **conflict-free cover set**  $F_{d,t}$  represents a set of flow rules that can be relocated to a remote switch without rule conflicts (see Sec. 3.2.5).

- Delegation templates are only calculated for delegation switches (not for switches without a bottleneck). A delegation template  $d \in D_{s,t}$  is always associated with delegation switch  $s$ . Selecting a delegation template  $d$  means an **aggregation rule** with match  $\vec{m}_d$  is installed in delegation switch  $s$  in the beginning of time slot  $t$  and flow rules from  $F_{d,t}$  are relocated to a remote switch (which reduces the flow table utilization in the delegation switches = bottleneck mitigation). It can be concluded that at least one delegation template has to be selected for each delegation switch.
- Utilization and cost coefficients are calculated differently for different **aggregation priorities** ( $\text{prio}_{\text{agg}}$ , a global parameter). If aggregation rules are installed with highest priority ( $\text{prio}_{\text{agg}}^{\text{highest}}$ ), all flow rules in the conflict-free cover set are relocated. If aggregation rules are installed with lowest priority ( $\text{prio}_{\text{agg}}^{\text{lowest}}$ ), only “new” flow rules that were installed after the beginning of time slot  $t$  are relocated. The problems below focus on low aggregation priority. Details can be found in Sec. 3.2.1 (definition) and Sec. 5.3 (discussion).

### 9.3 Problem Decomposition

The two basic decisions mentioned above – selection of delegation templates and allocation of remote switches – could be handled as one unified optimization problem. Because indirect rule aggregation is used, the amount of delegation templates per switch ( $|D_{s,t}|$ ) is expected to be small – in the range of 20 to 200 (number of physical ports) for the ingress port scheme, see Sec. 5.2.2.3. However, the number of switches affected from a bottleneck may be arbitrarily large which can result in a (too) large number of coefficients in the unified problem. This is especially problematic if multiple time slots are considered. It is shown later that the calculation of the decomposed problem – which is smaller and simpler – can take several milliseconds per switch in the worst case. And even if a very fast heuristic would be available: the time to prepare the input data for the unified problem alone would lead to scalability problems.

Fortunately, the flow table capacity bottleneck mitigation problem can also be modeled as two independent sub-problems. The problem decomposition is explained in Fig. 9.2. The top of the figure shows the first sub-problem which is called **Delegation Template Selection (DT-Select)** 3.1. The DT-Select problem is solved independently for each bottlenecked switch and selects a subset of the delegation templates to mitigate the bottleneck of this single switch. So the three boxes labeled as (a) represent individual optimization problems with individual inputs and outputs. This step, however, will not yet allocate the remote switches to the templates. It simply returns the subset of the selected delegation templates  $\overline{D}_{s,t}^*$  for each bottlenecked switch. The single asterisk

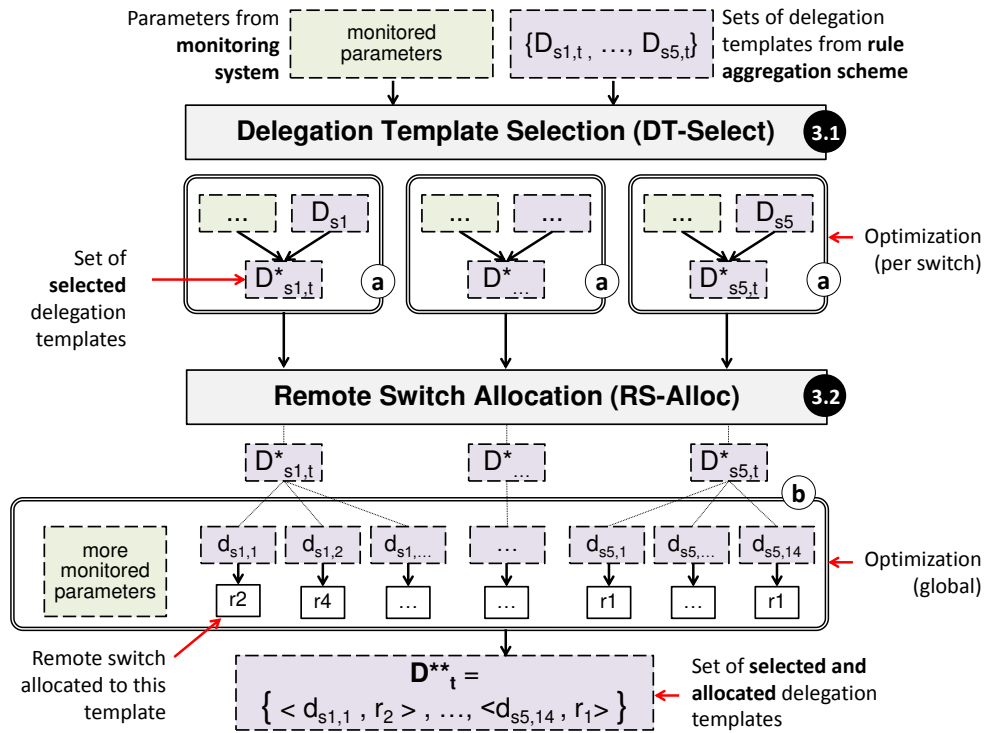


Figure 9.2: Problem Decomposition

(upper index) indicates that the first step of the delegation algorithm was executed and the set contains delegation templates suitable to mitigate the bottleneck.

The second step – the allocation part – is referred to as **Remote Switch Allocation (RS-Alloc)** 3.2. The RS-Alloc problem takes the set of selected delegation templates  $\{D_{s,t}^*\}$  (for all bottlenecked switches) and assigns each of it a remote switch based on flow table utilization and utilization of the links in the infrastructure. This step is executed only once per time slot and returns a set of selected and assigned delegation templates  $\{D_{s,t}^{**}\}$ . In contrast to the first step, the RS-Alloc problem is modeled as one global problem (shown in the box labeled as (b)). The output of RS-Alloc represents a solution for all bottlenecks in the network in time slot  $t$ . The double asterisk (upper index) indicates that both steps of the delegation algorithm were executed.

It can be seen in Fig. 9.2 that both sub-problems require input from the monitoring system. DT-Select primarily requires information about the flow rules contained in the conflict-free cover set that comes with each delegation template  $(\lambda, \delta)$ . RS-Alloc primarily requires information about flow table and link utilization of all switches / links in the

network and the number of flow rules and packets that must be relocated. All relevant monitored parameters are defined in Sec. 4.1.

## 9.4 Delegation Template Selection (DT-Select)

Delegation template selection (DT-Select) selects a subset  $D_{s,t}^* \subseteq D_{s,t}$  out of  $n$  delegation templates for switch  $s$  in time slot  $t$  so that i) the utilization of the flow table of switch  $s$  does not exceed its capacity and ii) overhead caused by flow delegation is minimized.

Sec. 9.4.1 first explains the general modeling approach. Afterwards, the decision variables and the required coefficients are explained (Sec. 9.4.2 to Sec. 9.4.4), followed by the definition of the single period DT-Select problem in Sec. 9.4.5 and its multi period extension in Sec. 9.4.6. Finally, the main challenge of the multi period DT-Select problem is discussed in Sec. 9.4.7.

### 9.4.1 Modeling Approach

The basic idea is to model DT-Select as a 0-1 knapsack problem. Selection-related knapsack problems are commonly used in the networking domain, e.g., for energy aware routing (select devices [SLX10]), delay tolerant forwarding (select relays [Gao+09]) or content caching (select caching locations [BGW10]). And there are also various related problems outside of the networking domain, e.g., project selection, budget control or cargo loading [SK75]. The 0-1 knapsack problem considers  $n$  different items. If formulated as a minimization problem (as it will be done here), each of the items is assigned a cost and a weight and the objective is to select items so that the sum of the costs is minimized and the sum of the weights does not exceed the capacity of the knapsack.

In DT-Select, the knapsack is the flow table of delegation switch  $s$  and the items are the delegation templates  $d \in D_{s,t}$ . The knapsack capacity is modeled as a negative value that represents the amount of flow rules that have to be relocated in order to mitigate the bottleneck. The weight of each template is given as a utilization coefficient  $u_{d,t}^{\text{Table}}$  that represents the amount of rules that will be relocated if template  $d$  is selected (i.e., added to the knapsack). And the cost of each template is given as a cost coefficient  $w_{d,t}$  that represents the overhead associated with the template, e.g., how much traffic is processed by the relocated flow rules (= more traffic is added to the link between delegation and remote switch). Goal is then to add delegation templates to the knapsack so that the cost is minimal and the capacity is non-negative. The time slot indices indicate that the variables may change between time slots.

### 9.4.2 Decision Variables

The decision variables in DT-Select are defined as follows:

**Definition 9.1: Decision Variables for DT-Select**

Selection of a delegation template  $d \in D_{s,t}$  for delegation switch  $s$  in time slot  $t$  is based on the following **decision variables**:

$$X_{d,t} := \begin{cases} 1, & \text{delegation template } d \text{ is selected in time slot } t \\ 0, & \text{delegation template } d \text{ is not selected in time slot } t \end{cases} \quad (9.1)$$

These decision variables are defined for each delegation template  $d \in D_{s,t}$  in time slot  $t$ . Undefined variables are interpreted as 0.

$X_{d,t} = 1$  means delegation template  $d$  is selected for bottleneck mitigation in time slot  $t$  (template is added to the knapsack).  $X_{d,t} = 0$  means delegation template  $d$  is not selected in time slot  $t$  and thus not required for bottleneck mitigation (not added to the knapsack). If a delegation template is selected, the aggregation rule associated with the delegation template is installed at the beginning of time slot  $t$  (if the template with the same aggregation match was not already selected in a previous time slot). If  $X_{d,t-1}$  is set to 1 and  $X_{d,t}$  is set to 0, a previously installed aggregation rule is removed at the beginning of time slot  $t$  to stop flow delegation.

### 9.4.3 Utilization Coefficients

The utilization coefficients  $u_{d,t}^{\text{Table}}$  for a single delegation template  $d \in D_{s,t}$  represent the amount of flow rules that will be relocated to the remote switch if delegation template  $d$  is selected. This means  $u_{d,t}^{\text{Table}}$  rules are removed from the flow table of switch  $s$ , i.e., this variable basically quantifies the “mitigation performance” of template  $d$  (in time slot  $t$ ). The utilization coefficients are the weights in the knapsack terminology. Basic idea is it to select the subset  $D_{s,t}^* \subseteq D_{s,t}$  of delegation templates in such a way that the current utilization minus the sum of the utilization of the selected templates is smaller than the flow table capacity of the delegation switch<sup>1</sup>:

$$u_{s,t}^{\text{Table}} - \sum_{d \in D_{s,t}^*} u_{d,t}^{\text{Table}} \leq c_s^{\text{Table}} \quad (9.2)$$

Recall that the current utilization  $u_{s,t}^{\text{Table}}$  does not consider flow delegation. Further recall that the delegation algorithm is only required if switch  $s$  is suffering from a flow table

<sup>1</sup>Eq. (9.2) is equivalent to  $\sum_{d \in D_{s,t}^*} u_{d,t}^{\text{Table}} \leq u_{s,t}^{\text{Table}} - c_s^{\text{Table}}$  where  $u_{s,t}^{\text{Table}} - c_s^{\text{Table}}$  is the negative knapsack capacity



capacity bottleneck in time slot  $t$ , i.e., the current utilization exceeds the capacity ( $u_{s,t}^{\text{Table}} > c_s^{\text{Table}}$ ). So DT-Select has to select at least one delegation template with a non-negative utilization coefficient to satisfy Eq. (9.2). If  $u_{d,t}^{\text{Table}}$  is large compared to  $u_{s,t}^{\text{Table}} - c_s^{\text{Table}}$ , it might be sufficient to select a single or only a few delegation templates to mitigate the bottleneck. If  $u_{d,t}^{\text{Table}}$  is small, more delegation templates might be required.

The utilization is calculated based on the amount of flow rules that can be relocated to the remote switch ( $\text{relocated}_{d,t}$ ) reduced by the overhead caused by the aggregation and backflow rules ( $\text{overhead}_{d,t}$ ):

$$u_{d,t}^{\text{Table}} := \text{relocated}_{d,t} - \text{overhead}_{d,t} \quad (9.3)$$

The first part ( $\text{relocated}_{d,t}$ ) depends on the selected aggregation priority. If aggregation rules are installed with highest priority ( $\text{prio}_{\text{agg}}^{\text{highest}}$ ), all flow rules in the conflict-free cover set  $F_{d,t}$  are relocated. If aggregation rules are installed with lowest priority ( $\text{prio}_{\text{agg}}^{\text{lowest}}$ ), only new flow rules installed after the beginning of time slot  $t$  are relocated. Using the lambda notation,  $\text{relocated}_{d,t}$  is defined as follows:

$$\text{relocated}_{d,t} := \begin{cases} \sum_{f \in F_{d,t}} \lambda_{f,t}^a = |F_{d,t}| & , \text{ if } \text{prio}_{\text{agg}}^{\text{highest}} \\ \sum_{f \in F_{d,t}} \lambda_{f,t}^i \leq |F_{d,t}| & , \text{ if } \text{prio}_{\text{agg}}^{\text{lowest}} \end{cases} \quad (9.4)$$

Eq. (9.4), however, can only be used in the single period case where only a single time slot is considered. In the multi period case, the calculations for  $\text{prio}_{\text{agg}}^{\text{lowest}}$  are more complex because a flow rule could have been relocated in one of the previous time slots which has to be taken into account.

The second part ( $\text{overhead}_{d,t}$ ) is defined as the amount of additional flow rules that are only required for flow delegation.  $\text{overhead}_{d,t}$  consists of two parts. First, the aggregation rule obviously contributes to the flow table utilization of the delegation switch if delegation template  $d$  is selected. And secondly, there is additional flow table overhead due to the backflow rules that are required to handle the return traffic from the remote switch (depends on the forwarding action of the flow rules in  $F_{d,t}$ ). This can be modeled in the following way:

$$\text{overhead}_{d,t} := 1 + \left| \left\{ s \mid \phi_{f,s} == 1, f \in F_{d,t} \right\} \right| \quad (9.5)$$

$\phi_{f,s}$  is defined in Sec. 4.2.3. It is a binary coefficient indicating flow rule  $f$  has a forwarding action that forwards packets to switch  $s \in S$ . The +1 in Eq. (9.5) represents the aggregation rule. The second addend is calculated as the cardinality of a helper set that contains all switches used in a forwarding action (each switch can only occur once).

This represents an upper bound<sup>2</sup> for the amount of backflow rules (one backflow rule for each unique forwarding action). Because both parts – aggregation rules as well as backflow rules – are by design limited by a maximum value of  $|D_{s,t}|$  (ingress port scheme), it can be a valid approach to replace Eq. (9.5) with a static expression equal to  $2 * |D_{s,t}|$ , as it was done, for example, in [BD17]. Note that a static overhead could be compensated by artificially reducing the capacity of the flow table accordingly, so it is possible to optimize this one step further and ignored the overhead-related part altogether. However, in case where  $|D_{s,t}|$  is very high (e.g.,  $> 100$ ), it is important to include the overhead into the decision process which does not allow for such a simplification.

#### 9.4.4 Cost Coefficients

The cost coefficients  $w_{d,t}$  for a single delegation template  $d \in D_{s,t}$  represent the overhead associated with selecting the template. This overhead consists of the amount of required aggregation and backflow rules ( $w_{d,t}^{\text{Table}}$ ), the amount of relocated packets ( $w_{d,t}^{\text{Link}}$ ) and the amount of required control messages ( $w_{d,t}^{\text{Ctrl}}$ ). Basic idea is it to select the subset  $D_{s,t}^* \subseteq D_{s,t}$  of delegation templates in such a way that Eq. (9.2) from the previous section is satisfied (knapsack constraint that ensures the bottleneck is mitigated) and, at the same time, the cost of the selection is minimized:

$$\text{minimize: } \sum_{d \in D_{s,t}^*} w_{d,t} \quad (9.6)$$

The three cost coefficients  $w_{d,t}^{\text{Table}}$ ,  $w_{d,t}^{\text{Link}}$ , and  $w_{d,t}^{\text{Ctrl}}$  mentioned above are explained in more detail below.

##### 9.4.4.1 Table Overhead

The table overhead cost coefficients  $w_{d,t}^{\text{Table}}$  define the amount of flow rules in the flow table of the delegation switch that are only required for flow delegation, i.e., the number of installed aggregation and backflow rules. In this work, the table overhead per delegation template is set to 1, i.e., only the aggregation rules are considered. The overhead could be greater than 1 – in case new backflow rules are required – but this is simplified here by assuming that all backflow rules are installed when the flow delegation system is started (could be included by replacing the static value 1 with Eq. (9.5)).

$$w_{d,t}^{\text{Table}} = 1 \quad (9.7)$$

Considering the table overhead in the minimization problem will prefer delegation templates with a large conflict-free cover set, because this will reduce the number of

---

<sup>2</sup>It is only an upper bound because backflow rules are shared across delegation templates

selected templates in total (and thus minimize the number of aggregation rules which is the table overhead).

#### 9.4.4.2 Link Overhead

The link overhead cost coefficients  $w_{d,t}^{\text{Link}}$  take the amount of relocated packets into account (in bits/s). These are the packets matched by the aggregation rule that have to be sent to the remote switch which is additional overhead for the infrastructure. These coefficients are calculated as follows:

$$w_{d,t}^{\text{Link}} := \begin{cases} \sum_{f \in F_{d,t}} \delta_{f,t} & , \text{ if } \text{prio}_{\text{agg}}^{\text{highest}} \\ \sum_{f \in F_{d,t}} \delta_{f,t} * \lambda_{f,t}^i & , \text{ if } \text{prio}_{\text{agg}}^{\text{lowest}} \end{cases} \quad (9.8)$$

Similar to  $\text{relocated}_{d,t}$  in Eq. (9.4), the link overhead cost coefficients also depend on the aggregation priority. If aggregation rules are installed with highest priority ( $\text{prio}_{\text{agg}}^{\text{highest}}$ ), all flow rules in the conflict-free cover set ( $F_{d,t}$ ) are relocated. The link overhead in this case is  $\sum_{f \in F_{d,t}} \delta_{f,t}$  where  $\delta_{f,t}$  represents the bits/s processed by flow rule  $f$  in time slot  $t$ . If aggregation rules are installed with lowest priority ( $\text{prio}_{\text{agg}}^{\text{lowest}}$ ), only new flow rules that were installed after the beginning of time slot  $t$  are relocated. If only one time slot is considered (single period problem), the link overhead for the low aggregation priority case is given as  $\sum_{f \in F_{d,t}} \delta_{f,t} * \lambda_{f,t}^i$  where  $\lambda_{f,t}^i$  selects the new flow rules (this coefficient is set to 1 if the rule was installed in time slot  $t$ ). In the multi period case, however, this cost coefficient cannot be calculated so easily because flow rules from previous time slots could also contribute to the link overhead (this problem will be discussed later).

#### 9.4.4.3 Control Message Overhead

The control message overhead cost coefficients  $w_{d,t}^{\text{Ctrl}}$  consider the amount of control messages necessary if delegation template  $d$  is selected in time slot  $t$ . In case of the single period problem, this is equal to the amount of flow rules relocated to the remote switch (each remote rule requires one control message) together with the control messages for aggregation and backflow rules. The latter is not included because this is covered already by the table overhead cost coefficients. And the former is identical to the utilization coefficients (in the single period case):

$$w_{d,t}^{\text{Ctrl}} := \text{relocated}_{d,t} = \begin{cases} \sum_{f \in F_{d,t}} \lambda_{f,t}^a & , \text{ if } \text{prio}_{\text{agg}}^{\text{highest}} \\ \sum_{f \in F_{d,t}} \lambda_{f,t}^i & , \text{ if } \text{prio}_{\text{agg}}^{\text{lowest}} \end{cases} \quad (9.9)$$

The control message overhead cost coefficients have the same problem as relocated $_{d,t}$  and the link overhead cost coefficients (calculation is more complex for the multi period case).

### 9.4.5 Single Period Problem

Based on the variables introduced in the previous sections, the single period DT-Select problem can be defined as an integer linear program in the following way (single period means only a single time slot is considered):

#### Definition 9.2: Single Period Delegation Template Selection (DT-Select)

**Inputs:** A single switch  $s \in S$ , flow table capacity  $c_s^{\text{Table}}$ , flow table utilization  $u_{s,t}^{\text{Table}}$ , a set of delegation templates  $D_{s,t}$ , utilization coefficients  $u_{d,t}^{\text{Table}}$  and overhead coefficients  $w_{d,t}$  for each  $d \in D_{s,t}$

**Output:** Set of selected delegation templates  $D_{s,t}^* \subseteq D_{s,t}$  for time slot  $t$

**Problem Formulation:**

$$\min_{X_{d,t}} \sum_{d \in D_{s,t}} X_{d,t} * w_{d,t} \quad (9.10)$$

$$\text{s.t. } u_{s,t}^{\text{Table}} - \sum_{d \in D_{s,t}} X_{d,t} * u_{d,t}^{\text{Table}} \leq c_s^{\text{Table}} \quad (9.11)$$

$$X_{d,t} \in \{0, 1\} \quad \forall_{d \in D_{s,t}} \quad (9.12)$$

This is a well-defined 0-1 knapsack problem and solving the above ILP will result in an optimal solution for DT-Select. The knapsack constraint in Eq. (9.11) ensures that the flow table capacity is not exceeded while the objective function in Eq. (9.10) ensures that the total cost (total overhead) is minimized. Eq. (9.12) ensures that each template can either be included in the knapsack or not.

The objective function uses a placeholder variable  $w_{d,t}$  to represent a mixture of the cost coefficients, i.e., this is a multi-objective formulation. This mixture consists of three parts explained in detail in Sec. 9.4.4 (the non-negative  $\omega_{\text{DTS}}$ -weights balance the different cost coefficients against each other):

$$w_{d,t} := \omega_{\text{DTS}}^{\text{Table}} w_{d,t}^{\text{Table}} + \omega_{\text{DTS}}^{\text{Ctrl}} w_{d,t}^{\text{Ctrl}} + \omega_{\text{DTS}}^{\text{Link}} w_{d,t}^{\text{Link}} \quad (9.13)$$

The optimal selection  $D_{s,t}^* \subseteq D_{s,t}$  – referred to as the set of selected delegation templates – is calculated as follows ( $X_{d,t}^*$  represent the optimal decision variables after the problem is solved):

$$D_{s,t}^* := \{d \in D_{s,t} \mid X_{d,t}^* = 1\} \tag{9.14}$$

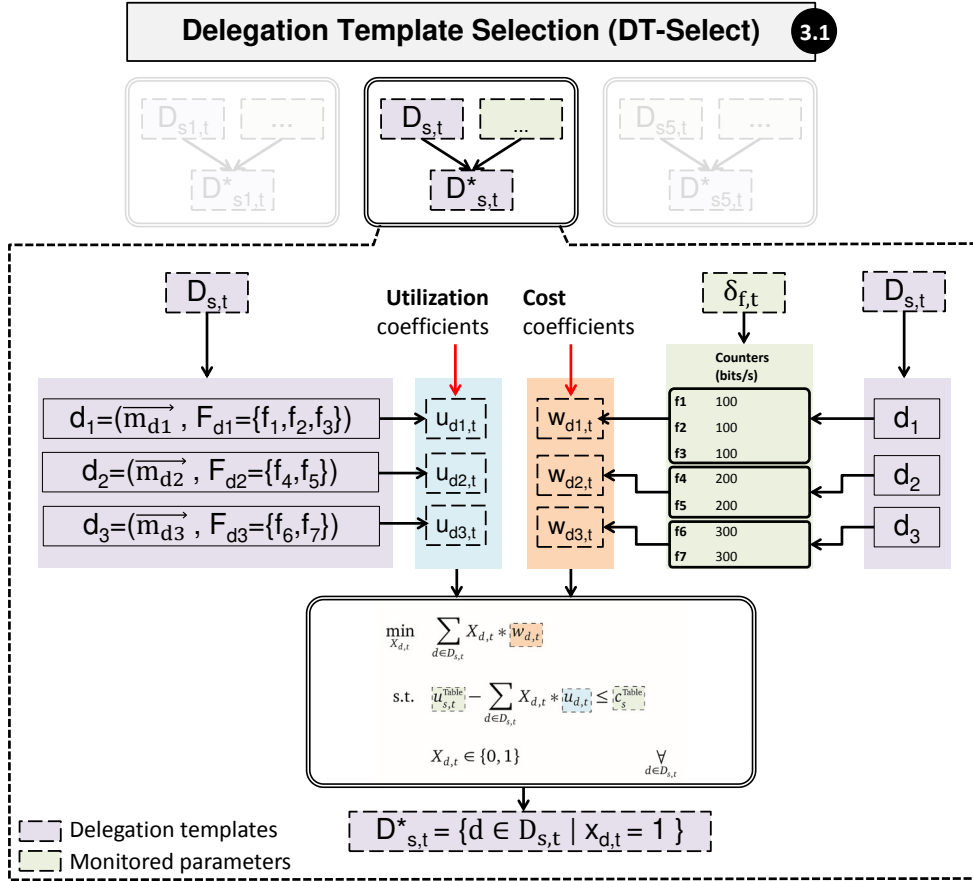


Figure 9.3: Single Period Delegation Template Selection

Fig. 9.3 visualizes the single period DT-Select problem and shows how the different coefficients are calculated. The utilization coefficients  $u_{d,t}^{\text{Table}}$  are derived from the delegation templates, more precisely, from the conflict-free cover set  $F_{d,t}$  within each template according to Eq. (9.3) - (9.5). The overhead coefficients  $w_{d,t}$  are also derived from  $F_{d,t}$ . In case of the link overhead cost coefficients, additional information about the processed bits ( $\delta_{f,t}$ ) is utilized as shown in Eq. (9.8).

Note that the 0-1 knapsack problem is known to be NP-hard [Kar72; Pap81]. However, there is a polynomial-time approximation scheme available [GJ78]. So at least the single

period problem can be solved easily. This changes if multiple time slots have to be considered which is discussed in the next section.

### 9.4.6 Multi Period Problem

The following updates the single period problem from Sec. 9.4.5 to a multi period problem. Only difference is that not a single time slot but a set of consecutive time slots  $T$  is considered. So the problem is to find optimal selections  $D_{s,t}^* \subseteq D_{s,t}$  for each  $t \in T$  so that the flow table utilization  $u_{s,t}^{\text{Table}}$  in each time slot  $t \in T$  is below  $c_s^{\text{Table}}$  while the overhead for flow delegation is minimized. This extended problem is defined as follows:

#### Definition 9.3: Multi Period Delegation Template Selection

**Inputs:** A single switch  $s \in S$ , flow table capacity  $c_s^{\text{Table}}$ , set of consecutive time slots  $T$ , flow table utilization  $u_{s,t}^{\text{Table}}$  for each time slot, sets of delegation templates  $D_{s,t}$  for each  $t \in T$ , utilization coefficients  $u_{d,t}^{\text{Table}}$  and overhead coefficients  $w_{d,t}$  for  $t \in T$ ,  $d \in D_{s,t}$

**Output:** Set of selected delegation templates  $D_{s,t}^*$  for each  $t \in T$

**Problem Formulation:**

$$\min_{X_{d,t}} \sum_{d \in D_{s,t}} \sum_{t \in T} X_{d,t} * w_{d,t} \quad (9.15)$$

$$\text{s.t. } u_{s,t}^{\text{Table}} - \sum_{d \in D_{s,t}} X_{d,t} * u_{d,t}^{\text{Table}} \leq c_s^{\text{Table}} \quad \forall_{t \in T} \quad (9.16)$$

$$X_{d,t} \in \{0, 1\} \quad \forall_{d \in D_{s,t}} \quad \forall_{t \in T} \quad (9.17)$$

At first glance, this is still a 0-1 knapsack problem. The objective function in Eq. 9.15 ensures that the cost is minimized for the specified set of time slots. And the knapsack constraints in Eq. 9.16 are identical to the knapsack constraint in the single period problem except that there are  $|T|$  of them (the constraint has to be fulfilled for each individual time slot).

However, the 0-1 knapsack problem requires that the items in the knapsack can be added and removed independently from each other. This means the utilization coefficients ( $u_{d,t}^{\text{Table}}$ ) and cost coefficients ( $w_{d,t}$ ) can be computed independently from the value of the decision variables. Unfortunately, it is shown in the next section that this is not possible.

### 9.4.7 Problem Analysis for DT-Select

Main challenge with respect to the multi period formulation of the DT-Select problem are the coefficients for cost and utilization. It was already mentioned this is more complex if multiple time slots are considered and the aggregation priority is set to low ( $\text{prio}_{\text{agg}}^{\text{lowest}}$ ). The reason is the coefficients cannot be calculated independently in this case, i.e., the utilization and the costs for the current time slot depend on decisions from previous time slots. This however, cannot be modeled as a linear problem. This is illustrated in Fig. 9.4 using the utilization coefficients as an example.

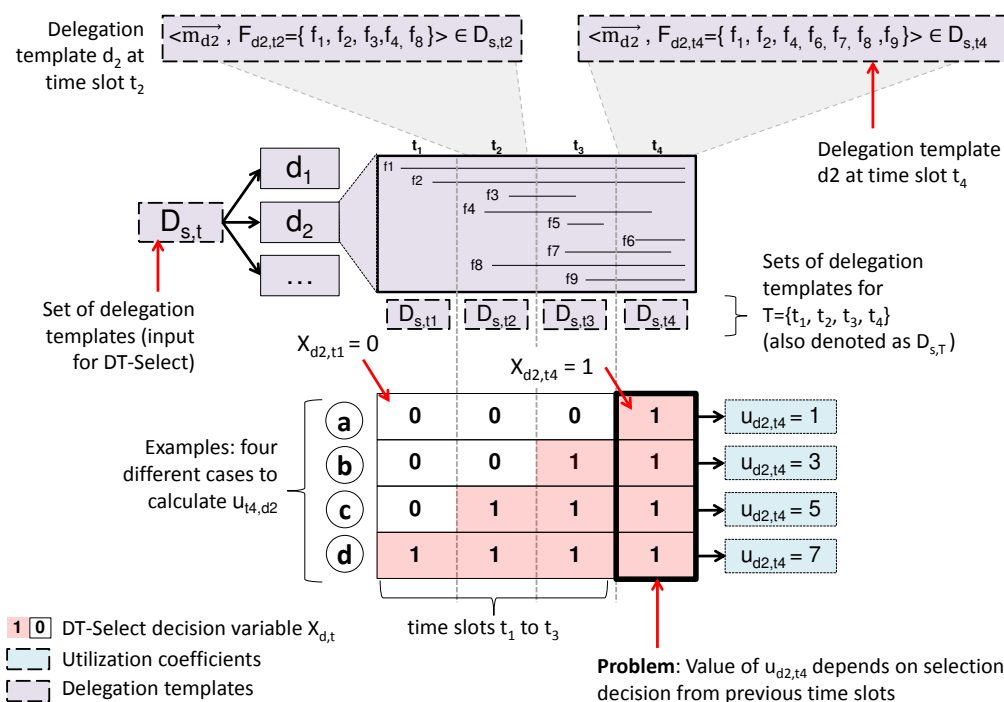


Figure 9.4: Problem DT-Select

The example shows a multi period scenario with four time slots. The large purple block in the middle depicts a set of delegation templates  $D_{s,t}$  where template  $d_2$  is highlighted. Flow rule  $f_1$  to  $f_9$  represent nine flow rules from the conflict-free cover set ( $F_{d_2,t}$ ) of  $d_2$ . The lines in the large purple block indicate the time when the flow rules in  $F_{d_2,t}$  are installed and active. Flow rule  $f_1$ , for example, is installed in time slot  $t_1$  (i.e.,  $\lambda_{f_1,t_1}^i = 1$  in lambda notation) and active in all four time slots ( $\lambda_{f_1,t}^a = 1, t = t_1, \dots, t_4$ ).

The bottom of the figure (the 4x4 grid with 0s and 1s) represent decision variables for DT-Select with respect to delegation template  $d_2$  and the four time slots. The first row of the 4x4 grid labeled as (a) represents one possible outcome of the DT-select problem for this delegation template, i.e., the template was only selected in time slot  $t_4$  and not

selected in all the other time slots. In this case, the utilization coefficient  $u_{d,t}^{\text{Table}}$  for time slot  $t_4$  can be calculated as follows (the overhead part from Eq. (9.3) is ignored here for simplicity):

$$\text{Case (a) in Fig. 9.4: } u_{d_2,t_4} = \begin{cases} \sum_{f \in F_{d_2,t_4}} \lambda_{f,t_4}^a = 7 & , \text{ if } \text{prio}_{\text{agg}}^{\text{highest}} \\ \sum_{f \in F_{d_2,t_4}} \lambda_{f,t_4}^i = 1 & , \text{ if } \text{prio}_{\text{agg}}^{\text{lowest}} \end{cases}$$

It is easy to see that the conflict-free cover set for template  $d_2$  and time slot  $t_4$  consists of seven rules (shown as  $F_{d_2,t_4}$  in the top right of the figure), so the utilization for the high priority case is calculated as  $\sum_{f \in F_{d_2,t_4}} \lambda_{f,t_4}^a = |F_{d_2,t_4}| = 7$ , which makes sense because all seven rules in the cover set will be relocated to the remote switch and the flow table utilization of the delegation switch is reduced by 7. In the low priority case ( $\text{prio}_{\text{agg}}^{\text{lowest}}$ ), only new flow rules installed after the beginning of time slot  $t_4$  are relocated. This is modeled as  $\sum_{f \in F_{d_2,t_4}} \lambda_{f,t_4}^i$  which calculates to 1 here because only one flow rule ( $f_6$ ) is installed in time slot  $t_4$ .

Now consider case (b) in the second row of the grid. In this case, delegation template  $d_2$  was already selected in time slot  $t_3$  (i.e.,  $X_{d_2,t_3} = 1$ ). The utilization coefficient for  $u_{d_2,t_4}$  (the same coefficient as in case (a)) is then calculated as:

$$\text{Case (b) in Fig. 9.4: } u_{d_2,t_4} = \begin{cases} \sum_{f \in F_{d_2,t_4}} \lambda_{f,t_4}^a = 7 & , \text{ if } \text{prio}_{\text{agg}}^{\text{highest}} \\ \sum_{f \in F_{d_2,t_4}} \lambda_{f,t_4}^i + \sum_{f \in F_{d_2,t_3}} \lambda_{f,t_3}^i \lambda_{f,t_4}^a = 3 & , \text{ if } \text{prio}_{\text{agg}}^{\text{lowest}} \end{cases}$$

The high priority case ( $\text{prio}_{\text{agg}}^{\text{highest}}$ ) is obviously identical to before because all flow rules are relocated. The low priority case, however, is different. The first part ( $\sum_{f \in F_{d_2,t_4}} \lambda_{f,t_4}^i$ ) is identical to case (a) because flow rule  $f_6$  is still relocated. However, there are also three flow rules installed in time slot  $t_3$  (the previous time slot) and two of them are still active in time slot  $t_4$ . Note that flow rule  $f_5$  is installed and removed in  $t_3$  and is not active – and thus not relevant – in time slot  $t_4$ . The flow rules that are still active ( $f_7$  and  $f_9$ ) must obviously be included in the utilization coefficient  $u_{d_2,t_4}$ . One way to model this is  $\sum_{f \in F_{d_2,t_3}} \lambda_{f,t_3}^i \lambda_{f,t_4}^a$ . This will select all flow rules installed in time slot  $t_3$  ( $\lambda_{f,t_3}^i$ ) that are still active in time slot  $t_4$  ( $\lambda_{f,t_4}^a$ ).

This pattern continues. In case (c), delegation template  $d_2$  is selected in three time slots ( $t_2$  to  $t_4$ ). There are two flow rules installed in time slot  $t_2$  that are still active in time



slot  $t_4$  which are  $f_4$  and  $f_8$ . These have to be added to the utilization coefficient (the high priority case is not shown any more, is identical in all four cases):

$$\text{Case } \textcircled{c} \text{ in Fig. 9.4: } u_{d_2, t_4} = \sum_{f \in F_{d_2, t_4}} \lambda_{f, t_4}^i + \sum_{f \in F_{d_2, t_3}} \lambda_{f, t_3}^i \lambda_{f, t_4}^a + \sum_{f \in F_{d_2, t_2}} \lambda_{f, t_2}^i \lambda_{f, t_4}^a = 5$$

In general: to calculate the utilization for delegation template  $d$  in time slot  $t$ , it is not enough to look at the situation at  $t$ . Assume  $T := \{t_1, t_2, \dots, t_{-2}, t_{-1}, t\}$ .  $t_{-1}$  represents the time slot before  $t$ ,  $t_{-2}$  the time slot before  $t_{-1}$  and so on.  $t_1$  is defined as the first time slot in  $T$ . If  $d$  is selected in time slot  $t_{-1}$ , the utilization coefficient has to consider all flow rules that were installed at  $t_{-1}$  and are still active at time slot  $t$ . And if  $d$  is also selected in time slot  $t_{-2}$ , the utilization coefficient has to consider all flow rules that were installed at  $t_{-2}$  and are still active at time slot  $t$ . And so on. This continues for all previous time slots where  $d$  was selected. This can be modeled with the DT-Select decision variables as follows:

$$u_{d,t}^{\text{Table}} := \left( \sum_{f \in F_{d,t}} \lambda_{f,t}^i X_{d,t} \right) + \left( \sum_{q=t_1}^{t_1} \sum_{f \in F_{d,q}} \lambda_{f,q}^i * \lambda_{f,t}^a * \prod_{v=q}^t X_{d,v} \right) \quad (9.18)$$

The first (linear) part of this formula counts the flow rules installed in time slot  $t$  if template  $d$  is selected. The second part keeps track of flow rules installed in previous time slots.  $q$  and  $v$  are iterator variables representing time slots. The easiest way to see that this formula properly models the utilization is to unroll the outer sum ( $\sum_{q=t_1}^{t_1}$ ):

$$\begin{aligned} u_{d,t}^{\text{Table}} &:= \left( \sum_{f \in F_{d,t}} \lambda_{f,t}^i X_{d,t} \right) + \left( \sum_{q=t_1}^{t_1} \sum_{f \in F_{d,q}} \lambda_{f,q}^i * \lambda_{f,t}^a * \prod_{v=q}^t X_{d,v} \right) \\ &= \left( \sum_{f \in F_{d,t}} \lambda_{f,t}^i X_{d,t} \right) \\ &\quad + \sum_{f \in F_{d,t_{-1}}} \lambda_{f,t_{-1}}^i * \lambda_{f,t}^a * (X_{d,t_{-1}} X_{d,t}) \quad // q = t_{-1} \\ &\quad + \sum_{f \in F_{d,t_{-2}}} \lambda_{f,t_{-2}}^i * \lambda_{f,t}^a * (X_{d,t_{-2}} X_{d,t_{-1}} X_{d,t}) \quad // q = t_{-2} \\ &\quad + \sum_{f \in F_{d,t_{-3}}} \lambda_{f,t_{-3}}^i * \lambda_{f,t}^a * (X_{d,t_{-3}} X_{d,t_{-2}} X_{d,t_{-1}} X_{d,t}) \quad // q = t_{-3} \\ &\quad + \dots \quad // q = \dots \\ &\quad + \sum_{f \in F_{d,t_1}} \lambda_{f,t_1}^i * \lambda_{f,t}^a * (X_{d,t_1} * \dots * X_{d,t_{-2}} X_{d,t_{-1}} X_{d,t}) \quad // q = t_1 \end{aligned}$$

It is easy to see that this maps to the example discussed above.  $\prod_{v=q}^t X_{d,v}$  will only evaluate to 1 if all decision variables in the product are set to 1. This ensures that not all previous time slots are considered but only those where the delegation template is selected continuously (for all time slots up to  $t$ ).

This forms a polynomial of degree  $|T|$  which requires non-linear programming to be solved – which is not in line with the requirements for the delegation algorithm. Instead, chapter see 9.16 introduces a modified approach for DT-Select that makes use of Eq. 9.18 in a different way (pre calculation of coefficients for a fixed assignments of the decision variables).

## 9.5 Remote Switch Allocation (RS-Alloc)

Remote Switch Allocation (RS-Alloc) is the second sub-problem of the delegation algorithm. It takes the output from DT-Select – which is a set of selected delegation templates  $D_{s,t}^*$  for all switches  $s \in S$  that suffer from a flow table capacity bottleneck – and allocates a remote switch for each selected delegation template based on the available free flow table capacity and link bandwidth in the network.

The discussion of RS-Alloc is structured in the same way as the previous section. First, Sec. 9.5.1 explains the general modeling approach (and introduces some additional terminology). Afterwards, the decision variables and the required coefficients are explained (Sec. 9.5.2 to Sec. 9.5.4), followed by the definition of the single period RS-Alloc problem in Sec. 9.5.5, and its multi period extension in Sec. 9.5.6. Finally, the main problem with the multi period RS-Alloc problem is discussed in Sec. 9.5.7.

### 9.5.1 Modeling Approach and Additional Terminology

The basic idea is to model RS-Alloc as a capacitated facility location problem. This well-known class of optimization problems has customers, each with an individual service demand that has to be serviced by one facility. Optimization goal is to decide which service demand is fulfilled by which facility so that the cost for servicing is minimized and the total amount of service a facility can provide is limited by its maximum capacity. This is a pretty good match to the RS-Alloc problem. Think of each delegation template as a customer while the remote switches (the switches with free capacity) are the facilities. A remote switch (facility) with spare flow table capacity (capacity of the facility) has to be allocated for each delegation template (customer).

There is, however, one important difference that has to be taken into account: the classical facility location problem models both, location and allocation decisions, i.e., there are two

different decision variables. The allocation decision is responsible for mapping customers to facilities. This has a direct counterpart in the RS-Alloc problem (allocating remote switches). The location decision, on the other hand, decides whether a facility at a specific location is open or not. In the case of RS-Alloc, this is not necessarily an active decision. One remote switch might be available (“open”) for one delegation switch but unavailable for another due to the topology of the network (recall that flow delegation will only consider remote switches in a 1-hop neighborhood). This is a conceptual difference in the sense that the location problem can be modeled as a constraint rather than a decision. This is possible because there is no explicit activation or setup cost for the remote switches (different for other problems where it is costly to open a new facility).

Before discussing this in more detail, the following two sub-sections define additional terminology for RS-Alloc. This is required to reduce the complexity in the problem formulation (e.g., avoid iterating over sets of selected delegation templates).

#### 9.5.1.1 Allocation Job

RS-Alloc includes all switches with a bottleneck, i.e., there are multiple sets of selected delegation templates  $D_{s,t}^*$  that have to be included in the problem formulation. To make this easier, allocation jobs are introduced as follows:

##### Definition 9.4: Allocation Job

An **allocation job**  $j \in J_t$  represents one single delegation template  $d \in D_{s,t}^*$ . This is a simple 1:1 mapping, except that the set  $J_t$  is not restricted to a single switch but contains all delegation templates of all bottlenecked switches:

$$J_t := \bigcup_{s \in S} D_{s,t}^* = \{ \dots, d_{j,t}, \dots \} \quad (9.19)$$

The following new variables are defined for working with allocation jobs:

- $d_{j,t}$  represents the delegation template from the 1:1 mapping, i.e., one of the elements in  $J_t$
- $s_j$  represents the delegation switch associated with  $j$
- $\vec{m}_j$  represents the aggregation match from  $d_{j,t}$
- $F_{j,t}$  represents the conflict-free cover set from  $d_{j,t}$ , i.e., the flow rules that have to be relocated to the remote switch

Fig. 9.5 explains this definition. There are five switches with bottlenecks in the example and DT-Select has calculated five sets of selected delegation templates  $D_{s_1,t}^*$  to  $D_{s_5,t}^*$  for

the given time slot  $t$  (shown in the top). The set of all allocation jobs  $J_t$  is defined as the union of these five sets. The union operator is visualized here with the black arrows that select every single delegation template from all five sets (34 allocation jobs in total in the example). The rest of the figure shows how the new helper variables from Def. 9.4 are used. Each variable represents a specific information associated with one single allocation job.  $s_j$  represents the delegation switch, i.e.,  $s_1$  for all allocation jobs created from  $D_{s_1,t}^*$ ,  $s_2$  for all allocation jobs created from  $D_{s_2,t}^*$ , and so on.  $d_{j,t}$  represents the delegation template itself, together with  $\vec{m}_j$  and  $F_{j,t}$ . The remote set  $R_{j,t}$  is defined in the next section. It basically contains all remote switches that could be allocated to this allocation job (in this time slot).

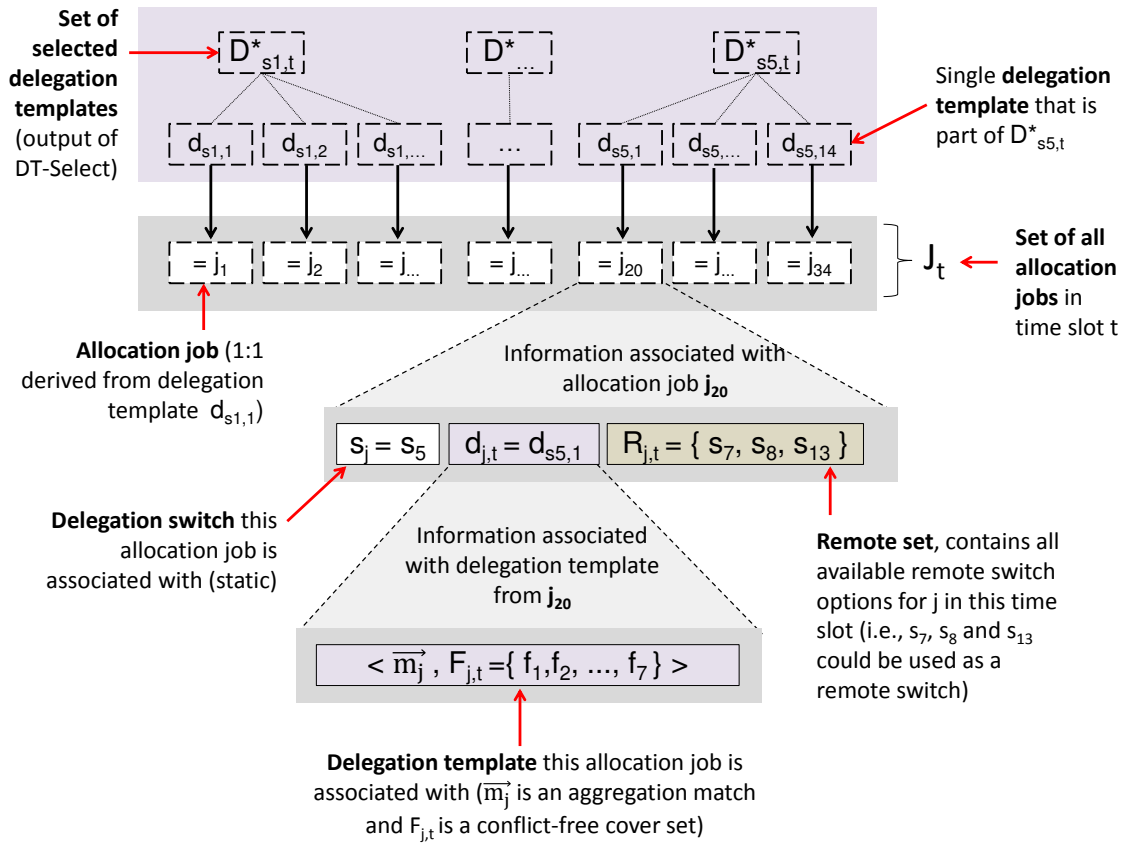


Figure 9.5: Allocation jobs and additional terminology

## 9.5.1.2 Remote Set

One important helper construct is the so-called remote set. This set defines the switches that can be used as a remote switch for one specific allocation job. More precisely: it contains all potential remote switches for allocation job  $j$  in time slot  $t$ .

**Definition 9.5: Remote Set**

A **remote set**  $R_{j,t} \subset S$  is a set of possible remote switches for allocation job  $j$ . It only includes a remote switch  $r$  when i) there is a link between  $s_j$  and  $r$ , ii)  $r$  has enough free flow table capacity at time slot  $t$  to store the flow rules associated with allocation job  $j$  and iii) the link between  $s_j$  and  $r$  has enough free bandwidth at time slot  $t$  to handle the redirected traffic (in both directions). These three conditions are modeled as follows:

$$R_{j,t} := \{s_B\} \cup \bigcup_{r \in S} \left\{ \begin{array}{l} \gamma_{s_j \rightarrow r} = \gamma_{r \rightarrow s_j} = 1 \text{ and} \\ c_r^{\text{Table}} - u_{r,t}^{\text{Table}} \geq u_{j,t}^{\text{Table}} \text{ and} \\ c_{r \rightarrow s_j}^{\text{Link}} - u_{r \rightarrow s_j,t}^{\text{Link}} \geq u_{j,t}^{\text{Link}} \text{ and} \\ c_{s_j \rightarrow r}^{\text{Link}} - u_{s_j \rightarrow r,t}^{\text{Link}} \geq u_{j,t}^{\text{Link}} \end{array} \right. \quad (9.20)$$

$s_B$  represents a backup switch with unlimited capacity and no restrictions but disproportionately high cost. This is required to ensure that the problem remains feasible if there is no optimal solution because of insufficient spare capacity.

$s_j$  is the delegation switch associated with the allocation job (static).  $r$  is the index variable that iterates over all possible remote switches.  $\gamma_{s_j \rightarrow r}$  is a monitored parameter set to 1 if a link exists between switch  $s_j$  and  $r$ .  $u_{j,t}^{\text{Table}}$  and  $u_{j,t}^{\text{Link}}$  represent the “utilization demand” of job  $j$  with respect to flow table capacity and link bandwidth – these variables are defined similar to the utilization coefficients in DT-Select.  $c_r^{\text{Table}} - u_{r,t}^{\text{Table}}$  is the free flow table capacity of switch  $r$ .  $c_{s_j \rightarrow r}^{\text{Link}} - u_{s_j \rightarrow r,t}^{\text{Link}}$  and  $c_{r \rightarrow s_j}^{\text{Link}} - u_{r \rightarrow s_j,t}^{\text{Link}}$  represent the available free link bandwidth between switch  $s_j$  and  $r$  (in both directions).

If one of the conditions from Eq. (9.20) is not fulfilled, switch  $r$  cannot be used as a remote switch in time slot  $t$  and will thus not be included in the remote set. Please note that the backup switch ( $s_B$ ) is always included in the remote set and this switch is never constrained by flow table or link capacities. If an allocation job is allocated to the backup switch, the problem without the backup switch is infeasible. However, this might only affect a small portion of the allocation jobs which is why the backup switch concept is used here (otherwise, the solver would return infeasible and all allocations – which can still be valuable – are lost).

### 9.5.2 Decision Variables

Decision variables in RS-Alloc have to allocate a remote switch to each allocation job. Each job has to be allocated to exactly one remote switch  $r \in R_{j,t}$ . The remote set  $R_{j,t}$  is used here because this set contains all possible allocation options (switches not in the remote set cannot be used because they are not connected to the delegation switch or they cannot provide the necessary resources).

#### Definition 9.6: Decision Variables for RS-Alloc

Remote switch allocation for allocation job  $j$  in time slot  $t$  is based on the following **decision variables**:

$$Y_{j \rightarrow r,t} := \begin{cases} 1, & \text{switch } r \in R_{j,t} \text{ is allocated to job } j \text{ in time slot } t \\ 0, & \text{switch } r \in R_{j,t} \text{ is not allocated to job } j \text{ in time slot } t \end{cases} \quad (9.21)$$

These decision variables are defined for each allocation job  $j \in J_t$  in time slot  $t$  and all remote switch options in the remote set  $R_{j,t}$ . Undefined variables are interpreted as 0. The arrow symbol ( $\rightarrow$ ) is used to emphasize the allocation process:  $j \rightarrow r$  is read as “ $r$  is allocated to  $j$ ” or “ $j$  is using remote switch  $r$ ”.

Recall that each allocation job represents one delegation template  $d_{j,t}$  and each delegation template belongs to one delegation switch  $s_j$ .  $Y_{j \rightarrow r,t} = 0$  says switch  $r$  is not used for this allocation job.  $Y_{j \rightarrow r,t} = 1$  says switch  $r$  is used. The latter means flow rules from  $F_{j,t}$  (the conflict-free cover set of  $d_{j,t}$ ) are relocated to remote switch  $r$  and all packets matched by aggregation match  $\vec{m}_j$  are forwarded from  $s_j$  to  $r$  (and back from  $r$  to  $s_j$  after they were processed). This allocation will “demand” two kinds of resources: flow table capacity of the remote switch  $r$  and bandwidth of the link between  $s_j$  and  $r$ . It is discussed in the next section how this utilization demand can be calculated.

### 9.5.3 Utilization Coefficients

The utilization coefficients  $u_{j,t}$  represent the “to be fulfilled utilization demand”<sup>3</sup> of allocation job  $j$  in time slot  $t$ . The term utilization demand fits well here because the allocation problem has to find remote switches so that i) the switch provides enough free flow table capacity for all relocated flow rules from delegation template  $d_{j,t}$  and ii) the link to the remote switch has enough free bandwidth for all packets that are relocated.

<sup>3</sup>The same notation as in DT-Select is used here (instead of a new “demand” notation) to be consistent with the terminology from the DT-Select problem and to avoid ambiguity with the notation of delegation templates (which use variable  $d$ )

Both can be interpreted as a demand that has to be fulfilled. Given the RS-Alloc decision variables, this can be modeled with three simple constraints (for one remote switch  $r$ ):

$$u_{r,t}^{\text{Table}} + \sum_{j \in J_t} u_{j,t}^{\text{Table}} Y_{j \rightarrow r,t} \leq c_r^{\text{Table}} \quad (9.22)$$

$$u_{s_j \rightarrow r,t}^{\text{Link}} + \sum_{j \in J_t} u_{j,t}^{\text{Link}} Y_{j \rightarrow r,t} \leq c_{s_j \rightarrow r}^{\text{Link}} \quad (9.23)$$

$$u_{r \rightarrow s_j,t}^{\text{Link}} + \sum_{j \in J_t} u_{j,t}^{\text{Link}} Y_{j \rightarrow r,t} \leq c_{r \rightarrow s_j}^{\text{Link}} \quad (9.24)$$

Set  $J_t$  contains all allocation jobs for time slot  $t$ . Decision variable  $Y_{j \rightarrow r,t}$  is only set to 1 if remote switch  $r$  is allocated to job  $j$  in time slot  $t$ . So the sum in all three equations will only consider those jobs that have remote switch  $r$  allocated to them.

Eq. (9.22) now ensures that the normal utilization of the remote switch ( $u_{r,t}^{\text{Table}}$ ) together with all relocated flow rules ( $\sum u_{j,t}^{\text{Table}}$  for all jobs that are allocated to  $r$ ) is below the capacity of the remote switch ( $c_r^{\text{Table}}$ ). The two other equations follow the exact same logic and ensure that there is enough free link bandwidth. Eq. (9.23) covers the relocated packets sent from delegation switch  $s_j$  to remote switch  $r$  and Eq. (9.24) covers the return path from remote to delegation switch.

It can be seen that two utilization coefficients are required, one for flow table utilization ( $u_{j,t}^{\text{Table}}$ ) and one for link bandwidth ( $u_{j,t}^{\text{Link}}$ ). These can be calculated in a similar way as the utilization coefficients for DT-Select (see Sec. 9.4.3). The first coefficient,  $u_{j,t}^{\text{Table}}$ , depends on the selected aggregation priority. If aggregation rules are installed with highest priority ( $\text{prio}_{\text{agg}}^{\text{highest}}$ ), all flow rules in the conflict-free cover set of  $d_{j,t}$  are relocated. The cover set is here given as  $F_{j,t}$ . If aggregation rules are installed with lowest priority ( $\text{prio}_{\text{agg}}^{\text{lowest}}$ ), only new flow rules that were installed after the beginning of time slot  $t$  are relocated.

$$u_{j,t}^{\text{Table}} := \begin{cases} \sum_{f \in F_{j,t}} \lambda_{f,t}^a & , \text{ if } \text{prio}_{\text{agg}}^{\text{highest}} \\ \sum_{f \in F_{j,t}} \lambda_{f,t}^i & , \text{ if } \text{prio}_{\text{agg}}^{\text{lowest}} \end{cases} \quad (9.25)$$

And the second coefficient,  $u_{j,t}^{\text{Link}}$ , is defined as the number of bits relocated from the delegation switch to the remote switch (and vice versa). This is also calculated from the conflict-free cover set  $F_{j,t}$ :

$$u_{j,t}^{\text{Link}} = \sum_{f \in F_{j,t}} \delta_{f,t} \quad (9.26)$$

Recall that  $\lambda_{f,t}^a$  is only set to 1 if flow rule  $f$  is active in time slot  $t$  and  $\lambda_{f,t}^i$  is only set to 1 if flow rule  $f$  is installed in time slot  $t$ .  $\delta_{f,t}$  represents the number of bits processed by flow rule  $f$  in time slot  $t$ .

### 9.5.4 Cost Coefficients

The cost coefficients  $w_{j \rightarrow r,t}$  represent the costs to be paid for allocating remote switch  $r$  to allocation job  $j$  in time slot  $t$ . This is equivalent to the costs that arise when flow rules of delegation template  $d_{j,t}$  are relocate to remote switch  $r$ . Basic idea is it to assign each allocation job a remote switch in such a way that the overall cost of all allocations is minimized:

$$\text{minimize: } \sum_{j \in J_t} w_{j \rightarrow r,t} \quad (9.27)$$

In the allocation problem, there are four important cost factors to be considered: i) free flow table capacity of the remote switch ( $w_{j \rightarrow r,t}^{\text{Table}}$ ), ii) free link bandwidth between delegation and remote switch ( $w_{j \rightarrow r,t}^{\text{Link}}$ ), iii) amount of control messages required ( $w_{j \rightarrow r,t}^{\text{Ctrl}}$ ), and iv) static costs associated with remote switches ( $w_{r,t}^{\text{Static}}$ ). All four cost factors are defined and explained in the following sections.

#### 9.5.4.1 Flow Table Capacity

This cost factor takes the available free flow table capacity at the remote switch into account. It basically takes the capacity of the remote switch at time slot  $t$  and subtracts the additional rules that would be added by allocation job  $j$  (number of remote rules in  $F_{j,t}$  that are relocated). The idea is that remote switches with higher spare capacity should be preferred.

$$w_{j \rightarrow r,t}^{\text{Table}} := -(c_r^{\text{Table}} - u_{r,t}^{\text{Table}} - u_{j,t}^{\text{Table}}) \quad (9.28)$$

The available flow table capacity of the remote switch in time slot  $t$  is given as  $c_r^{\text{Table}} - u_{r,t}^{\text{Table}}$ . The additional rules are given as  $u_{j,t}^{\text{Table}}$ . And because the allocation problem is modeled as a minimization problem, the cost factor is multiplied by  $-1$ .



### 9.5.4.2 Link Capacity

This cost factor takes the available free link bandwidth between delegation switch  $s_j$  and remote switch  $r$  into account. This link is burdened with all packets matched by the aggregation match  $\vec{m}_j$ . First, the packets are sent from  $s_j$  to remote switch  $r$ , i.e., utilization  $u_{r \rightarrow s_j, t}^{\text{Link}}$  increases. And after processing in the remote switch, the packets are sent back by the remote rules, i.e., utilization  $u_{s_j \rightarrow r, t}^{\text{Link}}$  increases. The idea is to prefer links with higher spare bandwidth.

$$w_{j \rightarrow r, t}^{\text{Link}} := -\left( \min(c_{s_j \rightarrow r}^{\text{Link}} - u_{s_j \rightarrow r, t}^{\text{Link}}, c_{r \rightarrow s_j}^{\text{Link}} - u_{r \rightarrow s_j, t}^{\text{Link}}) - u_{j, t}^{\text{Link}} \right) \quad (9.29)$$

$c_{s_j \rightarrow r}^{\text{Link}} - u_{s_j \rightarrow r, t}^{\text{Link}}$  is the free link bandwidth between  $s_j$  and  $r$  in time slot  $t$  and  $c_{r \rightarrow s_j}^{\text{Link}} - u_{r \rightarrow s_j, t}^{\text{Link}}$  is the free link bandwidth between  $r$  and  $s_j$ . Here, the minimum bandwidth of the two involved links is considered as the effective bandwidth (both directions are burdened with the same amount of additional packets).  $u_{j, t}^{\text{Link}}$  is the amount of bits associated with the allocation job. Multiplication by  $-1$  is required because the allocation problem is modeled as a minimization problem.

### 9.5.4.3 Control Messages

This cost factor considers the amount of control messages necessary if remote switch  $r$  is allocated to allocation job  $j$  in time slot  $t$ . In case of the single period problem (only a single time slot is considered), this is equal to the amount of flow rules that are installed in the remote switch:

$$w_{j \rightarrow r, t}^{\text{Ctrl}} := u_{j, t}^{\text{Table}} \quad (9.30)$$

This seems to be different from the above cost factors because it is independent from  $r$ , i.e., the factor is the same for all remote switches. However, this is only true in the single period case. In the multi period case, the allocation of the remote switch might change over time. Then, it makes a big difference whether consecutive time slots have the same allocation (no additional control messages required) or not (flow rules have to be relocated to another remote switch which results in additional control messages). This is what makes the multi period problem difficult.

### 9.5.4.4 Static Cost

The last cost factor reflects that different switches can have different characteristics, e.g., supported hardware features. It is also possible that different roles for switches exist such

as a top-of-rack switch, core switch or edge switch. To include this into the allocation problem, a static cost factor  $w_{r,t}^{\text{Static}}$  is defined that only depends on the switch. This is used here primarily to distinguish between regular hardware switches (switches in  $S$ ) and the backup switch ( $s_B$ ) that was introduced above in Sec. 9.5.1.2. Because the optimization problem should never use the backup switch (if feasible), this switch is assigned a very high static cost (10000 represents a high value that was found empirically with respect to all the other parameters):

$$w_{r,t}^{\text{Static}} := \begin{cases} 10000, & \text{if } r = s_B \\ 0, & \text{otherwise } (r \in S) \end{cases} \quad (9.31)$$

Note that this example does not depend on time slot  $t$ . In theory, however, it could also be possible to change static values based on the current time (not considered here further). The static cost coefficients can also be used to exclude arbitrary switches from the allocation problem – just as done above for the backup switch – by setting its static cost to a high value. In the remainder of this work, all switches in  $S$  are considered equal with a static cost coefficient of 0.

### 9.5.5 Single Period Problem

Based on the variables introduced in the previous sections, the single period RA-Alloc problem can be defined as an integer linear program in the following way (single period means only a single time slot is considered):

**Definition 9.7: Single Period Remote Switch Allocation Problem (RS-Alloc)**

**Inputs:** Set of switches  $S$ , set of allocation jobs  $J_t$  derived from DT-Select output, flow table capacity  $c_r^{\text{Table}}$  for each switch, flow table utilization  $u_{r,t}^{\text{Table}}$  for each switch, link capacity  $c_{s_j \rightarrow r}^{\text{Link}}$  for each link, utilization coefficients  $u_{j,t}$ , overhead coefficients  $w_{j \rightarrow r,t}$ , and remote sets  $R_{j,t}$  for each  $j \in J_t$

**Output:** A set of selected and allocated delegation templates  $D_t^{**}$  for time slot  $t$

**Problem Formulation:**

$$\min_{Y_{j \rightarrow r,t}} \sum_{j \in J_t} \sum_{r \in R_{j,t}} w_{j \rightarrow r,t} Y_{j \rightarrow r,t} \quad (9.32)$$

$$\text{s.t.} \quad \sum_{r \in R_{j,t}} Y_{j \rightarrow r,t} = 1 \quad \forall_{j \in J_t} \quad (9.33)$$

$$u_{r,t}^{\text{Table}} + \sum_{j \in J_t} u_{j,t}^{\text{Table}} Y_{j \rightarrow r,t} \leq c_r^{\text{Table}} \quad \forall_{r \in S} \quad (\text{see 9.22})$$

$$u_{s_j \rightarrow r,t}^{\text{Link}} + \sum_{j \in J_t} u_{j,t}^{\text{Link}} Y_{j \rightarrow r,t} \leq c_{s_j \rightarrow r}^{\text{Link}} \quad \forall_{r \in S} \quad (\text{see 9.23})$$

$$u_{r \rightarrow s_j,t}^{\text{Link}} + \sum_{j \in J_t} u_{j,t}^{\text{Link}} Y_{j \rightarrow r,t} \leq c_{r \rightarrow s_j}^{\text{Link}} \quad \forall_{r \in S} \quad (\text{see 9.24})$$

$$Y_{j \rightarrow r,t} \in \{0, 1\} \quad \forall_{r \in S} \quad \forall_{j \in J_t} \quad (9.34)$$

This is a well-defined capacitated facility location problem (facility location is modeled as a constraint) and solving the above ILP will result in an optimal solution for the RS-Alloc problem. Eq. (9.33) ensures that each delegation template (customer)  $j \in J$  is served by exactly one remote switch (facility). The utilization constraints were already explained in Sec. 9.5.3. Eq. (9.22) ensures that the flow table capacity of the remote switches is not exceeded. Eq. (9.23) and (9.24) ensures that the link between delegation and remote switch has enough free bandwidth. And Eq. (9.34) enforces binary decision variables.

The objective function uses a placeholder variable  $w_{j \rightarrow r,t}$  to represent a mixture of the cost coefficients, i.e., this is a multi-objective formulation. This mixture consists of four parts explained in detail in Sec. 9.5.4:

$$w_{j \rightarrow r,t} := \omega_{\text{RSA}}^{\text{Table}} w_{j \rightarrow r,t}^{\text{Table}} + \omega_{\text{RSA}}^{\text{Ctrl}} w_{j \rightarrow r,t}^{\text{Ctrl}} + \omega_{\text{RSA}}^{\text{Link}} w_{j \rightarrow r,t}^{\text{Link}} + \omega_{\text{RSA}}^{\text{Static}} w_{r,t}^{\text{Static}} \quad (9.35)$$

The non-negative  $\omega_{RSA}$ -weights balance the different cost coefficients against each other. To this regard, it is important to keep in mind that the values for  $w_{j \rightarrow r, t}^{Table}$  and  $w_{j \rightarrow r, t}^{Link}$ , for example, can be several orders of magnitude apart (if not normalized). The optimal allocation  $D_t^{**}$  – referred to as the set of selected and allocated delegation templates – is calculated as follows (the  $Y_{j \rightarrow r, t}^*$  variables represent the optimal decision variables after the problem is solved):

$$D_t^{**} := \{ \langle d_{j,t}, r \rangle \mid Y_{j \rightarrow r, t}^* = 1 \} \tag{9.36}$$

Each tuple  $\langle d_{j,t}, r \rangle$  contains one delegation template and the allocated remote switch. Note that RS-Alloc assigns remote switches  $r$  to allocation jobs. The output however, contains delegation templates (there is a 1:1 mapping between delegation templates and allocation jobs). This is done because only the RS-Alloc problem works with allocation jobs, all other building blocks work with delegation templates.

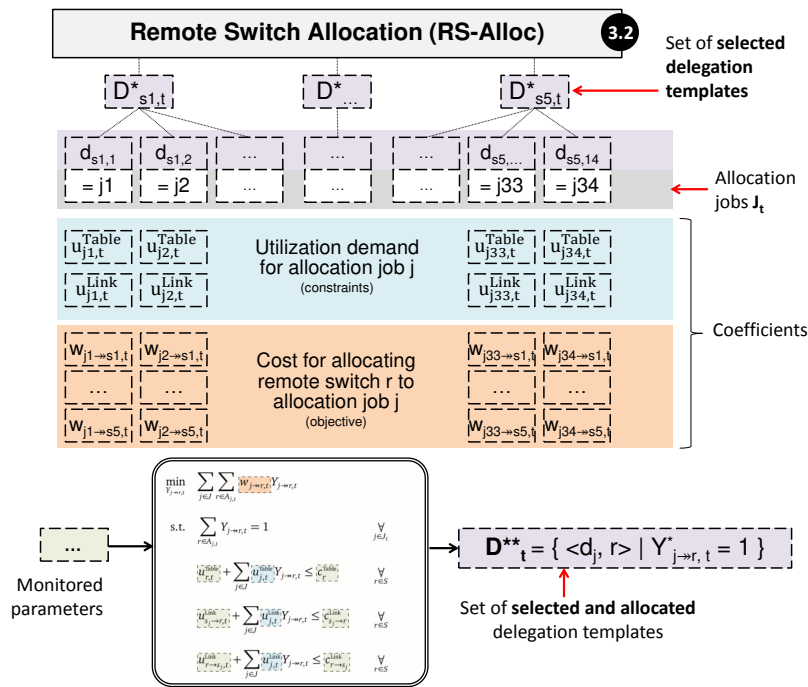


Figure 9.6: Single Period Remote Switch Allocation

Fig. 9.6 visualizes the single period RS-Alloc problem and shows how the different coefficients are used. The utilization coefficients  $u_{j,t}$  are shown in blue. The cost coefficients are shown in orange. Monitored parameters are shown in green.

### 9.5.6 Multi Period Problem

The following updates the single period problem from Sec. 9.5.5 to a multi period problem. The only difference is that the constraints are specified for each time slot and the objective function minimizes over the sum of all time slots.

#### Definition 9.8: Multi Period Remote Switch Allocation Problem

**Inputs:** A set of switches  $S$ , set of consecutive time slots  $T$ , set of allocation jobs  $J_t$  derived from the output of DT-Select for each  $t \in T$ , flow table capacity  $c_r^{\text{Table}}$  for each switch, link capacity  $c_{s_j \rightarrow r}^{\text{Link}}$  for each link, utilization coefficients  $u_{j,t}$ , overhead coefficients  $w_{j \rightarrow r,t}$ , and remote sets  $R_{j,t}$  for each  $j \in J_t$  and each  $t \in T$

**Output:** Sets of selected and allocated delegation templates  $D_t^{**}$  for each  $t \in T$

**Problem Formulation:**

$$\min_{Y_{j \rightarrow r,t}} \sum_{t \in T} \sum_{j \in J_t} \sum_{r \in R_{j,t}} w_{j \rightarrow r,t} Y_{j \rightarrow r,t} \quad (9.37)$$

$$\text{s.t.} \quad \sum_{r \in R_{j,t}} Y_{j \rightarrow r,t} = 1 \quad \forall_{t \in T} \quad \forall_{j \in J_t} \quad (9.38)$$

$$u_{r,t}^{\text{Table}} + \sum_{j \in J_t} u_{j,t}^{\text{Table}} Y_{j \rightarrow r,t} \leq c_r^{\text{Table}} \quad \forall_{t \in T} \quad \forall_{r \in S} \quad (9.39)$$

$$u_{s_j \rightarrow r,t}^{\text{Link}} + \sum_{j \in J_t} u_{j,t}^{\text{Link}} Y_{j \rightarrow r,t} \leq c_{s_j \rightarrow r}^{\text{Link}} \quad \forall_{t \in T} \quad \forall_{r \in S} \quad (9.40)$$

$$u_{r \rightarrow s_j,t}^{\text{Link}} + \sum_{j \in J_t} u_{j,t}^{\text{Link}} Y_{j \rightarrow r,t} \leq c_{r \rightarrow s_j}^{\text{Link}} \quad \forall_{t \in T} \quad \forall_{r \in S} \quad (9.41)$$

$$Y_{j \rightarrow r,t} \in \{0, 1\} \quad \forall_{t \in T} \quad \forall_{r \in S} \quad \forall_{j \in J_t} \quad (9.42)$$

The structure of the multi period problem is very similar to the single period one. In fact, the multi period problem could be decomposed into  $|T|$  individual single period problems that are solved in parallel (one problem instance per time slot), if the cost coefficients  $w_{j \rightarrow r,t}$  could be calculated independently. However, the next section explains why this is not possible.

### 9.5.7 Problem Analysis for RS-Alloc

Main challenge with respect to the multi period RS-Alloc problem is that the allocation can change between time slots which introduces quadratic complexity for  $w_{j \rightarrow r,t}^{\text{Ctrl}}$ . Recall

that this cost coefficient considers the amount of control messages necessary if remote switch  $r$  is allocated to allocation job  $j$  in time slot  $t$  (see Sec. 9.5.4.3).

Assume an allocation job with the same aggregation match ( $\vec{m}_j$ ) and delegation switch ( $s_j$ ) is defined for two time slots  $t_1$  and  $t_2$ . Further assume there are two remote switch options  $r_1$  and  $r_2$  in both time slots ( $R_{j,t_1} = R_{j,t_2} = \{r_1, r_2\}$ ). If remote switch  $r_1$  (or  $r_2$ ) is allocated to  $j$  in both time slots, the amount of flow rules that are installed and removed in the process of flow delegation is minimal and can be (for example) derived from  $d_{j,t_1}$  and  $d_{j,t_2}$ . But what if  $r_1$  is chosen for time slot  $t_1$  and  $r_2$  for  $t_2$ ? In this case, the flow rules are removed from  $r_1$  at the end of the first time slot and installed again in  $r_2$ , which is obviously more expensive in terms of control messages than using the same remote switch allocation for both time slots.

It can be concluded that  $w_{j \rightarrow r, t}^{\text{Ctrl}}$  has to be calculated in such a way that keeping the same remote switch allocation over multiple time slots is rewarded and changing the allocation between two consecutive time slots is penalized. Let us first assume a simple case where the control message cost is set to 0 if the allocation is not changed. This could be modeled as follows:

$$w_{j \rightarrow r, t}^{\text{Ctrl}} := \begin{cases} u_{j, t}^{\text{Table}}, & (1 - Y_{j \rightarrow r, t-1})Y_{j \rightarrow r, t} = 1 \\ 0, & \text{otherwise} \end{cases}$$

$w_{j \rightarrow r, t}^{\text{Ctrl}}$  will now evaluate to the amount of to be installed flow rules if remote switch  $r$  was not allocated to allocation job  $j$  in the previous time slot. In case  $r$  was already allocated, the cost is set to 0. It is easy to see that this requires a quadratic expression in the objective function:

$$\begin{aligned} \min_{Y_{j \rightarrow r, t}} \quad & \sum_{t \in T} \sum_{j \in J_t} \sum_{r \in R_{j, t}} Y_{j \rightarrow r, t} \left( \omega_{\text{RSA}}^{\text{Table}} w_{j \rightarrow r, t}^{\text{Table}} + \omega_{\text{RSA}}^{\text{Link}} w_{j \rightarrow r, t}^{\text{Link}} + \omega_{\text{RSA}}^{\text{Static}} w_{r, t}^{\text{Static}} \right) \quad (9.43) \\ & + \sum_{t \in T} \sum_{j \in J_t} \sum_{r \in R_{j, t}} (1 - Y_{j \rightarrow r, t-1}) Y_{j \rightarrow r, t} \omega_{\text{RSA}}^{\text{Ctrl}} w_{j \rightarrow r, t}^{\text{Ctrl}} \end{aligned}$$

Note that the other cost coefficients introduced in Sec. 9.5.4 do not suffer from this problem, i.e., these parts remain linear (first line of the equation).

## 9.6 Conclusion

This chapter shows how the delegation algorithm can be realized as a two-step optimization approach and introduces the two relevant sub-problems: DT-Select for delegation template selection and RS-Alloc for remote switch allocation.

- DT-Select is modeled as a 0-1 knapsack problem and calculated individually for each switch. It takes a set of delegation templates  $D_{s,t}$  provided by the rule aggregation scheme and returns a set of selected delegation templates  $D_{s,t}^* \subseteq D_{s,t}$ . DT-Select ensures that i) the utilization of the flow table of switch  $s$  does not exceed its capacity and ii) overhead caused by flow delegation is minimized.
- RS-Alloc is modeled as a capacitated facility location problem and calculated globally, once per time slot. It takes the output from DT-Select and allocates a remote switch for each selected delegation template based on the available free flow table capacity and link bandwidth in the network. It then returns a set of selected and allocated delegation templates  $D_t^{**}$  which are provided to the detour procedure.

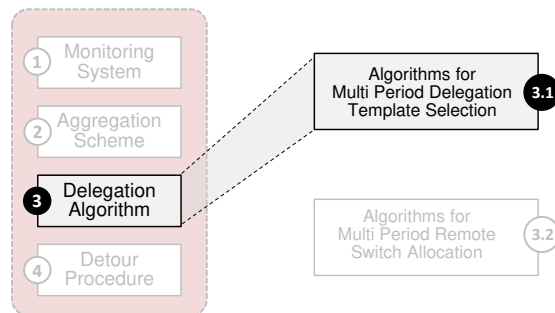
The chapter presents a single period and a multi period problem formulation for both sub-problems. It also explains how the relevant variables – decision variables, utilization coefficients and cost coefficients – are defined and calculated. In addition, the problems of the multi period formulations are discussed. For the multi period DT-Select problem, it is shown that the coefficients introduce non-linear dependencies. And for the multi period RS-Alloc problem it is shown that the control message cost coefficients lead to an objective function with a high number of quadratic terms. These problems are addressed by the multi period algorithms for DT-Select and RS-Alloc in the following two chapters.





# Multi Period Delegation Template Selection

---



This chapter introduces three algorithms for multi period delegation template selection (DT-Select). The first two algorithms (Select-Opt and Select-CopyFirst) use discrete optimization and exploit the flexibility of ILP modeling and the efficiency of modern ILP solvers. This is complemented with a simpler greedy algorithm for comparison.

- **Select-Opt** in Sec. 10.2 uses two key concepts to approach the multi period DT-Select problem. The problem is first linearized by pre-calculating all possible combinations of decisions for a fixed (small) amount of time slots which are then used in a multi-dimensional multiple-choice knapsack formulation. And a periodic optimization approach is introduced to solve the problem periodically with a limited future horizon of  $L$  time slots.
- **Select-CopyFirst** in Sec. 10.3 is a heuristic for Select-Opt that also uses the periodic approach but is based on a two-dimensional knapsack formulation which significantly reduces the problem space. The key idea is to only make a decision for the first time slot (select delegation template or not) and then “copy” this decision for all following time slots.

- **Select-Greedy** in Sec. 10.4 finally introduces a simple greedy heuristic for Select-CopyFirst which is very fast (below 0.25ms in all investigated cases) but less flexible and also less performant.

## 10.1 Problem

The multi period DT-Select problem is defined in Sec. 9.4.6. It can be written as an integer linear program in the following way:

### Recap: Multi Period Delegation Template Selection (see Def. 9.3)

**Inputs:** A single switch  $s \in S$ , flow table capacity  $c_s^{\text{Table}}$ , flow table utilization  $u_{s,t}^{\text{Table}}$ , set of consecutive time slots  $T$ , set of delegation templates  $D_{s,t}$  for each time slot, utilization coefficients  $u_{d,t}$  and overhead coefficients  $w_{d,t}$  for each  $d \in D_{s,t}$  and each time slot  $t \in T$

**Output:** Set of selected delegation templates  $D_{s,t}^* \subseteq D_{s,t}$  for each time slot  $t \in T$

**Problem Formulation:**

$$\min_{X_{d,t}} \sum_{d \in D_{s,t}} \sum_{t \in T} X_{d,t} * w_{d,t} \quad (\text{see 9.15})$$

$$\text{s.t. } u_{s,t}^{\text{Table}} - \sum_{d \in D_{s,t}} X_{d,t} * u_{d,t} \leq c_s^{\text{Table}} \quad \forall_{t \in T} \quad (\text{see 9.16})$$

$$X_{d,t} \in \{0, 1\} \quad \forall_{d \in D_{s,t}} \quad \forall_{t \in T} \quad (\text{see 9.17})$$

The problem associated with the above formulation was discussed in detail in Sec. 9.4.7. In a nutshell: the coefficients cannot be calculated independently from the decisions in different time slots which leads to a non-linear problem. This is true for both, the utilization coefficients  $u_{d,t}^{\text{Table}}$  which are required for the constraints in Eq. (9.16) and the cost coefficients  $w_{d,t}$  which are required in the objective function in Eq. (9.15). The utilization coefficients, for example, are calculated according to the following (non-linear) formula:

$$u_{d,t}^{\text{Table}} := \left( \sum_{f \in F_{d,t}} \lambda_{f,t}^i X_{d,t} \right) + \left( \sum_{q=t-1}^{t_1} \sum_{f \in F_{d,q}} \lambda_{f,q}^i * \lambda_{f,t}^a * \prod_{v=q}^t X_{d,v} \right) \quad (\text{see 9.18})$$

The first (linear) part of this formula counts the flow rules installed in time slot  $t$  if template  $d$  is selected. The second (non-linear) part keeps track of flow rules installed in previous time slots.  $q$  and  $v$  are iterator variables representing time slots.

## 10.2 DT-Select with Assignments (Select-Opt)

This section presents a periodic algorithm for the multi period DT-Select problem called Select-Opt. It is based on the multi-dimensional multiple-choice knapsack problem which is a generalization of the 0-1 knapsack problem. It uses the suffix “Opt” because an optimal result is returned for each individual optimization period (optimality is not guaranteed across different optimization periods).

### 10.2.1 General Idea

Select-Opt addresses the problem outlined in Sec. 10.1 with two concepts: assignment-based decision variables and periodic optimization.

**Assignment-based decision variables** use the fact that coefficients can be pre-calculated for all possible combinations of decisions if the amount of decisions is small ( $< 10$ ). DT-Select makes one decision per time slot  $t \in T$  for each delegation template  $d \in D_{s,t}$  – which is whether template  $d$  is selected in time slot  $t$  or not. Instead of having a non-linear term such as  $u_{d,t}^{\text{Table}} = \dots * \prod_{t=t_1}^{t_m} X_{d,t}$  (a polynomial of degree  $|T|$ ), the  $2^{|T|}$  possible combinations of all  $u_{d,t}^{\text{Table}}$  values are pre-calculated – called assignments in this work – and combined with  $2^{|T|}$  new decision variables that will select exactly one of the pre-calculated assignments. This concept is explained in more detail below.

The use of assignment-based decision variables linearizes the problem but the input size grows exponentially with  $|T|$  which makes it impracticable for larger problems (already 1000 different assignments per delegation template for  $|T| = 10$ ). In practice, however, it is usually not required to solve the problem for a large amount of time slots<sup>1</sup>. Instead, a practical algorithm for DT-Select can make use of **periodic optimization**. With periodic optimization, the problem is solved periodically once per time slot and in each of these optimization periods, a limited future horizon of  $L$  time slots is considered together with the status from the last optimization period. This is also explained in more detail below.

#### 10.2.1.1 Assignment-based Decision Variables

The main idea behind assignments is it to calculate coefficients for all combinations of decisions for a set of fixed time slots  $T = \{t_1, \dots, t_m\}$  and define a new problem based on assignments. Instead of using one decision variable per time slot ( $X_{d,t}$ ), decisions are defined over a set of assignments where each element in the set represents one concrete “assignment” of  $X_{d,t_1}, \dots, X_{d,t_m}$ . The concept becomes clear when looking at an example.

<sup>1</sup>This can be useful in certain cases, though. For example if the problem has to be executed on historical data to generate training input for machine learning.

Assume a single delegation template  $d$  and  $T = \{t_1, t_2\}$ , i.e., the problem has to consider two time slots. In the original problem (Def. 9.3), this case is modeled with two binary decision variables  $X_{d,t_1}$  and  $X_{d,t_2}$ , each of which can either take the value  $\boxed{0}$  or the value  $\boxed{1}$ . The decisions are represented here as small rectangles ( $\boxed{0}$  and  $\boxed{1}$ ) to support readability.  $\boxed{0}$  means the template is not selected in time slot  $t$  and  $\boxed{1}$  means the template is selected. So in total, there are four different “outcomes” when considering two time slots:

$$\begin{aligned}
 X_{d,t_1} = \boxed{0} \text{ and } X_{d,t_2} = \boxed{0} &\Rightarrow \text{case } \boxed{0}\boxed{0} &\Rightarrow \text{assignment } A_1 \\
 X_{d,t_1} = \boxed{0} \text{ and } X_{d,t_2} = \boxed{1} &\Rightarrow \text{case } \boxed{0}\boxed{1} &\Rightarrow \text{assignment } A_2 \\
 X_{d,t_1} = \boxed{1} \text{ and } X_{d,t_2} = \boxed{0} &\Rightarrow \text{case } \boxed{1}\boxed{0} &\Rightarrow \text{assignment } A_3 \\
 X_{d,t_1} = \boxed{1} \text{ and } X_{d,t_2} = \boxed{1} &\Rightarrow \text{case } \boxed{1}\boxed{1} &\Rightarrow \text{assignment } A_4
 \end{aligned}$$

Each of the four different outcomes above represents one assignment. In the first assignment ( $A_1$ ), delegation template  $d$  is not selected in both time slots. The second assignment ( $A_2$ ) represents the case where the template is selected in the second time slot only. The two remaining assignments ( $A_3$  and  $A_4$ ) represent the cases where the template is only selected in the first time slot and in both time slots. In general, there are  $2^m$  different assignments with respect to whether a delegation template  $d$  is selected or not when  $m$  time slots are considered. A formal definition of assignments and assignment sets is given below.

#### Definition 10.1: Assignment Set

An **assignment set**  $A_{d,T}$  for a single delegation template  $d \in D_s$  and a set of consecutive time slots  $T = \{t_1, \dots, t_m\}$  is defined as:

$$A_{d,T} := \left\{ A_i = \langle a_1, \dots, a_m \rangle \mid a_t = i_2[t], i = 1, \dots, 2^m \right\} \quad (10.1)$$

$i$  is a counting variable and  $i_2[t]$  represents the  $t$ -th position of the bit-representation of the counting variable. So the set  $A_{d,T}$  has  $2^m$  entries and each entry  $A_i$  is called an **assignment**. Each assignment consists of  $m$  values  $a_t \in \{\boxed{0}, \boxed{1}\}$ . And each value  $a_t$  represent a fixed decision for one time slot in  $T$  ( $d$  is selected or not).

An example with 4 time slot is shown in Fig. 10.1. This is the same scenario that was already used in Fig. 9.4.

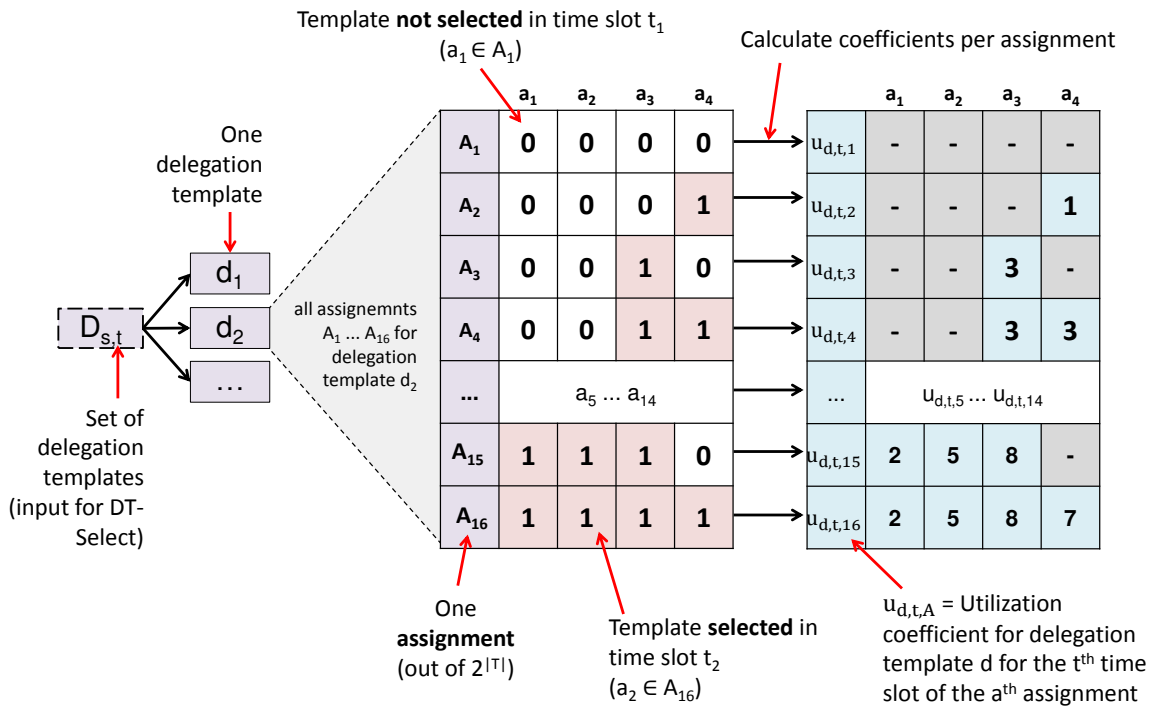


Figure 10.1: Example for assignments

The left part shows one set of delegation templates for switch  $s$  (note that there are four sets in this example,  $D_{s,t_1}$  to  $D_{s,t_4}$  that are not explicitly shown). The middle part shows the  $2^{|T|} = 16$  assignments for one of the delegation templates. The first assignment represents the case where the template is not selected in all four time slots  $(\boxed{0}\boxed{0}\boxed{0}\boxed{0})$ . The second assignment represents the case where the template is only selected in the last time slot  $(\boxed{0}\boxed{0}\boxed{0}\boxed{1})$ . And so on.

More importantly, the blue part on the right side shows that it is now possible to calculate coefficients for each assignment without dependencies – here on the example of utilization coefficients. Consider the fourth time slot  $t_4$  of the fourth assignment  $A_4$ . The utilization coefficient for this time slot is given as  $u_{d,t_4,A_4}$ , i.e., the utilization of delegation template  $d$  in the fourth time slot of assignment  $A_4 := \langle a_1 = \boxed{0}, a_2 = \boxed{0}, a_3 = \boxed{1}, a_4 = \boxed{1} \rangle$ .

Because all “decisions” within this assignment are fixed, the utilization coefficients can be calculated with Eq. (9.18) where the decision variables  $X_{d,t_1}, \dots, X_{d,t_4}$  are replaced with the values in the assignment, i.e., with  $a_1, \dots, a_4$ .  $u_{d,t_4,A_4}$  can therefore be calculated as follows:

$$\begin{aligned}
u_{d,t_4,A_2} &= \left( \sum_{f \in F_{d,t_4}} \lambda_{f,t_4}^i a_4 \right) + \left( \sum_{q=t_3}^{t_1} \sum_{f \in F_{d,q}} \left( \lambda_{f,q}^i * \lambda_{f,t_4}^a * \prod_{v=q}^{t_4} a_v \right) \right) \\
&= \left( \sum_{f \in F_{d,t_4}} \lambda_{f,t_4}^i a_4 \right) + \left( \sum_{f \in F_{d,t_3}} \lambda_{f,t_3}^i * \lambda_{f,t_4}^a * \prod_{v=t_3}^{t_4} a_v \right) + \left( \sum_{f \in F_{d,t_2}} \lambda_{f,t_2}^i * \lambda_{f,t_4}^a * \prod_{v=t_2}^{t_4} a_v \right) \\
&\quad + \left( \sum_{f \in F_{d,t_1}} \lambda_{f,t_1}^i * \lambda_{f,t_4}^a * \prod_{v=t_2}^{t_4} a_v \right) \\
&= \left( \sum_{f \in F_{d,t_4}} \lambda_{f,t_4}^i a_4 \right) + \left( \sum_{f \in F_{d,t_3}} \lambda_{f,t_3}^i \lambda_{f,t_4}^a a_3 a_4 \right) + \left( \sum_{f \in F_{d,t_2}} \lambda_{f,t_2}^i \lambda_{f,t_4}^a a_2 a_3 a_4 \right) \\
&\quad + \left( \sum_{f \in F_{d,t_1}} \lambda_{f,t_1}^i \lambda_{f,t_4}^a a_1 a_2 a_3 a_4 \right) \\
&= \left( \sum_{f \in F_{d,t_4}} \lambda_{f,t_4}^i \mathbb{1} \right) + \left( \sum_{f \in F_{d,t_3}} \lambda_{f,t_3}^i \lambda_{f,t_4}^a \mathbb{1} \mathbb{1} \right) + \left( \sum_{f \in F_{d,t_2}} \lambda_{f,t_2}^i \lambda_{f,t_4}^a \mathbb{0} \mathbb{1} \mathbb{1} \right) \\
&\quad + \left( \sum_{f \in F_{d,t_1}} \lambda_{f,t_1}^i \lambda_{f,t_4}^a \mathbb{0} \mathbb{0} \mathbb{1} \mathbb{1} \right) \\
&= \sum_{f \in F_{d,t_4}} \lambda_{f,t_4}^i + \sum_{f \in F_{d,t_3}} \lambda_{f,t_3}^i \lambda_{f,t_4}^a
\end{aligned}$$

This is the expected result: The first sum represents the flow rules installed in time slot  $t_4$  and the the second sum represents all flow rules installed in time slot  $t_3$  that are still active in time slot  $t_4$ . This way, it is possible to pre-calculate all coefficients for all time slots in all assignments.

Important remark: Coefficients for different delegation templates can be calculated independently from each other because the delegation templates are disjoint by definition. This is important, because dependencies between the delegation templates would increase the total amount of available assignments to  $2^{|T|*|D|}$  instead of  $|D| * 2^{|T|}$ .

### 10.2.1.2 Periodic Optimization

Periodic optimization means the optimization is executed periodically in the beginning of each time slot. The individual executions are called optimization periods. In each optimization period, a limited future horizon of  $L$  time slots is considered together with the status from the last optimization period (referred to as history  $H_s$ ).  $L$  is called look ahead factor.

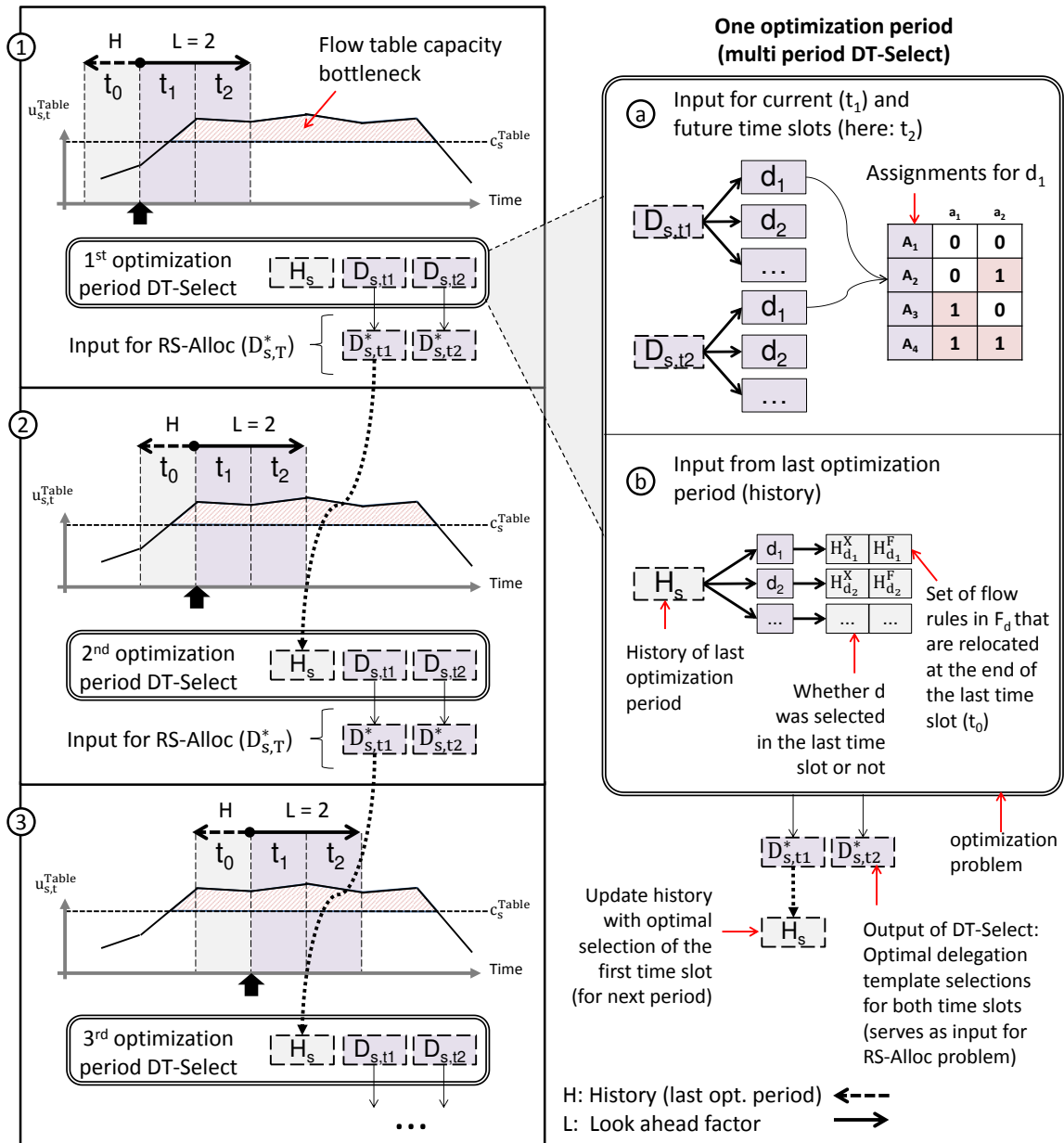


Figure 10.2: Periodic optimization

The periodic optimization approach is explained in Fig. 10.2. The example shows the execution for a single switch  $s$  (DT-Select is executed individually per switch) and the look ahead factor is set to  $L = 2$ , i.e., two time slots have to be considered. The three boxes in the left labeled as ①, ② and ③ are three subsequent optimization periods. The plot within each box represents the flow table utilization over time of switch  $s$ . This switch suffers from a flow table capacity bottleneck when the utilization ( $u_{s,t}^{\text{Table}}$ ) is above the dashed black line (capacity  $c_s^{\text{Table}}$ ).

In box ① of the example,  $T$  is given as  $\{t_1, t_2\}$  and the optimization is executed in the beginning of time slot  $t_1$  which is indicated here with a small black arrow. Note that  $u_{s,t_1}^{\text{Table}}$  represents the utilization at the end of time slot  $t_1$ . This means the optimization works with anticipated values, not with actually monitored values. Monitored values are only available for  $t_0$ , i.e., for the history. The periodic approach now takes two inputs which are shown in the right of the figure:

- ① Delegation templates  $D_{s,t}$  from the rule aggregation scheme for all time slots that are included in the look ahead. In this case ( $L = 2$ ), delegation templates for two time slots are required:  $D_{s,t_1}$  and  $D_{s,t_2}$ . Note that the ingress port scheme described in Sec. 5.2.2.3 will always return the “same” delegation templates, i.e., the aggregation match of  $d_1 \in D_{s,t_1}$  and  $d_1 \in D_{s,t_2}$  are identical. Only the conflict-free cover sets  $F_{d_1,t_1}$  and  $F_{d_1,t_2}$  will be different if new flow rules are installed or existing flow rules are removed between the end of  $t_1$  and the end of  $t_2$ . The information from  $D_{s,t_1}$  and  $D_{s,t_2}$  are used to create an assignment set  $A_{d,T} = \{A_1, \dots, A_4\}$  with four entries. This set represents all possible decisions for the considered time slots:
- $A_1$ : The delegation template is not selected at all ( $\overline{0}\overline{0}$ )
  - $A_2$ : The delegation template is selected only in time slot  $t_2$  ( $\overline{0}\overline{1}$ )
  - $A_3$ : The delegation template is selected only in time slot  $t_1$  ( $\overline{1}\overline{0}$ )
  - $A_4$ : The delegation template is selected in both time slots ( $\overline{1}\overline{1}$ )
- ② Information about the history of the last optimization period  $H_s$ . This is required because the utilization and cost coefficients are calculated differently if a delegation template was selected in the last optimization period. Take assignment  $A_3$  as an example with  $a_1 = \overline{1}$  and  $a_2 = \overline{0}$ . If the template was selected in the last optimization period, the utilization for the first time slot  $t_1$  would include all flow rules installed in the previous time slot (from the last period) that are still active. The same is true for  $A_4$ . To consider this in the optimization, two pieces of information are required: the selection decision for the given delegation template in the last optimization period ( $H_d^X$ ) and the set of flow rules that are relocated at the end of the last optimization period ( $H_d^F$ ).



Note that the information in  $D_{s,t_1}$  and  $D_{s,t_2}$  depend on information about the future, i.e., this part will work with predicted monitoring information in a real system. After the first optimization period, subsets  $D_{s,T}^* = \{D_{s,t_1}^*, D_{s,t_2}^*\}$  represent the optimal selections for both time slots. This is the input for the multi period remote switch allocation (RS-Alloc) problem which calculates optimal allocations for each selected delegation template.

In addition, the history for this switch ( $H_s$ ) is updated with the optimal result of the first time slot ( $D_{s,t_1}^*$ ). Note that the results for subsequent time slots (here:  $D_{s,t_2}^*$ ) are still required as input for the multi period RS-Alloc problem. The history, however, will only include the decision that is actually implemented with help of the detour procedure which is only the first time slot. If the algorithm is not executed in every time slot, this could of course be changed to include more than only the first time slot, but this is not considered here (the necessary changes would be minimal, though). In the context of delegation template selection, the history information in  $H_s$  is defined as follows:

#### Definition 10.2: History of a DT-Select optimization period

Assume the current optimization period starts with  $t_1$  and  $t_0$  is the time slot prior to  $t_1$ . The **history  $H_s$  of a DT-Select optimization period** for switch  $s$  is then defined as a tuple  $\langle H_d^X, H_d^F \rangle$  for each delegation template  $d \in D_s$  where  $H_d^X \in \{0, 1\}$  represents the decision whether delegation template  $d$  was selected in time slot  $t_0$  and  $H_d^F \subseteq F_{s,t_0}$  is a set of all flow rules that are active and relocated to the remote switch at the end of time slot  $t_0$ .

$$\forall_{d \in D_s} : H_d^X := \begin{cases} 1, & d \in D_{s,t_0}^* \\ 0, & d \notin D_{s,t_0}^* \end{cases} \quad (10.2)$$

$$\forall_{d \in D_s} : H_d^F := \{ f \mid f \in F_{s,t_0} \text{ and } f \text{ is relocated} \} \quad (10.3)$$

Eq. (10.3) can be calculated recursively between optimization periods using the following formula and  $H_d^F := \emptyset$  (each period starts with  $t_1$  as first time slot):

$$H_d^F := \begin{cases} \{ f \in F_{s,t_0} \mid \lambda_{f,t_0}^i = 1 \} \cup \{ f \in H_d^F \mid \lambda_{f,t_0}^a = 1 \}, & H_d^X = 1 \\ \emptyset, & H_d^X = 0 \end{cases} \quad \begin{matrix} \text{[Period=n]} \\ \text{[Period=n-1]} \\ \text{[Period=n-1]} \end{matrix}$$

Periodic optimization continues as long as there is a flow table capacity bottleneck. This approach is well-suited for flow delegation where the situation in the network changes continuously (reflected by changes in the conflict-free cover sets  $F_{d,t}$ ). And because  $L$  can be kept small – it is shown later that  $L = 3$  works best for many scenarios –, the problem of exponentially growing input length that comes with the assignment concept is avoided.

Furthermore, the approach could also be used to calculate the DT-Select problem for historical data by iterating over a large set of time slots  $T$  in small chunks of size  $L$ . In this case, no prediction is necessary because perfect future knowledge is available. However, this will not necessarily return the overall optimal result because sub optimal selections from previous optimization periods cannot be corrected. Instead, optimality is only ensured within the individual optimization periods. Nevertheless, it allows solving DT-Select iteratively for arbitrary sizes of  $T$  which is not possible otherwise.

### 10.2.2 Decision Variables

Decision variables for Select-Opt are defined on assignments instead of time slots:

#### Definition 10.3: Decision Variables for Select-Opt

Given a delegation template  $d$  and an assignment set  $A_{d,T}$ , the **decision variables for Select-Opt** are defined as:

$$X_{d,A} := \begin{cases} 1, & \text{Assignment } A \text{ is the selected assignment} \\ 0, & \text{Assignment } A \text{ is not the selected assignment} \end{cases} \quad (10.4)$$

These decision variables are defined for each delegation template  $d \in D_s$  and all assignments in  $A_{d,T}$ . Undefined variables are interpreted as 0.

The idea is to select exactly one out of the available  $2^{|T|}$  assignments in the assignment set  $A_{d,T}$  instead of selecting individual delegation templates for all time slots in  $T$ . This can be achieved with the following simple constraint:

$$\sum_{A \in A_{d,T}} X_{d,A} = 1 \quad \forall_{d \in D_s} \quad (10.5)$$

As a consequence, the utilization and cost coefficients have to be remodeled to work with assignments and the periodic optimization approach. It is explained in the following two sections how this is done.

### 10.2.3 Utilization Coefficients

Recall from Sec. 9.4.3 that the utilization coefficients for a single delegation template  $d \in D_{s,t}$  represent the amount of flow rules that will be relocated to the remote switch in time slot  $t$ . This is basically the “weight” of a delegation template in the knapsack terminology. Further recall that the utilization constraint in the multi period DT-Select problem is explicitly defined per time slot (see Sec. 9.4.6). The idea is to select subsets  $D_{s,t}^*$  for each  $t \in T$  so that the current utilization minus the sum of the utilization of the selected templates is smaller than the flow table capacity of the delegation switch in that time slot:

$$u_{s,t}^{\text{Table}} - \sum_{d \in D_{s,t}^*} u_{d,t}^{\text{Table}} \leq c_s^{\text{Table}} \quad \forall_{t \in T} \quad (10.6)$$

So it is not sufficient to calculate the utilization per assignment  $A \in A_{d,T}$  because each  $A$  represents the sum of all time slots in  $T$ , i.e., information about the utilization at the individual time slots would be lost. Instead, utilization coefficients for each time slot in assignment  $A$  are required so that it is possible to remodel the constraint from above into the following assignment-based constraint:

$$u_{s,t}^{\text{Table}} - \sum_{d \in D_s^*} \sum_{A \in A_{d,T}^*} u_{d,t,A}^{\text{Table}} \leq c_s^{\text{Table}} \quad \forall_{t \in T} \quad (10.7)$$

It is easy to see that this assignment-based formula is equivalent to Eq. (10.6) because  $A_{d,T}^*$  can only contain a single item according to the constraint in Eq. (11.7) – which is the selected optimal assignment. And it was already explained above in Sec. 10.2.1.1 how the utilization coefficients  $u_{d,t,A}^{\text{Table}}$  are calculated:

$$u_{d,t,A}^{\text{Table}} := \left( \sum_{f \in F_{d,t}} \lambda_{f,t}^i a_t \right) + \left( \sum_{q=t-1}^{t_1} \sum_{f \in F_{d,q}} \lambda_{f,q}^i * \lambda_{f,t}^a * \prod_{v=q}^t a_v \right) \quad (10.8)$$

$A = \langle a_1, \dots, a_t, \dots, a_m \rangle$  is one of the  $2^{|T|}$  assignments in  $A_{d,T}$  for delegation template  $d$ . And variable  $a_t \in \{\underline{0}, \underline{1}\}$  represents the selection decision in time slot  $t$ , i.e., this is not a decision variable but a static input for this specific assignment set (created according to Def. 10.1). So the first part of the formula counts the flow rules installed in time slot  $t$  if  $a_t$  is set to  $\underline{1}$ . The second part is required to keep track of flow rules installed in previous time slots. Assume the assignment set is  $A = \langle a_1 = \underline{0}, a_2 = \underline{0}, a_3 = \underline{1}, a_4 = \underline{1} \rangle$  (taken from the example in Sec. 10.2.1.1) and the utilization coefficient for time slot  $t_4$  has to be calculated. This includes all flow rules installed in  $t_4$  but also the flow rules installed in  $t_3$  that are still active. This is what  $\sum_{q=t-1}^{t_1} \sum_{f \in F_{d,q}} \lambda_{f,q}^i * \lambda_{f,t}^a * \prod_{v=q}^t a_v =$

$\sum_{f \in F_{d,t_4}} \lambda_{f,t_4}^i + \sum_{f \in F_{d,t_3}} \lambda_{f,t_3}^i \lambda_{f,t_4}^a$  calculates ( $t_{-1} = t_3$  in this example,  $q$  and  $v$  are iterator variables representing time slots).

What is left is to include the history of the last optimization period (see Def. 10.2). Note that the above formula makes no difference between the case where delegation template  $d$  is selected in the last optimization period and the case where  $d$  is not selected. However, this makes a difference. Consider the following example where the first value in grey represents the decision of the last optimization period (the history) and the four subsequent values belong to an assignment set  $A = \langle a_1 = \boxed{1}, a_2 = \boxed{1}, a_3 = \boxed{0}, a_4 = \boxed{1} \rangle$ .

$$\begin{array}{ccccc}
 t_0 & t_1 & t_2 & t_3 & t_4 \\
 \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
 \boxed{1} & \boxed{1} & \boxed{1} & \boxed{0} & \boxed{1} \\
 H_d^X & a_1 & a_2 & a_3 & a_4
 \end{array}$$

In this case, Eq. (10.8) will not return the correct utilization because there might be flow rules that were relocated in one of the last optimization periods which are still active. Consider a flow rule  $f$  installed in time slot  $t_0$  ( $\lambda_{f,t_0}^i = 1$ ) that is still active in time slot  $t_1$  ( $\lambda_{f,t_1}^a = 1$ ). This flow rule was relocated because the delegation template was selected in the last optimization period ( $H_d^X = \boxed{1}$ ). However, this rule is currently not covered by Eq. (10.8). The following extended formula for  $u_{d,t,A}^{\text{Table}}$  corrects this:

$$\begin{aligned}
 u_{d,t,A}^{\text{Table}} := & \left( \sum_{f \in F_{d,t}} \lambda_{f,t}^i a_t \right) + \left( \sum_{q=t-1}^{t_1} \sum_{f \in F_{d,q}} \lambda_{f,q}^i * \lambda_{f,t}^a * \prod_{v=q}^t a_v \right) \\
 & + \left( H_d^X * \prod_{q=t}^{t_1} a_q * \sum_{f \in H_d^F} \lambda_{f,t}^a \right)
 \end{aligned} \tag{10.9}$$

This equation is structured in three parts (the brackets). The first two parts prior to  $H_d^X$  are identical to Eq. (10.8). The new third part with  $H_d^X$  takes care of flow rules from previous optimization periods. Note that this part is only included if  $H_d^X = 1$ , i.e., the delegation template was selected in the previous period. In this case, all relocated flow rules from the previous period (given as set  $H_d^F$ ) that are still active in time slot  $t$  ( $\lambda_{f,t}^a = 1$ ) are also considered for this utilization coefficient.  $\prod_{q=t}^{t_1} a_q$  is required to make sure that the history is still relevant. Consider the fourth time slot from the above example. Because  $a_3 = \boxed{0}$ , all relocated flow rules were removed from the remote switch at the end of time slot  $t_2$ . So from the beginning of time slot  $t_3$ , the history is not relevant any more.

To see that the extended formula works as expected, consider the example from before with  $T = \{t_1, \dots, t_4\}$  and assignment  $A = \langle a_1 = \boxed{1}, a_2 = \boxed{1}, a_3 = \boxed{0}, a_4 = \boxed{1} \rangle$ . For this assignment, the four utilization coefficients  $u_{d,t_1,A}, \dots, u_{d,t_4,A}$  are calculated with Eq. (10.9) as:

$$\begin{aligned} u_{d,t_1,A} &:= \sum_{f \in F_{d,t_1}} \lambda_{f,t_1}^i + \sum_{f \in H_d^F} \lambda_{f,t_1}^a \\ u_{d,t_2,A} &:= \sum_{f \in F_{d,t_2}} \lambda_{f,t_2}^i + \sum_{f \in F_{d,t_2}} \lambda_{f,t_1}^i \lambda_{f,t_2}^a + \sum_{f \in H_d^F} \lambda_{f,t_2}^a \\ u_{d,t_3,A} &:= 0 \\ u_{d,t_4,A} &:= \sum_{f \in F_{d,t_4}} \lambda_{f,t_4}^i \end{aligned}$$

And this is indeed the expected result:

- For  $u_{d,t_1,A}$  in time slot  $t_1$ , all newly installed flow rules are relocated ( $\sum_{f \in F_{d,t_1}} \lambda_{f,t_1}^i$ ) and pre-existing relocated flow rules from previous optimization periods (the rules in  $H_d^F$ ) are also considered if they are still active in time slot  $t_1$  ( $\sum_{f \in H_d^F} \lambda_{f,t_1}^a$ ).
- For  $u_{d,t_2,A}$  in time slot  $t_2$ , all newly installed flow rules are relocated ( $\sum_{f \in F_{d,t_2}} \lambda_{f,t_2}^i$ ). And because  $a_1$  is also set to  $\boxed{1}$ , the second part of the formula will count all flow rules installed in time slot  $t_1$  that are still active in time slot  $t_2$  ( $\sum_{f \in F_{d,t_2}} \lambda_{f,t_1}^i \lambda_{f,t_2}^a$ ). In addition, all pre-existing relocated flow rules from previous optimization periods are considered as well if they are still active in time slot  $t_2$  ( $\sum_{f \in H_d^F} \lambda_{f,t_2}^a$ ).
- For  $u_{d,t_3,A}$  in time slot  $t_3$ , the utilization is 0 because the delegation template is not selected. Note that the history part – third part in Eq. (10.9) – is not considered because of  $a_3 = \boxed{0}$ . This is the reason why  $\prod_{q=t}^{t_1} a_q$  is required and why the lower index starts with  $t$  and not with  $t_{-1}$  as it is in the second part of Eq. (10.9).
- For  $u_{d,t_4,A}$  in time slot  $t_4$ , only the newly installed flow rules in time slot  $t_4$  are considered ( $\sum_{f \in F_{d,t_4}} \lambda_{f,t_4}^i$ ). The second and the third part of Eq. (10.9) are not considered because of  $a_3 = \boxed{0}$ .

#### 10.2.4 Cost Coefficients

Recall from Sec. 9.4.4 that the cost coefficients represent the overhead associated with selecting a delegation template. This overhead consists of the amount of required

aggregation and backflow rules ( $w_{d,t}^{\text{Table}}$ ), the amount of relocated packets ( $w_{d,t}^{\text{Link}}$ ) and the amount of required control messages ( $w_{d,t}^{\text{Ctrl}}$ ). Further recall these three coefficients are used in the objective function to minimize the cost of the optimal template selections for all  $t \in T$ :

$$\min_{X_{d,t}} \sum_{t \in T} \sum_{d \in D_{s,t}} X_{d,t} (\omega_{\text{DTS}}^{\text{Table}} w_{d,t}^{\text{Table}} + \omega_{\text{DTS}}^{\text{Ctrl}} w_{d,t}^{\text{Ctrl}} + \omega_{\text{DTS}}^{\text{Link}} w_{d,t}^{\text{Link}}) \quad (\text{see 9.15})$$

This is a multi-objective formulation and the non-negative  $\omega_{\text{DTS}}$ -weights balance the different cost coefficients against each other. The following re-models this into an objective function for Select-Opt that works with assignments and periodic optimization:

$$\min_{X_{d,A}} \sum_{d \in D_s} \sum_{A \in A_{d,T}} X_{d,A} * (\omega_{\text{DTS}}^{\text{Table}} w_{d,A}^{\text{Table}} + \omega_{\text{DTS}}^{\text{Ctrl}} w_{d,A}^{\text{Ctrl}} + \omega_{\text{DTS}}^{\text{Link}} w_{d,A}^{\text{Link}}) \quad (10.10)$$

First, the new objective function uses the decision variables  $X_{d,A}$  for Select-Opt from Def. 10.3. The outer sum iterates over all delegation templates and the inner sum iterates over all assignments in the assignment set  $A_{d,T}$  of the currently selected delegation template. And finally all three cost coefficients are defined per assignment, i.e., independently of individual time slots (which is different from the utilization coefficients discussed in the previous section!). This is possible here because the minimization in Eq. (9.15) is done over all time slots in  $T$  which maps perfectly to the definition of an assignment which represents all time slots in  $T$ .

Before the individual cost coefficients for Select-Opt are explained in more detail, it is important to have a quick look at how these coefficients are calculated in general. The first problem is that a cost coefficient for time slot  $t$  may depend on all previous time slots in the considered assignment and the history of the last optimization period (see Def. 10.2). This is very similar to the utilization coefficients and can be addressed with similar modeling techniques.

The second problem is more difficult. Unlike the utilization coefficients, the calculation of some cost coefficients can be different based on different preconditions ( $w_{d,A}^{\text{Link}}$  suffers from this and there are other possible cost coefficients not discussed here with the same problem). Consider an example with  $T = \{t_1, \dots, t_5\}$  and assignment  $A = \langle a_1 = \boxed{1}, a_2 = \boxed{0}, a_3 = \boxed{0}, a_4 = \boxed{1}, a_5 = \boxed{1} \rangle$ . Now assume the control message overhead cost has to be calculated for this assignment. To do so, four different cases ① to ④ have to be considered:

- ① With  $a_2$  set to  $\boxed{0}$  and  $a_3$  also set to  $\boxed{0}$ , no control messages are required in time slot  $t_3$  and the cost is 0.

- ② With  $a_3$  set to  $\boxed{0}$  and  $a_4$  set to  $\boxed{1}$ , a new aggregation rule is required in  $t_4$  and all newly installed flow rules in  $t_4$  are relocated.
- ③ With  $a_1$  set to  $\boxed{1}$  and  $a_2$  set to  $\boxed{0}$ , all flow rules relocated in  $t_1$  have to be moved back to the delegation switch. The history of the last optimization period has to be considered as well here.
- ④ With  $a_4$  set to  $\boxed{1}$  and  $a_5$  set to  $\boxed{1}$ , only the newly installed flow rules in  $t_5$  have to be considered. The aggregation rule is already installed.

Such differentiations cannot be modeled with one simple formula as it was possible for the utilization coefficients. To account for this, the cost coefficients are modeled with four different helper variables that map to the four cases above:

$$\left[ \begin{array}{l} w_{d,t,A}^{\boxed{0}\boxed{0}} := \boxed{\text{Cost if } a_t = \boxed{0} \text{ and } a_{t+1} = \boxed{0}} \quad \text{case } \textcircled{1} \\ w_{d,t,A}^{\boxed{0}\boxed{1}} := \boxed{\text{Cost if } a_t = \boxed{0} \text{ and } a_{t+1} = \boxed{1}} \quad \text{case } \textcircled{2} \\ w_{d,t,A}^{\boxed{1}\boxed{0}} := \boxed{\text{Cost if } a_t = \boxed{1} \text{ and } a_{t+1} = \boxed{0}} \quad \text{case } \textcircled{3} \\ w_{d,t,A}^{\boxed{1}\boxed{1}} := \boxed{\text{Cost if } a_t = \boxed{1} \text{ and } a_{t+1} = \boxed{1}} \quad \text{case } \textcircled{4} \end{array} \right] \quad (10.11)$$

Each case models one of four possible transitions.  $a_t$  represents the value of the assignment in time slot  $t$ .  $a_{t+1}$  represents the value of the assignment in time slot  $t+1$ .  $a_{t_0}$  is set to  $H_d^X$  (the decision from the previous optimization period) or to 0 if  $H_d^X$  is undefined. What is important here is that the calculated cost is always associated with the second time slot ( $t+1$ ) while time slot  $t$  is considered as a precondition. The first case above could be read as “cost for not selecting the delegation template if it was not selected in the previous time slot”. To emphasise this, the first upper index is shown in grey. Using this, it is now possible to determine the cost for assignment  $A$  in the following way:

$$w_{d,A} := \sum_{t=t_0}^{t_{m-1}} \boxed{(1-a_t) * (1-a_{t+1}) * w_{d,t,A}^{\boxed{0}\boxed{0}}} + \boxed{(1-a_t) * a_{t+1} * w_{d,t,A}^{\boxed{0}\boxed{1}}} \quad (10.12)$$

$$+ \boxed{a_t * (1-a_{t+1}) * w_{d,t,A}^{\boxed{1}\boxed{0}}} + \boxed{a_t * a_{t+1} * w_{d,t,A}^{\boxed{1}\boxed{1}}}$$

The formula iterates over all time slots in  $T$  and looks at two time slots per iteration. Note that the iteration starts at  $t_0$  and terminates at  $t_{m-1}$  which is important. In each iteration, the four cases discussed above are modeled with the helper variables from Eq.

(10.11). The four boxes are added to emphasize this and to support readability. The first box models case ① where two consecutive values in the assignment are set to  $\boxed{0}\boxed{0}$ . The second box models case ② where two consecutive values in the assignment are set to  $\boxed{0}\boxed{1}$ . And so on. For  $A = \langle a_1 = \boxed{1}, a_2 = \boxed{0}, a_3 = \boxed{1}, a_4 = \boxed{1} \rangle$  and  $H_d^X = \boxed{1}$ , the four iterations look like this (this also illustrates why the calculated cost is always associated with the second time slot of each iteration):

$$\begin{array}{cccc}
 t_0 & t_1 & & t_1 & t_2 & & t_2 & t_3 & & t_3 & t_4 \\
 \downarrow & \downarrow & & \downarrow & \downarrow & & \downarrow & \downarrow & & \downarrow & \downarrow \\
 \boxed{1} & \boxed{1} & \Rightarrow & \boxed{1} & \boxed{0} & \Rightarrow & \boxed{0} & \boxed{1} & \Rightarrow & \boxed{1} & \boxed{1} \\
 H_d^X & a_1 & & a_1 & a_2 & & a_2 & a_3 & & a_3 & a_4
 \end{array}$$

The beauty of this notation is that all cost coefficients can be displayed compactly with a set of helper variables according to Eq. (10.11) and the formula defined in Eq. (10.12) – as it is done now in the following sub-sections where the three cost coefficients for Select-Opt are defined.

#### 10.2.4.1 Rule Overhead

The rule overhead cost coefficients  $w_{d,A}^{\text{Table}}$  depend on the number of installed aggregation and backflow rules in the delegation switch. In this work, only the aggregation rules are considered while backflow rules are pre-installed in every switch (see Sec. 9.4.4.1). The translation to Select-Opt in this case is therefore very simple:

$$\left[ \begin{array}{ll}
 w_{d,t,A}^{\boxed{0}\boxed{0}\text{Table}} := 0 & \text{case ①} \\
 w_{d,t,A}^{\boxed{0}\boxed{1}\text{Table}} := 1 & \text{case ②} \\
 w_{d,t,A}^{\boxed{1}\boxed{0}\text{Table}} := 0 & \text{case ③} \\
 w_{d,t,A}^{\boxed{1}\boxed{1}\text{Table}} := 1 & \text{case ④}
 \end{array} \right] \quad (10.13)$$

The only relevant cases are case ② where a new delegation template is selected in time slot  $t_{+1}$  and case ④ where an already selected delegation template is selected again in time slot  $t_{+1}$ . Cost coefficients  $w_{d,A}^{\text{Table}}$  for an assignment  $A$  are then calculated as follows:

$$w_{d,A}^{\text{Table}} := \sum_{t=t_0}^{t_{m-1}} \boxed{(1 - a_t) * a_{t+1} * w_{d,t,A}^{\boxed{0}\boxed{1}\text{Table}}} + \boxed{a_t * a_{t+1} * w_{d,t,A}^{\boxed{1}\boxed{1}\text{Table}}} \quad (10.14)$$



$$= \sum_{t=t_0}^{t_{m-1}} (1 - a_t) * a_{t+1} + a_t * a_{t+1}$$

### 10.2.4.2 Link Overhead

The link overhead cost coefficients  $w_{d,A}^{\text{Link}}$  take the amount of relocated packets into account. These are the packets matched by the aggregation rule that have to be sent to the remote switch which is additional overhead for the infrastructure. For Select-Opt, these coefficients are calculated as follows:

$$\left[ \begin{array}{l} w_{d,t,A}^{\boxed{0}\boxed{0}\text{Link}} := 0 \quad \text{case } \textcircled{1} \\ w_{d,t,A}^{\boxed{0}\boxed{1}\text{Link}} := \sum_{f \in F_{d,t}} \delta_{f,t} \lambda_{f,t}^i \quad \text{case } \textcircled{2} \\ w_{d,t,A}^{\boxed{1}\boxed{0}\text{Link}} := 0 \quad \text{case } \textcircled{3} \\ w_{d,t,A}^{\boxed{1}\boxed{1}\text{Link}} := \left( \sum_{f \in F_{d,t}} \delta_{f,t} \lambda_{f,t}^i \right) + \left( \sum_{q=t-1}^{t_1} \sum_{f \in F_{d,q}} \delta_{f,t} \lambda_{f,q}^i \lambda_{f,t}^a \prod_{v=q}^t a_v \right) \\ \quad + \left( H_d^X \prod_{q=t}^{t_1} a_q \sum_{f \in H_d^F} \delta_{f,t} \lambda_{f,t}^a \right) \quad \text{case } \textcircled{4} \end{array} \right] \quad (10.15)$$

In case  $\textcircled{1}$  and case  $\textcircled{3}$ , the link overhead is 0 because the delegation template is not selected (recall that the cost is always defined for the second time slot of each iteration). In case  $\textcircled{2}$ , the template is selected in  $t_{+1}$  and was not selected in the previous time slot  $t$ . This means only the newly installed flow rules are relocated to the remote switch which in turn means that only these rules contribute to the link overhead. Case  $\textcircled{4}$  is more complex because the delegation template was already selected in the previous time slot. The given formula has the exact same structure as Eq. (10.9). The first part (first bracket) counts the bits/s for all flow rules installed in time slot  $t$  because  $a_t$  is set to  $\boxed{1}$ . The second part is required to keep track of flow rules installed in previous time slots – which is relevant here because the previous time slot is set to  $\boxed{1}$  (because this is case  $\textcircled{4}$ ). The third part with  $H_d^X$  includes the flow rules from previous optimization periods. This

is only required if  $H_d^X$  and all time slots between  $t$  and  $t_1$  are set to  $\boxed{1}$  (modeled with  $\prod_{q=t}^{t_1} a_q$ ). The cost coefficient  $w_{d,A}^{\text{Link}}$  for an assignment  $A$  is then calculated as follows:

$$w_{d,A}^{\text{Link}} := \sum_{t=t_0}^{t_{m-1}} \boxed{(1 - a_t) * a_{t+1} * w_{d,t,A}^{\boxed{0}\boxed{1}\text{Link}}} + \boxed{a_t * a_{t+1} * w_{d,t,A}^{\boxed{1}\boxed{1}\text{Link}}} \quad (10.16)$$

### 10.2.4.3 Control Message Overhead

The control message overhead cost coefficients  $w_{d,A}^{\text{Ctrl}}$  consider the amount of control messages necessary if delegation template  $d$  is selected in time slot  $t$ . The different cases for this cost coefficient were already discussed in the introduction to this section. The formal representation is as follows:

$$\left[ \begin{array}{l} w_{d,t,A}^{\boxed{0}\boxed{0}\text{Ctrl}} := \boxed{0} \quad \text{case } \textcircled{1} \\ w_{d,t,A}^{\boxed{0}\boxed{1}\text{Ctrl}} := \boxed{m_t + 1 + \sum_{f \in F_{d,t}} \lambda_{f,t}^i} \quad \text{case } \textcircled{2} \\ w_{d,t,A}^{\boxed{1}\boxed{0}\text{Ctrl}} := \boxed{1 + 2 * \left[ \left( \sum_{q=t-1}^{t_1} \sum_{f \in F_{d,q}} \lambda_{f,q}^i \lambda_{f,t}^a \prod_{v=q}^t a_v \right) + \left( H_d^X \prod_{q=t}^{t_1} a_q \sum_{f \in H_d^F} \lambda_{f,t}^a \right) \right]} \quad \text{case } \textcircled{3} \\ w_{d,t,A}^{\boxed{1}\boxed{1}\text{Ctrl}} := \boxed{m_t + \sum_{f \in F_{d,t}} \lambda_{f,t}^i} \quad \text{case } \textcircled{4} \end{array} \right] \quad (10.17)$$

$m_t$  is a penalty factor to ensure that it is more expensive to add (case  $\textcircled{2}$ ) or keep (case  $\textcircled{4}$ ) a delegation template selected if there is currently no flow table capacity bottleneck. To achieve this, the penalty factor is defined as follows:

$$m_t := \begin{cases} > 0, & \text{if } u_{s,t}^{\text{Table}} > c_s^{\text{Table}} \\ 0, & \text{otherwise} \end{cases} \quad (10.18)$$

Without such a penalty factor, the cost to “stop” flow delegation (which requires case  $\textcircled{3}$ : not selecting a delegation template that was previously selected) is high if the number of relocated flows is high. This can lead to the effect that some delegation templates

are selected longer than necessary. The cost coefficient  $w_{d,A}^{\text{Ctrl}}$  for an assignment  $A$  is then calculated as follows:

$$w_{d,A}^{\text{Ctrl}} := \sum_{t=t_0}^{t_{m-1}} \boxed{(1 - a_t) * a_{t+1} * w_{d,t,A}^{\boxed{0}\boxed{1}\text{Ctrl}}} \quad (10.19)$$

$$+ \boxed{a_t * (1 - a_{t+1}) * w_{d,t,A}^{\boxed{1}\boxed{0}\text{Ctrl}}} + \boxed{a_t * a_{t+1} * w_{d,t,A}^{\boxed{1}\boxed{1}\text{Ctrl}}}$$

### 10.2.5 Problem Formulation for Select-Opt

Based on the variables introduced in the previous sections, the Select-Opt problem (which is a multi period DT-Select problem with assignments that can be used with periodic optimization) is defined as an integer linear program in the following way:

#### Definition 10.4: Select-Opt Problem

**Inputs:** A single switch  $s \in S$ , flow table capacity  $c_s^{\text{Table}}$ , flow table utilization  $u_{s,t}^{\text{Table}}$ , set of consecutive time slots  $T$ , set of delegation templates  $D_{s,t}$  for each time slot, set of assignments  $A_{d,T}$  for each  $d \in D_s$ , pre-processed utilization coefficients  $u_{d,t,A}$  for each  $d \in D_{s,t}$ ,  $t \in T$ , and  $A \in A_{d,T}$ , pre-processed overhead coefficients  $w_{d,A}$  for each  $d \in D_s$  and  $A \in A_{d,T}$

**Output:** Set of selected delegation templates  $D_{s,t}^* \subseteq D_{s,t}$  for each time slot  $t \in T$

**Problem Formulation:**

$$\min_{X_{d,A}} \sum_{d \in D_s} \sum_{A \in A_{d,T}} X_{d,A} * (\omega_{\text{DTS}}^{\text{Table}} w_{d,A}^{\text{Table}} + \omega_{\text{DTS}}^{\text{Ctrl}} w_{d,A}^{\text{Ctrl}} + \omega_{\text{DTS}}^{\text{Link}} w_{d,A}^{\text{Link}}) \quad (10.20)$$

$$\text{s.t. } u_{s,t}^{\text{Table}} - \sum_{d \in D_s} \sum_{A \in A_{d,T}} X_{d,A} * u_{d,t,A}^{\text{Table}} \leq c_s^{\text{Table}} \quad \forall t \in T \quad (10.21)$$

$$\sum_{A \in A_{d,T}} X_{d,A} = 1 \quad \forall d \in D_s \quad (10.22)$$

$$X_{d,A} \in \{0, 1\} \quad \forall d \in D_s \quad \forall A \in A_{d,T} \quad (10.23)$$

This is an instance of the multi-dimensional multiple-choice knapsack problem. There are  $n$  different mutually disjoint classes  $A_{d,T} = \{A_1, \dots, A_n\}$  called assignment sets. Each delegation template  $d \in D_s$  defines one assignment set and each of these sets contains

exactly  $2^m$  items called assignments. For each assignment  $A \in A_{d,T}$  and each time slot  $t \in T$ , there is a utilization coefficient  $u_{d,t,A}^{\text{Table}}$ . And there is a cost coefficient  $w_{d,A}$  for each assignment  $A \in A_{d,T}$ . Goal is to choose exactly one item out of each assignment set so that the costs are minimized while the utilization is below the capacity in all time slots. Eq. (10.20) ensures that the objective function is minimized. Eq. (10.21) models the capacity constraint. This is done for each time slot separately, i.e., there are  $m = |T|$  capacity constraints. The multiple-choice constraints in Eq. (10.22) makes sure that exactly one out of  $2^m$  assignments is chosen. And Eq. (10.23) ensures that the decision variables are binary.

Technically, this falls into the same complexity class as the multi period DT-Select problem introduced in Def. 9.3 (both are NP-hard). Select-Opt, however, is much more difficult to solve in practice. This is because i) the input length as well as the number of decision variables grow exponentially with  $m$  and ii) the multidimensional multiple choice knapsack problem is one of the hardest members of the knapsack family [HLS10; Sho+13]. An empirical study from Han et. al [HLS10] on hard multidimensional multiple choice knapsack problems has found that this is especially true if the weights and profits (utilization and costs) are strongly correlated and the profits (costs) across the different classes (assignment sets) are similar – both is true for Select-Opt.

This is why the above problem should only be used with periodic optimization where the input length is restricted by look ahead factor  $L$ . The following section presents the Select-Opt algorithm that uses the Select-Opt problem in such a way.

### 10.2.6 Algorithm for Select-Opt

The Select-Opt algorithm is shown in Alg. 10. Goal, input and output are defined analogously to the Select-Opt problem from Def. 10.4. In addition, the periodic approach described in Sec. 10.2.1.2 is applied. The algorithm is structured into four steps that are executed once per optimization period. In the first step, the assignment sets  $A_{d,T}$  for each time slot and each delegation template of switch  $s$  are created, together with all required coefficients (lines 7-18).

In the second step, the Select-Opt problem is solved for the current optimization period based on the inputs and the calculations from step 1 (lines 20-23). If the problem is feasible, this returns a set of selected delegation templates for each time slot  $t \in T$  that is shown here as  $D_{s,T}^*$ . Note that the problem can get infeasible because Select-Opt only considers  $L$  time slots of the current optimization period. This happens if the flow table utilization is close to the capacity and develops unfavourably. Consider the following example. Assume a switch with  $c_s^{\text{Table}} = 100$  and a look ahead factor of  $L = 2$ . Further assume that the expected utilization is 85 flow rules for  $t_1$ , 96 for  $t_2$ , 98 for  $t_3$  and 150

**Algorithm 10:** Select-Opt Algorithm**Data:** Same input as Select-Opt problem from Def. 10.4, Look ahead factor  $L$ **Result:** For each optimization period: set of selected delegation templates

$$D_{s,t}^* \subseteq D_{s,t} \text{ for each time slot } t \in T \text{ with } |T| = L$$

```

1 Function select_opt_algorithm():
2   for  $d \in D_s$  do
3      $H_d^X \leftarrow 0$   $H_d^F \leftarrow \emptyset$ 
4   end
5   if  $u_{s,t}^{Table} > c_s^{Table}$  or  $\sum_{d \in D_s} H_d^X > 0$  then
6     for  $T = \{t_1, \dots, t_m\}$  do
7       /* Step 1: Initialize assignments and calculate coefficients */
8       for  $d \in D_s$  do
9          $A_{d,T} \leftarrow \{\}$ 
10        for  $i \in \{1, \dots, 2^{|T|}\}$  do
11           $A \leftarrow \text{new\_assignment}()$  // Def. 10.1
12          for  $t \in T$  do
13            /*  $i_2$  represents integer value  $i$  in binary form */
14             $A[t] = a_t \leftarrow i_2[t]$ 
15             $u_{d,t,A}^{Table} \leftarrow \text{utilization\_coefficient}(d, t, A)$  // Sec. 10.2.3
16          end
17           $A_{d,T} \leftarrow A_{d,T} \cup \{A\}$ 
18        end
19         $w_{d,A} \leftarrow \text{cost\_coefficient}(d, A)$  // Sec. 10.2.4
20      end
21      /* Step 2: solve Select-Opt (will return a set of selected delegation
22      templates  $D_{s,t}^*$  for each time slot in  $T$ ) */
23       $D_{s,T}^* = \{D_{s,t_1}^*, \dots, D_{s,t_m}^*\} \leftarrow \text{solve\_select\_opt}()$  // Def. 10.4
24      if  $D_{s,T}^* = \emptyset$  then
25         $D_{s,T}^* = \{D_{s,t_1}^*, \dots, D_{s,t_m}^*\} \leftarrow \text{fallback}()$ 
26      end
27      /* Step 3: update history and wait for next optimization period */
28      for  $d \in D_{s,t}$  do
29         $\langle H_d^X, H_d^F \rangle \leftarrow \text{update\_history}()$  // Def. 10.2
30      end
31      wait until next optimization period and start again from line 5
32    end
33  end

```

for  $t_4$ . Because the utilization peak in  $t_4$  is not “visible” ( $L$  is too small), the algorithm decides that no delegation template is selected in  $t_1$ . In the next optimization period, the utilization has increased to 96 and the expected utilization in  $t_4$  is included into the calculation. The problem is now infeasible, if more than 4 delegation templates are required to make sure that the utilization does not exceed the capacity in  $t_4$  because the capacity does only allow 4 more aggregation rules and each newly selected template requires an aggregation rule.

There are two options to cope with this problem, defining a safety margin (i.e., artificially reduce  $c_s^{\text{Table}}$ ) or install a fallback option to make sure that the problem becomes feasible again in future optimization periods. Because the first option introduces static overhead and the problem occurs rarely for look ahead values  $\geq 3$ , the second option is applied here (an empty result set in line 21 shows the problem was infeasible). Select-Greedy introduced in Sec. 10.4 is a good candidate for the fallback algorithm. The last step (lines 24-27) after calculation of  $D_{s,T}^*$  is updating the history and waiting for the next optimization period.

### 10.2.7 Summary

The Select-Opt algorithm calculates the multi period DT-Select problem using assignments to avoid that coefficients cannot be calculated independently from the decisions in different time slots and periodic optimization to avoid exponentially growing input length. The number of decision variables in the underlying ILP for a single optimization period is  $n * 2^L$  where  $n$  is the number of delegation templates and  $L$  is the look ahead factor. This allows for solving times in the millisecond range if  $L$  is sufficiently small. However, Select-Opt still requires costly pre-calculations (create assignment sets with  $2^L$  elements each, create  $n * 2^L$  decision variables, calculate  $n * L * 2^L$  utilization coefficients).

## 10.3 DT-Select with Restricted Assignments (Select-CopyFirst)

This section proposes a heuristic for Select-Opt that also uses periodic optimization with look ahead factor  $L$  but avoids the overhead of looking at all  $2^L$  possible assignments per delegation template. The idea is to use a different problem formulation that only makes a decision for the first time slot in  $T$  which significantly reduces the problem size. However, there are still multiple time slots per optimization period. To address this, the heuristic simply “copies” the decision of the first time slot to all subsequent time slots. This is why the approach is called Select-CopyFirst.

### 10.3.1 General Idea

Similar to Select-Opt, optimization is done periodically with a look ahead factor of  $L$ , i.e.,  $L$  time slots  $t_1, \dots, t_m$  (with  $m = L$ ) are considered in each optimization period. The key difference is that delegation template selection is restricted to the first time slot  $t_1$  while the remaining time slots  $t_2, \dots, t_m$  copy the decision from  $t_1$ . In other words: if delegation template  $d$  is selected in  $t_1$ , this template is selected for all  $t \in T$  (and vice versa if the template is not selected). The utilization and cost coefficients, however, are still calculated for all time slots, i.e., the utilization constraints must be fulfilled in all time slots and the cost is also minimized over all time slots.

Primary benefit of this restriction is the problem can be modeled as a 0-1 knapsack problem with two independent binary decision variables. There are only four cases for the selection of one delegation template  $d$  based on the history of the last optimization period (given as  $\langle H_d^X, H_d^F \rangle$ , see Def. 10.2) if selection decisions are restricted to  $t_1$ :

$$\left[ \begin{array}{l} \boxed{H_d^X = 0} \text{ and } X_{d,t_1} = 0 \quad \text{case ①} \\ \boxed{H_d^X = 0} \text{ and } X_{d,t_1} = 1 \quad \text{case ②} \\ \boxed{H_d^X = 1} \text{ and } X_{d,t_1} = 0 \quad \text{case ③} \\ \boxed{H_d^X = 1} \text{ and } X_{d,t_1} = 1 \quad \text{case ④} \end{array} \right] \quad (10.24)$$

If  $H_d^X$  is set to  $\boxed{0}$  (template  $d$  was not selected in the previous optimization period), the decision variable for time slot  $t_1$  (called  $X_{d,t_1}$  in Eq. (10.24)) can either be  $\boxed{0}$  or  $\boxed{1}$ . And the same two options exist for  $H_d^X = \boxed{1}$ . Note that the value of  $H_d^X$  is displayed in grey because this value represents the history of the last optimization period. This is very similar to the Select-Opt problem with a look ahead factor of  $L = 1$  with two important differences: the amount of choices in Select-CopyFirst is always limited to 4 and the coefficients can be calculated much easier.

### 10.3.2 Decision Variables

Decision variables for Select-CopyFirst are defined based on the value of  $H_d^X$ :

**Definition 10.5: Decision Variables for Select-CopyFirst**

Given a single delegation template  $d$  and the delegation template selection decision  $H_d^X$  from the previous optimization period, the **decision variables for Select-CopyFirst** are defined as:

$$X_d^{\boxed{0}} := \begin{cases} 1, & \text{if } H_d^X = \boxed{0} \text{ and template } d \text{ is selected in time slot } t_1 \\ 0, & \text{if } H_d^X = \boxed{0} \text{ and template } d \text{ is not selected in time slot } t_1 \end{cases}$$

and

$$X_d^{\boxed{1}} := \begin{cases} 1, & \text{if } H_d^X = \boxed{1} \text{ and template } d \text{ is selected in time slot } t_1 \\ 0, & \text{if } H_d^X = \boxed{1} \text{ and template } d \text{ is not selected in time slot } t_1 \end{cases}$$

These decision variables are defined for each delegation template  $d \in D_s$ . Undefined variables are interpreted as 0.

Note that these decision variables do not have a time slot index because they are by design only associated with the first time slot in  $T$ .

### 10.3.3 Utilization Coefficients

The utilization coefficients for a single delegation template  $d \in D_{s,t}$  represent the amount of flow rules that will be relocated to the remote switch in time slot  $t$  (“weight” in the knapsack terminology). For Select-CopyFirst, four different utilization coefficient are required, one for each case in Eq. (10.24). These are defined as follows:

$$\left[ \begin{array}{l} u_{d,t}^{\boxed{0}\boxed{0}} := 0 \quad \text{case } \textcircled{1} \\ u_{d,t}^{\boxed{0}\boxed{1}} := \sum_{q=t_1}^t \sum_{F_{d,q}} \lambda_{f,t}^i \quad \text{case } \textcircled{2} \\ u_{d,t}^{\boxed{1}\boxed{0}} := 0 \quad \text{case } \textcircled{3} \\ u_{d,t}^{\boxed{1}\boxed{1}} := \sum_{q=t_1}^t \left( \sum_{F_{d,q}} \lambda_{f,t}^i + \sum_{f \in H_d^F} \lambda_{f,t}^a \right) \quad \text{case } \textcircled{4} \end{array} \right] \quad (10.25)$$

These coefficients are much easier to calculate than the utilization coefficients from Select-Opt (see Sec. 10.2.3). In case  $\textcircled{1}$  and case  $\textcircled{3}$ , the delegation template is not selected in time slot  $t_1$  and the utilization is 0. Note that the delegation template is not selected for all time slots because of the decision from the first time slot ( $\boxed{0}$ ).



Similarly, in case ② and ④, the delegation template is selected in all time slots. In the former case, the delegation template was not selected in the previous optimization period ( $H_d^X = \mathbb{0}$ ), i.e., utilization coefficient  $u_{d,t}^{\mathbb{0}\mathbb{1}}$  must only consider newly installed flow rules between time slot  $t_1$  and time slot  $t$  ( $\sum_{q=t_1}^t \sum_{F_{d,q}} \lambda_{f,t}^i$ ). In the latter case, the delegation template was selected in the previous optimization period ( $H_d^X = \mathbb{1}$ ) and the flow rules in  $H_d^F$  that are still active in time slot  $t$  must be considered as well.

Using the above coefficients, it is possible to model the utilization constraint for Select-CopyFirst as follows:

$$u_{s,t}^{\text{Table}} - \sum_{d \in D_s} \left( (1 - H_d^X) * \boxed{X_d^{\mathbb{0}} * u_{d,t}^{\mathbb{0}\mathbb{1}}} + H_d^X * \boxed{X_d^{\mathbb{1}} * u_{d,t}^{\mathbb{1}\mathbb{1}}} \right) \leq c_s^{\text{Table}} \quad \forall_{t \in T} \quad (10.26)$$

### 10.3.4 Cost Coefficients

The cost coefficients represent the overhead associated with selecting a delegation template. This overhead consists of the amount of required aggregation and backflow rules ( $w_{d,t}^{\text{Table}}$ ), the amount of relocated bits ( $w_{d,t}^{\text{link}}$ ) and the amount of required control messages ( $w_{d,t}^{\text{Ctrl}}$ ).

#### 10.3.4.1 Rule Overhead

The rule overhead cost coefficients depend on the number of installed aggregation and backflow rules in the delegation switch. For Select-CopyFirst, these coefficients are calculated as follows:

$$\left[ \begin{array}{ll} w_d^{\mathbb{0}\mathbb{0}\text{Table}} := \boxed{0} & \text{case ①} \\ w_d^{\mathbb{0}\mathbb{1}\text{Table}} := \boxed{1} & \text{case ②} \\ w_d^{\mathbb{1}\mathbb{0}\text{Table}} := \boxed{0} & \text{case ③} \\ w_d^{\mathbb{1}\mathbb{1}\text{Table}} := \boxed{0} & \text{case ④} \end{array} \right] \quad (10.27)$$

#### 10.3.4.2 Link Overhead

The link overhead cost coefficients take the amount of relocated packets into account. These are the packets matched by the aggregation rule that have to be sent to the remote switch which is additional overhead for the infrastructure. For Select-CopyFirst, these coefficients are calculated as follows:

$$\left[ \begin{array}{l}
 w_d^{\boxed{0}\boxed{0}\text{Link}} := \boxed{0} \quad \text{case ①} \\
 w_d^{\boxed{0}\boxed{1}\text{Link}} := \boxed{\sum_{t \in T} \sum_{f \in F_{d,t}} \delta_{f,t} \lambda_{f,t}^i} \quad \text{case ②} \\
 w_d^{\boxed{1}\boxed{0}\text{Link}} := \boxed{0} \quad \text{case ③} \\
 w_d^{\boxed{1}\boxed{1}\text{Link}} := \boxed{\sum_{t \in T} \left( \left( \sum_{f \in F_{d,t}} \delta_{f,t} \lambda_{f,t}^i \right) + \left( \sum_{f \in H_d^F} \delta_{f,t} \lambda_{f,t}^a \right) \right)} \quad \text{case ④}
 \end{array} \right] \quad (10.28)$$

### 10.3.4.3 Control Message Overhead

The control message overhead cost coefficients consider the amount of control messages necessary if delegation template  $d$  is selected in time slot  $t$ . For Select-CopyFirst, these coefficients are calculated as follows:

$$\left[ \begin{array}{l}
 w_d^{\boxed{0}\boxed{0}\text{Ctrl}} := \boxed{0} \quad \text{case ①} \\
 w_d^{\boxed{0}\boxed{1}\text{Ctrl}} := \boxed{1 + \sum_{t \in T} \sum_{f \in F_{d,t}} \lambda_{f,t}^i} \quad \text{case ②} \\
 w_d^{\boxed{1}\boxed{0}\text{Ctrl}} := \boxed{1 + \sum_{f \in H_d^F} \lambda_{f,t}^a} \quad \text{case ③} \\
 w_d^{\boxed{1}\boxed{1}\text{Ctrl}} := \boxed{\sum_{t \in T} \sum_{f \in F_{d,t}} \lambda_{f,t}^i} \quad \text{case ④}
 \end{array} \right] \quad (10.29)$$

### 10.3.5 Problem Formulation for Select-CopyFirst

Based on the variables introduced in the previous sections, the Select-CopyFirst problem (which is restricted version of the Select-Opt problem) is defined as an integer linear program in the following way:

**Definition 10.6: Select-CopyFirst Problem**

**Inputs:** A single switch  $s \in S$ , flow table capacity  $c_s^{\text{Table}}$ , flow table utilization  $u_{s,t}^{\text{Table}}$ , set of consecutive time slots  $T$ , pre-processed utilization coefficients  $u_{d,t}$  for each  $d \in D_s$  and  $t \in T$ , pre-processed cost coefficients  $w_d$  for each  $d \in D_s$

**Output:** Set of selected delegation templates  $D_{s,t}^* \subseteq D_{s,t}$  for each time slot  $t \in T$

**Problem Formulation:**

$$\min_{X_d^{[0]}, X_d^{[1]}} \sum_{d \in D_s} \left( (1 - H_d^X) * \boxed{X_d^{[0]} * (w_d^{[0][1]\text{Table}} + w_d^{[0][1]\text{Link}} + w_d^{[0][1]\text{Ctrl}})} \right) \quad (10.30)$$

$$\begin{aligned} &+ H_d^X * \boxed{X_d^{[1]} * (w_d^{[1][1]\text{Link}} + w_d^{[1][1]\text{Ctrl}})} \\ &+ H_d^X * \boxed{(1 - X_d^{[1]}) * w_d^{[1][0]\text{Ctrl}}} \end{aligned} \quad (10.31)$$

$$\text{s.t. } u_{s,t}^{\text{Table}} - \sum_{d \in D_s} \left( (1 - H_d^X) * \boxed{X_d^{[0]} * u_{d,t}^{[0][1]}} \right. \\ \left. + H_d^X * \boxed{X_d^{[1]} * u_{d,t}^{[1][1]}} \right) \leq c_s^{\text{Table}} \quad \forall_{t \in T} \quad (10.32)$$

$$X_d^{[0]}, X_d^{[1]} \in \{0, 1\} \quad \forall_{d \in D_s} \quad (10.33)$$

This is a well-defined 0-1 knapsack formulation (more precisely, a two-dimensional multiple-choice knapsack problem) with two independent decision variables  $X_d^{[0]}$  and  $X_d^{[1]}$ . Independent means that either  $X_d^{[0]}$  or  $X_d^{[1]}$  are used based on the history of the previous optimization period ( $H_d^X$ ). This problem is much easier to solve compared to Select-Opt. There are only  $n = |D_s|$  decision variables and  $2 * n * L$  coefficient per optimization period.

### 10.3.6 Algorithm for Select-CopyFirst

The algorithm for Select-CopyFirst is identical to Alg. 10. Only difference is the coefficients in step 1 can be calculated more easily (as described above) and the optimization problem is changed from Select-Opt to Select-CopyFirst.

### 10.3.7 Summary

Select-CopyFirst is a heuristic for Select-Opt with a restricted solving space. The number of decision variables in the ILP is reduced from  $n * 2^L$  to  $n$  per optimization period where  $n$  is the number of delegation templates and  $L$  is the look ahead factor. The number of coefficients is reduced to  $2 * n * L$ . This allows for faster solving times and for much faster model construction times.

## 10.4 DT-Select with Greedy Strategy (Select-Greedy)

This section proposes a heuristic for Select-CopyFirst that uses a greedy selection strategy. This third algorithm is included here for two reasons. First, the previous algorithms Select-Opt and Select-CopyFirst fail if the problem gets infeasible, e.g., if the look ahead factor is too small. This requires a fallback solution that works with every scenario, i.e., is guaranteed to be feasible. And secondly, a greedy approach is a good baseline to evaluate the performance of the two other algorithms.

### 10.4.1 General Idea

Select-Greedy was first discussed in [BZ16]. This section presents an updated version of that algorithm based on delegation templates, utilization coefficients, and cost coefficients but the basic idea is still the same. Select-Greedy works very similar to Select-CopyFirst, i.e., is executed periodically and the decision for the first time slot is “copied” to all other time slots in  $T$ . As a result, the same coefficients can be used. For each optimization period<sup>2</sup> the algorithm distinguishes between two simple states:

- (1) If the current flow table utilization  $u_{s,t}^{\text{Table}}$  minus the utilization of all currently selected delegation templates is above the capacity  $c_s^{\text{Table}}$ , additional delegation templates are selected until the bottleneck is mitigated.
- (2) And if  $u_{s,t}^{\text{Table}}$  minus the utilization of all currently selected delegation templates is below capacity  $c_s^{\text{Table}}$ , Select-Greedy tries to unselect delegation templates as long as the capacity is not exceeded.

To include the costs, the delegation templates for both cases are sorted according to the cost coefficients. This is a much simpler scheme compared to Select-CopyFirst because the greedy algorithm can only take one of two actions: select or unselect. It will never select *and* unselect delegation templates in the same optimization period – which is

---

<sup>2</sup>The iterations of Select-Greedy are also called optimization periods here for simplicity, despite the fact that no real optimization takes place. This is easier because Def. 10.2 and the same notation as in Select-Opt and Select-CopyFirst can be used.

possible with the ILP-based algorithms if “switching” to another optimal selection leads to smaller costs.

### 10.4.2 Utilization Coefficients

The utilization coefficients for Select-Greedy are calculated in the same way as the utilization coefficients for Select-CopyFirst. However, it is not required to distinguish between different coefficients because Select-Greedy only needs the utilization if a previously unselected delegation template is selected or a selected delegation template is unselected. So there is only a single choice based on whether the delegation template was selected in the previous optimization period or not.

---

**Algorithm 11:** Utilization and cost coefficients for Select-Greedy

---

```

1 Function calculate_coefficients():
2   for  $t \in T$  do
3     for  $d \in D_{s,t}$  do
4       if  $H_d^X = 1$  then
5          $u_{d,t} \leftarrow u_{d,t}^{\boxed{1}\boxed{1}}$ 
6       else
7          $u_{d,t} \leftarrow u_{d,t}^{\boxed{0}\boxed{1}}$ 
8       end
9     end
10  end
11  for  $d \in D_s$  do
12    if  $H_d^X = 1$  then
13       $w_d \leftarrow \omega_{DTS}^{Table} w_d^{\boxed{1}\boxed{1}Table} + \omega_{DTS}^{Link} w_d^{\boxed{1}\boxed{1}Link} + \omega_{DTS}^{Ctrl} w_d^{\boxed{1}\boxed{1}Ctrl}$ 
14    else
15       $w_d \leftarrow \omega_{DTS}^{Table} w_d^{\boxed{0}\boxed{1}Table} + \omega_{DTS}^{Link} w_d^{\boxed{0}\boxed{1}Link} + \omega_{DTS}^{Ctrl} w_d^{\boxed{0}\boxed{1}Ctrl}$ 
16    end
17  end
18 end

```

---

The concrete calculations are shown in Alg. 11. If a delegation template  $d$  was selected in the previous optimization period ( $H_d^X = 1$ ), the value for  $u_{d,t}$  is calculated according to Eq. (10.25) case ④ which is  $u_{d,t}^{\boxed{1}\boxed{1}}$ . And if the delegation template was not selected ( $H_d^X = 0$ ), the utilization is calculated according to Eq. (10.25) case ② which is  $u_{d,t}^{\boxed{0}\boxed{1}}$ .

### 10.4.3 Cost Coefficients

The cost coefficients for Select-Greedy are also calculated in the exact same way as the cost coefficients for Select-CopyFirst and will thus not be explained again in all details. The concrete calculations are shown in Alg. 11.

If a delegation template  $d$  was selected in the previous optimization period ( $H_d^X = 1$ ), the cost is only relevant if the greedy algorithm should unselect the template. In this case, the cost coefficients are used to prioritize templates with high costs (these should be unselected first). This is  $w_d^{[1][1]}$ , i.e., the cost to be expected if the delegation template is selected again (= not unselected). And if the delegation template was not selected ( $H_d^X = 0$ ), the cost coefficients are used to prioritize templates with lowest cost (these should be selected first). This is  $w_d^{[0][1]}$ , i.e., the cost to be expected if the delegation template is selected. The non-negative  $\omega_{\text{dTS}}$ -weights balance the different cost coefficients against each other.

### 10.4.4 Algorithm for Select-Greedy

The details for Select-Greedy are given in Alg. 12. Goal, input and output are similar to Select-CopyFirst. The inner structure, however, is different. The algorithm is structured into five steps that are executed once per optimization period. In the first step (lines 6-7), the algorithm is initialized with the selections of the last period based on  $H_d^X$ . This means that, if no delegation templates are selected or unselected in the current optimization period, all selections from the previous optimization period ( $D_{s,t}^*$ ) are used again. This is fundamentally different from the ILP-based approaches above where  $D_{s,t}^*$  is a direct result of the respective optimization problem.

In the second step (lines 8-9) the algorithm explicitly calculates whether there is a bottleneck and a new delegation template must be selected ( $\Delta > 0$ ) or whether already selected delegation templates are unselected because the delegation switch has free flow table capacity ( $\Delta < 0$ ). This is also different from Select-Opt and Select-CopyFirst where bottleneck mitigation is modeled as a capacity constraint in the optimization problem which is not possible here. Variables  $t_{\max}$  and  $u_{\max}$  represent the time slot where the maximum utilization occurred within the set of considered time slots  $T$ . This is how the greedy algorithm accounts for the multi period nature of the problem. Delegation templates are selected in such a way that the worst case situation across all considered time slots is taken into account.

The third step is based on the value of  $\Delta$ . In case 3a with  $\Delta > 0$  (lines 11-16), the algorithm only works with the delegation templates that are currently not selected, i.e., with  $D_{s,t} \setminus D_{s,t}^*$ . These templates are sorted according to the cost coefficients so that the

**Algorithm 12:** Select-Greedy**Data:** Same input as Select-CopyFirst problem (Def. 10.6), Look ahead factor  $L$ **Result:** For each optimization period: set of selected delegation templates

$$D_{s,t}^* \subseteq D_{s,t} \text{ for each time slot } t \in T \text{ with } |T| = L$$

```

1 Function select_greedy():
2   for  $d \in D_s$  do
3      $H_d^X \leftarrow 0$   $H_d^F \leftarrow \emptyset$ 
4   end
5   if  $u_{s,t}^{\text{Table}} > c_s^{\text{Table}}$  or  $\sum_{d \in D_s} H_d^X > 0$  then
6     /* Step 1: calculate coefficients and initiate  $D_{s,t}^*$  */
7      $u_{d,t}, w_d \leftarrow \text{calculate\_coefficients}()$  // Alg. 11
8      $D_{s,t}^* \leftarrow \{d \mid H_d^X = 1, d \in D_{s,t}\}$ 
9     /* Step 2: case differentiation based on utilization */
10     $u_{\max}, t_{\max} \leftarrow \max_{t \in T} (\langle u_{s,t}^{\text{Table}}, t \rangle)$ 
11     $\Delta = u_{\max} - c_s^{\text{Table}} - \sum_{d \in D_{s,t}^*} u_{d,t_{\max}}$ 
12    if  $\Delta > 0$  then
13      /* Step 3a: select new delegation templates as required */
14      sort  $D_{s,t} \setminus D_{s,t}^*$  according to  $w_d$ 
15      for  $d \in D_{s,t} \setminus D_{s,t}^*$  do
16        if  $\Delta - u_{d,t_{\max}} > 0$  then
17           $D_{s,t}^* \leftarrow D_{s,t}^* \cup \{d\}$   $H_d^X \leftarrow 1$   $\Delta \leftarrow \Delta - u_{d,t_{\max}}$ 
18        end
19      end
20    else
21      /* Step 3b: unselect as many delegation templates as possible */
22      sort  $D_{s,t}^*$  according to  $w_d$ 
23      for  $d \in D_{s,t}^*$  do
24        if  $\Delta + u_{d,t_{\max}} < 0$  then
25           $D_{s,t}^* \leftarrow D_{s,t}^* \setminus \{d\}$   $H_d^X \leftarrow 0$   $\Delta \leftarrow \Delta + u_{d,t_{\max}}$ 
26        end
27      end
28    end
29    /* Step 4: update history and continue periodic optimization */
30    for  $d \in D_{s,t}$  do
31       $\langle H_d^X, H_d^F \rangle \leftarrow \text{update\_history}()$  // Def. 10.2
32    end
33    wait until next optimization period and start again in line 5
34  end
35 end

```

template with lowest cost is selected first. Selection continues until the utilization with flow delegation is below the capacity. In case 3b with  $\Delta < 0$  (lines 18-23), the algorithm only works with delegation templates that are already selected, i.e., with  $D_{s,t}^*$  (note that  $D_{s,t}^*$  was prepared in step 1 with data from the previous optimization period). These templates are sorted according to the cost coefficients so that the template with highest cost is selected first. The algorithm continues to unselect delegation templates as long as the utilization is below the capacity.

Step 4 is again identical to the previous algorithms: the history variables are updated (in this case only  $H_d^F$  because  $H_d^X$  was already updated in line 14 or 21) and the algorithm continues processing in the next optimization period.

## 10.5 Conclusion

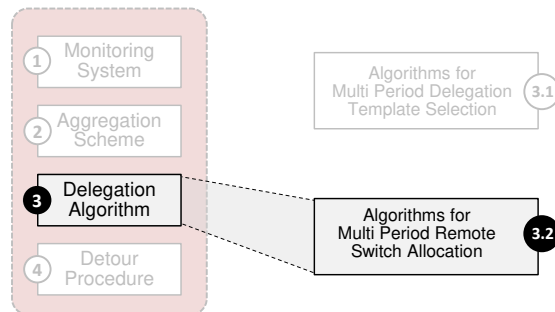
This chapter introduces three algorithms for the multi period DT-Select problem. All three use the periodic approach introduced in Sec. 10.2.1.2.

- **Select-Opt** pre-calculates all possible combinations of decisions for a fixed (small) amount of time slots (so-called assignments) and is based on a multi-dimensional multiple-choice knapsack formulation.
- **Select-CopyFirst** is a heuristic for Select-Opt based on a two-dimensional knapsack formulation which significantly reduces the problem space. The algorithm only make a decision for the first time slot and then “copies” this decision for all following time slots.
- **Select-Greedy** is a greedy heuristic for Select-CopyFirst that considers only two simple states: above capacity (select new delegation templates) and below capacity (unselect existing delegation templates).



# Multi Period Remote Switch Allocation

---



This chapter introduces an algorithm for multi period remote switch allocation. And because only a single algorithm for remote switch allocation is discussed here, this algorithm is simply called RS-Alloc. It uses a linear assignment-based knapsack problem with pre-calculation of all possible remote switch allocations for each allocation job, similar to the approach used in the last chapter.

The chapter is structured as follows. Sec. 11.1 summarizes the multi period RS-Alloc. The proposed algorithm to address this problem is then discussed in Sec. 11.2. Sec. 11.3 presents a pre-processing step to reduce the number of considered allocation assignments in order to improve scalability.

## 11.1 Problem

The multi period remote switch allocation problem can be written as an integer linear program in the following way:

**Recap: Multi Period Remote Switch Allocation Problem (see Def. 9.8)**

**Inputs:** A set of switches  $S$ , set of consecutive time slots  $T$ , set of allocation jobs  $J_t$  derived from the output of DT-Select for each  $t \in T$ , flow table capacity  $c_r^{\text{Table}}$  for each switch, link capacity  $c_{s_j \rightarrow r}^{\text{Link}}$  for each link, utilization coefficients  $u_{j,t}$ , overhead coefficients  $w_{j \rightarrow r,t}$ , and remote sets  $R_{j,t}$  for each  $j \in J_t$  and each  $t \in T$

**Output:** A set of selected and allocated delegation templates  $D_t^{**}$  for each  $t \in T$

**Problem Formulation:**

$$\min_{Y_{j \rightarrow r,t}} \sum_{t \in T} \sum_{j \in J_t} \sum_{r \in R_{j,t}} w_{j \rightarrow r,t} Y_{j \rightarrow r,t} \quad (\text{see 9.37})$$

$$\text{s.t.} \quad \sum_{r \in R_{j,t}} Y_{j \rightarrow r,t} = 1 \quad \forall_{t \in T} \quad \forall_{j \in J_t} \quad (\text{see 9.38})$$

$$u_{r,t}^{\text{Table}} + \sum_{j \in J_t} u_{j,t}^{\text{Table}} Y_{j \rightarrow r,t} \leq c_r^{\text{Table}} \quad \forall_{t \in T} \quad \forall_{r \in S} \quad (\text{see 9.39})$$

$$u_{s_j \rightarrow r,t}^{\text{Link}} + \sum_{j \in J_t} u_{j,t}^{\text{Link}} Y_{j \rightarrow r,t} \leq c_{s_j \rightarrow r}^{\text{Link}} \quad \forall_{t \in T} \quad \forall_{r \in S} \quad (\text{see 9.40})$$

$$u_{r \rightarrow s_j,t}^{\text{Link}} + \sum_{j \in J_t} u_{j,t}^{\text{Link}} Y_{j \rightarrow r,t} \leq c_{r \rightarrow s_j}^{\text{Link}} \quad \forall_{t \in T} \quad \forall_{r \in S} \quad (\text{see 9.41})$$

$$Y_{j \rightarrow r,t} \in \{0, 1\} \quad \forall_{t \in T} \quad \forall_{r \in S} \quad \forall_{j \in J_t} \quad (\text{see 9.42})$$

The challenge associated with the above formulation is that the allocation of a remote switch to one delegation template  $d \in D_s$  can change between time slots (see Sec. 9.5.7). This results in quadratic complexity because of the control message overhead cost coefficients  $w_{j \rightarrow r,t}^{\text{Ctrl}}$  which are the most important coefficient in this context. A realistic objective function for the above problem would look like this:

$$\begin{aligned} \min_{Y_{j \rightarrow r,t}} \quad & \sum_{t \in T} \sum_{j \in J_t} \sum_{r \in R_{j,t}} Y_{j \rightarrow r,t} \left( \omega_{\text{RSA}}^{\text{Table}} w_{j \rightarrow r,t}^{\text{Table}} + \omega_{\text{RSA}}^{\text{Link}} w_{j \rightarrow r,t}^{\text{Link}} + \omega_{\text{RSA}}^{\text{Static}} w_{r,t}^{\text{Static}} \right) \quad (\text{see 9.43}) \\ & + \sum_{t \in T} \sum_{j \in J_t} \sum_{r \in R_{j,t}} (1 - Y_{j \rightarrow r,t-1}) Y_{j \rightarrow r,t} \omega_{\text{RSA}}^{\text{Ctrl}} w_{j \rightarrow r,t}^{\text{Ctrl}} \end{aligned}$$

Eq. (9.43) requires  $|J| * |T| * |R|$  quadratic terms where  $|J|$  is the number of all allocation jobs per time slot,  $|T|$  is the number of considered time slots and  $|R|$  is the number of remote switch options for one allocation job. This is a scalability problem because  $|R|$  and  $|J|$  can both be large, even if only a small number of time slots is considered.

## 11.2 RS-Alloc with Assignments

RS-Alloc uses the same periodic optimization approach that was used for the multi-period DT-Select algorithms. This means an optimal result is returned for each individual optimization period but optimality is not guaranteed across different optimization periods.

### 11.2.1 General Idea

RS-Alloc translates the quadratic facility location problem shown above in Sec. 11.1 into a linear assignment-based knapsack problem. The general idea is very similar to the approach described in Sec. 10.2.1: RS-Alloc pre-calculates sets of all possible remote switch allocations for each allocation job. Before it is explained how assignments and periodic optimization work in the context of RS-Alloc, the next section first introduces some additional terminology.

#### 11.2.1.1 Additional Terminology

The definition of allocation jobs in Sec. 9.16 only considers a single time slot. Recall from that definition, that every job in set  $J_t$  represents one delegation template  $d_{j,t}$  for one delegation switch  $s_j$  in time slot  $t$ . Delegation template  $d_{j,t}$  consists of an aggregation match  $\vec{m}_j$  and a conflict-free cover set  $F_{j,t}$ .

Because of the ingress port scheme, there is a fixed set of  $n$  aggregation matches – one per ingress port – which means that delegation templates with the same aggregation match can and will be selected by DT-Select in different time slots. For RS-Alloc, it is important to know whether the same delegation template (same aggregation match, same delegation switch) is selected by DT-Select in multiple consecutive time slots. This is expressed with a new construct called  $T_j$ :

**Definition 11.1: Set  $T_j$**

$T_j \subseteq T$  is a set of consecutive time slot so that allocation job  $j$  is active in all time slots in  $T_j$ . Allocation job  $j$  is **active** in time slot  $t$  if  $J_t$  (see Def. 9.4) contains a delegation template with aggregation match  $\vec{m}_j$  that is associated with delegation switch  $s_j$ .

The first time slot in  $T_j$  is identical to the first time slot in  $T$  where a delegation template  $d \in D_s$  is selected. And the same delegation template is also selected in all other time slots in  $T_j$ . This concept is illustrated in Fig. 11.1 with an example.

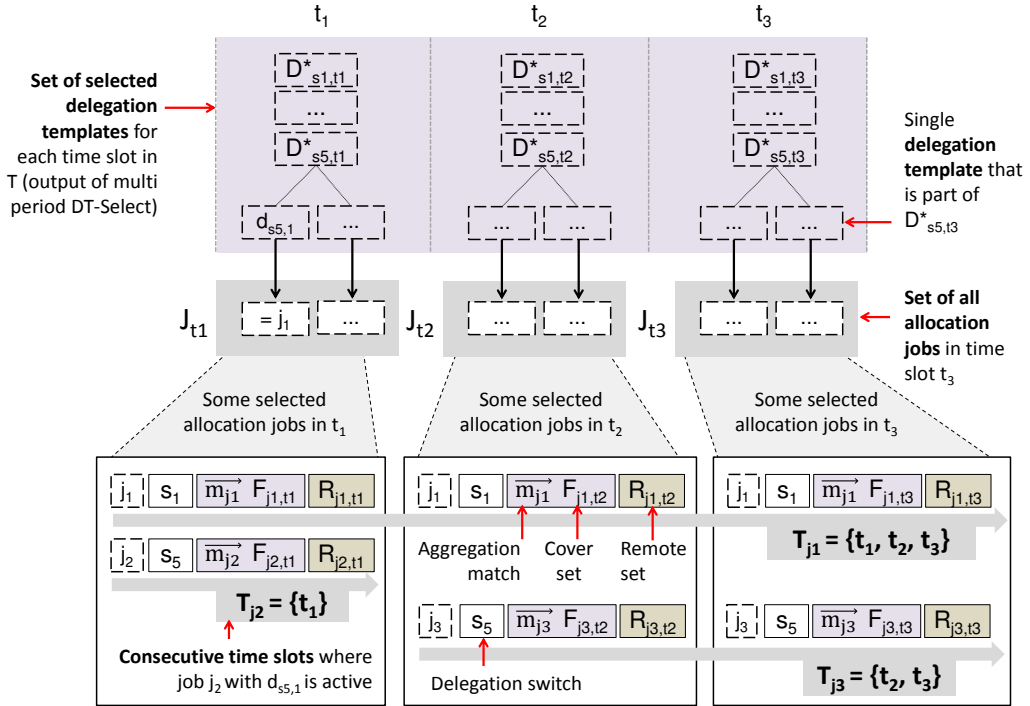


Figure 11.1: Illustration of set  $T_j$

The example shows three time slots  $T = \{t_1, t_2, t_3\}$  that are considered by DT-Select and RS-Alloc. The result of DT-Select for all three time slots is shown in the top. It consists of multiple sets of selected delegation templates ( $D_{s,t}^*$ ). RS-Alloc takes these inputs and creates one set of allocation jobs per time slot ( $J_t$ ). The bottom of the figure shows the  $J_t$ -sets in detail. There are two allocation jobs in each set ( $\bar{j}_1$  and  $\bar{j}_2$  in  $t_1$  and  $\bar{j}_1$  and  $\bar{j}_3$  in  $t_2$  and  $t_3$ ). The three small boxes next to  $\bar{j}$  refer to the delegation switch ( $s_j$ ), the delegation template ( $d_{s,t} = \langle \vec{m}_j, F_{j,t} \rangle$ ), and the remote set ( $R_{j,t}$ ) associated with this allocation job.

$T_j$  now indicates in which time slot allocation job  $j$  is active. Allocation job  $j_1$ , for example, is part of  $J_{t_1}, J_{t_2}$  and  $J_{t_3}$ , i.e. this job is active in  $T_{j_1} = \{t_1, t_2, t_3\}$ . Allocation job  $j_2$  is only part of  $J_{t_1}$ , i.e., is active in  $T_{j_2} = \{t_1\}$ . And allocation job  $j_3$  is part of  $J_{t_2}$  and  $J_{t_3}$ , i.e., is active in  $T_{j_3} = \{t_2, t_3\}$ . It is assumed here for simplicity that all time slots in  $T_j$  are consecutive. If the same delegation template is used again later (e.g., if  $j_2$  is used again in time slot  $t_3$ ), there would be two different sets of  $T_j$ .

Important: the time slots in  $T_j$  use the same notation like before, i.e.,  $t_1 \in T_j$  will be referred to as the first time slot in  $T_j$  and  $t_m$  will be referred to as the last time slot in  $T_j$  (even if the first / last time slot in  $T_j$  refers to a different time slot with respect to set  $T$ !).

In addition to  $T_j$ , a second helper construct is needed that collects all the different allocation jobs for a set of time slots  $T$  (without duplicates):

**Definition 11.2: Set  $J_T$**

The **set**  $J_T$  for a set of consecutive time slots  $T$  contains all allocation jobs that are active in one of the time slots in  $T$ :

$$J_T := \bigcup_{t \in T} J_t \quad (11.1)$$

Allocation jobs with the same delegation switch ( $s_j$ ) and aggregation match ( $\vec{m}_j$ ) that are active in multiple time slots in  $T$  are only added once.

### 11.2.1.2 Allocation Assignments

Instead of modeling the allocations in direct dependence of decision variables  $Y_{j \rightarrow r, t}$ , they are defined over a set of allocation assignments where each allocation assignment contains a fixed series of remote switches with exactly one remote switch to be used in each time slot.

**Definition 11.3: Allocation Assignment Set**

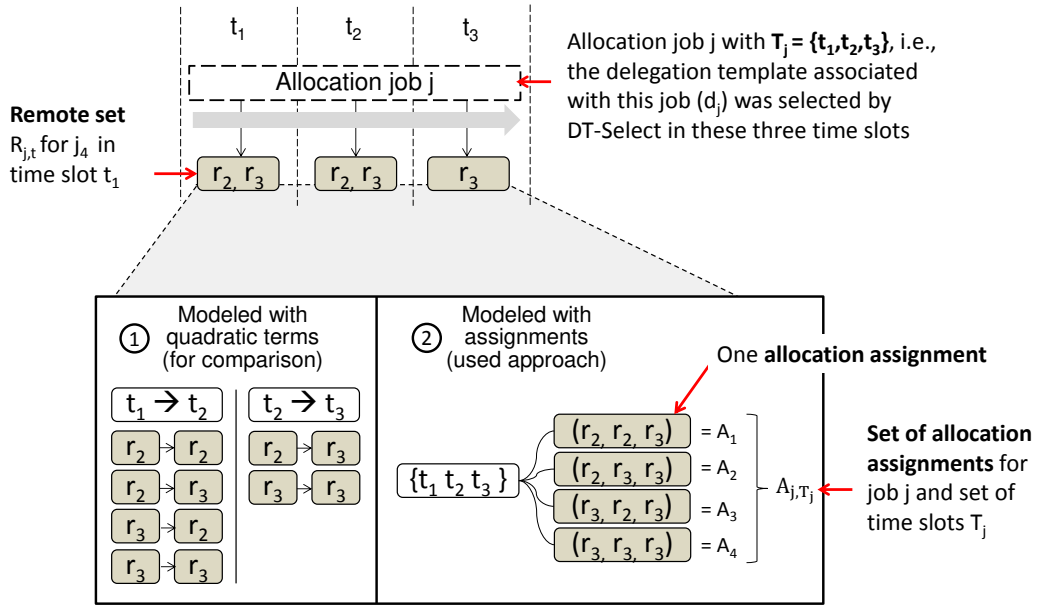
An **allocation assignment set**  $A_{j, T_j}$  for a single allocation job  $j \in J_T$  and a set of consecutive time slots  $T_j = \{t_1, \dots, t_m\}$  is defined as:

$$A_{j, T_j} := R_{j, t_1} \times R_{j, t_2} \times \dots \times R_{j, t_m} \quad (11.2)$$

$T_j$  is specified in Def. 11.1 above.  $R_{j, t}$  is the remote set of allocation job  $j$  in time slot  $t$ . The set of allocation assignments is calculated as the Cartesian power of the  $R_{j, t}$ , i.e., the set of all ordered pairs that can be build with the  $|T_j|$  involved remote sets. Each entry  $A_i := (r_1, \dots, r_m)$  in  $A_{j, T_j}$  is called an **allocation assignment** that consists of  $|T_j| = m$  remote switches, i.e., one specific remote switch allocation option for each time slot in  $|T_j|$ .

This definition is applied to each allocation job in  $J_T$ . Fig. 11.2 gives an example. The top of the figure shows one single allocation job  $j$  that is active in three time slots  $T_j = \{t_1, t_2, t_3\}$ .

The three smaller boxes (such as  $(r_2, r_3)$ ) represent allocation options from the remote set  $R_{j, t}$  in the given time slot. At time slot  $t_1$ ,  $r_2$  and  $r_3$  can be selected as a remote switch,



**Figure 11.2:** Assignment concept for RS-Alloc

i.e., both switches provide the necessary free flow table capacity and the link between  $s_j$  and either of the two options in  $R_{j,t_1}$  has enough free capacity. The second time slot is identical to the first. In the third time slot, only a single remote switch option ( $r_3$ ) remains. The backup switch that would usually be present in each set  $R_{j,t}$  (see Def. 9.5) is omitted here.

The bottom of the figure then illustrates the conceptual difference between the quadratic approach ① and the linearized approach with allocation assignments ②. The quadratic approach considers all pairs of remote switch options for two consecutive time slots to calculate cost coefficients. In this example, there are six cases to consider in total. There are four cases for  $t_1 \rightarrow t_2$  because of  $R_{j,t_1} \times R_{j,t_2} = \{(r_2, r_2), (r_2, r_3), (r_3, r_2), (r_3, r_3)\}$ . And there are two cases for  $t_2 \rightarrow t_3$  because of  $R_{j,t_2} \times R_{j,t_3} = \{(r_2, r_3), (r_3, r_3)\}$ . With respect to this one job, this will result in 6 quadratic terms for the control message cost coefficients and  $3 * \sum_{t \in T_j} |R_{j,t}| = 15$  linear terms for the other cost coefficients.

The assignment-based approach on the right side first creates a set of allocation assignments  $A_{j,T_j}$  for allocation job  $j$ . This set represents all the possible allocations of remote switches that can occur for  $t \in T_j$ . According to Def. 11.3, the assignment set is calculated

as the Cartesian power of the  $R_{j,t}$  which results in four individual allocation assignments here:

$$\begin{aligned} A_{j,T_j} &= R_{j,t_1} \times R_{j,t_2} \times R_{j,t_3} \\ &= \{r_2, r_3\} \times \{r_2, r_3\} \times \{r_3\} \\ &= \{(r_2, r_2, r_3), (r_2, r_3, r_3), (r_3, r_2, r_3), (r_3, r_3, r_3)\} \end{aligned}$$

Allocation assignment  $(r_2, r_2, r_3)$  represents the case where job  $j$  is served by remote switch  $r_2$  in the first two time slots of  $T_j$  and by  $r_3$  in the final time slot (because there are no other options for  $t_3$ ). The three other elements from the assignment follow the same logic.  $A_2 = (r_2, r_3, r_3)$  and  $A_3 = (r_3, r_2, r_3)$  represent the cases where the remote switch is changed between  $t_1$  and  $t_2$ .  $A_4 = (r_3, r_3, r_3)$  represents the case where  $j$  is served by  $r_3$  in all three time slots.

The benefits of the second approach are twofold. First, the assignment-based formulation results in a linear problem which is easier to solve in general. In addition, it can not only include quadratic dependencies, but basically all kinds of non-linear dependencies that are restricted to a single allocation job. Secondly, it is easier to rate different remote switch allocation options because the whole job (all time slots) is considered as one unit. This allows for easy and intuitive cost formulations. The major drawback, on the other hand, is the increased number of overall terms and constraints. While this example actually leads to less terms for the assignment-based approach, the situation for larger problem instances is quite different. Both approaches scale with the number of jobs  $|J|$ , the number of time slots  $|T_j|$  where the job is active and the amount of remote switch options  $|R_{j,t}|$ . The number of terms needed for the quadratic approach is  $\sum_{j \in J} \sum_{t \in T_j} |R_{j,t}|^2$  while the number of terms for the assignment-based approach is  $\sum_{j \in J} \prod_{t \in T_j} |R_{j,t}|$ , i.e., quadratic vs. exponential growth. This problem can be addressed with a pre processing step to only include the most promising assignments, which is shown later in Sec. see 9.16.

### 11.2.1.3 Periodic Optimization

Periodic optimization for RS-Alloc works similar as periodic optimization for DT-Select which was described in detail in Sec. 10.2.1.2. In each optimization period, a limited future horizon of  $L$  time slots is considered together with the status from the last optimization period (referred to as history  $H_j$ ). Main difference is that the history is defined for individual allocation jobs instead of all delegation templates.

**Definition 11.4: History of an RS-Alloc optimization period**

Let  $t_0$  be the time slot of the previous optimization period, i.e., the time slot prior to  $t_1 \in T$ . The **history  $H_j$  of an RS-Alloc optimization period** for allocation job  $j$  is then defined as a triple  $\langle H_j^X, H_j^F, H_j^r \rangle$  where  $H_j^X \in \{0, 1\}$  indicates whether delegation template  $d_{j,t_0}$  was selected in time slot  $t_0$ ,  $H_j^F \subseteq F_{s_j,t_0}$  is a set of all active flow rules that are relocated at the end of time slot  $t_0$  and  $H_j^r \in R_{j,t_0}$  is the remote switch that was allocated to  $d_{j,t_0}$  (this variable is only defined if  $H_j^X = 1$ ).

$$\forall_{j \in J_T} : H_j^X := \begin{cases} 1, & d_{j,t_0} \in D_{s_j,t_0}^* \\ 0, & d_{j,t_0} \notin D_{s_j,t_0}^* \end{cases} \quad (11.3)$$

$$\forall_{j \in J_T} : H_j^F := \{ f \mid f \in F_{s_j,t_0} \text{ and } f \text{ is relocated} \} \quad (11.4)$$

$$\forall_{j \in J_T} : H_j^r := \begin{cases} r, & \langle d_{j,t_0}, r \rangle \in D_{t_0}^{**} \\ \text{undefined}, & \langle d_{j,t_0}, r \rangle \notin D_{t_0}^{**} \end{cases} \quad (11.5)$$

Eq. (11.3) and Eq. (11.5) can be extracted directly from the result sets of time slot  $t_0$  ( $D_{s_j,t_0}^*$  and  $D_{t_0}^{**}$ ). Eq. (11.4) can be calculated recursively between optimization periods using the formula from Def. 10.2.

It is important to note that  $t_0$  in the above definition always refers to the previous optimization period, i.e., the time slot prior to  $t_1 \in T$  and not the time slot prior to  $t_1 \in T_j$ . In other words: the history is only defined for those allocation jobs that were already active in the previous optimization period. Further note that  $d_{j,t_0}$  and all  $d_{j,t}$  for  $t \in T_j$  refer to the same delegation template, i.e., they all have the same aggregation match  $\vec{m}_j$  and the same delegation switch  $s_j$ .

### 11.2.2 Decision Variables

Decision variables for RS-Alloc are defined with respect to allocation assignments instead of time slots:



**Definition 11.5: Decision Variables for RS-Alloc**

Given an allocation job  $j$  and an allocation assignment set  $A_{j,T_j}$  (see Def. 11.3), the **decision variables for Select-Opt** are defined as:

$$Y_{j,A} := \begin{cases} 1, & A \text{ is the selected allocation assignment} \\ 0, & A \text{ is not the selected allocation assignment} \end{cases} \quad (11.6)$$

These decision variables are defined for each allocation job  $j \in J_T$  (see Def. 11.2) and all allocation assignments in  $A_{j,T_j}$ . Undefined variables are interpreted as 0.

There are  $\prod_{t \in T_j} |R_{j,t}|$  allocation assignments per allocation job, one for each combination of remote switch options for all time slots where  $j$  is active. The idea is to select exactly one out of all these allocation assignments to make a decision for all remote switch allocations in  $T_j$ . This can be achieved with the following simple constraint:

$$\sum_{A \in A_{j,T_j}} Y_{j,A} = 1 \quad \forall_{j \in J_T} \quad (11.7)$$

As a consequence, the utilization and cost coefficients have to be remodeled to work with allocation assignments (and the periodic optimization approach). It is explained in the following two sections how this is done.

### 11.2.3 Utilization Coefficients

The utilization coefficients  $u_{j,t}$  represent the “to be fulfilled utilization demand” of allocation job  $j$  in time slot  $t$ . This means the remote switch provides enough free flow table capacity for all relocated flow rules from delegation template  $d_{j,t}$  and the link to the remote switch has enough free bandwidth for all packets that are relocated.

The utilization coefficients for RS-Alloc ( $u_{j,t}^{\text{Table}}$  and  $u_{j,t}^{\text{Link}}$ ) can be calculated easily because allocation assignments always represent a consecutive set of selected delegation templates (by definition). If job  $j$  was active in  $T_j = \{t_1, \dots, t_4\}$ , all four corresponding delegation

templates  $d_{j,t_1}, \dots, d_{j,t_4}$  were selected by DT-Select (the previous step in the delegation algorithm). Because of this, the two utilization coefficients are calculated as follows:

$$u_{j,t}^{\text{Table}} := \left( \sum_{q=t}^{t_1} \sum_{f \in F_{d_j,q}} \lambda_{f,q}^i * \lambda_{f,t}^a \right) + \left( H_j^X * \sum_{f \in H_j^F} \lambda_{f,t}^a \right) \quad (11.8)$$

$$u_{j,t}^{\text{Link}} := \left( \sum_{q=t}^{t_1} \sum_{f \in F_{d_j,q}} \lambda_{f,q}^i \delta_{f,t} \right) + \left( H_j^X * \sum_{f \in H_j^F} \delta_{f,t} \right) \quad (11.9)$$

Eq. (11.8) calculates the flow table utilization coefficients. The first bracket represents all newly installed flow rules that are relocated to the remote switch between  $t_1$  and  $t_m$  (possible because  $d_{j,t}$  is selected in all time slots). The second bracket takes care of flow rules from previous optimization periods. Note that this part is only included if  $H_j^X = 1$ , i.e., the allocation job was already active in the previous period. In this case, all relocated flow rules from the previous period (given as set  $H_j^F$ ) that are still active in time slot  $t$  ( $\lambda_{f,t}^a = 1$ ) are also considered for this utilization coefficient. This is basically a simplified version of the extended utilization formula from Eq. (10.9) that was used by the assignment-based DT-Select algorithm. Eq. (11.9) is identical except that  $\lambda_{f,t}^a$  is replaced with  $\delta_{f,t}$ .

Note that the two above coefficients are defined independently of the allocation assignment because the different remote switch allocation options do not change the utilization (neither the number of flow rules nor the number of bits to be relocated does depend on the used remote switch). This is a fundamental difference to the assignments used with DT-Select (Def. 10.1) where the assignment value for a specific time slot  $t$  inside an assignment could be set to  $\boxed{0}$  which did change the utilization. For RS-Alloc, however, this is not relevant because delegation template  $d_j$  was selected in all time slots in  $T_j$ .

The utilization constraints, however, do depend on the allocation assignments. This is because the constraints are checked for each remote switch  $r$  independently and it makes a difference whether  $r$  is allocated to a certain delegation template or not: the flow table utilization in  $r$  is higher in the first case and unchanged in the second case. Because allocation assignments represent a set of remote switches and not a set of binary variables (as it was the case with DT-Select), an additional helper construct is required to make use of the assignments:

**Definition 11.6: Helper Variable  $\alpha$** 

Given is an allocation assignment  $A \in A_{j,T_j}$  with  $A = (a_1, \dots, a_t, \dots, a_m)$  and  $a_t \in R_{j,t}$ . The binary **helper variable**  $\alpha_{j \rightarrow r,t}$  is then defined as:

$$\alpha_{j \rightarrow r,t} := \begin{cases} 1, & a_t = r \\ 0, & \text{otherwise} \end{cases} \quad (11.10)$$

This variable is set to 1 if remote switch  $r$  is used in time slot  $t$  of allocation assignment  $A$ . The variable is defined for all allocation jobs  $j \in J_T$ , all time slots  $t \in T_j$  and all remote switches in remote set  $R_{j,t}$ . Undefined variables are interpreted as 0.

The values for  $\alpha_{j \rightarrow r,t}$  can be directly deduced from the assignments. The following table shows an example mapping for  $T_j = \{t_1, t_2\}$  and  $R_{j,t_1} = R_{j,t_2} = \{r_1, r_2\}$ :

$A_1 = (r_1, r_1)$	$A_2 = (r_1, r_2)$	$A_3 = (r_2, r_1)$	$A_4 = (r_2, r_2)$
$\alpha_{j \rightarrow r_1, t_1} = 1$	$\alpha_{j \rightarrow r_1, t_1} = 1$	$\alpha_{j \rightarrow r_1, t_1} = 0$	$\alpha_{j \rightarrow r_1, t_1} = 0$
$\alpha_{j \rightarrow r_2, t_1} = 0$	$\alpha_{j \rightarrow r_2, t_1} = 0$	$\alpha_{j \rightarrow r_2, t_1} = 1$	$\alpha_{j \rightarrow r_2, t_1} = 1$
$\alpha_{j \rightarrow r_1, t_2} = 1$	$\alpha_{j \rightarrow r_1, t_2} = 0$	$\alpha_{j \rightarrow r_1, t_2} = 1$	$\alpha_{j \rightarrow r_1, t_2} = 0$
$\alpha_{j \rightarrow r_2, t_2} = 0$	$\alpha_{j \rightarrow r_2, t_2} = 1$	$\alpha_{j \rightarrow r_2, t_2} = 0$	$\alpha_{j \rightarrow r_2, t_2} = 1$

Given this helper variable, the three utilization constraints for RS-Alloc are defined as follows:

$$u_{r,t}^{\text{Table}} + \sum_{j \in J_t} \sum_{A \in A_{j,T_j}} \alpha_{j \rightarrow r,t} u_{j,t}^{\text{Table}} Y_{j,A} \leq c_r^{\text{Table}} \quad \forall_{t \in T} \quad \forall_{r \in S} \quad (11.11)$$

$$u_{s_j \rightarrow r,t}^{\text{Link}} + \sum_{j \in J_t} \sum_{A \in A_{j,T_j}} \alpha_{j \rightarrow r,t} u_{j,t}^{\text{Link}} Y_{j,A} \leq c_{s_j \rightarrow r}^{\text{Link}} \quad \forall_{t \in T} \quad \forall_{r \in S} \quad (11.12)$$

$$u_{r \rightarrow s_j,t}^{\text{Link}} + \sum_{j \in J_t} \sum_{A \in A_{j,T_j}} \alpha_{j \rightarrow r,t} u_{j,t}^{\text{Link}} Y_{j,A} \leq c_{r \rightarrow s_j}^{\text{Link}} \quad \forall_{t \in T} \quad \forall_{r \in S} \quad (11.13)$$

This is very similar to the constraints used in Eq. (9.39) to Eq. (9.41). The only difference is that  $\sum_{j \in J_t} u_{j,t}^{\text{Table}} Y_{j \rightarrow r,t}$  is replaced with  $\sum_{j \in J_t} \sum_{A \in A_{j,T_j}} \alpha_{j \rightarrow r,t} u_{j,t}^{\text{Table}} Y_{j,A}$ . The helper variable  $\alpha$  makes sure that only those utilization coefficients are considered that are relevant for the given allocation assignment  $A$  (only set to 1 if the remote switch from the assignment

is set to  $r$  and to 0 in all other cases). The two other cases for the link utilization are handled in the same way.

### 11.2.4 Cost Coefficients

The cost coefficients  $w_{j,A}$  represent the costs to be paid for using a certain allocation assignment. There are four important cost factors to consider: i) free flow table capacity of the remote switch ( $w_{j,A}^{\text{Table}}$ ), ii) free link bandwidth between delegation and remote switch ( $w_{j,A}^{\text{Link}}$ ), iii) amount of control messages required ( $w_{j,A}^{\text{Ctrl}}$ ), and iv) static costs associated with remote switches ( $w_A^{\text{Static}}$ ).

All four cost coefficients are explained in detail in Sec. 9.5.4. This section only describes the translation to the assignment-based formulations for RS-Alloc which is straightforward. Given an assignment  $A \in A_{j,T_j}$ , the cost coefficients can simply be calculated as  $w_{j,A} = \sum_{t \in T_j} w_{j \rightarrow a_t, t}$  in most cases where  $a_t$  denotes the  $t$ -th element (remote switch) of the allocation assignment.

$$w_{j,A}^{\text{Table}} := \sum_{t \in T_j} w_{j \rightarrow a_t, t}^{\text{Table}} = \sum_{t \in T_j} - \left( c_{a_t}^{\text{Table}} - u_{a_t, t}^{\text{Table}} - u_{j, t}^{\text{Table}} \right) \quad (11.14)$$

$$w_{j,A}^{\text{Link}} := \sum_{t \in T_j} w_{j \rightarrow a_t, t}^{\text{Link}} = \sum_{t \in T_j} - \left( \min \left( c_{s_j \rightarrow a_t}^{\text{Link}} - u_{s_j \rightarrow a_t, t}^{\text{Link}}, c_{a_t \rightarrow s_j}^{\text{Link}} - u_{a_t \rightarrow s_j, t}^{\text{Link}} \right) - u_{j, t}^{\text{Link}} \right) \quad (11.15)$$

$$w_{j,A}^{\text{Ctrl}} := \left( H_j^X * \text{get\_cost}(j, H_j^r, a_{t_1}, t_1) \right) + \sum_{t=t_1}^{t_{m-1}} \text{get\_cost}(j, a_t, a_{t+1}, t_{+1}) \quad (11.16)$$

$$w_A^{\text{Static}} := \sum_{t \in T_j} w_{a_t, t}^{\text{Static}} \quad (11.17)$$

Note that Eq. (11.16) – the control message cost coefficient – required a quadratic expression in the original problem. The assignment based formulation, however, can be calculated in a similar way as the other cost coefficients because all remote switches in the allocation assignment are fixed. One difference is that this coefficient has to include the history of the previous optimization period which is given as  $\langle H_j^X, H_j^F, H_j^r \rangle$ . In case allocation job  $j$  was active in the previous period ( $H_j^X = 1$ ), the cost for changing from remote switch  $H_j^r$  (allocated in the previous period) to remote switch  $a_{t_1}$  (allocated in the first time slot of the currently active period) is included here in the first bracket.

### 11.2.5 Problem Formulation for RS-Alloc

Based on the variables introduced in the previous sections, the RS-Alloc problem (which is a multi period RS-Alloc problem with allocation assignments that can be used with periodic optimization) is defined as an integer linear program in the following way:

#### Definition 11.7: RS-Alloc

**Inputs:** A set of switches  $S$ , set of consecutive time slots  $T$ , set of allocation jobs  $J_T$  (and  $J_t$  for all  $t \in T$ ), flow table capacity  $c_r^{\text{Table}}$  for each switch, link capacity  $c_{s_j \rightarrow r}^{\text{Link}}$  for each link, flow table utilization  $u_{r,t}^{\text{Table}}$  for each switch, link utilization  $u_{s_j \rightarrow r,t}^{\text{Link}}$  for each link, pre-computed allocation assignments  $A_{j,T_j}$  for each  $j \in J_T$ ,  $t \in T_j$ , utilization coefficients  $u_{j,t}$  for each  $j \in J_T$ ,  $t \in T_j$  and cost coefficients  $w_{j,A}$  for each  $j \in J_T$ ,  $A \in A_{j,T_j}$

**Output:** A set of selected and allocated delegation templates  $D_T^{**}$  for a set of consecutive time slots  $T$

#### Problem Formulation:

$$\min_{Y_{j \rightarrow r,t}} \sum_{j \in J_T} \sum_{A \in A_{j,T_j}} w_{j,A} Y_{j,A} \quad (11.18)$$

$$\text{s.t.} \quad \sum_{A \in A_{j,T_j}} Y_{j,A} = 1 \quad \forall_{j \in J_T} \quad (11.19)$$

$$u_{r,t}^{\text{Table}} + \sum_{j \in J_t} \sum_{A \in A_{j,T_j}} \alpha_{j \rightarrow r,t} u_{j,t}^{\text{Table}} Y_{j,A} \leq c_r^{\text{Table}} \quad \forall_{t \in T} \quad \forall_{r \in S} \quad (11.20)$$

$$u_{s_j \rightarrow r,t}^{\text{Link}} + \sum_{j \in J_t} \sum_{A \in A_{j,T_j}} \alpha_{j \rightarrow r,t} u_{j,t}^{\text{Link}} Y_{j,A} \leq c_{s_j \rightarrow r}^{\text{Link}} \quad \forall_{t \in T} \quad \forall_{r \in S} \quad (11.21)$$

$$u_{r \rightarrow s_j,t}^{\text{Link}} + \sum_{j \in J_t} \sum_{A \in A_{j,T_j}} \alpha_{j \rightarrow r,t} u_{j,t}^{\text{Link}} Y_{j,A} \leq c_{r \rightarrow s_j}^{\text{Link}} \quad \forall_{t \in T} \quad \forall_{r \in S} \quad (11.22)$$

$$Y_{j \rightarrow r,t} \in \{0, 1\} \quad \forall_{j \in J_T} \quad \forall_{t \in T_j} \quad \forall_{r \in S} \quad (11.23)$$

This is a multi-dimensional multiple-choice knapsack problem. There are  $|J_T|$  different mutually disjoint classes  $A_{j,T_j} = \{A_1, \dots, A_n\}$  called allocation assignment sets. Each allocation job  $j \in J_T$  defines one allocation assignment set and each of these sets contains  $\prod_{t \in T_j} |R_{j,t}|$  items called allocation assignments. For each allocation job  $j \in J_T$  and each time slot  $t \in T_j$ , there is a utilization coefficient  $u_{j,t}^{\text{Table}}$ . And there is a cost coefficient  $w_{j,A}$  for each allocation assignment  $A \in A_{j,T_j}$ . Goal is to choose exactly one item out of each allocation assignment set so that the costs are minimized while the utilization is

below the capacity in all time slots. Eq. (11.18) ensures that the objective function is minimized. The multiple-choice constraints in Eq. (11.19) make sure that exactly one allocation assignment is chosen for each allocation job. Eq. (11.20) models the flow table capacity constraints for the remote switches. Eq. (11.21) and Eq. (11.22) model the link capacity constraints for the links between delegation and remote switches. All capacity constraints are defined separately for each time slot and each remote switch, i.e., there are  $|S| * |T|$  capacity constraints (at most). And Eq. (11.23) ensures that the decision variables are binary.

The objective function uses a placeholder variable  $w_{j,A}$  to represent a mixture of the cost coefficients, i.e., this is a multi-objective formulation. This mixture consists of four parts as introduced above:

$$w_{j,A} = \left( \omega_{\text{RSA}}^{\text{Table}} w_{j,A}^{\text{Table}} + \omega_{\text{RSA}}^{\text{Link}} w_{j,A}^{\text{Link}} + \omega_{\text{RSA}}^{\text{Ctrl}} w_{j,A}^{\text{Ctrl}} + \omega_{\text{RSA}}^{\text{Static}} w_A^{\text{Static}} \right) \quad (11.24)$$

The non-negative  $\omega_{\text{RSA}}$ -weights balance the different cost coefficients against each other. The optimal allocation  $D_T^{**}$  – referred to as the set of selected and allocated delegation templates for all time slots in  $T$  – is calculated as follows:

$$D_T^{**} := \left\{ \langle d_{j,t}, a_t \rangle \mid a_t \in A \text{ with } Y_{j,A}^* = 1, t \in T_j, j \in J_T \right\} \quad (11.25)$$

The  $Y_{j,A}^*$  variables represent the optimal decision variables after the problem is solved. Each tuple  $\langle d_{j,t}, a_t \rangle$  contains one delegation template and the allocated remote switch.

### 11.2.6 Algorithm for RS-Alloc

The RS-Alloc algorithm is shown in Alg. 13. Goal, input and output are defined analogously to the RS-Alloc problem from Def. 11.7. In addition, the periodic approach described in Sec. 10.2.1.2 is applied.

The algorithm is structured into three steps that are executed once per optimization period. In the first step, the allocation assignment sets  $A_{j,T_j}$  for each allocation job  $j \in J_T$  are created, together with all required coefficients (lines 3-11). In the second step, the RS-Alloc problem is solved for the current optimization period based on the inputs and the calculations from step 1 (line 12). Because each remote set contains a backup switch ( $s_B$ , see Sec. 9.5.1.2) with unlimited capacity and no restrictions that be used with high cost, it is ensured that the problem is always feasible. The last step (lines 13-16) after calculation of  $D_T^{**}$  is updating the history (note that only the first time slot of  $T$  is used here) and waiting for the next optimization period.

**Algorithm 13:** RS-Alloc Algorithm**Data:** Same input as RS-Alloc problem from Def. 11.7, Look ahead factor  $L$ **Result:** For each optimization period: set of selected and allocated delegation templates  $D_T^{**}$  for each time slot  $t \in T$  with  $|T| = L$ 

```

1 Function alloc_opt_algorithm():
2   if  $J_T \neq \emptyset$  then
3     for  $j \in J_T$  do
4       /* Step 1: Initialize allocation assignments and coefficients */
5        $A_{j,T_j} \leftarrow R_{j,1} \times \dots \times R_{j,|T_j|}$  // Def. 11.3
6       for  $A \in A_{j,T_j}$  do
7         for  $t \in T_j$  do
8            $u_{j,t}^{\text{Table}}, u_{j,t}^{\text{Link}} \leftarrow \text{calculate\_utilization}()$  // Sec. 11.2.3
9         end
10         $w_{j,A} \leftarrow \text{calculate\_cost}()$  // Sec. 11.2.4
11      end
12      /* Step 2: solve RS-Alloc (will return a set of selected and assigned
13      delegation templates  $D_T^{**}$ ) */
14       $D_T^{**} \leftarrow \text{solve\_alloc\_opt}()$  // Def. 11.7
15      /* Step 3: update history and wait for next optimization period */
16      for  $d_{j,t} \in D_{t_1}^{**}$  do
17         $\langle H_j^X, H_j^F, H_j^r \rangle \leftarrow \text{update\_history}()$  // Def. 11.4
18      end
19      wait until next optimization period and start again from line 2
20    end
21  end

```

## 11.3 Pre-processing for Allocation Assignments

The RS-Alloc problem formulation in Def. 11.7 does not scale towards larger instances because the number of required allocation assignments is  $\prod_{t \in T_j} |R_{j,t}|$  per allocation job. However, this problem can be addressed with a **pre-processing step** in order to reduce the number of considered allocation assignments.

First, the general idea of pre-processing is explained in Sec. 11.3.1. Afterwards, two required concepts are introduced: allocation intervals in Sec. 11.3.2 and a stability metric for allocation intervals in Sec. 11.3.3. Sec. 11.3.4 finally introduces the pre-processing algorithm used in this thesis.

### 11.3.1 General Idea

The idea is to exploit the observation that the vast majority of the allocation assignments will never be considered by the solver and could thus be removed before the optimization. This will not change the problem formulation or the algorithm. The only thing that is updated is the way how the allocation assignment set for allocation job  $j$  is calculated.

Consider an allocation job that is active for three consecutive time slots  $T_j = \{t_1, t_2, t_3\}$  and the same three remote switch options are available in all three time slots, i.e.,  $R_{j,t_1} = R_{j,t_2} = R_{j,t_3} = \{r_1, r_2, r_3\}$ . This will result in  $3 * 3 * 3 = 27$  different allocation assignments in  $A_{j,T_j}$ :

$$\begin{aligned} A_1 &= (r_1, r_1, r_1) \\ A_2 &= (r_1, r_1, r_2) \\ &\dots \\ A_{26} &= (r_3, r_3, r_2) \\ A_{27} &= (r_3, r_3, r_3) \end{aligned}$$

However, based on the available free flow table capacity and link bandwidth in the network, not all 27 allocation assignments have the same relevance with respect to the optimization. Assume the flow table utilization for remote switch  $r_1$  and  $r_2$  ( $u_{r,t}^{\text{Table}}$ ) is low in all three time slots (lots of free capacity) while the flow table utilization of  $r_3$  is high (almost no free capacity). In this case, all assignments with  $r_3$  have a very high cost and are less important because the solver will presumably prefer solutions with  $r_1$  and  $r_2$  (because of how the cost coefficients are calculated, see Sec. 11.2.4). And because of  $w_{j,A}^{\text{ctrl}}$ , assignments that use the same remote switch in consecutive time slots – such as  $(r_1, r_1, r_1)$  and  $(r_2, r_2, r_2)$  – are preferred over assignments where the remote switch is changed frequently, e.g.,  $(r_2, r_1, r_2)$ . The former assignment is assumed to be “more stable” than the latter one.

Removing less important allocation assignments prior to the optimization will reduce the number of objective terms and coefficients significantly. This cannot guarantee that a fully optimal result is found because the removed assignments are not included in the search space. However, experiments have shown that the approach with reduced assignments achieves very promising results in practice.



### 11.3.2 Allocation Interval

To realise the above idea, it is first necessary to split the consecutive time slots in  $T_j$  into multiple disjoint sets of consecutive time slots. These smaller sets are called allocation intervals for allocation job  $j$  in the following:

#### Definition 11.8: Allocation Interval

An **allocation interval**  $\hat{T}_{i,j}$  is a subset of set  $T_j$ . The **set of allocation intervals**  $\hat{T}_j$  for a set of consecutive time slots  $T_j$  is defined as follows:

$$\hat{T}_j := \{\hat{T}_{1,j}, \dots, \hat{T}_{n,j}\} \left| \begin{array}{l} \hat{T}_{i,j} \subseteq T_j, i = 1, \dots, n \text{ and} \\ \bigcup_{i=1}^n \hat{T}_{i,j} = T_j, \bigcap_{i=1}^n \hat{T}_{i,j} = \emptyset \text{ and} \\ \text{all } T_{i,j} \text{ are ordered} \end{array} \right. \quad (11.26)$$

$T_j$  represents all time slots where allocation job  $j$  is active (see Def. 11.1).

An allocation job with  $T_j = \{t_1, \dots, t_4\}$ , for example, can be split into  $\hat{T}_j = \{\hat{T}_{1,j}, \hat{T}_{2,j}\} = \{\{t_1, t_2\}, \{t_3, t_4\}\}$  or  $\hat{T}_j = \{\hat{T}_{1,j}, \hat{T}_{2,j}\} = \{\{t_1\}, \{t_2, t_3, t_4\}\}$  or any other valid combination of allocation intervals. In addition, a new operator  $\sqcap$  for allocation intervals is defined as follows:

$$\sqcap_{t \in \hat{T}_{i,j}} R_{j,t} := \left\{ (r, r, \dots, r) \left| r \in \bigcap_{t \in \hat{T}_{i,j}} R_{j,t} \text{ and } |(r, r, \dots, r)| = |\hat{T}_{i,j}| \right. \right\} \quad (11.27)$$

This is similar to calculating the intersections of  $R_{j,t}$  for all  $t \in \hat{T}_{i,j}$  except that each resulting element in the intersection set is “expanded” into a tuple of size  $|\hat{T}_{i,j}|$ . In other words: the new operator takes a set of time slots  $t \in \hat{T}_{i,j}$  and a set of remote switch options  $R_{j,t}$  for each time slot and creates a set of tuples of length  $|\hat{T}_{i,j}|$  for each element in the intersection of all  $R_{j,t}$ . Assume a job with  $R_{j,t} = \{r_1, r_2\}$  for all time slots, i.e., the intersection will contain both switches in all cases. In this example, the  $\sqcap$  operator returns the following results:

$$\begin{aligned} \sqcap_{t \in \{t_1\}} R_{j,t} &:= \{(r_1), (r_2)\} \\ \sqcap_{t \in \{t_1, t_2\}} R_{j,t} &:= \{(r_1, r_1), (r_2, r_2)\} \end{aligned}$$

$$\begin{aligned} \bigcap_{t \in \{t_1, t_2, t_3\}} R_{j,t} &:= \{(r_1, r_1, r_1), (r_2, r_2, r_2)\} \\ &\dots \\ \bigcap_{t \in \{t_1, t_2, t_3, \dots, t_n\}} R_{j,t} &:= \{(r_1, r_1, r_1, \dots, r_1), (r_2, r_2, r_2, \dots, r_2)\} \end{aligned}$$

The following table contains some additional examples to clarify the idea. The first column shows the allocation interval, the second column the remote sets for each time slot from the allocation interval and the two remaining columns show the result of the normal intersection operator and the  $\bigcap$  operator from Eq. (11.27):

$\hat{T}_{i,j}$	$R_{j,t}$	$\bigcap_{t \in \hat{T}_{i,j}} R_{j,t}$	$\bigcap_{t \in \hat{T}_{i,j}} R_{j,t}$
$\{t_1\}$	$R_{j,t_1} = \{r_1, r_2\}$	$\{r_1, r_2\}$	$\{(r_1), (r_2)\}$
$\{t_1, t_2\}$	$R_{j,t_1} = \{r_1, r_2\}$ $R_{j,t_2} = \{r_1, r_3\}$	$\{r_1\}$	$\{(r_1, r_1)\}$
$\{t_1, t_2, t_3\}$	$R_{j,t_1} = \{r_1, r_2, r_3\}$ $R_{j,t_2} = \{r_1, r_3\}$ $R_{j,t_3} = \{r_1, r_3\}$	$\{r_1, r_3\}$	$\{(r_1, r_1, r_1), (r_3, r_3, r_3)\}$
$\{t_1, t_2, t_3, t_4\}$	$R_{j,t_1} = \{r_1, r_2, r_3\}$ $R_{j,t_2} = \{r_1, r_3\}$ $R_{j,t_3} = \{r_1, r_3\}$ $R_{j,t_4} = \{r_1, r_2\}$	$\{r_1\}$	$\{(r_1, r_1, r_1, r_1)\}$

### 11.3.3 Stability Metric

For pre-processing, it is required to compare different allocation intervals with each other. Basic idea is it to transform  $T_j$  into a set of  $n$  allocation intervals  $\hat{T}_j := \{\hat{T}_{1,j}, \dots, \hat{T}_{n,j}\}$  so that allocation job  $j$  is most likely allocated to a single remote switch for all time slots in all the  $n$  allocation intervals (i.e., the remote switch is not changed within one single allocation interval). This property is referred to as allocation interval stability:

**Definition 11.9: Allocation Interval Stability**

The **allocation interval stability**  $\hat{w}_{i,j}$  for an allocation interval  $\hat{T}_{i,j}$  is defined as:

$$\begin{aligned} \hat{w}_{i,j} &:= \text{get\_stability} \left( \hat{T}_{i,j} \right) & (11.28) \\ &= |\hat{T}_{i,j}| * \min_{t \in \hat{T}_{i,j}} \left( \left( \sum_{r \in \bigcap_{t \in \hat{T}_{i,j}} R_{j,t}} \hat{m}_{r,t} \right) * \left( \sum_{r \in \bigcap_{t \in \hat{T}_{i,j}} R_{j,t}} u_{r,t}^{\text{Table}} - u_{j,t}^{\text{Table}} \right) \right) \end{aligned}$$

with helper variable  $\hat{m}_{r,t}$  defined as:

$$\hat{m}_{r,t} := \begin{cases} 1, & \text{if } u_{r,t}^{\text{Table}} > u_{j,t}^{\text{Table}} \\ 0, & \text{otherwise} \end{cases}$$

The stability metric in Eq. (11.28) was found empirically (can be seen as an example). It is based on the assumption that the control message cost coefficient ( $w_{j,A}^{\text{Ctrl}}$ ) is the dominant factor when calculating the minimum cost for an allocation assignment. To take this into consideration, the stability metric returns higher (better, more stable) values if more free flow table capacity is available (more free capacity means higher chances that the same remote switch can be used) and scales this value with the number of different remote switches that are available for the currently selected delegation job. The metric consists of three conceptual parts:

- $|\hat{T}_{i,j}| * \min_{t \in \hat{T}_{i,j}} (\dots)$ : The main part of the formula consists of the minimum operator that compares the score (more free capacity = better score) for each individual time slot in the allocation interval. It uses the minimum of all scores because this will be the bottleneck the optimization algorithm has to deal with. It is further multiplied by the number of time slots in  $\hat{T}_{i,j}$  to make sure that “longer” allocation intervals with high stability are preferred.
- $\left( \sum_{r \in \bigcap_{t \in \hat{T}_{i,j}} R_{j,t}} u_{r,t}^{\text{Table}} - u_{j,t}^{\text{Table}} \right)$ : This calculates the individual score for one time slot. It iterates over all remote switch options that are present in all time slots of the allocation interval (intersection) and compares the utilization demand ( $u_{j,t}^{\text{Table}}$ ) with the current flow table utilization of the remote switch ( $u_{r,t}^{\text{Table}}$ ). The higher this value, the more free flow table capacity is available. If the value is negative, at least one remote switch in the allocation interval does not provide enough free flow table capacity.
- $\left( \sum_{r \in \bigcap_{t \in \hat{T}_{i,j}} R_{j,t}} \hat{m}_{r,t} \right)$ : This represents the number of remote switches that can provide enough free capacity for allocation job  $j$  in time slot  $t$ . This value is multiplied to

the individual score from above because the chances for using the same remote switch is higher if there are multiple alternatives.

### 11.3.4 Pre processing Algorithm

Given the concepts introduced above (allocation interval and stability), the goal of the pre-processing step can be described as follows: Find allocation intervals that are most stable and use these intervals to calculate the allocation assignment set  $A_{j,T_j}$ . If  $n$  allocation intervals were found ( $n$  can be different for every allocation job), the allocation assignment set is calculated as follows:

$$A_{j,T_j} := \prod_{t \in \hat{T}_{1,j}} R_{j,t} \times \prod_{t \in \hat{T}_{2,j}} R_{j,t} \times \dots \times \prod_{t \in \hat{T}_{n,j}} R_{j,t} \quad (11.29)$$

The challenge is to select the allocation intervals  $\hat{T}_{i,j}$  in such a way, that the search space is not too restricted and the solver can still make relevant decisions. This is done by Alg. 14 that will recursively split  $T_j$  into allocation intervals based on the stability metric from Sec. 11.3.3.

Fig. 11.3 illustrates the general process. The example in the figure consists of a single allocation job  $j$  that is active in nine time slots  $T_j = \{t_1, \dots, t_9\}$ . For most of the time slots – all except  $t_5$  and  $t_6$  – there are two remote switch options  $R_{j,t} = \{r_2, r_3\}$ . For  $t_5$  and  $t_6$ , there is only a single remote switch ( $r_2$ ) available. The backup switch that is usually included in  $R_{j,t}$  is omitted here to keep the example simple. Pre-processing now works as follows (the numbers match the circled numbers in the figure):

- (1) The algorithm starts with  $n = 1$ , i.e., with a single allocation interval  $\hat{T}_{1,j} = T_j$ . This is the initialization step (line 3 in Alg. 14). If the number of allocation assignments in the result set (pre-defined value, e.g., 100) is not yet reached, the algorithm starts. In the first step, it updates the final result set (the allocation assignment set  $A_{j,T_j}$ ) based on the  $n$  elements currently stored in  $\hat{T}_j$ . In this case,  $\hat{T}_j$  contains only a single allocation interval which results in one allocation assignment  $A_1 = (r_2, \dots, r_2)$ .
- (2) The next step of the algorithm consists of three parts: interval selection (2a), split index selection (2b) and split execution (2c). In the first part, the process iterates over all current allocation intervals in  $\hat{T}_j$  and selects the one with lowest stability. In the second part, the selected allocation interval is split into two smaller intervals (if possible). And in the third part, the two new allocation intervals replace the old one and the algorithm starts over.

**Algorithm 14:** Pre-processing for RS-Alloc

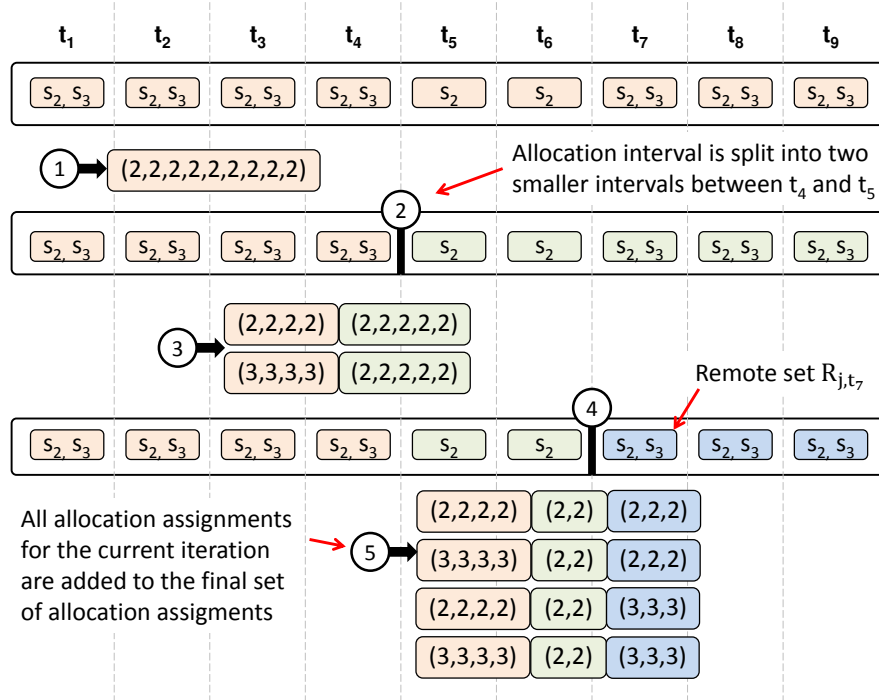
---

```

1 Function preprocess_assignments(depth = 1):
2   if depth = 1 then
3      $\hat{T}_j = \{\hat{T}_{1,j}\} = \{T_j\} = \{t_1, \dots, t_m\}$     $A_{j,T_j} \leftarrow \{\}$  // Initialize
4   end
5   if  $|A_{j,T_j}| <$  required number of allocation assignments then
6     /* Step 1: update allocation assignments in the final result set based
7       on the n elements currently stored in  $\hat{T}_j$  */
8      $A_{j,T_j} \leftarrow \prod_{t \in \hat{T}_{1,j}} R_{j,t} \times \dots \times \prod_{t \in \hat{T}_{n,j}} R_{j,t}$ ,  $n = |\hat{T}_j|$  // Eq. (11.27)
9     /* Step 2a: select allocation interval with minimum stability */
10     $\hat{w}_{i,j}^{\text{Min}}, \hat{T}_{i,j}^{\text{Min}} \leftarrow \langle \infty, \emptyset \rangle$ 
11    for  $\hat{T}_{i,j} \in \hat{T}_j$  do
12      if  $|\hat{T}_{i,j}| > 1$  then
13         $\hat{w}_{i,j} \leftarrow \text{get\_stability}(\hat{T}_{i,j})$  //see Def. 11.9
14        if  $\hat{w}_{i,j} < \hat{w}_{i,j}^{\text{Min}}$  then
15           $\hat{w}_{i,j}^{\text{Min}}, \hat{T}_{i,j}^{\text{Min}} \leftarrow \langle \hat{w}_{i,j}, \hat{T}_{i,j} \rangle$ 
16        end
17      end
18    end
19    if  $\hat{T}_{i,j}^{\text{Min}} \neq \emptyset$  then
20      /* Step 2b: split index selection */
21       $\hat{w}_{i,j}^{\text{Min}}, \hat{T}_{i,j}^{\text{Left}}, \hat{T}_{i,j}^{\text{Right}} \leftarrow \langle -\infty, \emptyset, \emptyset \rangle$ 
22      for  $t \in \hat{T}_{i,j}^{\text{Min}}$  do
23         $\hat{T}_{i,j}^{\text{Left}} \leftarrow \{t_1, \dots, t\}$     $\hat{T}_{i,j}^{\text{Right}} \leftarrow \{t_{+1}, \dots, t_m\}$ 
24         $\hat{w}_{i,j}^{\text{Left}} \leftarrow \text{get\_stability}(\hat{T}_{i,j}^{\text{Left}})$     $\hat{w}_{i,j}^{\text{Right}} \leftarrow \text{get\_stability}(\hat{T}_{i,j}^{\text{Right}})$ 
25        if  $\hat{w}_{i,j}^{\text{Left}} + \hat{w}_{i,j}^{\text{Right}} > \hat{w}_{i,j}^{\text{Min}}$  then
26           $\hat{w}_{i,j}^{\text{Min}}, \hat{T}_{i,j}^{\text{Left}}, \hat{T}_{i,j}^{\text{Right}} \leftarrow \langle \hat{w}_{i,j}^{\text{Left}} + \hat{w}_{i,j}^{\text{Right}}, \hat{T}_{i,j}^{\text{Left}}, \hat{T}_{i,j}^{\text{Right}} \rangle$ 
27        end
28      end
29      /* Step 2c: Replace  $\hat{T}_{i,j}^{\text{Min}}$  with the splitted allocation interval
30        (number of elements in  $\hat{T}_j$  increases from n to n+1) */
31       $\hat{T}_j = \{\dots, \cancel{\hat{T}_{i,j}^{\text{Min}}}, \hat{T}_{i,j}^{\text{Left}}, \hat{T}_{i,j}^{\text{Right}}, \dots\}$ 
32      preprocess_assignments(depth+1) // start next iteration
33    end
34  end
35 end

```

---



**Figure 11.3:** Pre-processing algorithm for RS-Alloc

- Basic idea of step 2a is to select the interval that will most probably have different remote switches after optimization. Allocation intervals the solver can handle as one unit (the most stable intervals) should not be selected. Because of  $n = 1$ , the decision is trivial in the first step ( $\hat{T}_{1,j}$  is selected).
  - Basic idea of step 2b is to select a split index for the selected allocation interval in such a way that one of the resulting new intervals (left or right) will be stable, i.e., the chance that the interval is split in the next iteration of the algorithm is small. In the example it is assumed that the first four time slots form the most stable interval. Therefore, the split is executed between  $t_4$  and  $t_5$  which will result in two intervals  $\hat{T}_{1,j}^{\text{Left}} = \{t_1, \dots, t_4\}$  and  $\hat{T}_{1,j}^{\text{Right}} = \{t_5, \dots, t_9\}$ .
  - In step 2c, the old allocation interval ( $\hat{T}_{1,j}$ ) is replaced by the two new ones ( $\hat{T}_{1,j}^{\text{Left}}$  and  $\hat{T}_{1,j}^{\text{Right}}$ ).  $n$  is increased to 2 and the function calls itself with  $\text{depth}=2$ .
- (3) Repetition of step (1). Set  $\hat{T}_j$  now consists of two items  $\hat{T}_{1,j}$  and  $\hat{T}_{2,j}$  (left and right from above), i.e.,  $n = 2$ . The algorithm updates the final result set via  $A_{j,T_j} \leftarrow \prod_{t \in \hat{T}_{1,j}} R_{j,t} \times \prod_{t \in \hat{T}_{2,j}} R_{j,t}$  with  $\prod_{t \in \hat{T}_{1,j}} R_{j,t} = \{(r_2, r_2, r_2, r_2), (r_3, r_3, r_3, r_3)\}$  and  $\prod_{t \in \hat{T}_{2,j}} R_{j,t} = \{(r_2, r_2, r_2, r_2, r_2)\}$ . This results in two new allocation assignment to be added to  $A_{j,T_j}$ , i.e.,  $A_1 = (r_2, \dots, r_2)$  and  $A_2 = (r_3, r_3, r_3, r_3, r_2, \dots, r_2)$ . Note that

allocation assignment  $A_1$  was already calculated in step (1). This can be handled more efficiently by calculating the final result set  $A_{j,T_j}$  only in the last step (the number of allocation assignments for the check in line 5 can be calculated by multiplication of the cardinality of the allocation intervals).

- (4) Repetition of step (2). Because there are now two allocation intervals, one has to be selected for splitting. As mentioned earlier, the first interval ( $t_1$  to  $t_4$ ) is considered the most stable in this example and will thus not be selected. Instead, the second allocation interval is split into two new allocation intervals between  $t_6$  and  $t_7$ . This results in two new allocation intervals  $\hat{T}_{2,j}^{\text{Left}} = \{t_5, t_6\}$  and  $\hat{T}_{2,j}^{\text{Right}} = \{t_7, t_8, t_9\}$ . The old interval  $\hat{T}_{2,j}$  is replaced by the two new ones.  $n$  is increased to 3, i.e., there are now three allocation intervals in set  $\hat{T}_j$ .
- (5) Repetition of step (1) for  $n = 3$  which results in four allocation assignments in total. The two steps will now continue in turns until a termination criteria is met, e.g., the number of allocation assignments in the final set  $A_{j,T_j}$  is larger than a threshold.

This process will create a set of allocation assignments that contains the most relevant assignments for the optimization step. Bad allocation assignments are avoided because the most stable allocation intervals are not splitted. Consider the first allocation interval in the example in Fig. 11.3. Simply put, most stable means that one or both options ( $r_2$  and  $r_3$ ) have plentiful free resources between  $t_1$  and  $t_4$  while less stable intervals have less resources. The chances that the first allocation interval can be handled as a whole is thus much higher. Splitting the allocation interval will only result in a potentially huge number of inferior assignments with high costs that are ignored by the solver.

## 11.4 Conclusion

This chapter introduces an ILP-based algorithm for the multi period RS-Alloc problem based on a linear assignment-based knapsack formulation. The algorithm calculates coefficients for all combinations of possible remote switch allocations for each allocation job. To reduce the number of allocation assignments, a pre-processing step removes assignments that will probably never be considered by the solver.





Part IV  
Evaluation



# Preliminaries and Methodology

---

The evaluation part – which consists of Chapters 12 to 16 – investigates feasibility, performance, overhead, and scalability of the flow delegation approach based on a wide range of different scenarios. It is structured as follows:

- (1) Chapter 12 (this chapter) defines **assumptions, terminology and the evaluation methodology**. This includes central definitions such as capacity reduction factor, failure rate, and flow delegation performance. It also explains in detail how the synthetic scenarios for evaluation are generated.
- (2) Chapter 13 conducts a small **case study** to demonstrate that the approach is feasible. This is done for four conceptually different scenarios including a worst case example, a best case example and two examples in between.
- (3) Chapter 14 investigates the expected **performance** of the approach for 5000 different scenarios based on capacity reduction factor (bottleneck severity) and failure rate. It also investigates the performance with respect to over- and underutilization.
- (4) Chapter 15 investigates the **overhead** associated with the algorithms developed in the thesis which includes table overhead, link overhead and control overhead.
- (5) Chapter 16 investigates **runtime and scalability**. This includes the modeling and solving time of the developed algorithms. It also investigates how the approach scales with number of delegation templates and number of switches.

## 12.1 Network Model and Assumptions

The network model used for evaluation consists of a set of switches  $S$  and a set of hosts  $H$  where each host  $h \in H$  is attached to exactly one switch  $s \in S$ . All switches are connected to the same controller. All connection between hosts and switches are bidirectional and static, i.e., do not change over time. All switches have the same flow table capacity  $c^{\text{Table}}$  and all links have the same link capacity  $c^{\text{Link}}$ . Links between switches and hosts have unlimited capacity in both directions. The following additional assumptions are made to simplify scenario generation and the overall evaluation process<sup>1</sup>.

First, it is assumed that each flow in the network has a single source  $h_{\text{src}} \in H$  and a single destination  $h_{\text{dst}} \in H$  and traffic is always forwarded on the shortest path between  $h_{\text{src}}$  and  $h_{\text{dst}}$ .  $h_{\text{src}}$  starts sending traffic at a certain point in time given as  $\tau^{\text{start}}$ . It is assigned a fixed bit rate  $b$  and a fixed amount of bits  $\delta$  that have to be transferred to  $h_{\text{dst}}$ . The bit rate is defined as a constant rate that does not change over time, i.e., effects such as slow start of a TCP connection are not considered.  $h_{\text{src}}$  stops sending traffic at  $\tau^{\text{stop}} = \tau^{\text{start}} + \frac{\delta}{b}$ .

Regarding the flow rules, it is assumed that a new flow rule  $f$  is installed in each switch on the path from  $h_{\text{src}}$  to  $h_{\text{dst}}$  at  $\tau^{\text{install}} = \tau^{\text{start}}$ . Propagation delays and processing delays of the controller are ignored. Communication is defined unidirectional, i.e., acknowledgements sent in the return direction are not considered and no flow rules are installed in the return direction by default. Flow rules have a minimum lifetime  $n_{\text{lifetime}}$  which is set to a value between 1 and 5 seconds. The point in time the flow rule is removed is then calculated as  $\tau^{\text{remove}} = \max(\tau^{\text{stop}}, \tau^{\text{install}} + n_{\text{lifetime}})$ .

Each flow rule performs an exact match on a certain set of packet header fields such as IP or MAC addresses, i.e., no wildcard matches are used. The distance between two time slots is set to one second, i.e., the delegation algorithm is executed once per second. It is assumed that the monitoring system can provide all necessary monitored parameters as defined in Table 4.1. Estimations about future time slots which are required for look-ahead factors  $L > 1$  are based on perfect knowledge, not on a prediction mechanism. With respect to the aggregation scheme, it is assumed that the ingress port based scheme from Sec. 5.2.2.3 is used. This means there is one delegation template per physical port in each time slot.

---

<sup>1</sup>These simplifications can be made because we are here only interested in the general behavior and performance of the flow delegation approach and the involved algorithms, e.g., with respect to the number of flow rules, the parameterization of the algorithms or other higher level parameters – and not in the exact behavior expected from a real network. Also note that the flow delegation approach was successfully tested in an emulated setting without these simplifications (Chapter 8).

Given the above assumptions, it is now possible to specify a single flow rule for the evaluation as a nine-tuple  $\langle s_f, h_{src}, h_{dst}, SP, \tau^{install}, \tau^{removed}, \delta, b, n_{lifetime} \rangle$ . The nine values in the tuple are explained in Table 12.1. A set of flow rules following this specification together with a topology form a *scenario* which is explained in the next section. Note that match, action and priority are defined implicitly in this model, i.e., are not included in the nine-tuple.

Flow rule definition:  $\langle s_f, h_{src}, h_{dst}, SP, \tau^{install}, \tau^{removed}, \delta, b, n_{lifetime} \rangle$

Parameter	Description
$s_f$	The switch this flow rule is installed at. Can be identical to $s_{src}$ and/or $s_{dst}$ (see below).
$h_{src}$	The host that generates the traffic processed by this flow rule
$h_{dst}$	The host that receives the traffic processed by this flow rule
$SP = (s_{src}, \dots, s_f, \dots, s_{dst})$	The path between $h_{src}$ and $h_{dst}$ where $h_{src}$ is attached to $s_{src}$ and $h_{dst}$ is attached to $s_{dst}$
$\tau^{install}$	Point in time at which the flow rule is installed in the flow table of switch $s_f$
$\tau^{removed}$	Point in time at which the flow rule is removed from the flow table of switch $s_f$
$\delta$	Total number of bits sent by $h_{src}$ and thus also the number of bits processed by this flow rule
$b$	Constant bit rate of $h_{src}$ in bits/s
$n_{lifetime}$	Minimum lifetime of the flow rule. Static value, identical for all flow rules in the network.
$\vec{m}$	Not required, is implicitly given by $h_{src}$ and $h_{dst}$
$\vec{a}$	Not required, is implicitly given by the path between $h_{src}$ and $h_{dst}$ and the position of $s_f$ in this path
prio	Not required, has no impact

**Table 12.1:** Flow rule definition for the evaluation. The three variables in the bottom which are part of the original flow rule definition (Def. 2.8) are specified implicitly, i.e., are not included in the nine-tuple.

## 12.2 Important Terminology

This section introduces important definitions required in the following chapters. Sec. 12.2.1 and Sec. 12.2.2 define the terms scenario and scenario set. Sec. 12.2.3 defines the capacity reduction factor which is used here to model bottleneck situations of different severity. Sec. 12.2.4 defines the failure rate which is used to specify whether flow delegation can successfully handle a specific situation or not. Sec. 12.2.5 defines how flow delegation performance is measured in this thesis. Sec. 12.2.6 defines the flow table utilization ratio that is used here to distinguish scenarios based on their available free capacity. Sec. 12.2.7 summarizes other important terms.

### 12.2.1 Scenario

Core idea of the performed evaluation is it to create different scenarios with a wide range of characteristics – with respect to number of flow rules, traffic volume etc. – and investigate how the flow delegation approach deals with these characteristics.

#### Definition 12.1: Scenario

A **scenario** is a quadruple  $\langle S, H, Y, F \rangle$  where  $S$  is a set of switches,  $H$  is a set of hosts,  $Y$  is a set of links connecting switches / hosts and  $F$  is a set of flow rules where each flow rule is defined as a nine-tuple according to Table 12.1.

There are two other important parameters for a scenario:  $c^{\text{Link}}$  and  $c^{\text{Table}}$ .  $c^{\text{Link}}$  defines the link capacity between switches and is set to a constant value of 1Gbit/s.  $c^{\text{Table}}$  represents the flow table capacity of the switches. This value is set to the maximum flow table utilization in the scenario and will be later modified with the capacity reduction factor:

$$c^{\text{Table}} = u_{\max}^{\text{Table}} = \max_{s \in S, t \in T} (u_{s,t}^{\text{Table}}) \quad (12.1)$$

Set  $T$  is the set of all considered time slots where at least one flow rule is installed in the network.  $u_{s,t}^{\text{Table}}$  is the flow table utilization without flow delegation in switch  $s$  in time slot  $t$ . Note that all switches have the same flow table capacity  $c^{\text{Table}}$ , according to the assumptions in Sec. 12.1.

The generation process for creating scenarios is explained in detail below in Sec. 12.3.1. All scenarios considered in this work span over a time period of 400 seconds and consider up to 600.000 individual flow rules installed during this time period.

### 12.2.2 Scenario Set

Goal is it to investigate how flow delegation performs in different scenarios, i.e., without introducing bias by only selecting scenarios where the approach can work in its “comfort zone”. To support this idea and still allow reproducibility, the investigated scenarios are “bundled” in so-called scenario sets with well documented characteristics that are made available to the public (see Appendix D).

#### Definition 12.2: Scenario Set

A **scenario set** is a collection of  $n$  different scenarios created with random parameters in such a way that a) the contained scenarios cover a range of different characteristics and b) the different scenario sets are comparable to each other. The latter means the cumulative distribution function of certain key characteristics has to follow a pre-determined pattern.

The second part – comparability across scenario sets – is only required because some experiments do not allow for large scenario sets. An example for this is given in Appendix A.3.1. In this case, there are 729 different combinations of experiments to be executed for each single scenario which does not scale in terms of experiment execution time if  $n$  is too large. However, because the results of these experiments were also used with larger scenario sets, it is required that the two sets are somehow comparable to justify the outcome of those experiments.

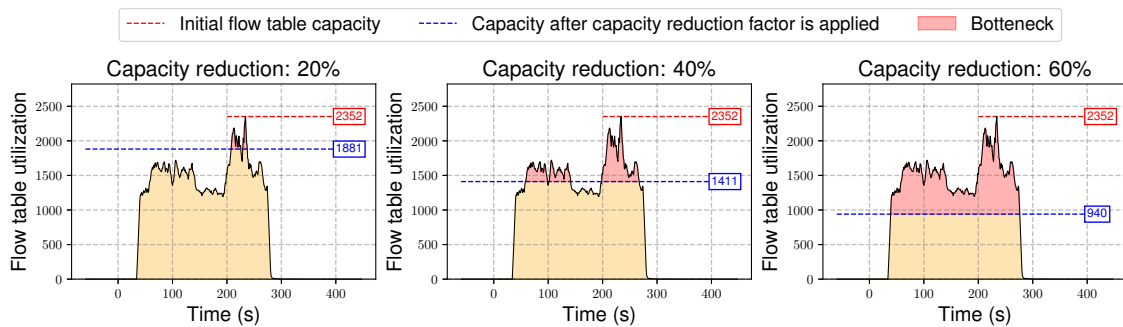
### 12.2.3 Capacity Reduction Factor

Next, it is important to define a way to distinguish scenarios with respect to “bottleneck severity”. One obvious candidate for such a distinction would be the number of flow rules that need to be relocated over the course of an experiment. However, this value cannot be easily used as an input parameter because it depends on too many other variables. A much simpler way to distinguish the severity of different bottleneck situations (which can be easily controlled from an experiment point of view) is the so-called capacity reduction factor.

#### Definition 12.3: Capacity Reduction Factor

The **capacity reduction factor** (capacity reduction for short) is given as a percentage value between 0% and 100% and specifies the size of the flow table compared to the maximum flow table utilization  $u_{\max}^{\text{Table}}$ . If the capacity reduction factor is set to 40%, for example, the flow table capacity of all switches is set to  $u_{\max}^{\text{Table}} - \frac{40}{100} * u_{\max}^{\text{Table}} = 0.6 * u_{\max}^{\text{Table}}$ .

This is a simple and intuitive way to describe bottleneck severity because higher factors will automatically result in more flow rules that need to be relocated over the course of an experiment, regardless of how the scenario is generated. Fig. 12.1 gives an example. It shows the flow table utilization over time for one switch of one scenario. The red dashed line marks the maximum flow table utilization  $u_{\max}^{\text{Table}}$ , which is 2352 flow rules in this example. The blue line represents the capacity of the flow table for different capacity reduction factors.



**Figure 12.1:** *Illustration of capacity reduction*

In the leftmost plot, the capacity reduction factor is set to 20% which results in a flow table capacity of 1881 flow rules. The two other plots show the same scenario with higher capacity reduction factors, i.e., with less capacity and thus more severe bottlenecks. Flow rules that need to be relocated over the course of an experiment are shown in red. It is easy to see that there is a direct correlation between capacity reduction and bottleneck severity, i.e., the number of flow rules that need to be relocated over the course of an experiment (the red area gets larger for higher factors).

And the above definition has another important benefit: it can be used as a performance metric for flow delegation. Take the three example plots in Fig. 12.1 that show the same scenario with different capacity reduction factors. We can now ask the following question to evaluate the performance: “How high can we set the capacity reduction factor before flow delegation is unable to mitigate the bottleneck?”. If the answer is 20%, only the situation in the leftmost plot of Fig. 12.1 can be handled<sup>2</sup>. If the answer is 40%, the performance is obviously better because the situation in the middle can be handled as well – and all situations in between.

However, it is not yet clear what it means that a situation can be handled by flow delegation. This is addressed first with the failure rate definition in the next section. The term flow delegation performance is formally introduced afterwards.

<sup>2</sup>And all situations with factors below 20%



### 12.2.4 Failure Rate

We say that flow delegation can “handle” a situation – i.e., a combination of scenario and capacity reduction factor as described in Fig. 12.1 – if all flow rules in the red area can be relocated to a remote switch. This, however, is only possible if remote switches with enough spare flow table capacity are available. If this is not the case, flow delegation will use a so-called backup switch (BS) (see Def. 9.5) to store relocated flow rules that do not fit in one of the remote switches. The backup switch is a fake remote switch without capacity constraints but disproportionately high cost that is only used if the RS-Alloc problem is getting infeasible otherwise.

The failure rate is now defined as the fraction of the flow rules that are not handled by a remote switch in hardware over the course of an experiment. This is an important performance metric for flow delegation. If the failure rate for a scenario with a specific capacity reduction factor is 0%, flow delegation can handle this specific situation without the help of the backup switch, i.e., all flow rules are handled in hardware. If the failure rate is above 0%, some flow rules are relocated to the backup switch and the situation cannot be handled, at least not without limitations. A more formal definition of the failure rate is shown below.

#### Definition 12.4: Failure Rate

The **failure rate** counts the amount of flow rules allocated to the backup switch over the course of an experiment. This value is multiplied with a normalization factor (denominator) that represents all used flow rules over the course of the experiment to get a percentage value:

$$\left( \frac{\sum_{t \in T} u_t^{\text{BS}}}{\sum_{s \in S} \sum_{t \in T} u_{s,t}^{\text{Table}}} \right) * 100 \quad (12.2)$$

$u_t^{\text{BS}}$  represents the flow rules allocated to the backup switch BS in time slot  $t$ .

Note that the above definition explicitly includes the time domain. Assume a trivial scenario with one hardware switch  $s_1$  and only a single flow rule that is installed in  $s_1$  for 100 consecutive time slots. A 0% failure rate means the flow rule is installed in  $s_1$  the whole time and the backup switch is never used. A 10% failure rate means the flow rule is relocated to the backup switch for 10 time slots because of a bottleneck (and the fact that there is no remote switch available). A 100% failure rate means the flow rule is relocated to the backup switch in all 100 time slots, i.e., over the course of the complete experiment.

### 12.2.5 Flow Delegation Performance

Flow delegation performance indicates “how well” flow delegation can deal with a specific situation. This is primarily determined by the failure rate, i.e., the amount of flow rules that cannot be relocated to another hardware switch – which should be close to 0%. However, the failure rate largely depends on the capacity reduction factor: it is much easier to achieve a 0% failure rate for lower capacity reduction factors. The term flow delegation performance brings these two aspects together:

#### **Definition 12.5: Flow Delegation Performance**

The **flow delegation performance** is defined as the maximum capacity reduction factor that can be applied to a scenario so that, when flow delegation is used with this scenario, the failure rate does not exceed a specified maximum. The maximum accepted failure rate is a configurable parameter that must be specified by the network operator. Acceptable maximum failure rates investigated in this thesis are 0% (no failures), 0.1%, and 1%. Failure rates higher than 1% are considered impracticable.

Note that flow delegation performance is always defined with respect to a specific scenario. For the following example of how to use this definition, assume the maximum accepted failure rate is 0%, i.e., all excess flow rules must be relocated to a remote switch. In other words: there is a “failure” if only a single flow rule has to be allocated to the backup switch<sup>3</sup>.

The performance of flow delegation with respect to a scenario can now be described as follows: A performance of XYZ% means flow delegation can handle the scenario with a maximum applied capacity reduction factor of XYZ% while still achieving a 0% failure rate. This implies two important statements:

- The scenario can be handled with 0% failure rate for all capacity reduction factors up to XYZ%
- If a higher capacity reduction factor is applied, the failure rate is above 0%.

And because the flow delegation approach should be tested against scenario sets rather than individual scenarios, the performance is usually specified for a percentile of multiple investigated scenarios. A typical performance result would be: “flow delegation can achieve a 0% failure rate for all capacity reduction factors up to XYZ% for 90% of the investigated scenarios”. The key performance value here is XYZ% because this value

<sup>3</sup>Note that the flow delegation approach does not really fail in the sense of an error / exception. It only means some flow rules are allocated to the backup switch because the network can not provide sufficient spare resources.

represents the bottleneck severity. But it is important to keep in mind that this value always has to be interpreted with respect to the percentile of the investigated scenarios and the maximum accepted failure rate.

### 12.2.6 Flow Table Utilization Ratio

It is expected – and will be shown later – that the available free capacity in the network correlates with flow delegation performance. To distinguish between scenarios with large and small amount of free flow table capacity, the following definition is introduced:

#### Definition 12.6: Flow Table Utilization Ratio

The **flow table utilization ratio** for a scenario is defined as the ratio between average flow table utilization of all switches in the scenario and the maximum flow table utilization of all switches. The latter is given as  $u_{\max}^{\text{Table}}$ . In lambda notation, the ratio is calculated as follows:

$$\frac{\frac{1}{|S|} \sum_{s \in S} \sum_{t \in T_s} \frac{1}{|T_s|} \sum_{f \in F_{s,t}} \lambda_{f,t}^a}{u_{\max}^{\text{Table}}} \quad (12.3)$$

In the above definition,  $T_s$  is the set of all time slots for switch  $s$  with at least one active flow rule. Time slots where no rules are installed are not considered. This new terminus is introduced here because the flow table utilization ratio is a good indicator for the available free flow table capacity in the network – which is a key parameter for flow delegation performance. In addition, it is easy to derive the flow table utilization ratio from basic monitoring data in an existing network. This is important because a measured ratio provides network operators with a first estimate on the potential performance to be expected from flow delegation.

### 12.2.7 Other Terminology

**Bottleneck Situation:** The term (bottleneck) situation refers to a specific scenario in combination with a capacity reduction factor. Recall the example in Fig. 12.1. The three plots show the same scenario but three different situations. The plot in the left shows the situation with a capacity reduction factor of 20%. The plot in the middle shows the same scenario but a more severe situation with a capacity reduction factor of 40%.

**Experiment:** An experiment is a single scenario applied to a specific parameterization of the flow delegation approach – e.g., using a specific algorithm for delegation template selection with specific weights and a specific look-ahead factor. The same scenario can be used in multiple experiments if different parameters are used.

**Experiment Series:** A series of experiments conducted to investigate a certain aspect of the flow delegation approach. Most experiment series represent a scenario set evaluated against a specific parameterization, i.e., flow delegation is used for each scenario in the scenario set with one or multiple parameterizations.

**Samples:** A single execution of DT-Select or RS-Alloc. Because the flow delegation algorithm is executed in multiple optimization periods, the associated linear programming problems are solved multiple times during one experiment. Assume a scenario with three switches  $s_1$ ,  $s_2$ ,  $s_3$  that spans a time period of 400 seconds.  $s_1$  suffers from a flow table capacity bottleneck for 25 seconds and  $s_3$  suffers from a bottleneck for 50 seconds (but fully overlapping with the first bottleneck). Given that the flow delegation algorithm is executed once per second, this example will result in 75 executions of DT-Select and 50 executions of RS-Alloc. Each of these executions is referred to here as a sample. Samples of DT-Select are called DTS-Samples. Samples for RS-Alloc are called RSA-Samples.

**Data set:** After an experiment is executed, scenario, parameters and results are stored in a result file. A collection of these result files for multiple experiments is called a data set. All results presented in this work are associated with exactly one data set. All data sets and the tools for browsing the data sets are publicly available (see Appendix D).

**Scenario ID:** Many of the plots and explanations used in this thesis specify a so-called scenario ID. This is a unique identifier between 0 and 500000 that can be used in the evaluation environment to recreate the exact same scenario, e.g., for reproducibility or to test the scenario with another parameterization.

## 12.3 Evaluation Methodology

The methodology used for evaluation is depicted in Fig. 12.2. It consists of four steps: scenario generation, pre-processing, experiment execution and post-processing. All conducted experiments follow this methodology. Before the individual steps are explained in more detail, the overall process is briefly summarized below.

- ① **Scenario generation:** Single scenarios are generated as scale-free topologies based on the Barabasi-Albert model [AB02] and populated with a configurable number of hosts, following the approach in [Coh+14b]. Afterwards, traffic and flow rules for the hosts are generated based on data from [Geb+12] and [JRB18]. Specific bottlenecks situations can be enforced in certain parts of the topology (hotspots) or certain time frames (temporal bottlenecks).
- ② **Pre-processing:** An initial scenario set with 500.000 scenarios is generated based on the process from step 1 with a randomized set of parameters. This initial scenario

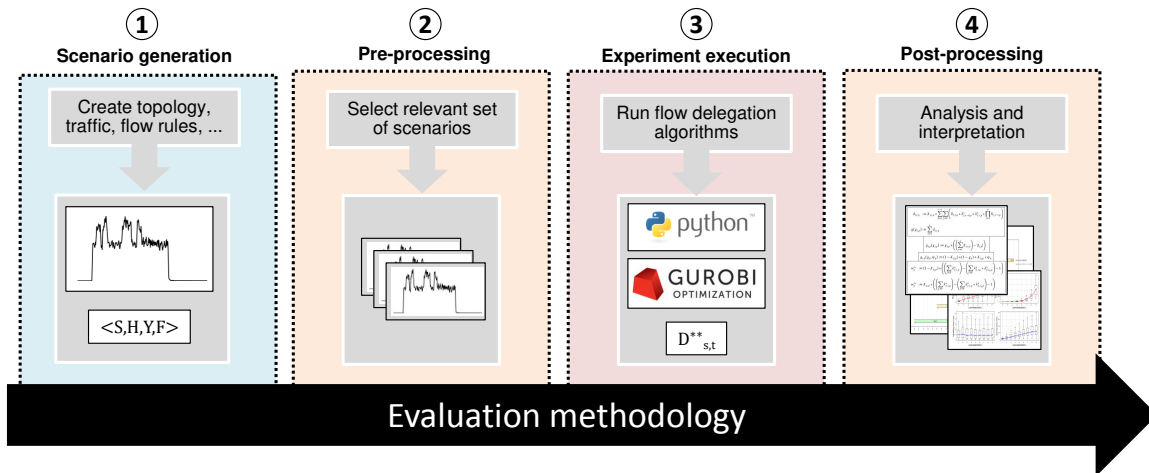


Figure 12.2: Evaluation methodology

set is then reduced in two steps to create scenario sets of smaller size with similar characteristics.

- ③ **Experiment execution:** Execution of the actual experiment which means DT-Select and RS-Alloc are calculated individually for each scenario with the given parameterization (which is specified by the experiment creator). Each such experiment results in a result file that consists of the scenario and the output of the two algorithms. It is also possible to validate results with a flow-level simulator (optional).
- ④ **Post-processing:** The data obtained from step ③ is analyzed analytically to determine the flow table utilization of the scenario with and without flow delegation (and several other metrics). Results of multiple experiments are stored in a data set. Final step is analysis and interpretation of the obtained data.

### 12.3.1 Scenario Generation

This section describes the scenario generation process. Goal is it to create a synthetic scenario with realistic topology and plausible characteristics for traffic and flow table utilization. Recall from Def. 12.1 that such a scenario is defined as a quadruple  $\langle S, H, Y, F \rangle$ .

A scenario is generated in two subsequent steps. The first step is **topology generation** which will return a three-tuple  $\langle S, H, Y \rangle$ .  $S$  is a set of switches,  $H$  is a set of hosts and  $Y$  represents the links between the switches and hosts. This is discussed in Sec. 12.3.1.1. The second and more complex step is **flow rule set generation**. The flow rule set  $F$  contains all flow rules installed into the switches in  $S$ . This is discussed in Sec. 12.3.1.2.

### 12.3.1.1 Topology Generation

The topology of a scenario  $\langle S, H, Y \rangle$  is created with Alg. 15 in two steps: The first step (line 5-12) creates links between the switches in  $S$  and the second step (line 13-18) connects each host in  $H$  with a switch in  $S$ . Links are specified as binary variables as it was introduced in Sec. 4.1:  $y_{s \rightarrow r}$  is set to 1 if a link exists between switch  $s \in S$  and  $r \in S$ . If no such link exists, the variable is set to 0. The same notation is used for links between hosts and switches.

The first step of Alg. 15 is based on the Barabasi-Albert model which is often used in the context of software-based networks [LPW16; Luo+16; Kos+17; Xu+17; Zha+19; OCF19]. It generates random scale-free topologies, i.e., topologies where the node degree distribution follows a power law. The model argues that the scale-free nature of a topology is based on two mechanisms that are commonly found in many real-world communication networks [AB02]: i) the network can be described as an open system where new nodes are added continuously starting from a small initial set of nodes and ii) the likelihood of connecting a new node to an existing node depends on the degree of the existing node. These two mechanisms are described in the model as growth and preferential attachment.

The algorithm has three simple input parameters: number of switches  $|S|$ , number of hosts  $|H|$  and a third parameter  $1 \leq m < |S|$ . The latter is a special parameter from the Barabasi-Albert model that serves two purposes. It defines the size of the initial set and also the number of links that are added in each iteration. Because the growth mechanism requires that at least one switch is not part of the initial set, the parameter is restricted to a value  $m < |S|$ . At initialization, the topology consists of  $m_0 = m$  switches and no links (line 2) – the initial set mentioned above that represents the starting point for the growth mechanism. New switches and links are then added in  $|S| - m_0$  iterations in the first loop (line 5). In each iteration, one new switch is added and connected to  $m$  existing switches, i.e., the model adds  $m$  new links per iteration (lines 6-10). To take preferential attachment into account, the endpoints for each of the  $m$  new links are chosen according to the probability in Eq. 12.4.  $r \in S$  is one of the already existing switches in  $S$ .

$$\Pi(r) = \frac{\text{degree}(r)}{\sum_{s \in S} \text{degree}(s)} \quad (12.4)$$

This will preferably attach new links to switches with higher node degree. After all remaining switches are added, the topology consists of  $|S|$  switches and  $m * (|S| - m_0)$  links and the model ensures that the node degree distribution of the switches follows a power law. In the second step, the hosts in  $H$  are randomly attached to one of the switches in  $S$  (lines 13-18).

**Algorithm 15:** Topology Generation

---

**Data:** number of switches  $|S|$  and hosts  $|H|$  to be generated, model parameter  $m$   
**Result:** topology information  $\langle S, H, Y \rangle$  for one scenario

```

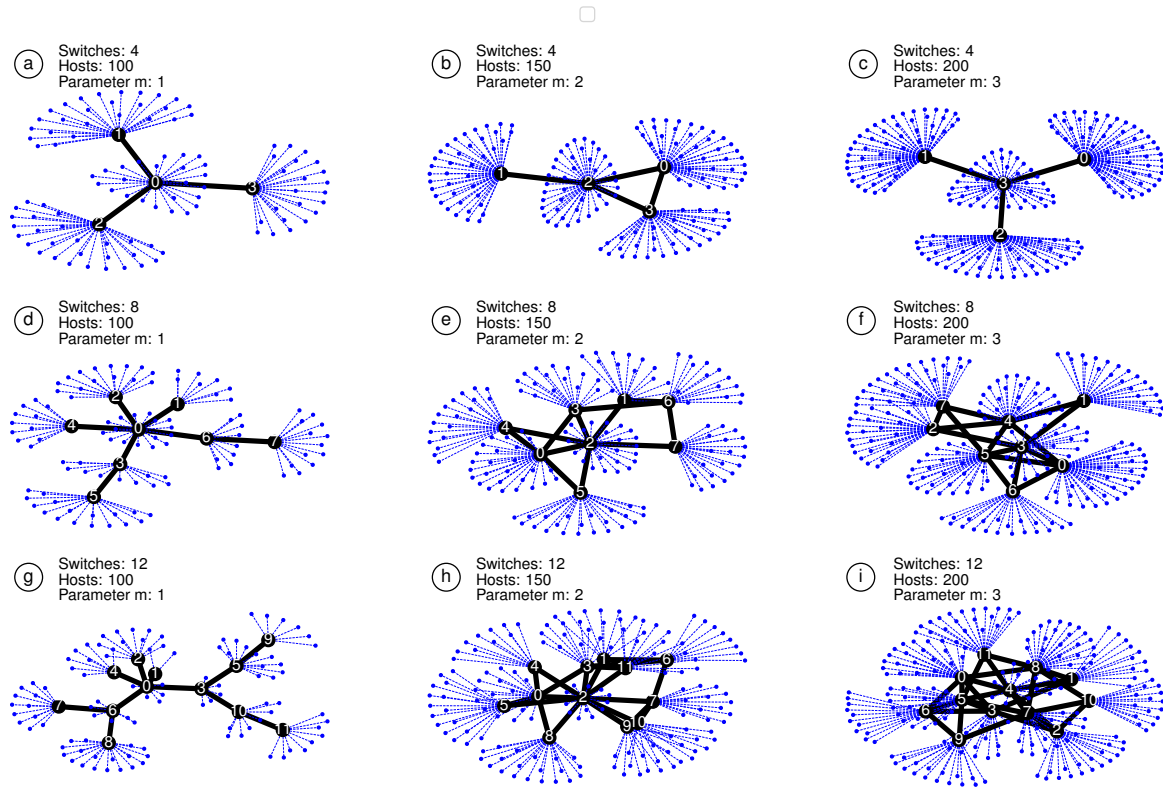
1 Function generate_topology( $|S|, |H|, m$ ):
2    $S = \{s_1, \dots, s_m\}$  // initial set of  $m$  switches
3    $H = \{\}$  // initial set of hosts (empty)
4    $Y = \{\}$  // initial set of links (empty)
   /* Step 1: create switch topology (Barabasi-Albert model) */
5   for  $s \in \{s_{m+1}, \dots, s_{|S|}\}$  do
   /* add  $m$  new bi-directional links between  $s$  and  $r \in S$  */
6     for  $i \in \{1, \dots, m\}$  do
7        $r \leftarrow$  select a switch  $r \in S$  with probability  $\Pi(r) = \frac{\text{degree}(r)}{\sum_{s \in S} \text{degree}(s)}$ 
8        $y_{s \rightarrow r} \leftarrow y_{r \rightarrow s} \leftarrow 1$ 
9        $Y \leftarrow Y \cup \{y_{s \rightarrow r}, y_{r \rightarrow s}\}$  // update link set
10      end
11       $S \leftarrow S \cup \{s\}$  // update switch set
12    end
   /* Step 2: attach hosts to topology */
13  for  $h \in \{h_1, \dots, h_{|H|}\}$  do
14     $s \leftarrow$  select a random switch  $s \in S$  (equally distributed)
15     $y_{s \rightarrow h} \leftarrow y_{h \rightarrow s} \leftarrow 1$ 
16     $Y \leftarrow Y \cup \{y_{s \rightarrow h}, y_{h \rightarrow s}\}$  // update link set
17     $H \leftarrow H \cup \{h\}$  // update host set
18  end
19  return  $\langle S, H, Y \rangle$ 
20 end

```

---

Fig. 12.3 shows examples of generated topologies with different parameters. The switches are shown as black circles labeled from 0 to  $|S| - 1$ . Links between switches are shown as black lines. The hosts are shown as small blue dots. Links between switches and hosts are shown as blue lines. Topology (a) in the top left corner, for example, was generated with  $|S| = 4$ ,  $|H| = 100$ , and  $m = 1$ . The topologies further to the right represent higher values of  $|H|$  and  $m$ . The topologies further to the bottom represent higher values of  $|S|$ .

Note that the Barabasi-Albert model was only chosen because it can be easily parameterized (with parameter  $m$ ) and creates very different but still realistic topologies. The created topologies, however, are not necessarily representative for all kinds of networks. Data center networks, for example, have a different structure in most cases. It is important to mention here that flow delegation was also successfully tested in data center topologies [BD17], i.e., the approach is not conceptually limited to a certain kind of topology.



**Figure 12.3:** Example topologies created with Alg. 15

### 12.3.1.2 Flow Rule Set Generation

Flow rule set generation is more complex than topology generation. The process takes the topology from the last section and creates the flow rules as follows:

- (1) Create a set of inter-arrival time values  $T_{iat}$ . Values in this set specify the time between two consecutive flow rule installations from the perspective of the controller. Assume the first flow rule is installed at some point in time  $\tau$  and  $T_{iat}$  is given as  $\{t_a, t_b, t_c, \dots\}$ . The second flow rule is then installed at  $\tau + t_a$ , the third flow rule is installed at  $\tau + t_a + t_b$  and so on.
- (2) Values in  $T_{iat}$  are artificially reduced to increase the number of flow rules installed per second. This can be done either for all values in the set using some linear factor to scale up the scenario as a whole. Or the reduction is restricted to specific time windows to create periods where the network temporarily suffers from increased load. The latter is referred to as “temporal bottlenecks” from here on.



- (3) Individual flow rules are then generated iteratively in five sub-steps:
- Select a random pair of hosts  $\langle h_{src}, h_{dst} \rangle$  from the topology
  - Calculate the number of bits sent from  $h_{src}$  to  $h_{dst}$  and the constant bit rate<sup>4</sup>
  - Calculate install and removal time for the flow rule(s) associated with the communication between  $h_{src}$  and  $h_{dst}$ . The install time is realized as a global time offset.
  - Create a new flow rule for each switch on the shortest path between  $h_{src}$  and  $h_{dst}$
  - Select next value in  $T_{iat}$  and add this value to the global time offset for consecutive flow rules (see step c)

The above process is shown in Alg. 16. It has two kinds of inputs: the topology  $\langle S, H, Y \rangle$  that was created in the last section with Alg. 15 and parameters to further specify how the rules are generated, given here as  $n_{xyz}$ . These parameters are explained in Table 12.2 below, listed in the order in which they are used in Alg. 16.

Parameter	Description
$n_{pairs}$	<b>Number of host pairs</b> $\langle h_{src}, h_{dst} \rangle$ selected from set $H$ . For each selected pair, one flow rule is installed in all switches on the path between $h_{src}$ and $h_{dst}$ .
$n_{iat\_scale}$	<b>Global scale parameter for inter-arrival time values</b> in $T_{iat}$ . Specifies the time required to install all flow rules in seconds. If set to 350s (default value), $T_{iat}$ is scaled in such a way that the sum of the inter-arrival time values equals 350 seconds.
$n_{bneck}$	<b>Number of temporal bottlenecks</b> added to the scenario. Each temporal bottleneck is further specified with a duration and an intensity value.
$n_{bneck\_duration}$	<b>Duration value for temporal bottlenecks</b> in seconds. A value of 10s means that all temporal bottlenecks last for 10 seconds.
$n_{bneck\_intensity}$	<b>Intensity value for temporal bottlenecks</b> given as a percentage value $> 100\%$ . All inter-arrival time values affected by a temporal bottleneck are scaled by $\frac{100}{n_{bneck\_intensity}}$ .

<sup>4</sup>As mentioned in Sec. 12.1, the bit rate for a single sender ( $h_{src}$ ) is assumed to be constant over time. Different senders, however, can have different constant bit rates.

$n_{\text{isr}}$	<b>Inter switch ratio.</b> Probability that $h_{\text{src}}$ and $h_{\text{dst}}$ are not attached to the same switch if a new pair is selected, given as a percentage value between 0% and 100%
$n_{\text{hs}}$	<b>Number of hotspots</b> in the scenario. A hotspot is a subset of the hosts in $H$ that are preferred when host pairs are selected. Is further specified with an intensity value.
$n_{\text{hs\_intensity}}$	<b>Intensity value for hotspots.</b> Determines the number of re-selections that are done if a selected host does not belong to a hotspot. Default value is 5.
$n_{\text{traffic\_scale}}$	<b>Global scale parameter for processed traffic</b> given as a percentage value $> 100\%$ . All traffic processed by a flow rule is scaled linearly by $\frac{100}{n_{\text{traffic\_scale}}}$ .
$n_{\text{lifetime}}$	<b>Minimum lifetime of a flow rule.</b> A global parameter given in seconds that is used for all flow rules.

**Table 12.2:** Parameters for flow rule set generation

Output of the algorithm is a set of flow rules  $F_s$  for each switch  $s \in S$ . These flow rules are specified as introduced in Sec. 12.1, i.e., as a nine-tuple. The remainder of this section will now explain the motivation and design decisions associated with the individual steps of Alg. 16<sup>5</sup>.

The **first step** (line 3) is construction of suitable inter-arrival time values  $T_{\text{iat}}$ . This is done with a gamma distribution based on results presented in [Geb+12]. The authors in [Geb+12] analyzed different distribution functions (exponential, log-normal, gamma) to model flow inter-arrival times. For that purpose, traffic of 600 customers of a German access provider was captured over a period of 14 days (3.3 TBytes of observed data). They found that a gamma distribution with shape  $k = 0.4754$  and scale  $\theta = 13.7300$  resulted in a good match with the observed data. The same approach and parameters are used here to model the inter-arrival time of flow rules. Note that the function in line 3 returns a set of  $n_{\text{pairs}}$  values, one for each flow rule installed in the third step below.

In the **second step** (line 4-5), the calculated inter-arrival time values are adjusted. Line 4 first applies a global scale parameter  $n_{\text{iat\_scale}}$ . This is required because the above gamma distribution was designed for a single switch while a scenario usually consists of multiple switches. However, instead of using  $\frac{1}{|S|}$  as a traffic-independent multiplier

<sup>5</sup>See <https://github.com/kit-tm/fdeval/blob/master/topo/scenario.py> for implementation details.

**Algorithm 16:** Flow Rule Set Generation

**Data:** topology  $\langle S, H, Y \rangle$  and parameters for flow rule generation ( $n_{xyz}$ , explained in Table 12.2)

**Result:** flow rule set  $F_s$  for each switch  $s \in S$

```

1 Function generate_flow_rule_sets( $\langle S, H, Y \rangle, n_{xyz}$ ):
2    $F_s = \{\}$   $\forall s \in S$ 
   /* Step 1: Calculate a set of  $n_{pairs}$  inter-arrival time values using a gamma
   distribution with shape  $k$  and scale  $\theta$  [Geb+12] */
3    $T_{iat} \leftarrow$  get_iat_samples_from_gamma_distribution( $k, \theta, n_{pairs}$ )
   /* Step 2: Modify the set of inter-arrival time values to include global
   scale factor and temporal bottlenecks */
4    $T_{iat} \leftarrow$  apply_global_scale_factor( $T_{iat}, n_{iat\_scale}$ )
5    $T_{iat} \leftarrow$  apply_temporal_bottlenecks( $T_{iat}, n_{bneck}, n_{bneck\_intensity}, n_{bneck\_duration}$ )
   /* Step 3: Iteratively add new flow rules */
6    $\tau^{offset} \leftarrow 10$ 
7   for  $i \in \{1, \dots, n_{pairs}\}$  do
   /* Step 3a: Select a new random host pair based on inter switch ratio
   ( $n_{isr}$ ) and topology hotspot parameters ( $n_{hs}$  and  $n_{hs\_intensity}$ ) */
8    $\langle h_{src}, h_{dst} \rangle \leftarrow$  get_host_pair( $H, n_{isr}, n_{hs}, n_{hs\_intensity}$ )
   /* Step 3b: Calculate flow length in bits based on [JRB18] and select
   bit rate for the sending host */
9    $\delta \leftarrow$  get_sample_from_jurkiewicz_mixture()
10   $b \leftarrow$  get_bitrate( $\delta, n_{traffic\_scale}$ )
   /* Step 3c: Calculate install and removal time for the new flow rule */
11   $\tau^{install} \leftarrow \tau^{offset}$ 
12   $\tau^{remove} \leftarrow \tau^{offset} + \max(\min(\frac{\delta}{b}, 35), n_{lifetime})$ 
   /* Step 3d: Calculate shortest path between  $h_{src}$  and  $h_{dst}$  and add flow
   rule to the flow rule set of each switch on the path */
13   $sp = \{s_{src}, \dots, s_{dst}\} \leftarrow$  get_shortest_path( $h_{src}, h_{dst}$ )
14  for  $s \in \{s_{src}, \dots, s_{dst}\}$  do
15     $f \leftarrow \langle s, h_{src}, h_{dst}, sp, \tau^{install}, \tau^{remove}, \delta, b, n_{lifetime} \rangle$ 
16     $F_s \leftarrow F_s \cup \{f\}$ 
17  end
   /* Step 3e: Increase offset by next inter-arrival time value and
   continue with the next flow rule */
18   $\tau^{offset} \leftarrow \tau^{offset} + T_{iat}[i]$ 
19 end
20 return  $F_s$ 
21 end

```

(which is a valid alternative approach),  $\frac{\sum T_{iat}}{n_{iat\_scale} * 1000}$  is used here. That means all values in  $T_{iat}$  are multiplied with this value. This ensures that flow rules are always installed over a time period of  $n_{iat\_scale}$  seconds which makes it easier to design and plot experiments. The additional factor of 1000 in the denominator is required because inter-arrival time values are given in milliseconds and the scale parameter is given in seconds.

Line 5 further adjusts the values in  $T_{iat}$  to emulate **temporal bottlenecks**. The number of temporal bottlenecks to be considered is given as  $n_{bneck}$ . For each temporal bottleneck, a random index  $x$  is selected between 1 and  $|T_{iat}|$ . This index represents the start of the temporal bottleneck. Next, a second index  $y > x$  is selected so that  $y$  is minimal and Eq. (12.5) is fulfilled, i.e., the bottleneck spans a duration of  $n_{bneck\_duration}$  seconds.

$$\lfloor \frac{\sum_{i=x}^{x+y} T_{iat}[i]}{1000} \rfloor \geq n_{bneck\_duration} \quad (12.5)$$

All values in  $|T_{iat}|$  between index  $x$  and index  $y$  are then reduced to increase the number of flow rules installed in this time period, i.e., they are multiplied with a value  $< 1$ . This multiplier is determined by the scale parameter for bottleneck intensity  $n_{bneck\_intensity}$  which is given as a value  $> 100$ . A value of 125, for example, means that the inter-arrival times in  $T_{iat}$  are multiplied with  $\frac{100}{125} = 0.8$  during the bottleneck. Higher intensity values result in lower multipliers and a higher number of installed flow rules per second. All temporal bottlenecks currently have the same duration and intensity. To simulate that such temporal bottlenecks do not occur instantaneously, the multipliers ( $y - x$  in total) are modeled as a normal distribution with  $\mu = \frac{100}{n_{bneck\_intensity}}$ . Smaller multipliers are used if the indices are close to  $x$  and  $y$  (at the “edge” of the temporal bottleneck) and larger multipliers are used for indices near  $x + \frac{y-x}{2}$  (at the center). The subsequent section gives examples of how this mechanism works with different parameterizations.

The **third step** (line 7-19) will create the individual flow rules, roughly following the methodology used in [Coh+14b]. It consists of five sub steps.

**Sub step 3a** (line 8) selects a random pair of hosts  $\langle h_{src}, h_{dst} \rangle$  from set  $H$ . The selection is controlled by three parameters. The inter switch ratio  $n_{isr}$  determines the probability that the two selected hosts are attached to two different switches so that traffic between  $h_{src}$  and  $h_{dst}$  has to be forwarded over at least two switches. With  $n_{isr} = 50\%$ , half of the selected pairs is attached to different switches. With  $n_{isr} = 20\%$ , only 20% of the pairs are attached to different switches. This parameter is important because higher inter switch ratios are more likely to cause bottlenecks in switches with high node degree (because less traffic is exchanged locally at one switch).

The two other parameters that control the selection of the host pairs are referred to as hotspot parameters. A **hotspot** is a subset  $H_{hs} \subseteq H$  and hosts in this subset are preferred in the host pair selection process. The number of hotspots is given as  $n_{hs}$ . For each

hotspot, the hosts attached to a randomly selected switch are added to  $H_{hs}$ . The hotspot mechanism itself works as follows. Each time a new host pair is selected, it is checked whether  $h_{src}$  is present in  $H_{hs}$ . If this is the case, the selection process is finished. If this is not the case, however, a new pair is selected. The maximum number of re-selections is constrained by the second hotspot parameter  $n_{hs\_intensity}$ . Using this mechanism, bottlenecks can be simulated for arbitrary switches – including those with a low node degree. Examples are provided in the next section.

**Sub step 3b** (line 9-10) generates the traffic. This includes the number of bits sent from  $h_{src}$  to  $h_{dst}$  and the bitrate of the sending host. The number of bits are calculated in a similar way as the inter-arrival time values. Because [Geb+12] does not provide distributions for this use case, we instead use the results presented in [JRB18]. The authors used a very similar setup, collected 275 TBytes of traffic with four billion flows in a campus network (AGH University of Science and Technology) and extracted different mixture models that are publicly available on GitHub<sup>6</sup>. The concrete mixture used to create a sample for  $\delta$  in line 9 makes use of several uniform and log-normal distributions and can be found here: <https://github.com/kit-tm/fdeval/blob/master/topo/static.py><sup>7</sup>. Given the number of bits  $\delta$ , the bitrate of the sending host has to be determined which is done in line 10. The idea is that high volume flows are transmitted with higher bitrate. The relationship between  $\delta$  and  $b$  is modeled as a simple square root function. A step function representing a static set of bit rates for different classes of applications was also tested with similar results but discarded because of the required additional parameters.

The traffic scale parameter  $n_{traffic\_scale}$  is used for scenarios where the flow rules process more traffic, e.g., because multiple hosts are aggregated behind one rule. If used,  $\delta$  is multiplied with  $\frac{100}{n_{traffic\_scale}}$ , i.e., the number of bits is increased linearly. Without this parameter, the majority of the created flow rules will process a very small amount of traffic (less than 1000 bytes) which is a favourable situation for the flow delegation approach because the amount of additional traffic caused by relocation is small (would lead to biased results).

**Sub step 3c** (line 11-12) calculates the install and removal times for the new flow rule. The install time is determined with a global offset variable –  $\tau^{offset}$  in line 6 – which is updated in each iteration in sub step 3e. The removal time is calculated with the help of  $\delta$  and  $b$ . Parameter  $n_{lifetime}$  represents the minimum lifetime of all flow rules and is used here if  $\frac{\delta}{b}$  is too small – e.g., only a couple of milliseconds. This is similar to the idle

<sup>6</sup>See <https://github.com/piotrjurkiewicz/flow-models>, last accessed 2020-02-25. Mixtures for flow duration and inter-arrival time are unfortunately not yet available here.

<sup>7</sup>The original version of this mixture given as a set of json files created by Piotr Jurkiewicz can be found here: [https://github.com/piotrjurkiewicz/flow-models/tree/master/data/agh\\_2015/mixtures/all/length](https://github.com/piotrjurkiewicz/flow-models/tree/master/data/agh_2015/mixtures/all/length), last accessed 2020-02-25

timeout mechanism used in OpenFlow networks. Because scenarios in this evaluation are limited to 400 seconds, a static maximum lifetime of 35 seconds is used. This is necessary because the above process also generates a small amount ( $< 0.001\%$ ) of flow rules that would last for hundreds of seconds which makes plotting and exception handling more difficult.

**Sub step 3d** (line 13-17) calculates the shortest path between  $h_{src}$  and  $h_{dst}$  and adds a new flow rule to set  $F_s$  if  $s$  is included in the shortest path. It is easy to see that all nine variables specified in Table 12.1 are available at this point. The new flow rule is constructed in line 15 and added to  $F_s$  in line 16.

**Sub step 3e** (line 18) increases the offset that is used in the next iteration in sub step 3c. It therefore extracts the next value from the set of inter-arrival time values ( $T_{iat}[i]$ ) and adds it to  $\tau^{offset}$ . The procedure then continues with sub step 3a until all pairs are handled, i.e., there are  $n_{pairs}$  iterations in total.

### 12.3.1.3 Examples

This section shows examples for flow rule set generation. To better illustrate the idea, the majority of the parameters is kept static<sup>8</sup>. All examples are created with 6 switches and 100 hosts. Parameter  $m$  for the Barabasi-Albert model is set to 1. In addition, the global seed for the pseudo random number generator was set to a fixed value so that all examples use the same topology.

The parameters for flow rule generation are listed in Table 12.3. The number of selected pairs ( $n_{pairs}$ ) is set to 100.000 in all five examples. The global scale parameter for the inter-arrival time values ( $n_{iat\_scale} = 350s$ ), the global scale parameter for processed traffic ( $n_{traffic\_scale} = 100\%$ ) and the minimum lifetime of the flow rules ( $n_{lifetime} = 3s$ ) are also kept static across the examples. The bottleneck-specific parameters, however, are varied to highlight different aspects:

- **Example 1:** low inter switch ratio, ignores all other bottleneck-specific parameters
- **Example 2:** high inter switch ratio, ignores all other bottleneck-specific parameters
- **Example 3:** uses a hotspot, i.e., the hosts attached to one random switch (red circle) are selected more frequently
- **Example 4:** uses two temporal bottlenecks that last for 50s each
- **Example 5:** combines a hotspot with five partially overlapping temporal bottlenecks

<sup>8</sup>The scenarios shown here are not included in the actual evaluation. They were created manually, only for illustration purposes in this section.

Parameter	Example 1 (Fig. 12.4)	Example 2 (Fig. 12.5)	Example 3 (Fig. 12.6)	Example 4 (Fig. 12.7)	Example 5 (Fig. 12.8)
$n_{\text{pairs}}$	100.000				
$n_{\text{iat\_scale}}$	350s				
$n_{\text{bneck}}$	-	-	-	2	5
$n_{\text{bneck\_duration}}$	-	-	-	50s	50s
$n_{\text{bneck\_intensity}}$	-	-	-	150%	150%
$n_{\text{isr}}$	20%	75%	50%	50%	50%
$n_{\text{hs}}$	-	-	1	-	1
$n_{\text{hs\_intensity}}$	-	-	5	-	5
$n_{\text{traffic\_scale}}$	100%				
$n_{\text{lifetime}}$	3s				

**Table 12.3:** Flow rule generation parameters for the five examples in Sec. 12.3.1.3. The parameters are explained in Table 12.2. Ignored parameters are shown as a minus sign.

**Example 1:** The topology – which is the same for all five examples – is shown in the left. Switches are shown as black circles labeled from 0 to 5. Links between switches are shown as black lines. The hosts are shown as small blue dots. Links between switches and hosts are shown as blue lines. Example 1 in Fig. 12.4 is now constructed with an inter switch ratio of 20%, i.e., traffic exchanged between two hosts has to be forwarded over more than one switch in 20% of the cases. This means the majority of the generated flow rules was only installed in a single switch.

This results in a scenario where the flow tables of the switches are used more or less uniformly, simply because the host pairs are selected uniformly as well. The flow table utilization over time for this case is shown in the right of Fig. 12.4. The black circles refer to the switch number in the topology. Only the switches in the center of the topology – primarily switch 0 – have a slightly higher flow table utilization which can be explained with the inter switch ratio of 20%.

Another important aspect that can be observed in all six plots on the right of Fig. 12.4 is the global scaling mechanism for the inter-arrival time values. Because  $n_{\text{iat\_scale}}$  is set to 350 seconds, the values in  $T_{\text{iat}}$  are scaled in such a way that all flow rules are installed within a 350 second period.

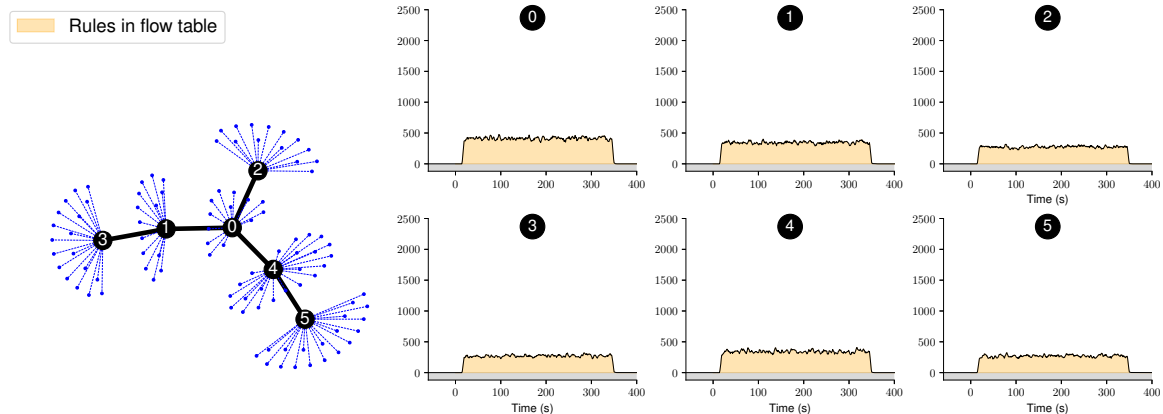


Figure 12.4: Example 1 with small inter switch ratio of 20%

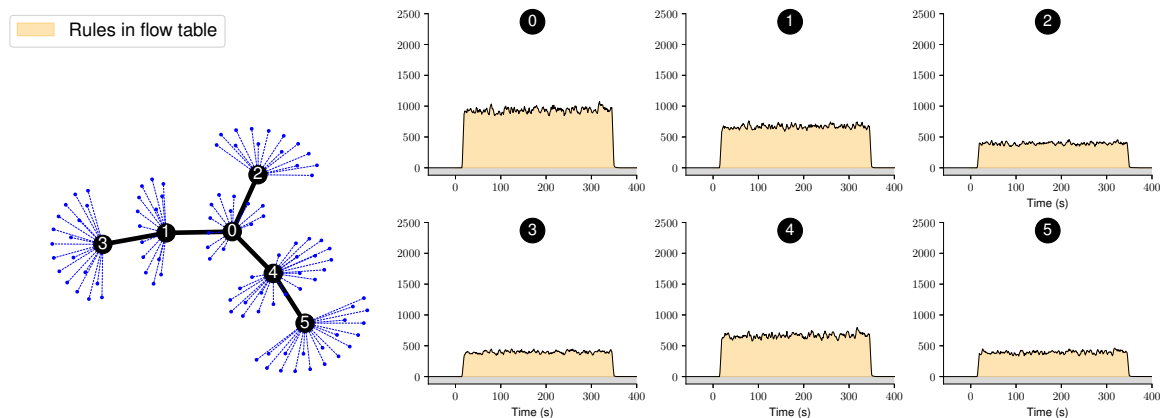


Figure 12.5: Example 2 with large inter switch ratio of 75%

**Example 2:** This example is identical to example 1 except that the inter switch ratio was increased from 20% to 75%. In this case, the majority of the generated flow rules is installed in multiple switches (in contrast to before). It can be seen in Fig. 12.5 that the flow table utilization differs based on the node degree. Switches with a higher degree in the center of the topology (switch 0) require significantly more flow rules than switches with a smaller degree at the edge (switches 2, 3, and 5). This is because host pairs are still selected uniformly. The probability that a selected pair creates a shortest path including switch 0 (with high node degree) is obviously much higher than the probability that the path includes switch 3 (with small degree).

**Example 3:** In this example, the inter switch ratio is set to 50%. But unlike before, the host pairs are not selected uniformly any more due to the hotspot mechanism. As shown by the red circle in the topology in Fig. 12.6, switch 2 was randomly selected as a hotspot.



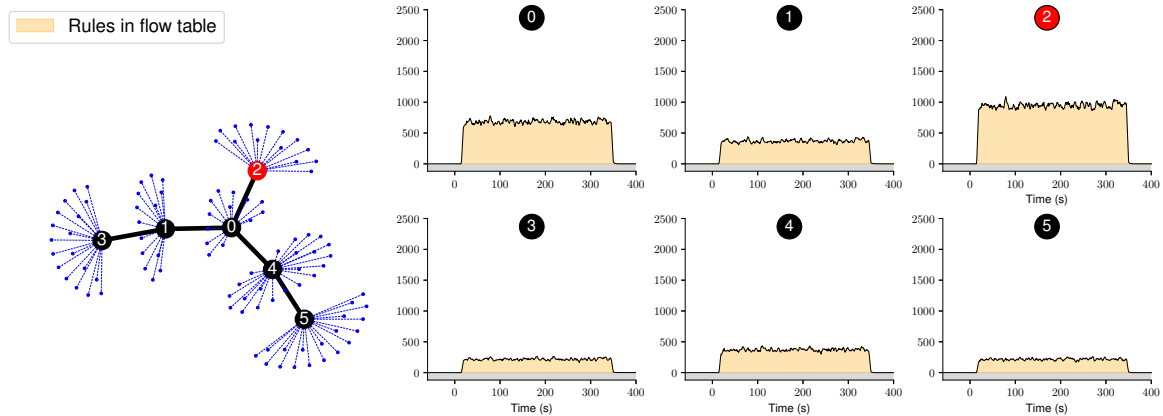


Figure 12.6: Example 3 with a hotspot (red circle) at switch 2

This means all hosts attached to switch 2 were added to set  $H_{hs}$ . Now, when a new host pair is selected, it is checked whether  $h_{src}$  is present in  $H_{hs}$ . If this is not the case, a new pair is selected which grants another chance to select a host from  $H_{hs}$ . This re-selection is done up to 5 times in this example because  $n_{hs\_intensity}$  is set to 5. As a result, the probability that a flow rule has to be installed in switch 2 is increased. This effect is clearly visible in the large flow table utilization of switch 2 in the top right of Fig. 12.6.

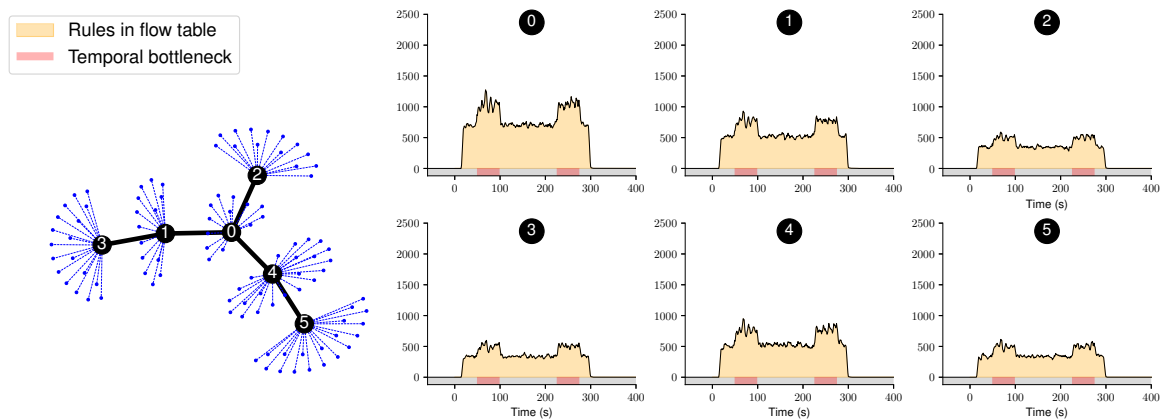
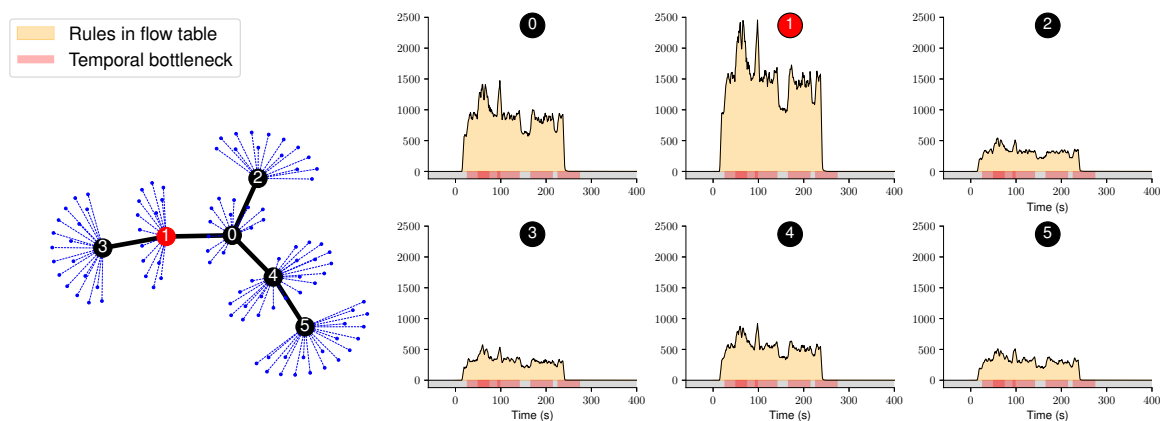


Figure 12.7: Example 4 with two temporal bottlenecks

**Example 4:** This example illustrates the mechanism behind the temporal bottlenecks. Because  $n_{bneck} = 2$ , there are two distinct temporal bottlenecks. These two bottlenecks are visualized with small red rectangles in Fig. 12.7, directly above the x-axis. The first temporal bottleneck starts at the 50s mark. The second temporal bottleneck starts after 230s. Both last for approx. 50s. Because temporal bottlenecks are applied to the global

set of all inter-arrival time values, all switches experience an increase in installed flow rules in the respective time windows. For the same reasons as above, the effect is stronger for switches with high node degree.

Unlike the hotspot mechanism, temporal bottlenecks reduce the overall time period between the first and last installed flow rule because a subset of the values in  $T_{iat}$  is multiplied with a value  $< 1$ . Because the global scale parameter  $n_{iat\_scale}$  is applied prior to the temporal bottlenecks, the total duration is reduced. This effect can be observed in the six plots in the right of Fig. 12.7. While previous examples span over a time period of 350 seconds, this example is finished after approx. 300 seconds. This could be altered by using the global scale parameter after the temporal bottlenecks. However, in this case, the temporal bottlenecks would be artificially extended to a value larger than  $n_{bneck\_duration}$ . Note that this effect is not important for the interpretation of the scenarios because a real network would populate the flow tables continuously, i.e., is not restricted to a maximum time window.



**Figure 12.8:** Example 5 with hotspot (red circle) and temporal bottlenecks

**Example 5:** The last example in Fig. 12.8 shows a combination of the three introduced bottleneck mechanisms – inter switch ratio, hotspots, and temporal bottlenecks. The inter switch ratio is set to 50%, the number of temporal bottlenecks is set to 5 and there is a hotspot in switch 1. Note that the temporal bottlenecks can “overlap” if the duration value is large enough because the indices for the start of the bottleneck are selected randomly. This can result in situations where the demand for flow rules increases very fast, as it is shown, for example, in switch 1.

### 12.3.2 Pre-Processing

The previous section explained how a single scenario is generated, which is step ① in the evaluation process outlined in Fig. 12.2. Step ② (pre-processing) is discussed in this section and consists of two tasks:

- (1) Generation of 500.000 scenarios with randomized **parameter selection**
- (2) Selection of relevant scenarios to **construct the scenario sets** used for evaluation

#### 12.3.2.1 Parameter Selection

Goal is it to test the flow delegation approach with scenarios with a wide range of characteristics – with respect to number of flow rules, traffic volume etc. However, due to the high number of parameters in the scenario generation process, it is not possible to simply test with all combinations of parameters. And it is also difficult to select a subset of the parameters manually without excessive testing (which combination of parameters leads to which result). Manual selection also comes with the danger of introducing bias.

For the above reasons, all parameters are selected randomly – and independently from each other in the majority of the cases. This leads to a variety of scenarios with different combinations of parameters. Table 13.1 shows the different parameters that are required and the range of allowed values, specified by minimum and maximum. All parameters are given as integer values. Parameters labeled with U (last column) are selected uniformly between the specified minimum and maximum. Parameters labeled with N use a normal distribution to prefer the smaller values. This is used if higher values of the parameter near the maximum tend to create extreme scenarios, for example, scenarios with a very high number of temporal bottlenecks.

Parameters labeled with S are also selected uniformly but from a prepared set, i.e., not all values between minimum and maximum are available<sup>9</sup>. Parameters labeled with \* depend on the selection of another parameter. This is important for the  $m$  parameter used for topology generation (has to be smaller than  $|S|$ ) and for the number of hosts where it makes sense to have a dependency on the number of switches.

Parameter	Description	Min	Max	
$ S $	Number of switches	2	15	U
$ H $	Number of hosts	$5 *  S $	$20 *  S $	S (*)
$m$	Used for topology generation	1	$ S  - 1$	U (*)

<sup>9</sup>See <https://github.com/kit-tm/fdeval/blob/master/topo/custom/pbce/exp800-2.py> for details

$n_{seed}$	Seed	1	1.000.000	U
$n_{reduction}$	Capacity reduction factor	20	99	U
$n_{pairs}$	Number of host pairs	25.000	250.000	U
$n_{iat\_scale}$	Global scale for $T_{iat}$	280	350	S
$n_{bneck}$	Number of temporal bottlenecks	0	20	N
$n_{bneck\_duration}$	Bottleneck duration	1	50	U
$n_{bneck\_intensity}$	Bottleneck intensity	110	280	N
$n_{isr}$	Inter switch ratio	20	80	S
$n_{hs}$	Number of hotspots	0	4	N
$n_{hs\_intensity}$	Hotspot intensity	0	10	N
$n_{traffic\_scale}$	Global traffic scale	25	12.500	S
$n_{lifetime}$	Minimum flow rule lifetime	1	5	U

(\*) = Parameter selection depends on another parameter

U = Parameter is selected uniformly

N = Parameter is selected following a normal distribution

S = Parameter is selected uniformly from a prepared set

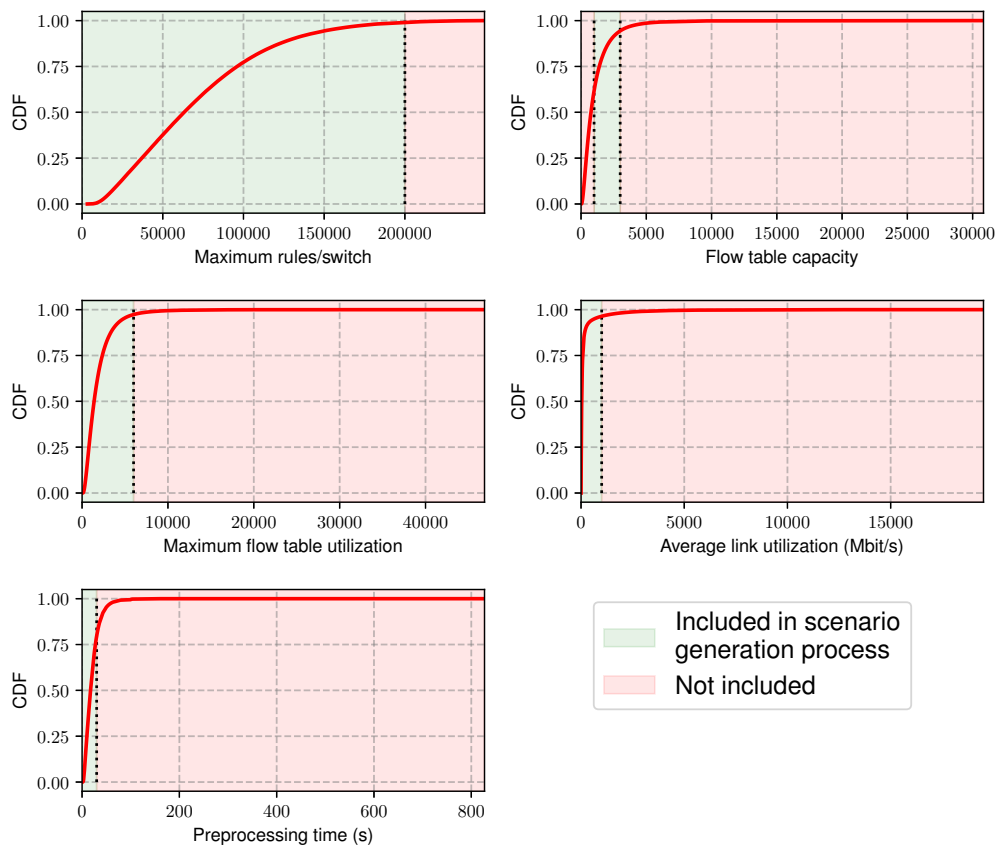
**Table 12.4:** *Random parameter selection*

The characteristics of the created scenarios with the above parameters are discussed in the next section.

### 12.3.2.2 Construction of Scenario Sets

Not all 500.000 scenarios generated with the random parameters in Table 13.1 can be used in the evaluation (resource constraints). It is therefore necessary to select a subset of the scenarios. More precisely, two scenario sets are generated. One large scenario set with 5.000 scenarios called  $Z_{5000}$  and a smaller set with 100 scenarios called  $Z_{100}$ . The smaller set is needed because certain experiments require many different combinations of experiments to be executed for each single scenario which does not scale with  $Z_{5000}$  in terms of experiment execution time.

In a first step, scenarios that are unrealistic or would consume too much evaluation resources are excluded from the original set of 500.000 scenarios. This reduces the amount of scenarios to 98.821 and is based on five criteria. These criteria are shown with their cumulative distribution functions in Fig. 12.9. The dotted black vertical lines indicate the boundaries used to include or exclude a scenario. In the end, only scenarios that fall into the green region are considered.



**Figure 12.9:** Characteristics of 500.000 random scenarios before selection

- The first plot in the top left shows the maximum number of flow rules that are installed into a single switch over the period of an experiment. Scenarios with more than 200.000 rules are excluded because such a high number of rules is not considered realistic. However, this excludes only a small fraction of the scenarios (< 1%).
- The second plot in the top right shows the flow table capacity of the switches taking the capacity reduction factor into account. This criterion is restricted to values between 1.000 and 3.000 which represents realistic values for currently available hardware switches.
- The third plot shows the maximum flow table utilization if flow delegation is not used. This criterion is restricted to values below 6.000. Given that the minimum table capacity is 1.000, this allows for scenarios where the utilization is 600% above the maximum capacity – an extreme case for the flow delegation approach.

- The fourth plot shows the average link utilization of all links between the switches in the topology. Because the maximum link capacity is set to 1000 Mbit/s, scenarios with higher values are excluded.
- The last plot shows the pre-processing time required for the scenario generation process. Because this value is strongly correlated with the experiment execution time, it is used here to exclude scenarios that will most likely run into a timeout later on because of limited evaluation resources (CPU processing time).

The characteristics of the reduced set with now 98.821 scenarios are shown in detail in Appendix B. A subset of the characteristics – including the five criteria from above – can also be seen in Fig. 12.10 and Fig. 12.11 (the dashed black line).

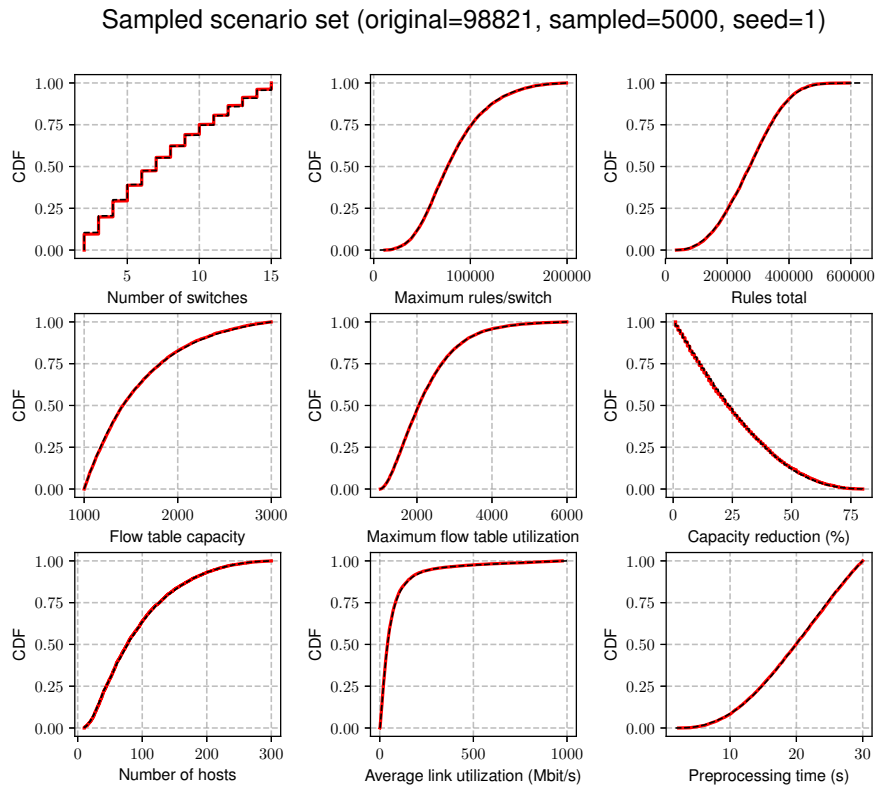
To further reduce the number of scenarios to 5000 and 100, the 98.821 scenarios are processed with a simple sampling algorithm. The idea is to select a subset of the scenarios so that the difference between the original distribution of the characteristics and the sampled distribution of the characteristics is minimized<sup>10</sup>. The result for  $Z_{5000}$  with 5000 scenarios is shown in Fig. 12.10.

It can be seen that the distribution of the original scenario set with 98.821 scenarios – shown as the dashed black line – and the distribution for the subset with 5000 scenarios is almost identical. Benefit of this approach is that the subset contains a similar collection of scenarios which allows it to apply results achieved with a smaller set to the larger set. The result for  $Z_{100}$  with only 100 scenarios is shown in Fig. 12.11. Despite the small amount of scenarios, it is still possible to approximate the original distribution in this case.

The two scenario sets are available online. Please refer to Appendix D for further details on how these data sets can be used and reproduced. Furthermore, Appendix B.1 shows the CDFs for all characteristics and parameters. In addition, Appendix B.3 gives a detailed overview of individual scenarios that can be found in  $Z_{100}$ . It is also important to mention that the results in this work were not only tested against  $Z_{100}$  and  $Z_{5000}$ . The experiments were also successfully validated with alternative sets for  $Z_{100}$  (two alternative sets) and  $Z_{5000}$  (one alternative set). These alternative scenario sets are also available online.

---

<sup>10</sup>The sampling algorithm is implemented in [https://github.com/kit-tm/fdeval/blob/master/plottter/agg\\_01\\_d001\\_scenario\\_selection.py](https://github.com/kit-tm/fdeval/blob/master/plottter/agg_01_d001_scenario_selection.py). It takes a random subset and calculates seven different percentiles (1%, 25%, 50%, 75%, 90%, 95%, and 99%) for 9 selected characteristics. It then iteratively re-selects scenarios to improve the mean error between the percentiles of the original scenario set and the sampled scenario set, averaged over the selected characteristics.



**Figure 12.10:** *Nine characteristics of the scenarios in  $Z_{5000}$  (red line) compared to the same characteristics of the original set with 98.821 scenarios (dashed black line)*

### 12.3.3 Experiment Execution

The two previous sections explained how scenarios are generated in step ① and how scenario sets are created in step ②. The third step of the methodology is experiment execution, i.e., flow table utilization and several other metrics are investigated for the created scenarios, with and without flow delegation.

#### 12.3.3.1 Environment

The evaluation environment created for this work is written in Python 3.6.8 and available on Github (<https://github.com/kit-tm/fdeval>). The only external dependency despite common open source python libraries (such as networkx, numpy, scipy, etc) is Gurobi which is used in version 8.0.1 for modeling and solving ILPs. The code will not be explained here in detail. However, the following table lists selected entry points to important source files in the code.

---

Entry point	Description
main.py	Executes a single experiment <a href="https://github.com/kit-tm/fdeval/blob/master/main.py">https://github.com/kit-tm/fdeval/blob/master/main.py</a>
test.py	Executes an experiment series <a href="https://github.com/kit-tm/fdeval/blob/master/test.py">https://github.com/kit-tm/fdeval/blob/master/test.py</a>
experiment.py	Represents one instance of an experiment <a href="https://github.com/kit-tm/fdeval/blob/master/topo/custom/pbce/exp800-2.py">https://github.com/kit-tm/fdeval/blob/master/topo/custom/pbce/exp800-2.py</a>
scenario.py	Implements the scenario generation process <a href="https://github.com/kit-tm/fdeval/blob/master/topo/scenario.py">https://github.com/kit-tm/fdeval/blob/master/topo/scenario.py</a>
dts.py	Implements DT-Select algorithms <a href="https://github.com/kit-tm/fdeval/blob/master/engine/solve_dts.py">https://github.com/kit-tm/fdeval/blob/master/engine/solve_dts.py</a>
rsa.py	Implements RS-Alloc algorithms <a href="https://github.com/kit-tm/fdeval/blob/master/engine/solve_rsa.py">https://github.com/kit-tm/fdeval/blob/master/engine/solve_rsa.py</a>
translate_dts.py	Translation to lambda notation for DT-Select <a href="https://github.com/kit-tm/fdeval/blob/master/engine/solve_dts_data.py">https://github.com/kit-tm/fdeval/blob/master/engine/solve_dts_data.py</a>
translate_rsa.py	Translation to lambda notation for RS-Alloc <a href="https://github.com/kit-tm/fdeval/blob/master/engine/solve_rsa_data.py">https://github.com/kit-tm/fdeval/blob/master/engine/solve_rsa_data.py</a>
simulator.py	Discrete event simulator for flow rule delegation <a href="https://github.com/kit-tm/fdeval/blob/master/core/simulator.py">https://github.com/kit-tm/fdeval/blob/master/core/simulator.py</a>

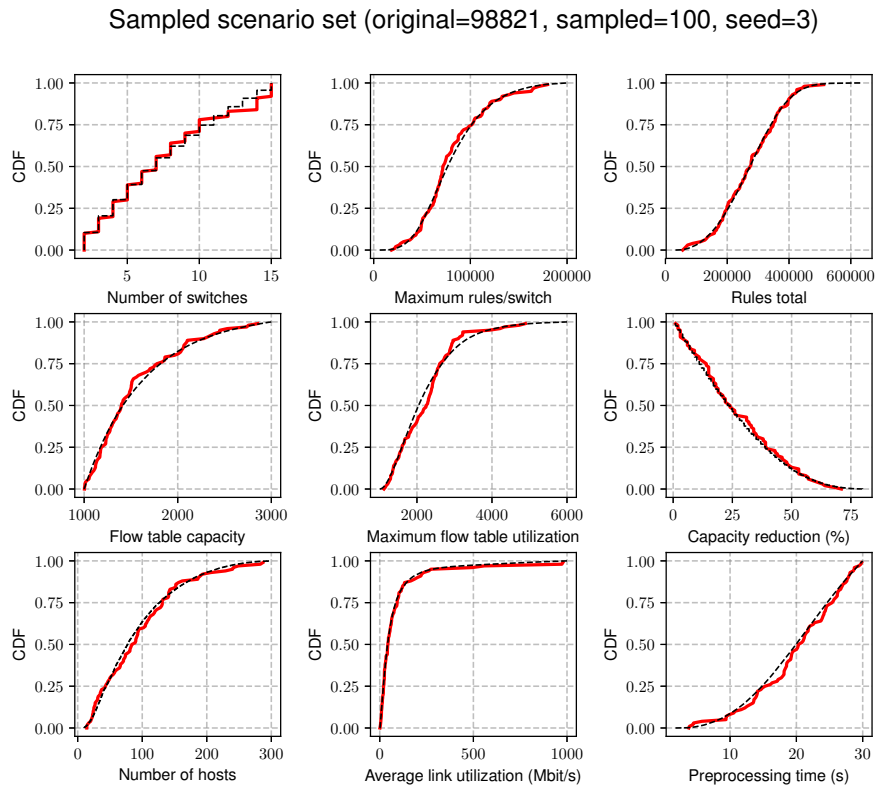
**Table 12.5:** *Important entry points of the evaluation environment*

The workflow for running an experiment series (using the source files from the table above) is discussed in the next section.

### 12.3.3.2 Execution of an Experiment Series

An experiment series consists of multiple experiments with different parameters. Such a series is executed as follows (the \*.py entry points are explained in Table 12.5): In a





**Figure 12.11:** *Nine characteristics of the scenarios in  $Z_{100}$  (red line) compared to the same characteristics of the original set with 98.821 scenarios (dashed black line)*

first step, the parameters for the experiment series are written into a parameter database called `series.db`. The series is started with a call to `test.py` using a reference to the folder that contains the `series.db` parameter database. This tool uses `asynio` to run all experiments specified in `series.db` (in parallel). The tool can be aborted at any time. It will delete failed result sets at the next start and continue with the missing experiments. This will also consider updated parameters in the parameter database. Each experiment is executed in a new process – instance of `main.py` – to avoid memory leaks. Within `main.py`, the following steps are executed per experiment:

- (1) `main.py` will first create a global experimentation environment called “`ctx`” where all important data structures are stored (parameters, scenario, statistics, ...).
- (2) Parameters are extracted from `series.db` and stored in `ctx`. If a scenario ID is used, it serves as a seed to generate the corresponding random parameters. This is done in `experiment.py`.

- (3) The scenario generator in `scenario.py` is called and creates the scenario based on the parameters. This step will also perform a first analysis of the scenario to determine various metrics such as maximum flow table utilization or maximum link utilization (without flow delegation).
- (4) The created scenario is handed over to `rsa.py` where the flow delegation algorithm is executed. The process is implemented iteratively. `rsa.py` calls `dts.py` to get access to the results of the delegation template selection algorithm (if required). The translation to lambda notation is also executed iteratively in each optimization period.
- (5) After the algorithm is finished, the situation with flow delegation is calculated. This takes the optimal solutions from DT-Select and RS-Alloc and applies them to the scenario. Statistics for the scenario with flow delegation applied are then determined based on these calculations.
- (6) The optimal solutions from DT-Select and RS-Alloc and the original scenario without flow delegation can be validated in an event discrete simulator. The statistics obtained from the simulator are compared to the statistics obtained analytically in the last step.
- (7) The statistics are written into a result file that is stored on disk together with additional files for debugging (such as the output from the terminal).

Step 6 was introduced to ensure that the analytical results are calculated correctly – which is not trivial due to the complexity of the involved code. To achieve this, the scenario is executed in a discrete event simulator created for this thesis that can simulate basic flow rule life cycles<sup>11</sup>. The implementation of the simulator is rather simple and should be easy to understand by studying the code. What is important here: the validation step is performed completely independent from the analytical part in `rsa.py` and `dts.py`. The simulator takes the flow rule installation and removal events calculated by the scenario generation process and creates the appropriate events in the simulator. It also takes the selected and allocated delegation templates (the output of DT-Select and RS-Alloc) for each time slot and executes these decisions inside the simulator. A periodic process will then count the installed flow rules in each switch once per second. These values are then compared to the flow table utilization calculated by `rsa.py`.

---

<sup>11</sup>The simulator is implemented in <https://github.com/kit-tm/fdeval/tree/master/core>, last accessed 2020-07-20

## 12.3.3.3 Parameterization

An experiment series needs to be parameterized. The most important parameters in this context are shown in Table 12.6. The three parameters in the top are the key parameters.  $n_{\text{scenario\_id}}$  defines the scenario IDs that should be used in the experiment series. The second-level parameters for scenario generation are then determined automatically based on the specified scenario IDs. This concept is used for the majority of the experiments. Only the experiments for the case study and the scalability experiments need manually defined scenarios which is explained in the respective sections.  $n_{\text{reduction}}$  defines the capacity reduction factor to control the severity of the bottlenecks. Note that, for  $Z_{5000}$ , each scenario is already associated with a randomly determined capacity reduction factor so that all experiments use the same setup. And  $n_{\text{dts\_algo}}$  is used to specify the DT-Select algorithm.

Parameter	Description	Possible values
$n_{\text{scenario\_id}}$	Used scenario IDs	The scenario IDs in $Z_{5000}$ or $Z_{100}$
$n_{\text{reduction}}$	Capacity reduction factor	Values between 1% and 80%
$n_{\text{dts\_algo}}$	DT-Select algorithm	Select-Opt, Select-CopyFirst, Select-Greedy
Parameter	Description	Used default values (see Appendix A)
$L_{\text{dts}}$	Look-ahead factor DT-Select	3
$L_{\text{rsa}}$	Look-ahead factor RS-Alloc	3
$n_{\text{assignments}}$	Assignments for RS-Alloc	50
$n_{\text{dts\_weights}}$	DT-Select weights	$\omega_{\text{DTS}}^{\text{Table}} = 6$ $\omega_{\text{DTS}}^{\text{Link}} = 2$ $\omega_{\text{DTS}}^{\text{Ctrl}} = 1$
$n_{\text{rsa\_weights}}$	RS-Alloc weights	$\omega_{\text{RSA}}^{\text{Table}} = 1$ $\omega_{\text{RSA}}^{\text{Link}} = 0$ $\omega_{\text{RSA}}^{\text{Ctrl}} = 5$

**Table 12.6:** *Parameterization options*

The five parameters in the bottom of Table 12.6 represent the configuration of the algorithms. The developed algorithms in this thesis need to be parameterized with a look-ahead factor (represents the amount of future time slots to consider), with the number of allocation assignments in case of RS-Alloc and with weights to control the different objectives. All experiments conducted in this thesis use the default parameterization that is shown here. It is explained in detail in the parameter study in Appendix A how these default parameters were determined.

The default parameterization works well with all scenarios investigated here. However, better results may be achieved for individual scenarios if other parameterizations are used. However, this and advanced mechanisms such as automatic parameter tuning with

the help of machine learning (for example) are considered future work and not discussed here further.

There are various other implementation-specific parameters such as timeout values, debugging options etc. that are not listed in Table 12.6. A complete list of all parameters supported by the evaluation environment can be found in <https://github.com/kit-tm/fdeval/blob/master/topo/custom/pbce/exp800-2.py>. In the majority of the cases, however, the used default parameters should work just fine.

#### 12.3.4 Post-Processing

The final post-processing step uses a set of tools specified in <https://github.com/kit-tm/fdeval/tree/master/plotter> to access the result files stored on disk. This step has to be done individually for each experiment series and is not explained here. Instead, the details are explained in the respective setup sections of the individual experiments. Implementation specific details can be studied directly in the code.

# Case Study

---

This chapter shows how the flow delegation approach works from a functional perspective. It consists of a case study with four selected scenarios from  $Z_{100}$  that represent conceptually different situations. The chapter is structured as follows. Sec. 13.1 explains how the candidates for the case study are selected. Sec. 13.2 presents the four selected candidate scenarios. Sec. 13.3 introduces the experiment setup and Sec. 13.4 discusses the results.

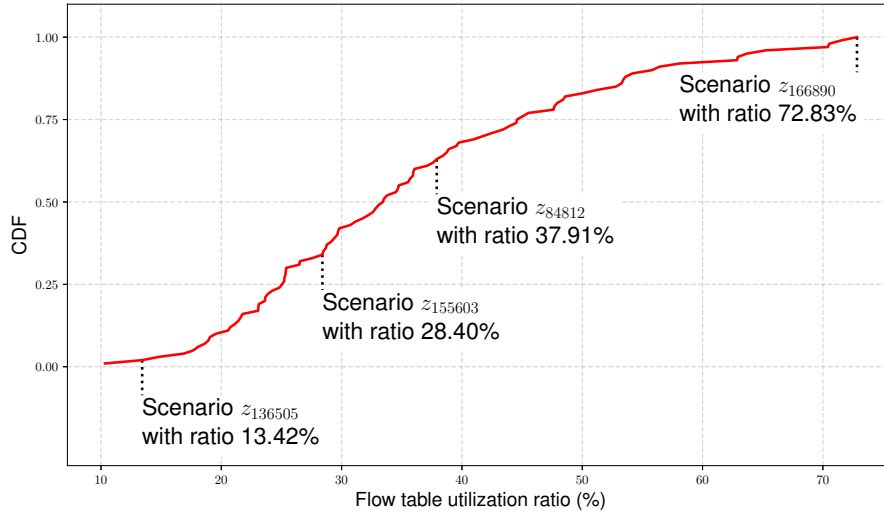
## 13.1 Candidate Selection

The candidates for the case study are selected based on the flow table utilization ratio from Def. 12.6. Fig. 13.1 shows a cumulative distribution function of the ratios for the 100 scenarios in  $Z_{100}$ . Small ratios represent scenarios with a large amount of free capacity. High ratios represent scenarios with little free capacity. In  $Z_{100}$ , the ratios span from approx. 10% to 70% and 50% of the scenarios have a ratio below 34%.

The remainder of this section will now investigate four scenarios with different flow table utilization ratios. This includes two extreme cases – scenario  $z_{136505}$  with a ratio of 13.42% and scenario  $z_{166890}$  with a ratio of 72.83% – and two cases in between. The index numbers represent the scenario id used in the implementation and data sets.

## 13.2 Description of Investigated Scenarios

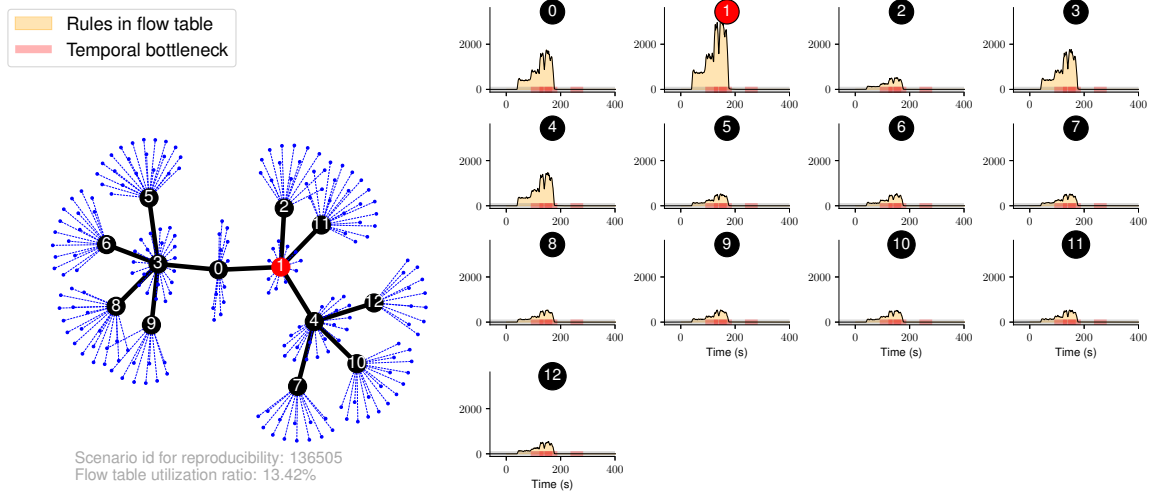
The four scenarios highlighted in Fig. 13.1 are associated with the following randomized scenario generation parameters:



**Figure 13.1:** Candidate selection for the case study

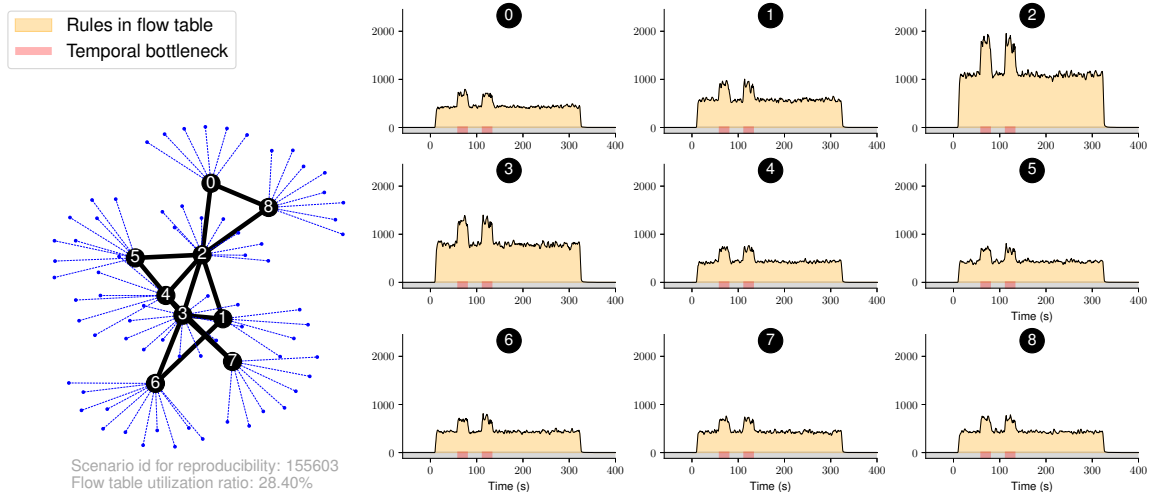
Parameter	Description	$z_{136505}$ (13.42%)	$z_{155603}$ (28.40%)	$z_{84812}$ (37.91%)	$z_{166890}$ (72.83%)
$ H $	Number of hosts	203	70	156	56
$ S $	Number of switches	13	9	12	5
$m$	Topology generation	1	2	1	2
$n_{seed}$	Seed	136505	155603	84812	166890
$n_{pairs}$	Number of host pairs	83758	222892	209242	231821
$n_{iat\_scale}$	Global scale for $T_{iat}$	300	350	300	350
$n_{bneck}$	Temporal bottlenecks	4	2	4	0
$n_{bneck\_duration}$	Bottleneck duration	47	23	14	0
$n_{bneck\_intensity}$	Bottleneck intensity	190	160	110	0
$n_{isr}$	Inter switch ratio	50	70	40	50
$n_{hs}$	Number of hotspots	1	0	0	0
$n_{hs\_intensity}$	Hotspot intensity	4	0	0	0
$n_{traffic\_scale}$	Global traffic scale	500	500	75	75
$n_{lifetime}$	Minimum rule lifetime	3	2	3	3

**Table 13.1:** Scenario generation parameters for the example scenarios in Fig. 13.1

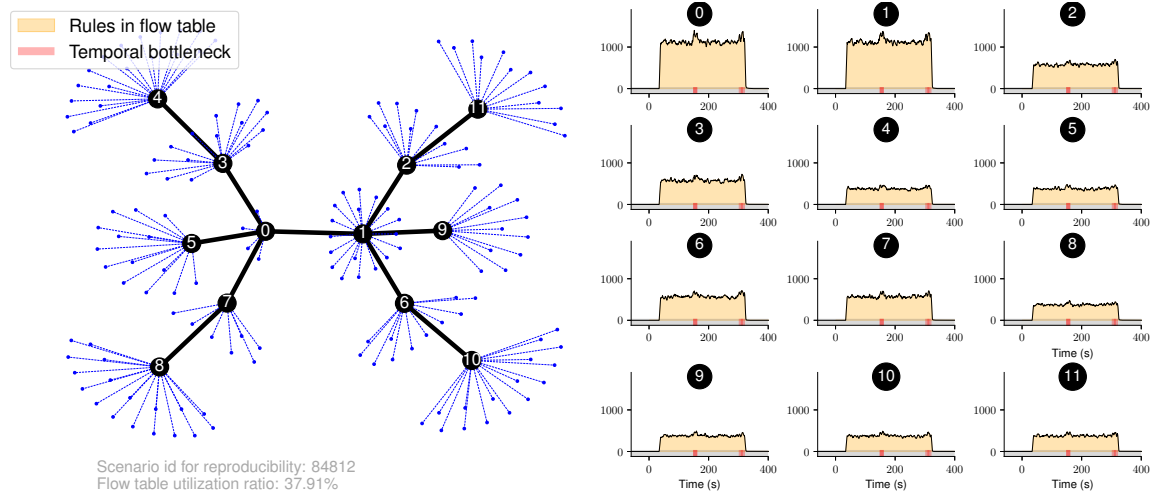


**Figure 13.2:** Scenario  $z_{136505}$  with a flow table utilization ratio of 13.42%

The scenarios in the table are listed from left to right with increasing flow table utilization ratio. The ratio is shown in brackets in the table header. In general, it can be seen that the scenarios with smaller ratios have higher bottleneck parameters. Scenario  $z_{136505}$  has a hotspot and four temporal bottlenecks. In addition, bottleneck duration is close to the maximum (47 out of 50) and bottleneck intensity (190) is high. Scenario  $z_{155603}$  (duration 23, intensity 160) and  $z_{84812}$  (duration 14, intensity 110) are also created with temporal bottlenecks. From these four examples, only scenario  $z_{166890}$  is created without the use of bottleneck mechanisms which results in a high flow table utilization ratio.



**Figure 13.3:** Scenario  $z_{155603}$  with a flow table utilization ratio of 28.40%



**Figure 13.4:** Scenario  $z_{84812}$  with a flow table utilization ratio of 37.91%

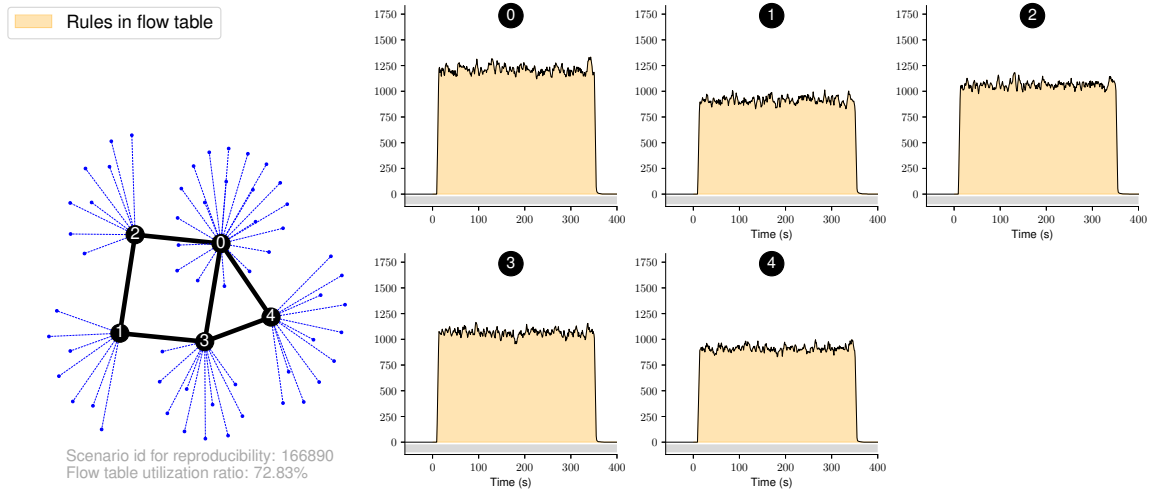
This makes sense because more – and more intensive – temporal bottlenecks and hotspots result in a scenario with higher maximum flow table capacity which is used as the denominator in Eq. 12.3. So the ratio does not only correlate with the available free capacity, it also correlates with number and intensity of the bottlenecks.

**Scenario**  $z_{136505}$  is visualized in Fig. 13.2. The visualization is used in the same way as in Sec. 12.3.1.3. The scenario consists of 13 switches with a hotspot in switch 1 (red circle). This switch and the three other switches with high node degree (0,3,4) require a high amount of flow rules compared to the switches at the edge with smaller node degree (2, 11, ...). It is expected that scenarios with such rapidly changing flow table utilization patterns can benefit from flow delegation.

**Scenario**  $z_{155603}$  is visualized in Fig. 13.3. The scenario consists of 9 switches. The overall situation is similar to scenario  $z_{136505}$ , except that the switches have a higher flow table utilization in general. On the one hand, there is less free capacity available. At the same time, however, the node degree of the switches with higher utilization (primarily 2) is higher compared to the last scenario (more options for remote switches).

**Scenario**  $z_{84812}$  is visualized in Fig. 13.4. It consists of 12 switches, two of which form the core of the network (0, 1). Because communication between the left and the right part of the network has to involve both core switches, the amount of flow rules required in these switches is significantly higher compared to rest of the network. The two small temporal bottlenecks are barely significant in this example.





**Figure 13.5:** Scenario  $z_{166890}$  with a flow table utilization ratio of 72.83%

**Scenario**  $z_{166890}$  is visualized in Fig. 13.5. In this scenario, the 5 involved switches are all highly utilized. There is little free flow table capacity that could be used by flow delegation.

### 13.3 Experiment Setup

To investigate how flow delegation performs with respect to the four above scenarios, a series of experiments is conducted, individually for each scenario. The scenario is kept fix and then used with capacity reduction factors between 1% and 80%. A capacity reduction factor of 5%, for example, will set the flow table capacity used in the experiment to 95% of the maximum flow table utilization. A factor of 10% will set the capacity to 90% of the maximum flow table utilization. All experiments are performed with Select-CopyFirst as DT-Select algorithm and the default parameterization from Appendix A.4. The most important parameters for this experiment series are listed in Table 13.3.

Parameter	Description	Used Values
$n_{\text{scenario\_id}}$	Used scenario IDs	136505, 155603, 84812, 166890
$n_{\text{reduction}}$	Capacity reduction factor	1%, 2%, ..., 79%, 80%
$n_{\text{dts\_algo}}$	DT-Select algorithm	Select-CopyFirst
$n_{\text{dts\_lookahead}}$	DT-Select look-ahead	3
$n_{\text{rsa\_lookahead}}$	RS-Alloc look-ahead	3
$n_{\text{rsa\_assignments}}$	RS-Alloc assignments	50

$n_{\text{dts\_weights}}$	DT-Select weights	$\omega_{\text{DTS}}^{\text{Table}} = 6$	$\omega_{\text{DTS}}^{\text{Link}} = 2$	$\omega_{\text{DTS}}^{\text{Ctrl}} = 1$
$n_{\text{rsa\_weights}}$	RS-Alloc weights	$\omega_{\text{RSA}}^{\text{Table}} = 1$	$\omega_{\text{RSA}}^{\text{Link}} = 0$	$\omega_{\text{RSA}}^{\text{Ctrl}} = 5$

**Table 13.2:** Important experiment parameters

There are 80 result sets per scenario and 320 result sets in total in the final data set. The parameters in the bottom are the same for all 320 experiments. All other parameters are set to their default values.

## 13.4 Results

The results of the case study are discussed in two steps. Sec. 13.4.1 first presents an exemplary analysis of scenario  $z_{136505}$ . This introduces the correlation between capacity reduction factor and failure rate which is essential for later discussions. It also provides a visual explanation of the flow delegation approach and its limitations. Sec. 13.4.2 will then discuss the implications of the flow table utilization ratio.

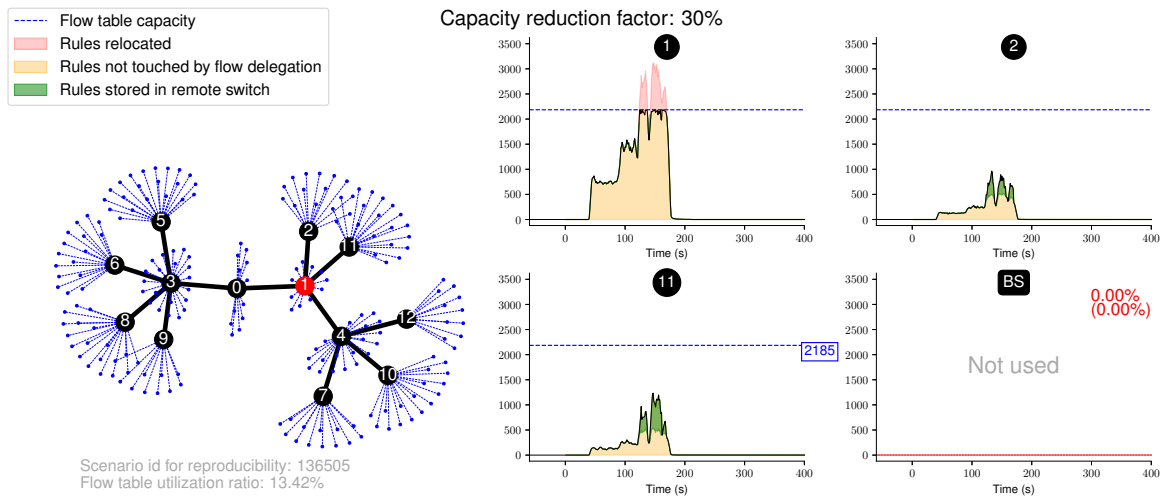
### 13.4.1 Capacity Reduction and Failure Rate

**Findings:** This section illustrates how flow delegation works for example scenario  $z_{136505}$ . It is shown that the approach can achieve a 0% failure rate with a capacity reduction factor of 40% (capacity = 1873, maximum utilization = 3122).

We are interested in the maximum capacity reduction factor that we can apply to the example scenarios so that no flow rules are relocated to the backup switch, i.e., with a failure rate of 0%. The following discusses this process on the example of scenario  $z_{136505}$ . More precisely, the following four experiments are discussed below:

	$n_{\text{scenario\_id}}$	$n_{\text{reduction}}$	Utilization	Capacity	Failure rate
Fig. 13.6	136505	30%	3122	2185	0.00%
Fig. 13.7	136505	40%	3122	1873	0.00%
Fig. 13.8	136505	50%	3122	1561	0.21%
Fig. 13.9	136505	60%	3122	1248	2.73%

**Table 13.3:** Discussed experiments in Sec. 13.4.1



**Figure 13.6:** Scenario  $z_{136505}$  with capacity reduction of 30%

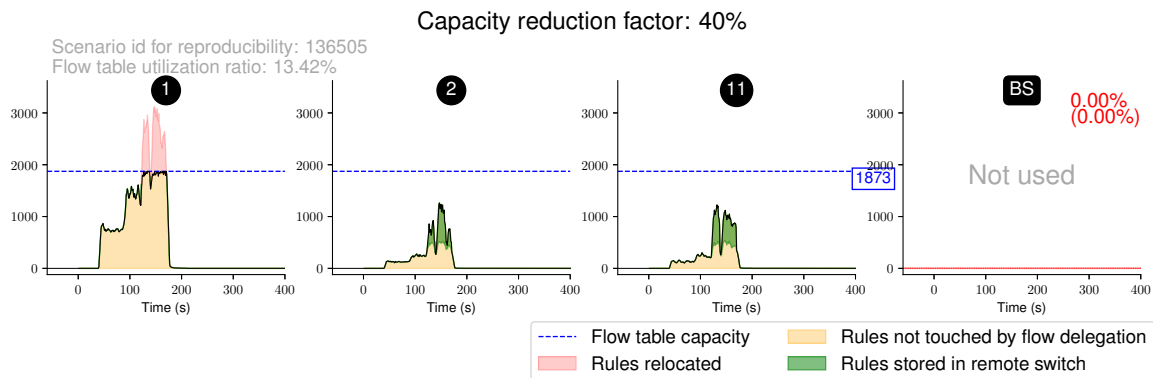
Fig. 13.6 shows the first experiment for a **capacity reduction factor of 30%**. The scenario has a maximum flow table utilization of 3122 rules (see peak in switch 1), i.e., a switch with at least 3122 rules is required to not run into a flow table capacity bottleneck. In other words: without flow delegation or a similar approach, the network operator has only two options. She either invests in infrastructure with sufficient resources (over-provisioning, expensive) or has to deal with the consequences of the bottleneck which might not be acceptable. Flow delegation reduces the required amount of investments. Let us assume the operator buys switches with a flow table capacity of 2185 flow rules, as it is shown in Fig. 13.6. This is a 30% reduction compared to a switch with a capacity of 3122 flow rules – and potentially significantly cheaper in terms of capital and operational expenditure.

The figure shows four switches on the right. The fact that the other switches are not shown here means flow delegation will only interact with these three switches (1, 2, and 11) in this situation. The fourth switch labeled BS is the backup switch. The area colored in red in switch 1 shows flow rules that are relocated to mitigate the bottleneck in switch 1. In this case, switch 2 and switch 11 are used as remote switches. The green area in switches 2 and 11 represents the relocated remote rules. The area colored in yellow shows flow rules that are not touched by the flow delegation approach, including additional rules such as aggregation and backflow rules. There are three important observations to make in Fig. 13.6.

- (1) None of the three switches suffers from a capacity bottleneck if flow delegation is enabled.

- (2) The backup switch is not used, i.e., all flow rules are handled in hardware. The failure rate is 0.00%, shown here in red in the top right of the backup switch.
- (3) There is still plenty of free capacity available in the two remote switches. As a result, the capacity reduction factor could potentially be increased to values above 30% without failures.

The two first observations demonstrate the feasibility of the flow delegation approach for this example. The third observation motivates the concept to incrementally increase the capacity reduction factor. Fig. 13.7 shows the same scenario with a **capacity reduction factor of 40%** instead of 30%.

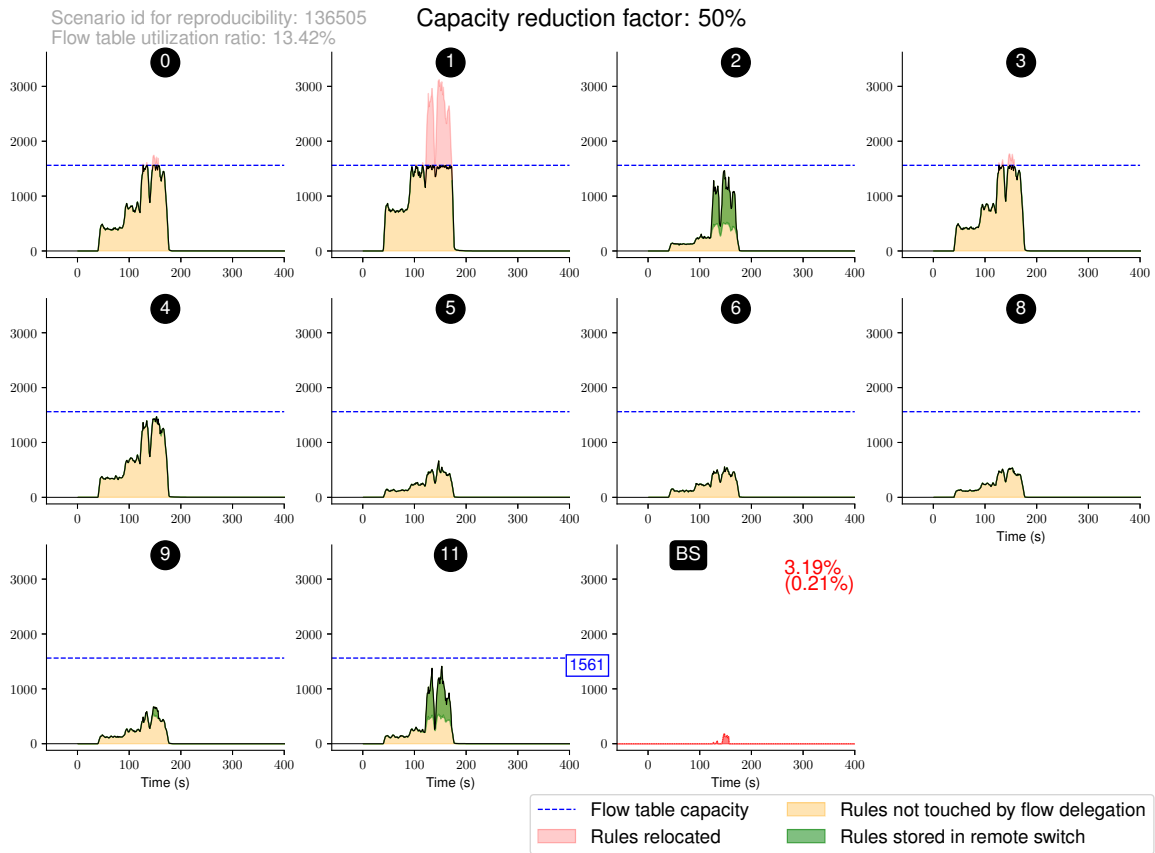


**Figure 13.7:** Scenario  $z_{136505}$  with capacity reduction of 40%

The topology is the same as above (not shown again). With 40% reduction, the flow table capacity for this experiment is 1873 rules. It can be seen that the same two remote switches are chosen. And while the failure rate is still at 0%, the flow table utilization of remote switches 2 and 11 is higher than in Fig. 13.6 because of the increased number of rules relocated from switch 1.

Fig. 13.8 shows the scenario with a **capacity reduction factor of 50%** (capacity of 1561 rules). 10 of the 13 switches interact with the flow delegation approach. There are again several important observations.

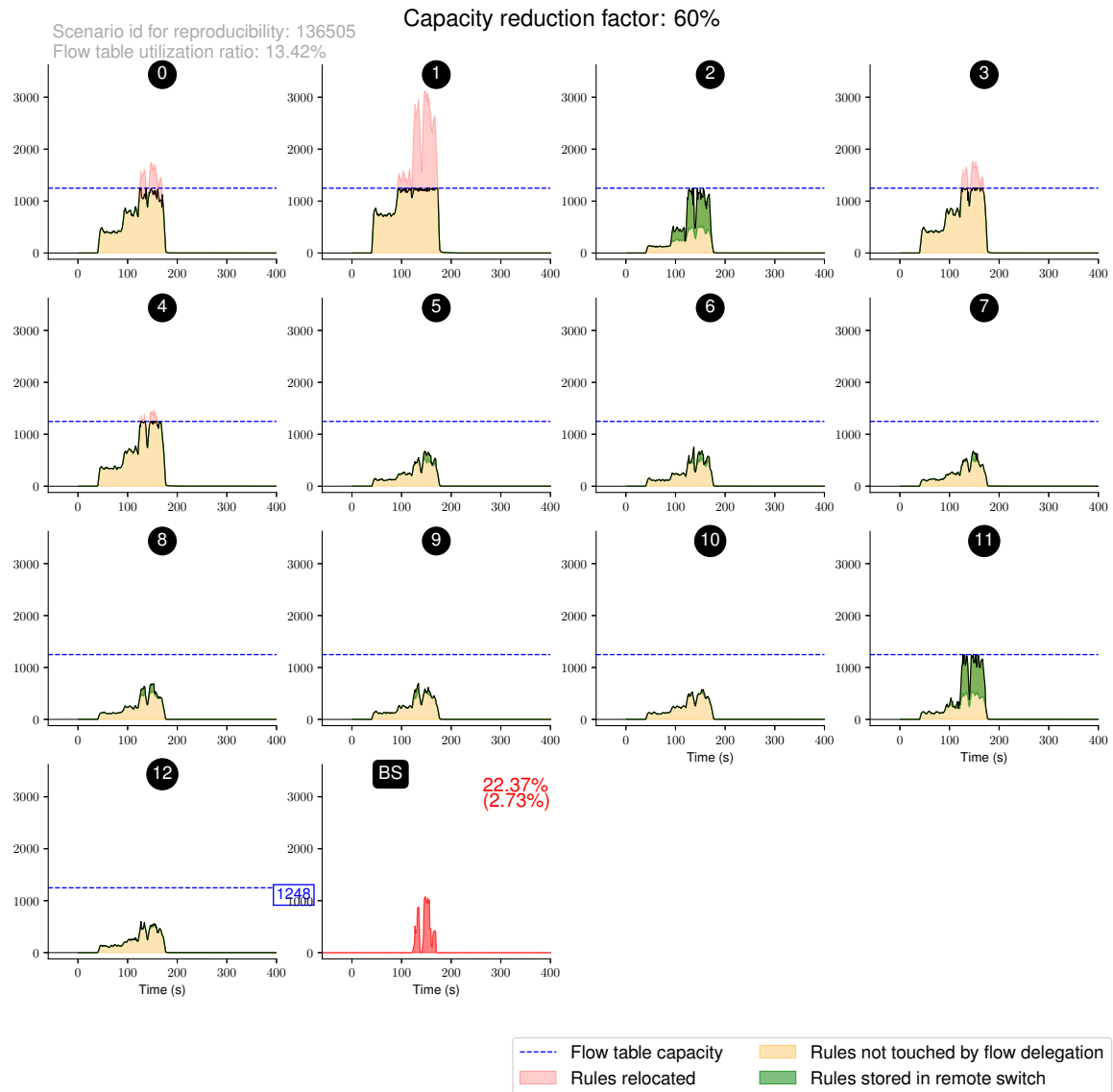
- (1) Unlike before, there are now multiple bottlenecked switches (0, 1, and 3).
- (2) The previously used remote switches (2, 11) are highly utilized during the bottleneck phase, i.e., there is not much free flow table capacity left. In fact, all neighboring switches of switch 1 are highly utilized during the bottleneck phase.



**Figure 13.8:** Scenario  $z_{136505}$  with capacity reduction of 50%

- (3) Several other switches such as 5, 6, and 8 still provide spare resources. They can, however, only be used for the switch 3 bottleneck due to the physical connections in the topology (no link to switch 1).
- (4) Not all flow rules can be successfully relocated to a remote switch any more. The failure rate in this experiment is 0.21%, i.e., 0.21% of all flow rules – or 3.19% of the to be relocated flow rules in the red area – are relocated to the backup switch.

Because the failure rate is not 0%, the above experiment with 50% reduction is considered a “failure”. It means flow delegation cannot relocate 100% of the bottlenecked flow rules to another hardware switch because the accessible spare resources are insufficient in this case. This is a fundamental limitation of the approach. It is important to mention, however, that 50% capacity reduction is already a significant factor – in terms of potential savings but also in terms of overhead (discussed later). If the capacity reduction factor is increased further, the failure rate will increase as well. Fig. 13.9 illustrates this with a capacity reduction factor of 60% and a failure rate of 2.73%.



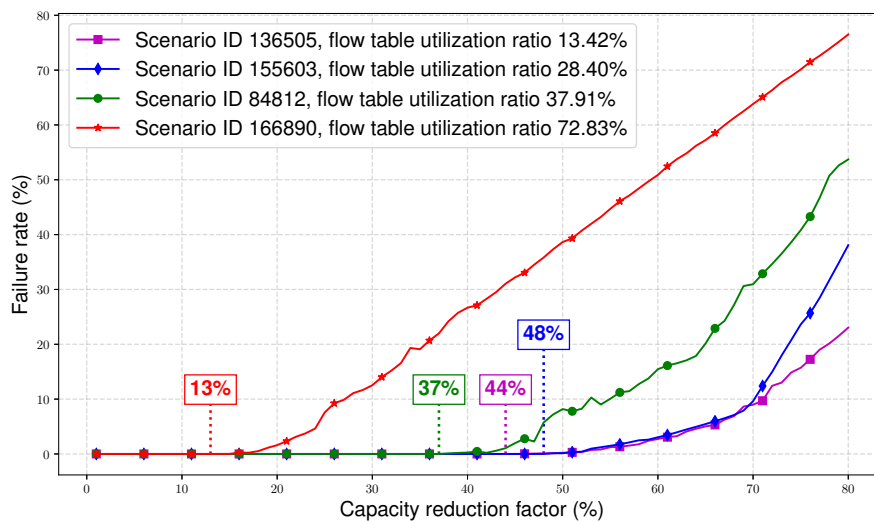
**Figure 13.9:** Scenario  $z_{136505}$  with capacity reduction of 60%

### 13.4.2 Flow Table Utilization Ratio

**Findings:** Scenarios with lower flow table utilization ratio (more free capacity) can achieve higher capacity reduction factors with 0% failure rate. The example with lowest ratio ( $z_{136505}$ ) achieved a reduction factor of 44%. The example with highest ratio ( $z_{166890}$ ) achieved only a reduction factor of 13%.

The previous section showed that scenario  $z_{136505}$  can achieve a capacity reduction factor of at least 40% with a 0% failure rate. This section now investigates how the four selected example scenarios perform with respect to capacity reduction and how this correlates with the flow table utilization ratio.

Fig. 13.10 plots the failure rate for different capacity reduction factors. The x-axis represents the different experiments with a capacity reduction factor between 1% and 80%. The y-axis denotes the failure rate recorded for each experiment. And the four lines represent the four example scenarios.

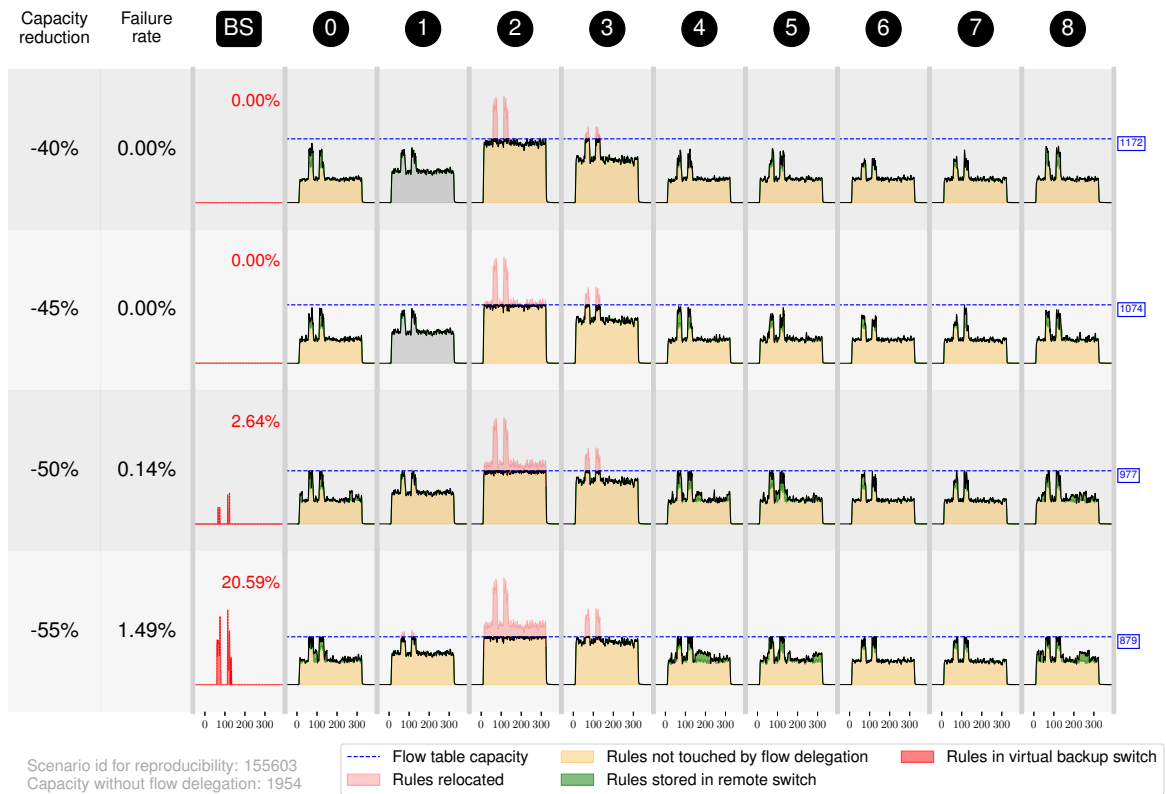
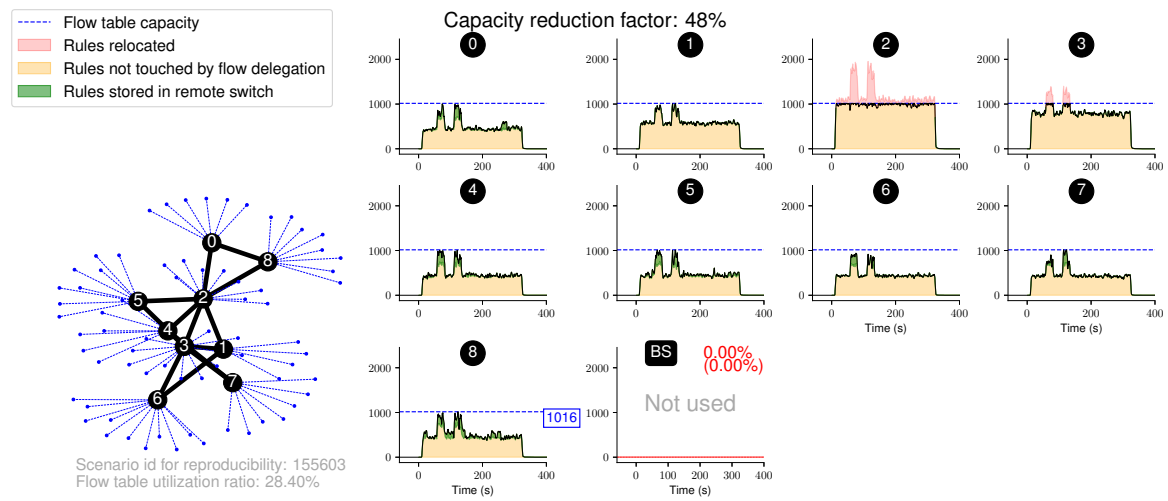


**Figure 13.10:** Failure rate for the example scenarios based on capacity reduction

The colored rectangles in the plot then indicate the highest capacity reduction factor that was achieved with 0% failure rate for the respective scenario. In case of  $z_{136505}$  (magenta, utilization ratio of 13.42%), the highest capacity reduction factor without failures is 44%. This is in line with the results from the previous section – at least 40% according to Fig 13.7 and less than 50% according to Fig. 13.8.

Note that the courses of the curves are of limited value here because even a failure rate of 1% could already be unacceptable in practice (depends on the specific design and tasks of the network). The important point is the value specified in the rectangle while the difference between the curves has little meaning. They are only shown here to underline the strong correlation between failure rate and flow table utilization ratio: it is easy to see that higher ratios will result in higher failure rates. The following will now look at the three remaining scenarios in more detail.

**Observations with respect to  $z_{155603}$**  (blue curve in Fig. 13.10, utilization ratio of 28.40%): In direct comparison with  $z_{136505}$ , scenario  $z_{155603}$  can achieve an even higher



**Figure 13.11:** Highest capacity reduction factor without failures for scenario  $z_{155603}$  (top) and examples for different capacity reduction factors (bottom)



capacity reduction factor of 48%, despite its higher utilization ratio. This is because the utilization ratio is not the only factor that is important here. Consider the situation depicted in Fig. 13.11. The top of the figure shows scenario  $z_{155603}$  with a capacity reduction factor of 48%, i.e., the maximum with 0% failure rate for this case. The part in the bottom shows the same scenario with four selected capacity reduction factors between 40% and 55%. The switches in grey are not used.

It is easy to see that the switches in  $z_{155603}$  have a much higher flow table utilization than the switches in the scenario from the last section (Fig. 13.7). This is expected as the utilization ratio is higher. However, the node degree of the bottlenecked switch in  $z_{155603}$  (switch 2, degree 6) is also higher than the node degree of the bottlenecked switch in  $z_{136505}$  (switch 1, degree 4). As a result, the latter scenario has less remote switch options available. And it can in fact be seen in Fig. 13.8, for example, that only two of the four available remote switches can be used because the other two are fully utilized. Scenario  $z_{155603}$ , on the other hand, can make use of all six available remote switches which results in a higher capacity reduction factor.

**Observations with respect to  $z_{84812}$**  (green curve in Fig. 13.10, utilization ratio of 37.91%): A detailed view of this scenario with flow delegation is shown in Fig. 13.12. It represents a long-lasting bottleneck that is caused by two overloaded switches in the core of the topology. This scenario is mainly included here because it suffers from a (rarely occurring) limitation of the two-step optimization approach.

Take switch 1, for example. This switch has four remote switch options. Switch 0 cannot be used because it is bottlenecked as well while the remaining three switches (2, 9, 6) have free capacity and could be used. However, the majority of the flow rules are relocated to switch 9 and switches 2 and 6 are barely used. There are two primary reasons for such a behavior. One is link utilization. If the link between switch 1 and switch 2, for example, is fully utilized, the flow delegation approach cannot relocate rules to switch 2. This is not the case here (not visible in the plot). What happens here is a significant amount of the flow rules in switch 1 is associated with the delegation templates representing the ingress ports that are connected to switch 0 – and also partially switch 2 and switch 6. Note that especially the delegation template for the ingress port towards switch 0 can be very large because this template has to cover all flow rules that communicate from the left side of the network to the right.

This is unproblematic if the flow delegation problem is solved globally because the free capacities of the remote switches are automatically included in the delegation template selection process – e.g, by not selecting a larger template in exchange for several smaller ones. This thesis, however, uses a two-step approach and delegation template selection does currently not know about the free capacity in the remote switches. It can therefore happen that the delegation templates selected by DT-Select are simply “too big” for

allocation. This can be easily solved in future work by additional constraints with respect to remote capacities while still using the two-step approach.

**Observations with respect to  $z_{166890}$**  (red curve in Fig. 13.10, utilization ratio of 72.83%): A detailed view of this scenario is given in Fig. 13.13. The highest capacity reduction factor achieved with 0% failure rate is 13%, shown in the top of the figure. The scenario is clearly limited by the available spare capacity in the network and the small number of remote switch options – a worst case scenario for the flow delegation approach. However, even in this scenario it is possible to achieve a capacity reduction factor larger than 10%.

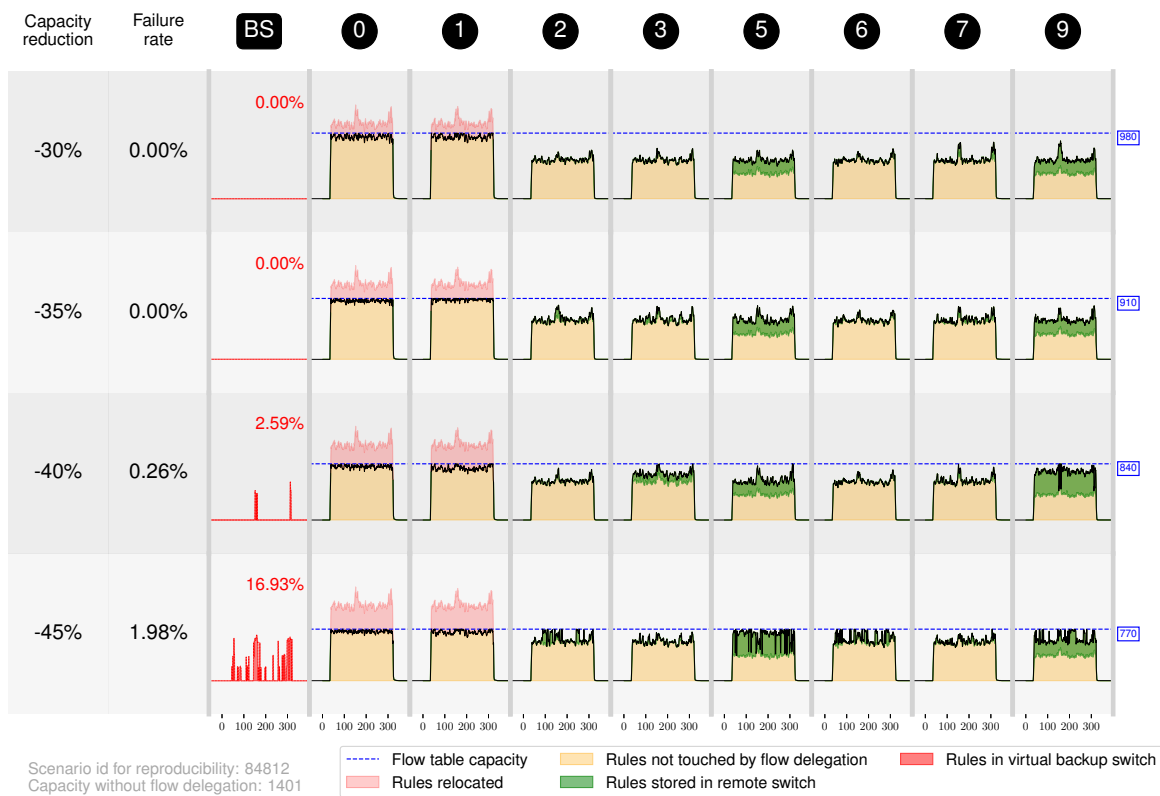
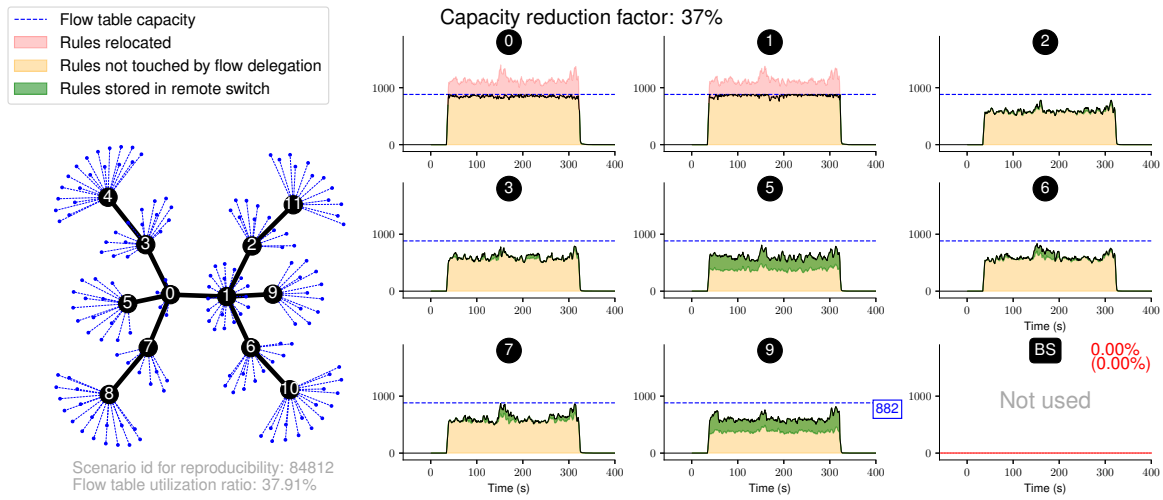


Figure 13.12: Highest capacity reduction factor without failures for scenario  $z_{84812}$  (top) and examples for different capacity reduction factors (bottom)

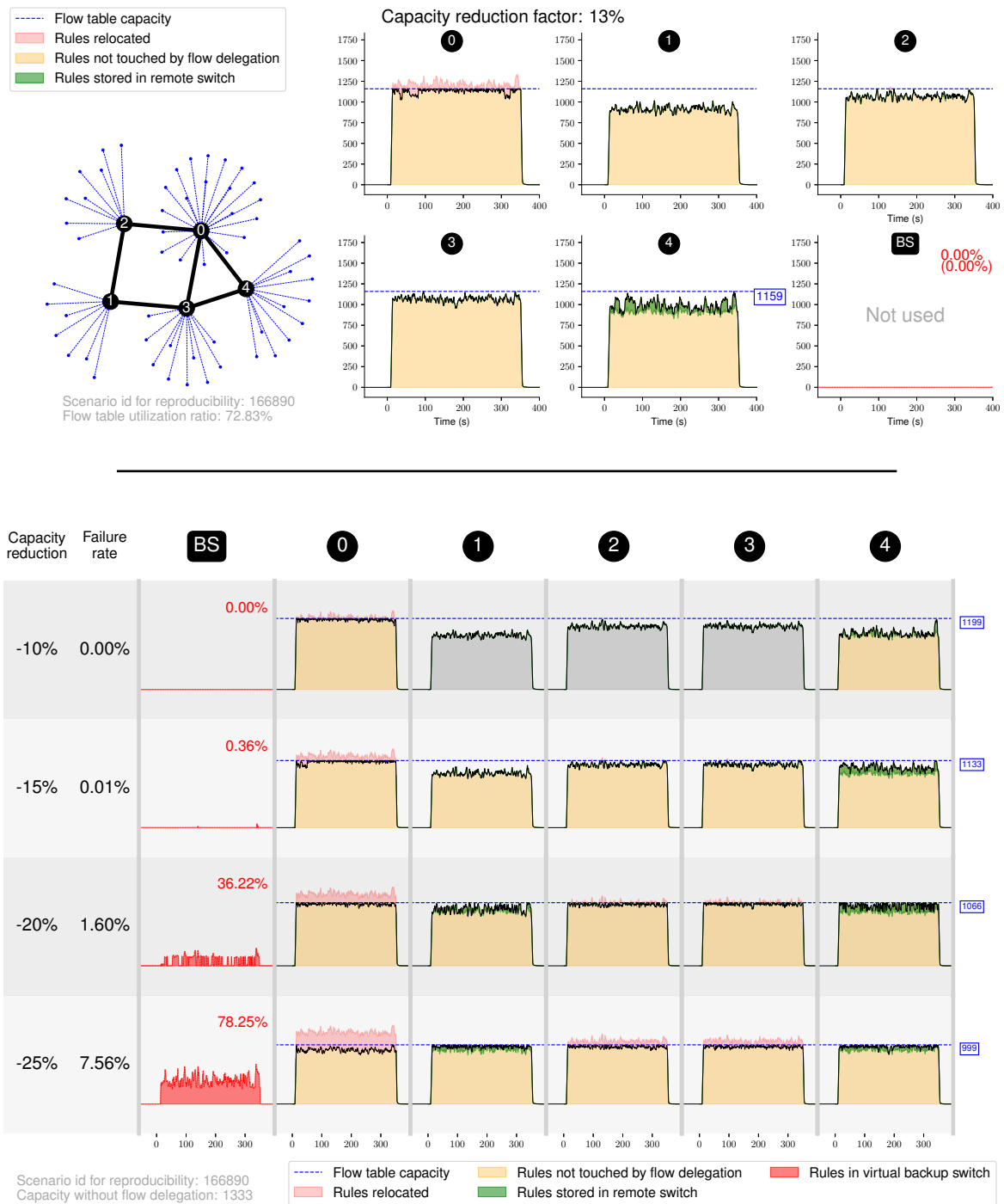


Figure 13.13: Highest capacity reduction factor without failures for scenario  $z_{166890}$  (top) and examples for different capacity reduction factors (bottom)

# Performance

---

Flow delegation performance was defined in Def. 12.5 and indicates “how well” flow delegation can deal with a specific situation. It is specified as the maximum capacity reduction factor (=bottleneck severity) that can be applied to a scenario so that the failure rate does not exceed 0%, 0.1% or 1%. Failure rates higher than 1% are considered impracticable. Another performance indicator also discussed here is over- and underutilization. Overutilization occurs if the flow table utilization exceeds the capacity despite the fact that flow delegation is used. Underutilization occurs when flow rules are relocated to a remote switch despite the fact that the delegation switch has free entries left in its flow table. This chapter investigates the following aspects with respect to flow delegation performance:

- (1) **Performance without Failures:** Sec. 14.1 investigates the relationship between capacity reduction factor (bottleneck severity) and the ability to handle a scenario with 0% failure rate. It is shown that 90% of the experiments achieve a 0% failure rate for capacity reduction factors up to 7% and 50% of the experiments achieve a 0% failure rate for capacity reduction factors up to 28%.
- (2) **Performance with small Failure Rates:** Not all use cases demand for a 0% failure rate. Sec. 14.2 investigates the performance if small failure rates of 0.1% and 1% are considered acceptable. It is shown that this relaxation can improve flow delegation performance by a factor of 3.
- (3) **Performance based on Utilization Ratio:** Sec. 14.3 investigates how the available free capacity impacts the performance. It is shown that flow delegation achieves better performance when applied to scenarios with low utilization ratio which is equivalent to more free flow table capacity.

- (4) **Over- and Underutilization:** Sec. 14.4 investigates over- and underutilization. It is shown that no overutilization occurs with Select-Opt and Select-CopyFirst and underutilization is below 5% in most cases which is considered a good value.

## 14.1 Performance without Failures

Sec. 13.4.1 already showed that flow delegation can achieve a 0% failure rate when the capacity reduction factor is set to values between 13% and 48%. However, these results are based on individual examples. This section will now investigate the important relationship between capacity reduction factor and failure rate for a broader set of scenarios.

### 14.1.1 Experiment Setup

Two experiment series are performed in this context. The first series executes all scenarios in  $Z_{5000}$  with random capacity reduction factors between 1% and 80% .

	Parameter	Description	Used Values
Series 1	$n_{\text{scenario\_id}}$	Used scenario IDs	All scenario IDs in $Z_{5000}$
	$n_{\text{reduction}}$	Capacity reduction factor	Random between 1% and 80%
Series 2	$n_{\text{scenario\_id}}$	Used scenario IDs	All scenario IDs in $Z_{100}$
	$n_{\text{reduction}}$	Capacity reduction factor	1%, 2%, ..., 79%, 80%
Same for both series	$n_{\text{dts\_algo}}$	DT-Select algorithm	Select-CopyFirst
	$n_{\text{dts\_lookahead}}$	DT-Select look-ahead	3
	$n_{\text{rsa\_lookahead}}$	RS-Alloc look-ahead	3
	$n_{\text{rsa\_assignments}}$	RS-Alloc assignments	50
	$n_{\text{dts\_weights}}$	DT-Select weights	$\omega_{\text{DTS}}^{\text{Table}} = 6$ $\omega_{\text{DTS}}^{\text{Link}} = 2$ $\omega_{\text{DTS}}^{\text{Ctrl}} = 1$
	$n_{\text{rsa\_weights}}$	RS-Alloc weights	$\omega_{\text{RSA}}^{\text{Table}} = 1$ $\omega_{\text{RSA}}^{\text{Link}} = 0$ $\omega_{\text{RSA}}^{\text{Ctrl}} = 5$

**Table 14.1:** Experiment parameters for Sec. 14.1

However, the capacity reduction factors are not distributed uniformly. This is because the random factors were already selected in the scenario generation process and there is a (realistic) bias towards smaller factors. This bias comes from the fact that higher capacity reduction factors are correlated with extreme scenarios with higher pre-processing and experiment execution times which is one of the five conditions for excluding a scenario

from the original set (see Sec. 12.3.2.2). 50% of the scenarios have a capacity reduction factor between 1% and 24%. And only 12% of the scenarios have capacity reduction factors above 50%.

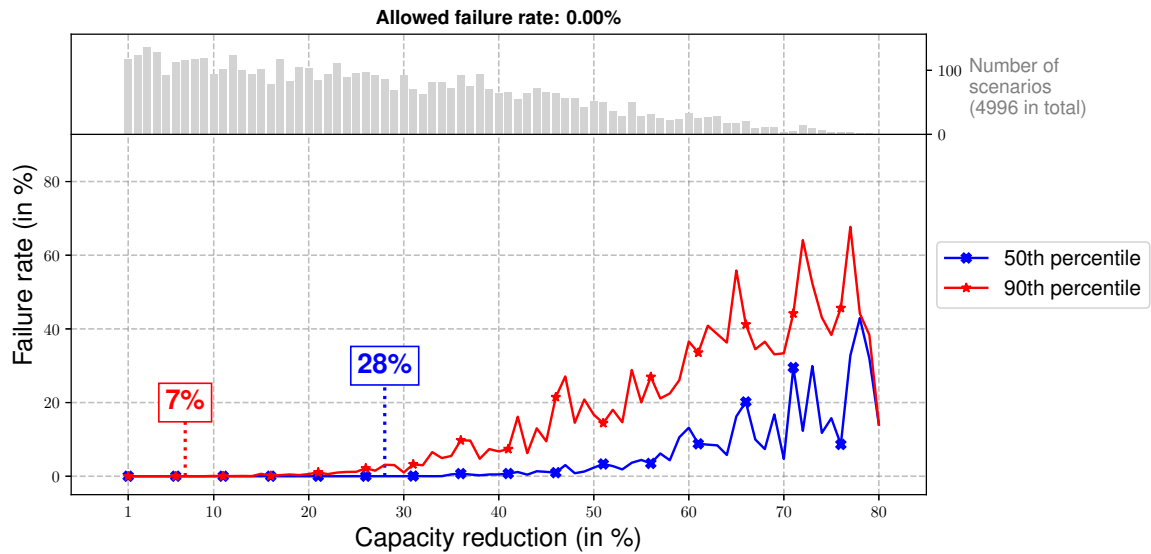
To make sure that this bias does not distort the result, a second experiment series is designed for validation. In this second series based on  $Z_{100}$ , all capacity reduction factors between 1% and 80% are used following a uniform distribution. The smaller scenario set is used here because using  $Z_{5000}$  with all 80 different values consumes too much CPU resources. All experiments for both series are performed with Select-CopyFirst as DT-Select algorithm and the default parameterization from Appendix A.4. The most important parameters are listed in Table 14.4. The data set for the first series contains 4996 result sets, 4 experiments were aborted because of a timeout. The data set for the second series contains 7907 result sets, 93 experiments were aborted because of a timeout.

### 14.1.2 Results

**Findings:** 90% of the experiments achieve a 0% failure rate for all capacity reduction factors up to 7%. 50% of the experiments achieve a 0% failure rate for all capacity reduction factors up to 28%.

To investigate the relationship between capacity reduction factor and failure rate, the data sets are processed as follows. In a first step, the failure rate is calculated for each experiment. Step two puts the experiments into 80 different groups, one group for each used capacity reduction factor. And step three calculates the 50th ( $p_{50}$ ) and the 90th ( $p_{90}$ ) percentile of the failure rates for each groups, i.e., 50% (90%) of the observed failure rates in the group are below  $p_{50}$  ( $p_{90}$ ). Percentiles are used here because  $Z_{5000}$  was generated randomly and includes extreme scenarios where the failure rate is  $> 0\%$  despite the fact that a very low capacity reduction factor is applied (below 5%). Without percentiles, the result would be dominated by these outliers. Extended plots with additional percentiles – including the 100th percentile that represents all 4996 experiments – are shown in Appendix C.

The results for  $Z_{5000}$  are shown in Fig. 14.1. The x-axis represents capacity reduction factors in ascending order, i.e., the 80 groups from above. The y-axis denotes the maximum failure rate of either 50% (blue curve) or 90% (red curve) of the experiments in the group associated with the x-value. The histogram in gray in the top denotes the number of experiments inside each group. Note that the groups for the smaller reduction factors contain more experiments. For very high reduction factors above 70%, most of the groups contain less than 5 entries. The small amount of data points in this area explain



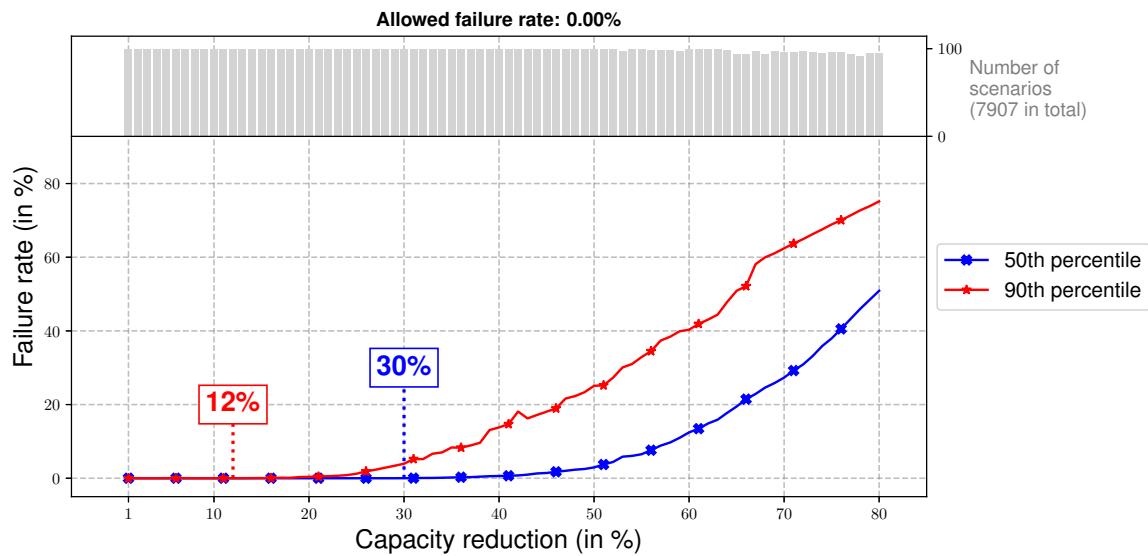
**Figure 14.1:** Capacity reduction for 4996 experiments based on  $Z_{5000}$

the absence of a clear trend in the curves – this is different for the second experiment series discussed below. The values close to 80% on the x-axis are thus not considered representative, they are just included here for completeness.

The actual flow delegation performance for the two percentiles is shown in the colored rectangles in Fig. 14.1. These values denote the highest capacity reduction factor that could be handled with a 0% failure rate. It can be concluded from the blue rectangle that 50% of the experiments achieve a 0% failure rate for all capacity reduction factors up to 28%. It is important to mention that this does *not* mean that 50% of the scenarios achieve a capacity reduction of 28% (this would be a completely different statement). It says that 50% of the experiments with a capacity reduction factor of 28% and lower (!) achieve a failure rate of 0%. Similarly, the red rectangle indicates that 90% of the experiments achieve a 0% failure rate for all capacity reduction factors up to 7%. Note that  $Z_{5000}$  represents thousands of different scenarios and the values might be higher (or smaller) for individual scenarios. This is discussed in more detail in subsequent sections.

These numbers can be interpreted as follows. The 7% value (red) is meant as a worst case baseline. It can be expected that, regardless of the actual situation, flow delegation can deal with a bottleneck that falls into the 7% reduction range while maintaining a 0% failure rate. It is shown later that this range can be extended significantly by allowing failure rates slightly above 0%. The 28% value (blue) is meant as an optimistic expectation that the majority of the bottleneck situations in the 28% range can be handled without





**Figure 14.2:** Capacity reduction for 7907 experiments based on  $Z_{100}$

failures. While 7% and 28% does not sound impressive at first glance, the potential gain in practice is significant. Assume an existing network is equipped with switches that support 1000 flow rules. For this network, 28% reduction is sufficient to mitigate bottleneck situations with 1387 flow rules, i.e., more than 38% above the maximum capacity of the switches. 28% reduction also means the network operator can replace the switches with a smaller (cheaper) model that only supports 720 flow rules, i.e., if the 1000 rules are only required in rare high load scenarios.

Another important observation from Fig. 14.1 is that higher capacity reduction factors in the area at and above 50% lead to unacceptable high failure rates. It can further be seen that the y-values increase “slowly” for x-values between 7% and approx. 20% in case of the red curve. This means that only a small fraction of the flow rules can not be allocated to a remote switch. A similar observation can be made for the blue curve up to an x-value of approx. 40%. This is the main motivation to also investigate cases with failure rates slightly above 0% (next section).

Fig. 14.2 shows the same plot for the second experiment series based on  $Z_{100}$ . It can be seen in the histogram in the top that each capacity reduction factor now has the same amount of experiments (around 100). Only some higher capacity reduction factors lead to complex optimization problems which triggered a timeout to save resources. These experiments are not included here (93 out of 8000). Note that the curves now follow a clear trend compared to the results in Fig. 14.1. This and the overall numbers – 12%

reduction for 90% of the experiments and 30% reduction for 50% of the experiments – validate the results achieved with  $Z_{5000}$ .

## 14.2 Performance with Small Failure Rates

In certain contexts, it can be acceptable that a small fraction of the flow rules is *not* processed by a remote switch but processed by a backup switch in software or handled in some other way, e.g., with a flow rule eviction mechanism (see Sec. 2.4.2.2). In such cases, it is not mandatory to achieve 0% failure rate. This section discusses the impact on flow delegation performance if a slightly increased failure rate of 0.1% or 1% is considered acceptable.

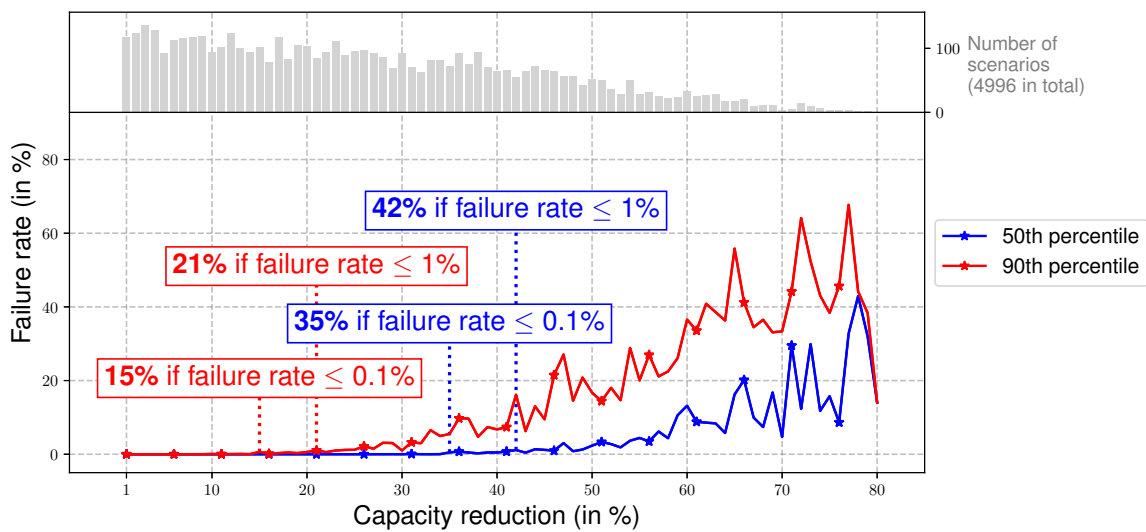
### 14.2.1 Experiment Setup

This investigation also uses the two experiment series described in Sec. 14.1.1. The first series is based on scenario set  $Z_{5000}$  with random capacity reduction factors between 1% and 80%. The second series is based on scenario set  $Z_{100}$  with all capacity reduction factors between 1% and 80%.

### 14.2.2 Results

**Findings:** Allowing a slightly increased failure rate of 0.1% or 1% improves the performance of the flow delegation approach significantly – up to a factor of 3 compared to the results with 0% failure rate.

Fig. 14.3 shows the same data as Fig. 14.1. The plot is designed in the exact same way as explained in Sec. 14.1.2, with one small exception: it is now assumed that a small failure rate is acceptable, i.e., flow delegation is allowed to put some rules into a backup switch. Note that this has no impact on the actual data – it basically only changes the definition of when the flow delegation approach has failed with respect to a certain scenario and capacity reduction factor. In the last section, any experiment with a failure rate above 0% was considered a failure, even if only a single rule was sent to the backup switch. Fig 14.3, on the other hand, shows the flow delegation performance if a failure rate of 0.1% or 1% is considered acceptable. It is important to mention here that the values of 0.1% and 1% refer to the total amount of flow rules installed in all switches over the course of the experiment. With 300.000 rules (average value for scenarios in  $Z_{5000}$ ), this would be 300 rules in case of 0.1% and 3.000 rules in case of 1%.

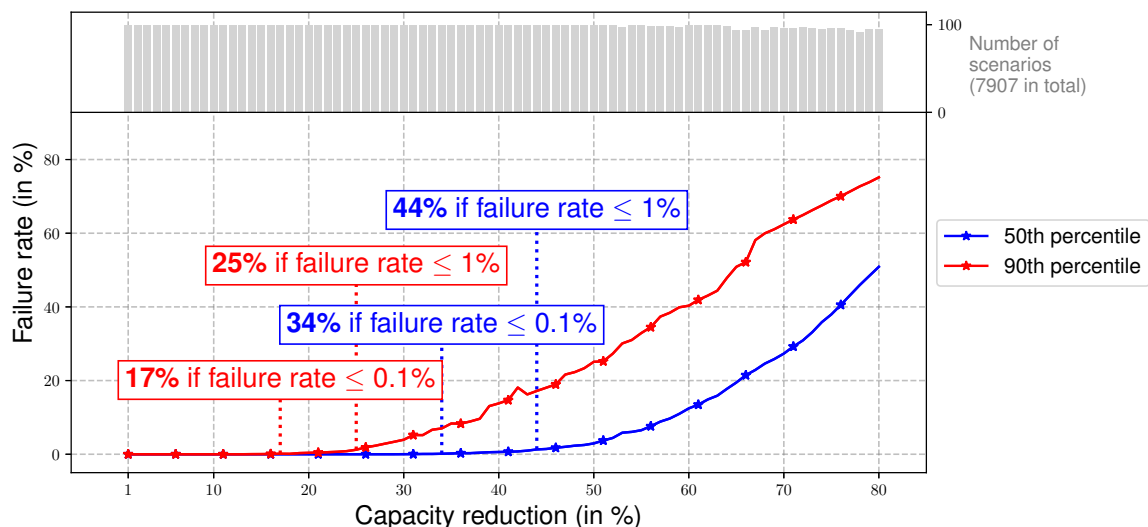


**Figure 14.3:** Capacity reduction for 4996 experiments based on  $Z_{5000}$ . In contrast to Fig. 14.1, rectangles represent a maximum allowed failure rate of 0.1% and 1%.

It can be seen in Fig. 14.3 that a relaxation of the accepted failure rate increases the performance significantly. 90% of the experiments can be handled with capacity reductions of up to 15% if a failure rate below 0.1% is accepted. This is increased by a factor of 2.14 when compared to the numbers from Fig. 14.1 (7%). If a failure rate of 1% is accepted, the performance is further increased to 21% (factor 3). For 50% of the experiments, the values are increased from 28% to 35% with 0.1% failure rate and to 42% with 1% failure rate. The effect is noticeably stronger for smaller capacity reduction factors which makes sense because the approach does not scale beyond reduction factors of 50% and there is little room for improvement for higher factors. The second experiment series based on  $Z_{100}$  shown in Fig. 14.4 shows a similar behavior which supports the results.

## 14.3 Performance with Different Utilization Ratios

The performance results presented so far do not differentiate between scenario characteristics. However, it is expected – and was already shown for individual scenarios in Sec. 13.4.1 – that the available free capacity in the network correlates with the failure rate and the maximum capacity reduction that can be achieved without failures. It is investigated in this section how the available free capacity affects flow delegation performance.



**Figure 14.4:** Capacity reduction for 7907 experiments based on  $Z_{100}$ . In contrast to Fig. 14.2, rectangles represent a maximum allowed failure rate of 0.1% and 1%.

### 14.3.1 Experiment Setup

The basic setup is the same as before, i.e., the two experiment series based on  $Z_{5000}$  and  $Z_{100}$  described in Sec. 14.1.1 are used. However, to investigate the impact of the available free capacity, the experiments are grouped into seven disjoint classes based on flow table utilization ratio. Recall from Def. 12.6 that this ratio quantifies the available free capacity. Table 14.2 shows the seven classes for  $Z_{5000}$ . The majority of the classes covers ratios in a 10% region. Only the first and the last class cover a slightly larger region – 20% in case of  $c_1$  and 30% in case of  $c_7$  – because of the low number of available samples.

	Class	Mean ratio	Samples	0% FR	0.1% FR	1% FR
$c_1$	0%-20%	15.89	880	66.48%	71.82%	84.20%
$c_2$	20%-30%	24.81	1539	67.12%	73.49%	85.77%
$c_3$	30%-40%	34.61	1152	62.07%	66.32%	79.77%
$c_4$	40%-50%	44.61	737	48.30%	54.55%	67.84%
$c_5$	50%-60%	54.60	398	32.66%	40.95%	56.28%
$c_6$	60%-70%	64.44	199	23.62%	31.16%	45.23%
$c_7$	70%-100%	76.08	91	7.69%	14.29%	27.47%

**Table 14.2:** Flow table utilization classes for experiments based on  $Z_{5000}$

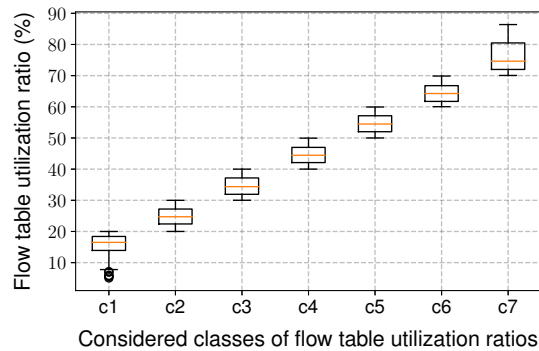
	Class	Mean ratio	Samples	0% FR	0.1% FR	1% FR
$c_1$	0%-20%	16.74	781	54.93%	60.82%	73.50%
$c_2$	20%-30%	25.34	2495	46.81%	51.62%	63.45%
$c_3$	30%-40%	35.24	2071	36.36%	41.43%	51.86%
$c_4$	40%-50%	45.19	1120	30.80%	35.09%	43.12%
$c_5$	50%-60%	53.89	800	20.75%	24.88%	33.38%
$c_6$	60%-70%	63.68	320	21.25%	23.75%	30.00%
$c_7$	70%-100%	71.32	320	4.38%	9.38%	19.06%

**Table 14.3:** Flow table utilization classes for experiments based on  $Z_{100}$

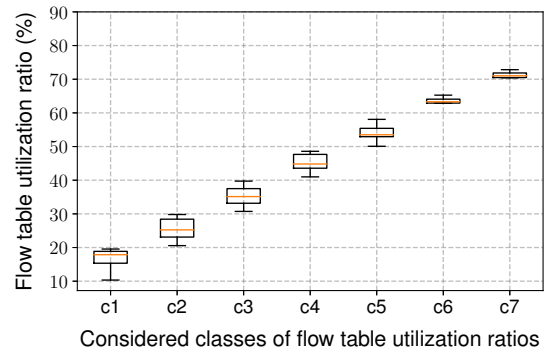
The first class  $c_1$ , for example, contains 880 experiments with a flow table utilization ratio between 0% and 20%, with a mean of 15.89%. The three additional values listed in the table specify the fraction of the experiments with a failure rate of 0%, up to 0.1% and up to 1%. The numbers confirm the results from the previous section in the sense that the fraction of experiments that can achieve an acceptable failure rate increases if the accepted maximum failure rate itself is increased. For class  $c_7$ , for example, the fraction of experiments with a failure rate up to 0.1% is significantly higher than the fraction of experiments that achieved a 0% failure rate. The effect is visible across all seven classes. However, it is noticeably stronger for scenarios with a high utilization ratio. This is the expected behavior because less free capacity leads to more difficult problem formulations and increases the chance that no suitable remote switch allocation is available for a subset of the rules.

The classes for the second experiment series based in  $Z_{100}$  look similar, shown in Table 14.3. There are, however, two important remarks. First, there are only 100 different scenarios in this scenario set. As a result, there are only 100 different utilization ratios (experiments for the same scenario have the same ratio). Classes  $c_6$  and  $c_7$ , for example, consists of only four different scenarios. The 320 samples come from the 80 different experiments per scenario. Second, the fraction of experiments that can achieve an acceptable failure rate is lower compared to the numbers in Table 14.2. This is because the experiments for  $Z_{100}$  use a uniformly distributed capacity reduction factor. Experiments with a high capacity reduction factor, however, can not achieve acceptable failure rates because of missing spare capacity.

Fig. 14.5 and Fig. 14.6 show the distribution of the individual ratios for both experiment series. The x-axis lists the seven classes. The y-axis denotes the flow table utilization ratio. The data is displayed in box plot form showing minimum, maximum, median as



**Figure 14.5:** Utilization ratios  $Z_{5000}$



**Figure 14.6:** Utilization ratios  $Z_{100}$

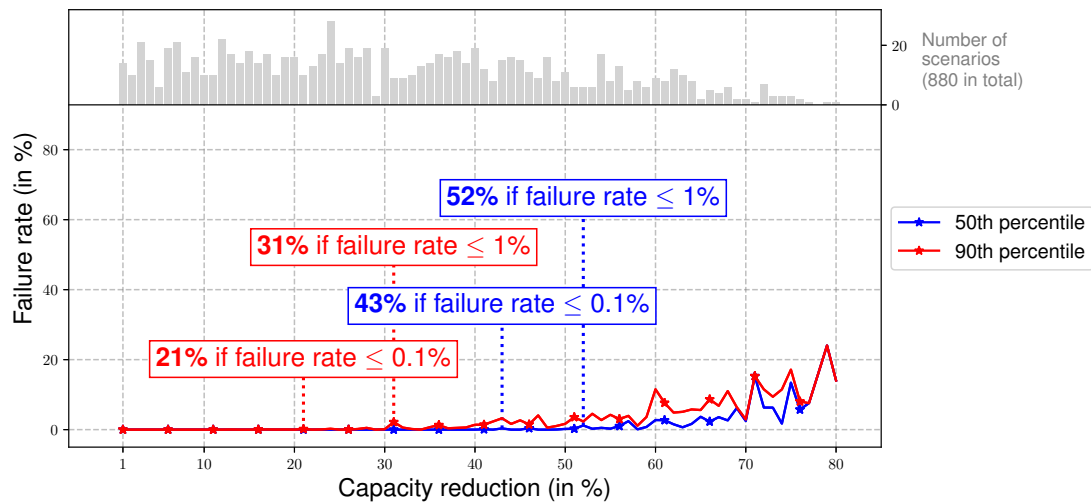
well as 25th and 75th percentiles. In case of  $Z_{5000}$  (in the left), it can be seen that the classes between 20% and 70% are well balanced, i.e., there are experiments for almost all possible values of utilization ratios. The two other classes include the extreme cases with very high or very low ratios. For  $Z_{100}$ , the number of different utilization values is noticeably smaller. The first five classes cover the majority of utilization ratios, but are not as well balanced as in  $Z_{5000}$ .  $c_6$  and  $c_7$  are highly unbalanced – results with respect to utilization ratio that are based on these classes should be treated with care (not used in this thesis).

### 14.3.2 Results

**Findings:** Flow delegation can achieve better performance in scenarios with low utilization ratio where more free capacity is available.

An example for the impact of the utilization ratio on flow delegation performance is given in Fig. 14.7. The plot is designed in the same way as before except that only the 880 experiments of utilization class  $c_1$  are considered (see Table 14.2). The x-axis represents capacity reduction factors in ascending order. The y-axis denotes the maximum failure rate of either 50% (blue curve) or 90% (red curve) of the experiments associated with the given capacity reduction factor (x-value). The histogram in gray in the top denotes the number of experiments for the different reduction factors. The colored rectangles define the flow delegation performance for an accepted failure rate of 0.1% and 1%.

In the original plot with all experiments in Fig 14.7, the performance for the 90th percentile was 17% in case of 0.1% failure rate and 25% in case of 1% failure rate. In Fig. 14.7 where only the experiments in  $c_1$  are considered, the performance is increased to



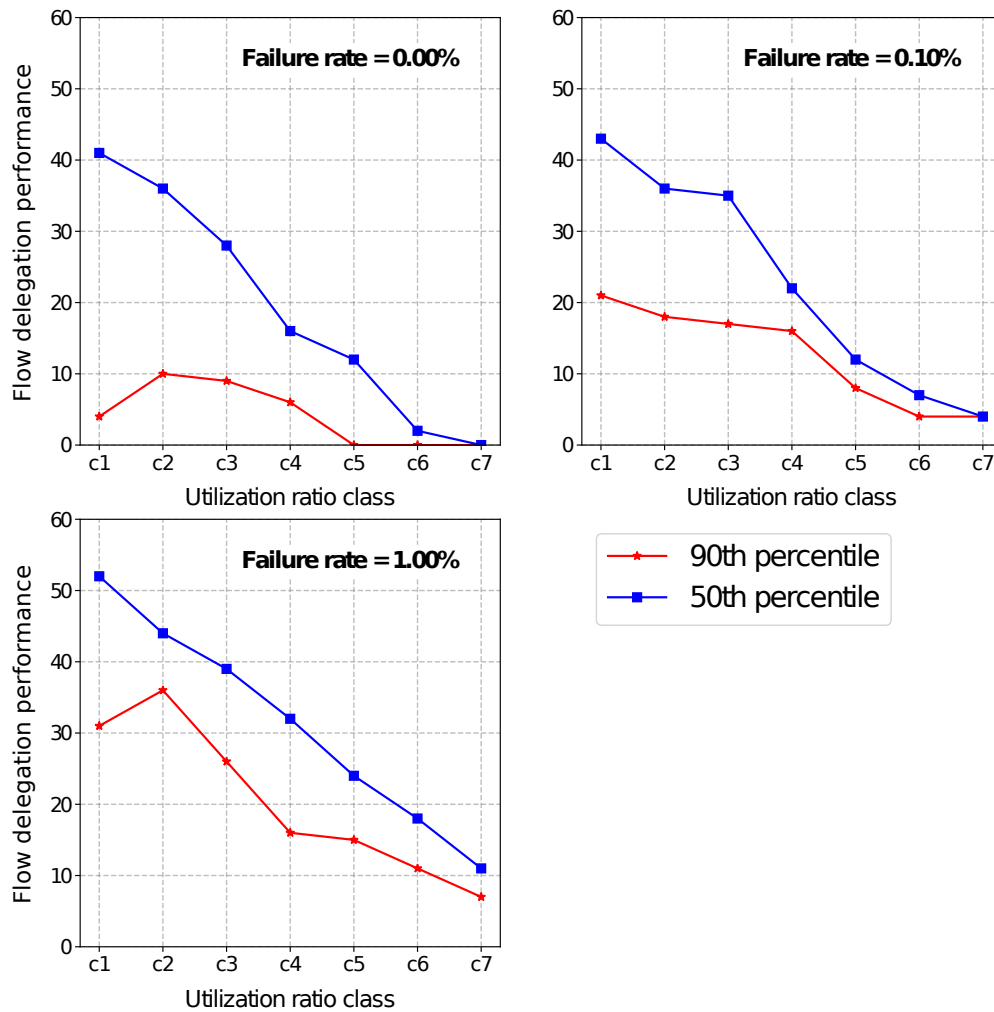
**Figure 14.7:** Flow delegation performance of class  $c_1$  for non-zero failure rates

21% and 31%. The performance values for the 90th percentile are also increased, from 34% to 43% (failure rate 0.1%) and from 44% to 52 % (failure rate 1%).

Fig. 14.8 shows the flow delegation performance of all seven classes and considered failure rates (0%, 0.1%, and 1%). It can be seen that higher utilization ratios result in a higher flow delegation performance. In class  $c_1$  with a low utilization ratio, 50% of the experiments achieve a 0% failure rate for capacity reduction factors up to 41%. The performance is reduced from 41% to 36% in class  $c_2$ , to 28% in class  $c_3$  and this trend continues linearly until the performance reaches 0% in class  $c_7$ . This is expected because  $c_7$  contains the experiments with highest utilization ratio and flow delegation reaches its conceptual limitations. It can also be observed that the results for an accepted failure rate of 0.1% and 1% are similar and that a higher accepted failure rate (e.g., 1% instead of 0.1%) results in a higher performance.

## 14.4 Over- and Underutilization

Two other important key performance indicators for flow delegation are over- and underutilization. Overutilization describes a situation where the flow table utilization in the delegation switch exceeds the capacity for one or multiple time slots, i.e., a capacity bottleneck was not mitigated. Underutilization describes a situation where flow rules are relocated to a remote switch despite the fact that the delegation switch has free entries



**Figure 14.8:** Flow delegation performance for different utilization ratios and different accepted failure rates

left in its flow table. This section investigates these two key performance indicators when flow delegation is applied to the scenarios in  $Z_{5000}$ .

#### 14.4.1 Experiment Setup

Similar to the setup in Sec. 14.1.1, Over- and underutilization is examined for the scenarios in  $Z_{5000}$  with random capacity reduction factors between 1% and 80%. Because the different DT-Select algorithms show different behavior with respect to over- and underutilization, three experiments are executed for each scenario with different algorithms



(Select-Opt, Select-CopyFirst, Select-Greedy). The data set therefore contains 15.000 result sets in total.

Parameter	Description	Used Values
$n_{\text{scenario\_id}}$	Used scenario IDs	All scenario IDs in $Z_{5000}$
$n_{\text{reduction}}$	Capacity reduction factor	Random between 1% and 80%
$n_{\text{dts\_algo}}$	DT-Select algorithm	Select-Opt, Select-CopyFirst, Select-Greedy

**Table 14.4:** Experiment parameters for Sec. 14.4

It is important to mention that all three algorithms (Select-Opt, Select-CopyFirst, Select-Greedy) in conjunction with the ingress port based aggregation scheme from Sec. 5.2.2.3 worked as expected for 100% of the investigated scenarios, i.e., all scenarios in  $Z_{5000}$  were handled successfully. In case of Select-Opt and Select-CopyFirst, this means the optimization problem was feasible in 100% of the cases.

The following defines metrics for over- and underutilization that are required for the investigations in this section. Overutilization is calculated individually for each switch. It is given as a percentage value that is defined as follows:

**Definition 14.1: Overutilization**

**Overutilization** for a switch  $s \in S$  counts the amount of flow rules above the capacity in all time slots after the flow delegation algorithm is executed and applies a normalization factor (denominator) to get a percentage value:

$$\left( \frac{\sum_{t \in T} \max(u_{s,t}^{\text{FD}} - c_s^{\text{Table}}, 0)}{\sum_{t \in T} \max(u_{s,t}^{\text{Table}} - c_s^{\text{Table}}, 0)} \right) * 100 \quad (14.1)$$

$c_s^{\text{Table}}$  is the capacity of the switch.  $u_{s,t}^{\text{Table}}$  denotes the flow table utilization of switch  $s$  in time slot  $t$  *without flow delegation* (see Def 2.12). And  $u_{s,t}^{\text{FD}}$  denotes the flow table utilization of switch  $s$  in time slot  $t$  *with flow delegation*. The latter term is introduced here to simplify the equation. Using the notation from previous chapters,  $u_{s,t}^{\text{FD}}$  can be calculated as follows:

$$u_{s,t}^{\text{FD}} = u_{s,t}^{\text{Table}} - 2 * |D_{s,t}^*| - \sum_{d \in D_{s,t}^*} u_{d,t}^{\text{Table}} \quad (14.2)$$

$D_{s,t}^*$  is the set of selected delegation templates by the DT-Select algorithm in time slot  $t$  and  $u_{d,t}^{\text{Table}}$  represents the amount of rules that will be relocated if delegation template  $d$  is selected. (see Sec. 9.4.1). The first part ( $2 * |D_{s,t}^*|$ ) accounts for the aggregation and backflow rules because each delegation template results in one aggregation rule and the number of backflow rules is constrained by  $|D_{s,t}^*|$  due to the ingress-port aggregation scheme.

The numerator iterates over all time slots and subtracts the capacity from the current flow table utilization value (with flow delegation). The max function ensures that only positive values are taken into account, i.e., time slots where the utilization stays below the capacity are evaluated to 0. The normalization factor in the denominator represents the amount of flow rules above the capacity if flow delegation is not used. This value is always bigger because flow delegation will always reduce the amount of flow rules in the flow table if used. The  $*100$  is used to get a percentage value. This way, overutilization can be i as the fraction of flow rules above the capacity that are not removed by flow delegation.

Underutilization is also calculated individually for each switch. It is given as a percentage value that is defined as follows:

#### Definition 14.2: Underutilization

**Underutilization** for a switch  $s \in S$  counts the amount of free capacity in the flow table for time slots where the flow table utilization without flow delegation is above the capacity. This value is multiplied with a normalization factor (denominator) to get a percentage value:

$$\left( \frac{\sum_{t \in T^{\text{Bneck}}} \max(c_s^{\text{Table}} - u_{s,t}^{\text{FD}}, 0)}{\sum_{t \in T^{\text{Bneck}}} c_s^{\text{Table}}} \right) * 100 \quad (14.3)$$

Set  $T^{\text{Bneck}}$  contains all time slots  $t \in T$  where the flow table utilization without flow delegation is above the capacity:  $T^{\text{Bneck}} := \{t \in T \mid u_{s,t}^{\text{Table}} > c_s^{\text{Table}}\}$

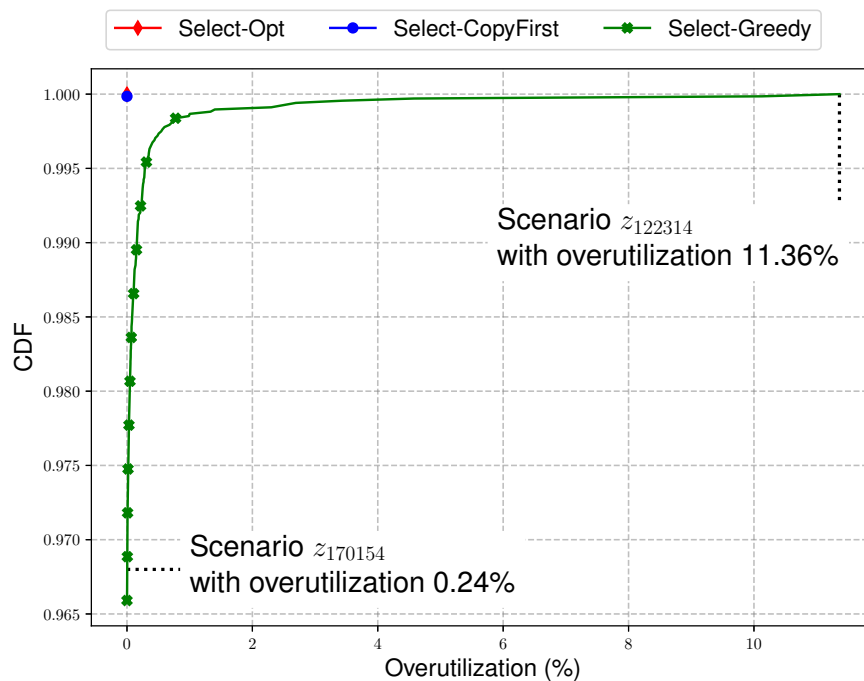
The variables are the same as before.  $c_s^{\text{Table}}$  is the capacity of the switch,  $u_{s,t}^{\text{Table}}$  denotes the flow table utilization of switch  $s$  in time slot  $t$  *without flow delegation* and  $u_{s,t}^{\text{FD}}$  denotes the flow table utilization of switch  $s$  in time slot  $t$  *with flow delegation*.

Underutilization is a common phenomenon caused by the fact that DT-Select makes decisions based on delegation templates and not individual flow rules. It is thus difficult to perfectly utilize all entries in the flow table of the delegation switch. High underutilization indicates that available flow table capacity at the delegation switch is wasted, i.e., lower values are better.

## 14.4.2 Results for Overutilization

**Findings:** No overutilization occurs in any experiment when used with Select-Opt and Select-CopyFirst. Select-Greedy causes overutilization in approx. 3.5% of the experiments but the bottleneck situation is resolved quickly in all cases.

The measured overutilization for all scenarios in  $Z_{5000}$  is shown as a cumulative distribution function in Fig. 14.9. The x-axis denotes the calculated overutilization value given as a percentage value according to Eq. 14.1. The y-axis represents the fraction of experiments with an overutilization value below the x-value. The two scenario IDs are used as examples below.

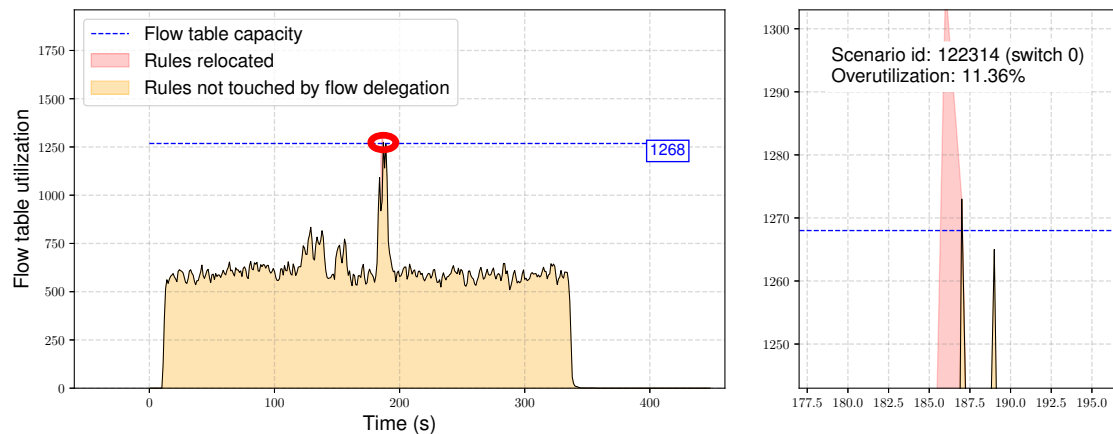


**Figure 14.9:** Overutilization results for Select-Opt, Select-CopyFirst, and Select-Greedy

It can be seen in the figure that both, Select-Opt and Select-CopyFirst have an overutilization value of 0% in all cases. This is the expected result. With flow delegation, there can only be overutilization if the delegation template selection algorithm selects too few delegation templates. The two ILP-based algorithms Select-Opt and Select-CopyFirst, however, use a knapsack constraint for the capacity, i.e.,  $u_{s,t}^{FD}$  is guaranteed to be below

$c_s^{\text{Table}}$  as long as the problem is feasible. Because none of the problem formulations that are required for the experiments here was infeasible, there is no overutilization<sup>1</sup>.

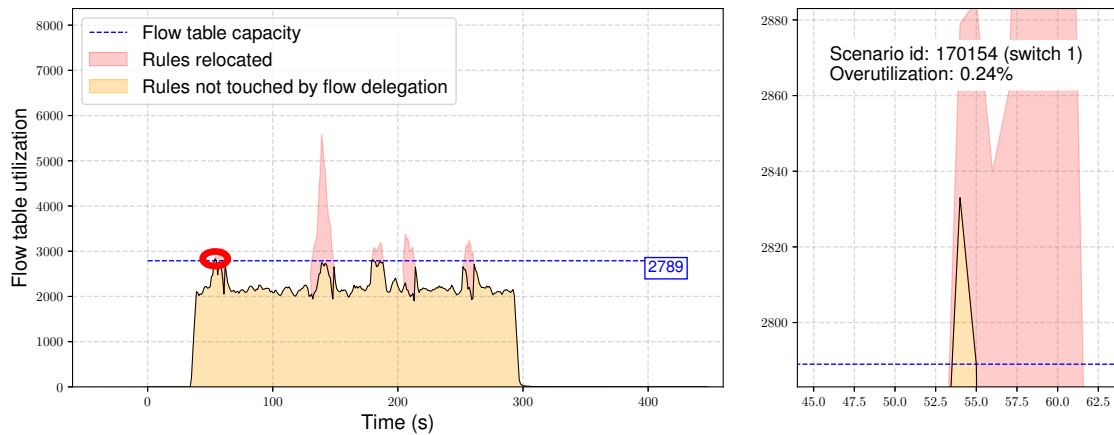
Because Select-Greedy does not use linear programming, it is not guaranteed that there is no overutilization. Fig. 14.9 shows that approx. 3.5% of the 5000 experiments with Select-Greedy resulted in an overutilization value  $> 0\%$  and the maximum overutilization value is 11.36%. The actual results in practice, however, are not as critical as the raw values might indicate. This is due to the normalization factor that is applied in Eq. 14.1. This normalization factor is not a constant but depends on the severity of the bottleneck. For further illustration, the flow table utilization over time and the capacity of the experiment with 11.36% overutilization is shown in Fig. 14.10. The red circle in the left plot shows the situation where the utilization exceeds the capacity. A zoom-in is shown on the right side.



**Figure 14.10:** Overutilization in scenario  $z_{122314}$

It can be seen that, in this case, the bottleneck itself lasts for a very short time and the amount of flow rules that fall into the red region (relocated rules) is small. Because the normalization factor counts the flow rules in the red region above the blue line (at time slot boundaries), the calculated overutilization value is high. It could theoretically also exceed 100%. To account for this observation, we also calculate the experiments with the highest overutilization value without normalization – the worst case in absolute numbers, so to speak. The result is shown in Fig. 14.11.

<sup>1</sup>If a problem formulation would get infeasible, Select-Greedy is used as fallback mechanism. So even if no such case is present in  $Z_{5000}$ , the performance results for Select-Greedy show the anticipated results for a potential infeasible scenario.



**Figure 14.11:** Overutilization in scenario  $z_{170154}$

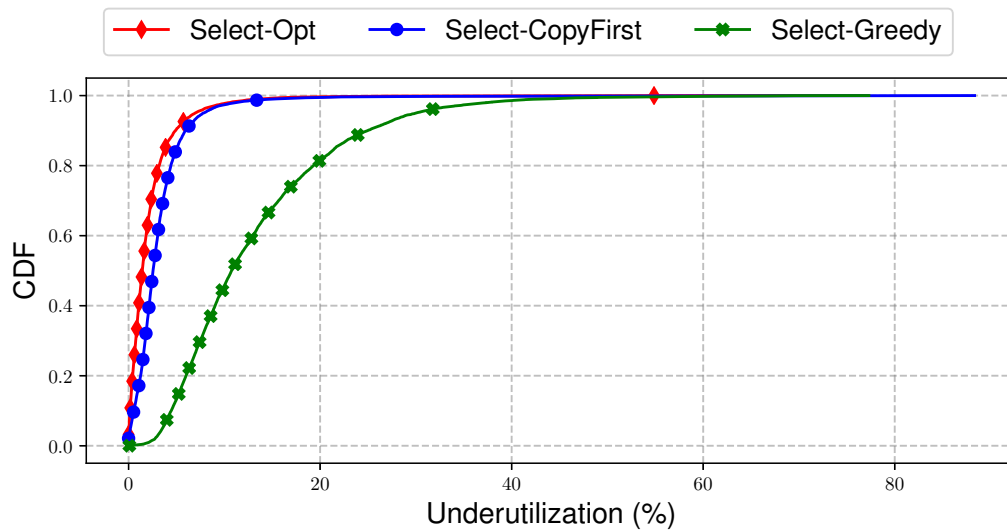
The figure shows that this situation is more severe than the one seen before because more than 30 flow rules are affected. However, even in this worst case example, the situation is resolved in 3 time slots. This is also true for all other experiments with overutilization. It is concluded that Select-Greedy can still be used if occasional overutilization is acceptable. Note that the normalized overutilization value of the example in Fig. 14.11 is only 0.24% because the bottleneck (red area) above the capacity is much larger.

Final remark: overutilization is fundamentally different from the failure rate where flow rules are sent to the backup switch because not enough free capacity is available. A non-zero failure rate (RS-Alloc was forced to use the backup switch) is expected if free capacity in the remote switches is scarce while overutilization (DT-Select failed) will never occur in the optimal case.

### 14.4.3 Results for Underutilization

**Findings:** Underutilization with Select-Opt is below 5% in 90% of the cases. Select-CopyFirst performs slightly worse and stays below 5% underutilization in 80% of the cases. Select-Greedy can only achieve underutilization below 10.74% in 50% of the cases.

The measured underutilization for all scenarios in  $Z_{5000}$  is shown as a cumulative distribution function in Fig. 14.12. The x-axis denotes the calculated underutilization value given as a percentage value according to Eq. 14.3. The y-axis represents the fraction of scenarios with underutilization below the x-value.

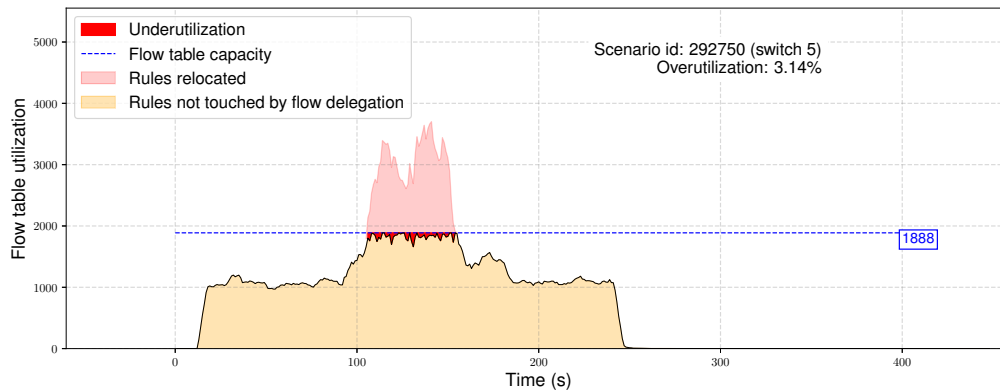


**Figure 14.12:** Underutilization results for *Select-Opt*, *Select-CopyFirst*, and *Select-Greedy*

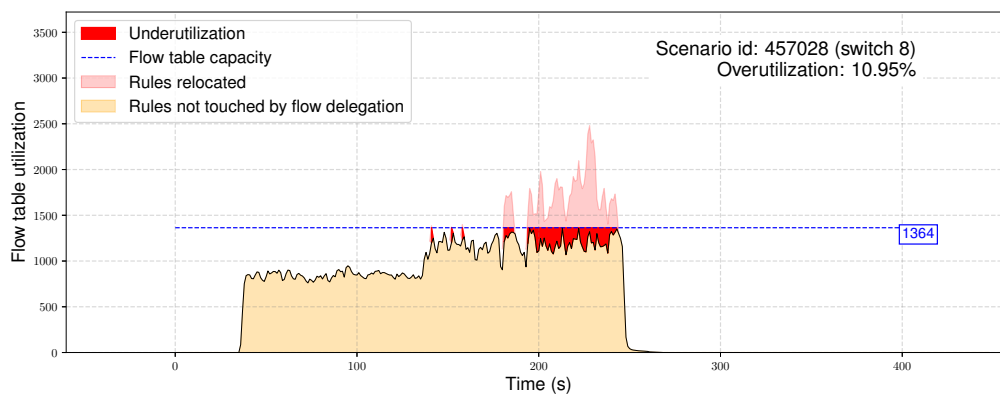
It can be seen in the figure that, as expected, all algorithms suffer from a certain degree of underutilization. *Select-Opt* has the best performance, followed by *Select-CopyFirst* and – with some distance – *Select-Greedy*. In 50% of the experiments, using *Select-Opt* results in a maximum underutilization of 1.42% compared to 2.55% with *Select-CopyFirst* and 10.75% with *Select-Greedy*.

Less underutilization means more flow rules can be stored in the delegation switch which reduces the required additional link bandwidth and also the number of flow rules affected by delegation. Reducing the number of remote rules is also beneficial from the perspective of the end users – recall from Sec. 8.4 that the detour via the remote switch increases the end-to-end delay by up to 0.1ms. With underutilization below 5% in at least 80% of the cases, both *Select-Opt* and *Select-CopyFirst* show promising results. *Select-Greedy* should only be used if the other two approaches are not applicable because the number of unnecessarily relocated flow rules is significantly higher. This shows the general limitation of the greedy-based approach compared to the much more flexible ILP-based algorithms.

To get a better understanding of the practical implications of underutilization, Fig. 14.13 shows a typical example with 3.14% underutilization which is considered a good value. It is clearly visible in the plot that the flow table resources in the delegation switch are utilized properly when compared to a similar example with 10.95% underutilization shown in Fig. 14.14.

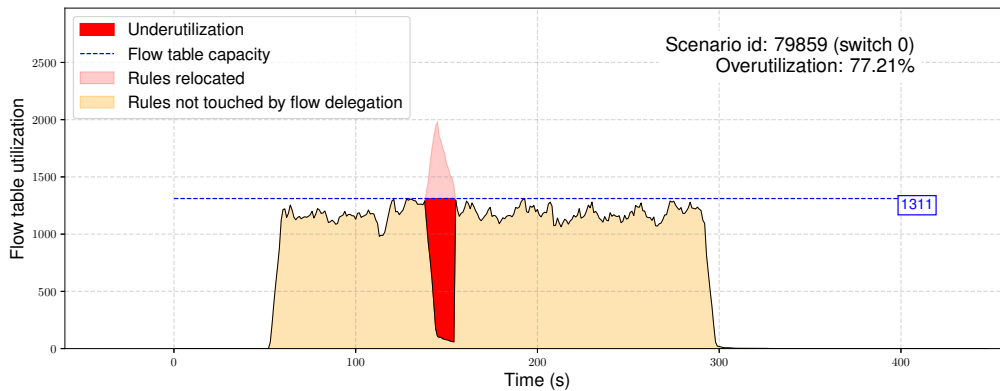


**Figure 14.13:** Underutilization in scenario  $z_{292750}$  (with *Select-CopyFirst*)



**Figure 14.14:** Underutilization in scenario  $z_{457028}$  (with *Select-Greedy*)

The last aspect that has to be discussed with respect to underutilization are the extreme outliers in the top of Fig. 14.9: a small fraction of approx. 0.1% of the experiments show underutilization values between 40% and 80%. This is significantly higher than what is achieved in other experiments. 80% underutilization means that only 20% of the flow table is used in the delegation switch – which is far from reasonable. Such an experiment is shown in Fig. 14.15. This experiment was conducted with *Select-Opt* but the same effect occurs if *Select-CopyFirst* or *Select-Greedy* are used. It can be seen in the darker red area that all flow rules from the delegation switch are relocated away around the 150s mark. What seems like an error at first glance can be explained with the scenario generation parameters that are used for this experiment.

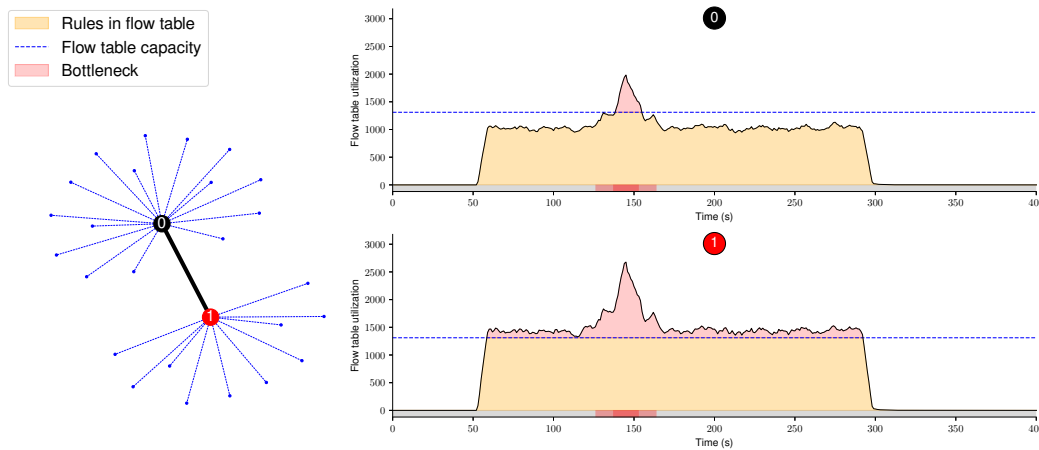


**Figure 14.15:** Underutilization in scenario  $z_{79859}$  (with Select-Opt)

The scenario used is shown in Fig. 14.16. The fact that only 2 switches are used is not essential but simplifies the explanation, the effect can also occur with more than 2 switches. It can be seen that the switches share the same bottleneck. This alone, of course, is not enough to explain the high underutilization. The critical point is the hotspot in switch 1 in conjunction with a very high inter switch ratio ( $n_{isr}$ ). For this scenario, the inter switch ratio was set to  $n_{isr} = 70\%$ . This means that only 30% of the flow rules handle “local” traffic between two hosts while the other 70% redirect traffic to either switch 0 or switch 1. To understand why this is a problem, take switch 0 as an example which is the switch shown in Fig. 14.15. The delegation template selection algorithm has 16 delegation templates to choose from (recall that we use the ingress-port based aggregation scheme). 15 of these templates are associated with the links connected to the hosts shown in blue. The last template is associated with the link towards switch 1. What happens now is that, because of the hotspot and the high inter switch ratio, the majority of the communication pairs select  $h_{src}$  attached to switch 1 (due to the hotspot) and  $h_{dst}$  attached to switch 0 (due to high  $n_{isr}$ ). Consequentially, the 15 templates associated with the links connected to the hosts of switch 0 will rarely start a new communication and the cover sets of the delegation templates will be very small. So small, that these 15 templates alone are not sufficient to mitigate the bottleneck. In result, the algorithm has no other choice than selecting the delegation template that is associated with switch 1. The conflict free cover set of this template, however, is so large that it contains the majority of all the flow rules in the flow table. This explains why all the flow rules are suddenly relocated and the underutilization gets as high as 77.21% in this case. It was confirmed that all outliers can be explained with this specific effect.

In practice, this effect can be easily compensated by setting a threshold for the maximum number of flow rules in the conflict-free cover set and ignoring all templates in the





**Figure 14.16:** Detailed view of scenario  $z_{79859}$  to explain the effect seen in Fig. 14.15

selection process that exceed the threshold. This was not done here to make sure that the anticipated and expected effect actually occurs and to determine its probability (extreme cases occur in approx. 0.1% of the experiments).



# Overhead

---

Bottleneck mitigation with the help of flow delegation does not come for free. This chapter investigates the overhead that is associated with the approach. There are three major kinds of overhead that need to be considered (see Sec. 9.4.4 for further details):

- (1) **Table overhead:** the additional rules installed in the flow table of the delegation switch. It is shown in Sec. 15.3 that 99% of the investigated experiments can be handled with a maximum table overhead of 10 aggregation rules per time slot with all three DT-Select algorithms.
- (2) **Link overhead:** the amount of bits relocated from delegation switch to remote switch and vice versa. It is shown in Sec. 15.4 that 50% of the investigated scenarios can be handled with a maximum link overhead of less than 13 Mbit per switch and time slot in case of Select-Opt and 19 Mbit per switch and time slot in case of Select-CopyFirst. Select-Greedy causes significantly more link overhead.
- (3) **Control overhead:** the additional control messages required to keep the flow delegation system running. It is shown in Sec. 15.5 that most investigated scenarios can be handled with less than 100 additional control messages per second and that Select-Greedy causes significantly more control overhead than the two other algorithms.

The following section first investigate the three above kinds of overhead when flow delegation is applied to an example scenario. Subsequent section then investigate the overhead for all scenarios in  $Z_{5000}$ .

## 15.1 Experiment Setup

This section uses the same setup as in Sec. 14.4.1, i.e., the scenarios in  $Z_{5000}$  with the three different delegation template selection algorithms and random capacity reduction factors between 1% and 80%. Table overhead, link overhead and control overhead are specified per time slot and calculated individually for each switch. Table overhead is given as an average value of required aggregation rules per time slot:

### Definition 15.1: Table overhead

**Table overhead** for a switch  $s \in S$  counts the amount of aggregation rules installed in the flow table of the delegation switch, divided by the number of time slots with bottlenecks:

$$\frac{1}{|T^{\text{Bneck}}|} * \sum_{t \in T^{\text{Bneck}}} |D_{s,t}^*| \quad (15.1)$$

Set  $T^{\text{Bneck}}$  contains all time slots  $t \in T$  where the flow table utilization without flow delegation is above the capacity:  $T^{\text{Bneck}} := \{t \in T \mid u_{s,t}^{\text{Table}} > c_s^{\text{Table}}\}$ .  $D_{s,t}^*$  is the set of selected delegation templates in time slot  $t$ .

Note that the backflow rules are not included in Eq. (15.1) because it is assumed that a static set of  $|D_{s,t}^*|$  backflow rules is installed as soon as the flow delegation system is enabled (see Sec. 9.4.4.1). The real overhead is thus increased by a constant value of approx. 10 to 20 rules.

Link overhead represents the additional bandwidth required between delegation switch and remote switch(es).

### Definition 15.2: Link overhead

**Link overhead** for a switch  $s \in S$  counts the bits relocated by all installed aggregation rules, divided by the number of time slots with bottlenecks:

$$\frac{1}{|T^{\text{Bneck}}|} * \sum_{t \in T^{\text{Bneck}}} \sum_{d \in D_{s,t}^*} w_{d,t}^{\text{Link}} \quad (15.2)$$

Set  $T^{\text{Bneck}}$  contains all time slots  $t \in T$  where the flow table utilization without flow delegation is above the capacity:  $T^{\text{Bneck}} := \{t \in T \mid u_{s,t}^{\text{Table}} > c_s^{\text{Table}}\}$ .  $D_{s,t}^*$  is the set of selected delegation templates in time slot  $t$ .  $w_{d,t}^{\text{Link}}$  represents the link overhead cost coefficient first introduced in Sec. 9.4.4.2 that represents the relocated bits for template  $d$  in time slot  $t$ .

It is important to mention that link overhead always refers to the amount of bits that need to be relocated *per switch*. This does not mean that all bits are relocated via the same link. If different delegation templates are allocated to different remote switches, the overhead is shared among these links.

Control overhead is finally given as the average amount of additional control messages that are required per time slot in order to realize flow delegation:

**Definition 15.3: Control overhead**

**Control overhead** for a switch  $s \in S$  counts the additional control messages sent by the flow delegation system, divided by the number of time slots with bottlenecks:

$$\frac{1}{|T^{\text{Bneck}}|} * \sum_{t \in T^{\text{Bneck}}} \sum_{d \in D_{s,t}^*} w_{d,t}^{\text{Ctrl}} \quad (15.3)$$

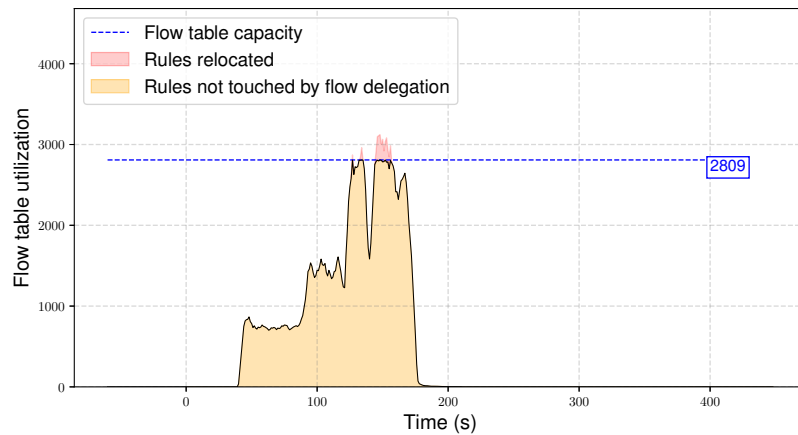
Set  $T^{\text{Bneck}}$  contains all time slots  $t \in T$  where the flow table utilization without flow delegation is above the capacity:  $T^{\text{Bneck}} := \{t \in T \mid u_{s,t}^{\text{Table}} > c_s^{\text{Table}}\}$ .  $D_{s,t}^*$  is the set of selected delegation templates by the DT-Select algorithm in time slot  $t$ .  $w_{d,t}^{\text{Ctrl}}$  represents the control message overhead cost coefficient first introduced in Sec. 9.4.4.3 that represents required additional control messages for template  $d$  in time slot  $t$ .

## 15.2 Overhead Example

This section presents an example for the above overhead definitions based on a concrete scenario. First recall that all three kinds of overhead in Sec. 15.1 are defined for DT-Select and quantify resources used by the flow delegation system that would not be used otherwise:

- Table overhead quantifies the amount of aggregation rules installed in the delegation switch (uses space in the flow table)
- Link overhead quantifies the traffic that is relocated by the aggregation rules (uses bandwidth of one or multiple links connected to the delegation switch)
- Control overhead quantifies the additional control messages sent by the flow delegation system (uses bandwidth of the control channel and also computational resources on the switch required for processing the control messages)

To get a better understanding what this means in raw numbers and how the different DT-Select algorithms perform in terms of overhead, consider the example given in Fig.



(a) Select-Opt

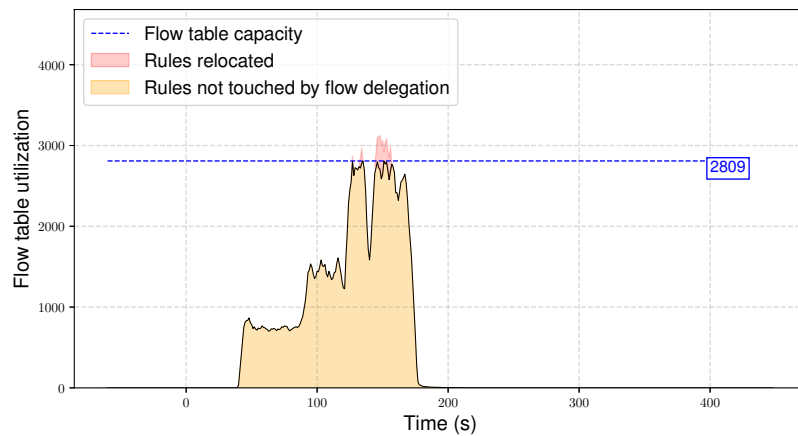
**Statistics**

Capacity reduction: 10%  
 Time slots above capacity: 15  
 DTS Alg.: Select-Opt  
 Delegation templates: 15  
 Overutilization: 0.00%  
 Underutilization: 0.61%

**Overhead**

Table: 36 (2.40)  
 Link: 360.79 Mbit (24.05)  
 Control: 1022 (68.13)

Scenario id: 136505 (switch 1)  
 Capacity without flow delegation: 3122



(b) Select-CopyFirst

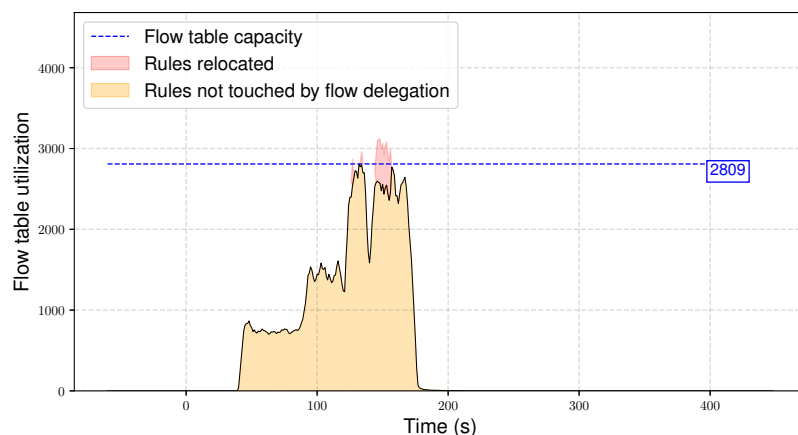
**Statistics**

Capacity reduction: 10%  
 Time slots above capacity: 15  
 DTS Alg.: Select-CopyFirst  
 Delegation templates: 15  
 Overutilization: 0.00%  
 Underutilization: 3.16%

**Overhead**

Table: 35 (2.33)  
 Link: 883.41 Mbit (58.89)  
 Control: 1236 (82.40)

Scenario id: 136505 (switch 1)  
 Capacity without flow delegation: 3122



(c) Select-Greedy

**Statistics**

Capacity reduction: 10%  
 Time slots above capacity: 15  
 DTS Alg.: Select-Greedy  
 Delegation templates: 15  
 Overutilization: 0.00%  
 Underutilization: 9.30%

**Overhead**

Table: 23 (1.53)  
 Link: 1844.57 Mbit (122.97)  
 Control: 1142 (76.13)

Scenario id: 136505 (switch 1)  
 Capacity without flow delegation: 3122

**Figure 15.1:** Overhead example using scenario  $z_{136505}$

15.1. The three plots show three different experiments executed with scenario  $z_{136505}$  which was already discussed in Sec. 13.4.1. The applied capacity reduction factor is 10% in all three cases, i.e., the switch has a capacity of 2809 rules which is 90% of the maximum flow table utilization of the scenario (3122 rules). The plot in the top shows the flow table utilization over time for the experiment that uses Select-Opt to calculate the delegation templates. The plot in the middle shows the flow table utilization over time if Select-CopyFirst is used. And the last plot in the bottom shows the flow table utilization over time if Select-Greedy is used. The dashed blue line is the capacity of the switch. The rules in yellow are not touched by the flow delegation approach. The rules in red are relocated to a remote switch (not shown).

As a first observation, it can be seen from the general statistics (next to each plot on the right side) that 15 time slots are affected by a bottleneck over the course of the three experiments. It can also be seen that **all three DT-Select algorithms can successfully mitigate the two bottleneck situations** that occur between the 100 and 200 second mark. However, the caused overhead differs dependent on the used algorithm. The overhead details are listed below the general statistics. The absolute values are shown first. The average values per time slot are shown in brackets after the absolute values. The value in brackets is the value from the definitions in Sec. 15.1.

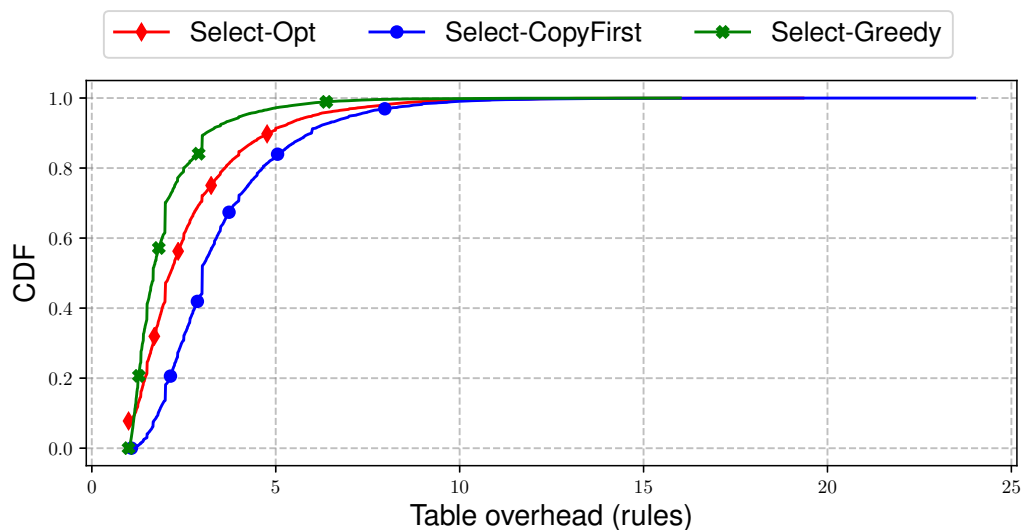
- Table overhead: Select-Opt requires 36 aggregation rules in total which is 2.4 rules per time slot on average. Select-CopyFirst requires 35 aggregation rules (2.33 per time slot on average) and select greedy requires 23 aggregation rules. It is explained below in the following sections where these differences come from.
- Link overhead: Select-Opt selects the delegation templates in such a way that the aggregation rules will relocate 360.79 Mbit in total during the 15 time slots that are affected by a bottleneck. This is 24.05 Mbit in each time slot on average. Note that the relocated traffic can be distributed over multiple links if more than one remote switch is selected (not shown here). What is shown, however, is that the two other algorithms return selections that are inferior in terms of link overhead. The selected templates in case of Select-CopyFirst result in 883.41 Mbit in total and the templates selected by Select-Greedy result in 1844.57 Mbit.
- Control overhead. In case of Select-Opt, 1022 additional control messages are sent from the flow delegation system which is 68.13 control messages per time slot on average. These control messages are required if a delegation template is unselected and the currently relocated rules have to be moved back to the delegation switch. The two other algorithms need a similar amount of control messages, 1236 in case of Select-CopyFirst and 1142 in case of Select-Greedy.

It can be concluded that – at least for this example scenario – there is a **gap between the three DT-Select algorithms in terms of overhead**. This gap and a more complete view on overhead caused in different scenarios is investigated in the following sections. This is done individually for each kind of overhead.

### 15.3 Results for Table Overhead

**Findings:** 99% of the scenarios in  $Z_{5000}$  can be handled with a maximum table overhead of 10 aggregation rules per time slot, regardless of the used DT-Select algorithm. Select-Greedy causes the lowest table overhead, followed by Select-Opt and Select-CopyFirst.

The measured table overhead for all scenarios in  $Z_{5000}$  is shown as a cumulative distribution function in Fig. 15.2. The x-axis denotes the calculated table overhead given as the average amount of required aggregation rules per time slot according to Eq. 15.1. The y-axis represents the fraction of experiments with a table overhead below the x-value.



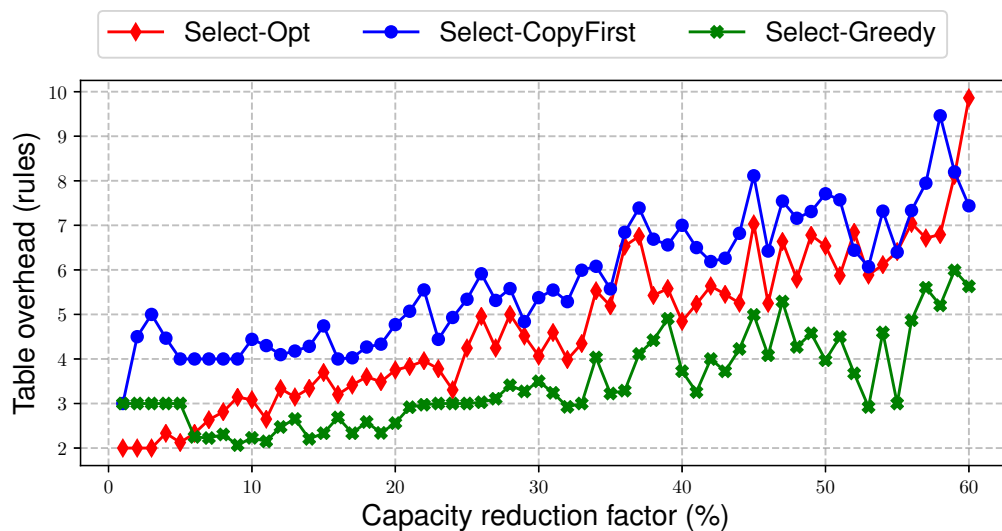
**Figure 15.2:** Table overhead results for *Select-Opt*, *Select-CopyFirst*, and *Select-Greedy*

It can be seen that all three algorithms can handle 99% of the experiments with less than 10 aggregation rules (on average). For 50% of the experiments, the table overhead is at 1.67 rules in case of Select-Greedy, 2.13 rules in case of Select-Opt and 3 rules in case of Select-CopyFirst. Together with the backflow rules, the total amount of additional rules stored in a delegation switch stays below 30 rules in almost all cases. This is less than



1% of the capacity in case of a 3000 rule switch. It is concluded that table overhead is not a critical limitation of the flow delegation approach.

The fact that Select-Greedy has the lowest table overhead is directly linked to the observation that this algorithm causes the highest underutilization: its lack of flexibility. Other than the ILP-based approaches, Select-Greedy can only select or unselect delegation templates in one time slot. It can not select one template in exchange for another one. It therefore tends to just stick with a once selected delegation template. This means most delegation templates are used for a long time which again means that more rules of this template are relocated (recall that only the rules after installation of the aggregation rule are relocated). Because more relocated rules are associated with a single template, less templates are required (lower table overhead) but it is more difficult to efficiently utilize the space in the flow table of the delegation switch (higher underutilization). The two other algorithms switch the templates much more frequently to improve the objective function which leads to a smaller amount of relocated rules associated with each template (higher table overhead, lower underutilization).



**Figure 15.3:** Table overhead grouped by capacity reduction factor

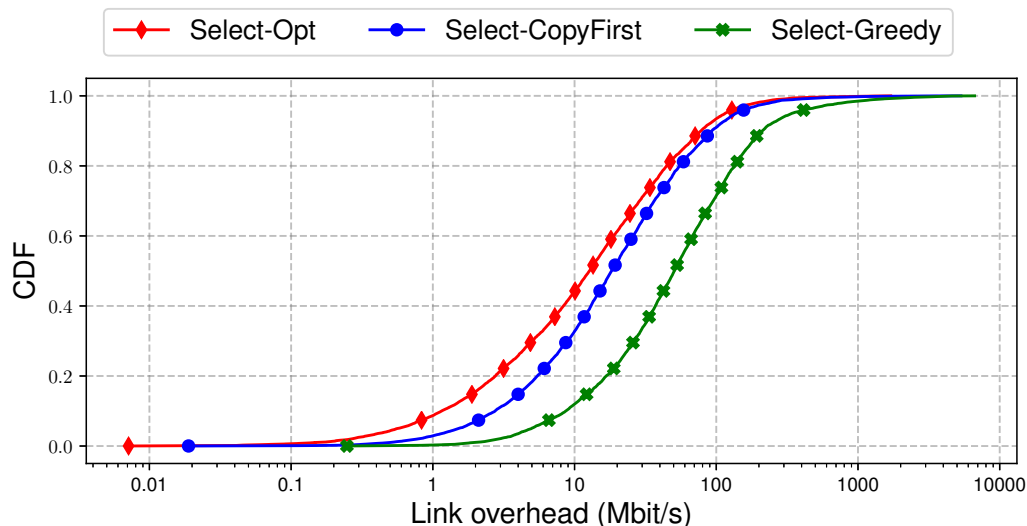
Fig. 15.3 shows the table overhead for different capacity reduction factors. A value on the x-axis represents all experiments with a specific capacity reduction factor. The corresponding y-value is the 90th percentile of the table overhead for these experiments. It can be seen that, up to a capacity reduction factor of 20%, the table overhead is below 5 rules for all three algorithms. In addition, the curves for all three algorithms follow a linear trend. So lower capacity reduction factors (less severe bottlenecks) can be handled with less aggregation rules and smaller table overhead. The fluctuations for

higher capacity reduction factors are related to the smaller number of samples in this area (see Sec. 14.1.1).

## 15.4 Results for Link Overhead

**Findings:** 50% of the scenarios in  $Z_{5000}$  can be handled with a maximum link overhead of less than 13 Mbit/s per switch (not per link!) in case of Select-Opt, 19 Mbit/s in case of Select-CopyFirst and 51 Mbit/s in case of Select-Greedy. Smaller capacity reduction factors require less bandwidth.

The measured link overhead for all scenarios in  $Z_{5000}$  is shown as a cumulative distribution function in Fig. 15.4. The x-axis in log-scale denotes the calculated link overhead given as the average amount of relocated bits per switch and time slot according to Eq. 15.2. The y-axis represents the fraction of scenarios with a link overhead below the x-value.

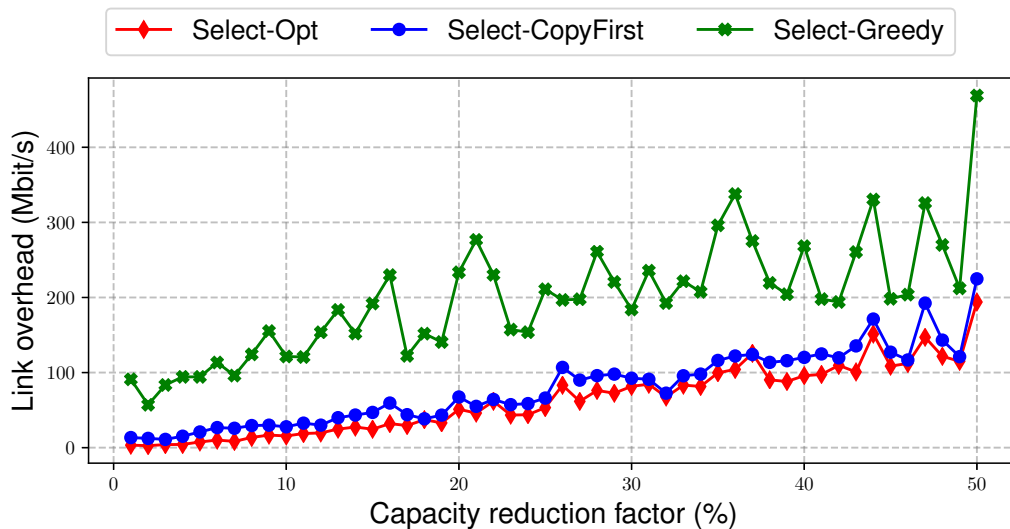


**Figure 15.4:** Link overhead results for *Select-Opt*, *Select-CopyFirst*, and *Select-Greedy*

It can be seen in the figure that Select-Opt (red) has the best link overhead results, followed by Select-CopyFirst (blue) and Select-Greedy (green). If the delegation templates are calculated with Select-Opt, the amount of relocated traffic is below 12.74 Mbit on average per time slot in 50% of the experiments, below 44.63 Mbit in 80% of the experiments and below 277.15 Mbit in 99% of the experiments. If the delegation templates are calculated with Select-CopyFirst, the amount of relocated traffic is below 18.26 Mbit on average per time slot in 50% of the experiments, below 55.12 Mbit in 80%

of the experiments and below 363.19 Mbit in 99% of the experiments. With Select-Greedy, the amount of relocated traffic is below 50.63 Mbit on average per time slot in 50% of the experiments, below 134.19 Mbit in 80% of the experiments and below 1392.05 Mbit in 99% of the experiments.

The significantly higher link overhead caused by Select-Greedy shows that this algorithm should only be used if none of the two other algorithms is applicable, e.g., because of resource constraints or if a backup solution is required when one of the optimization problems is getting infeasible. The difference between Select-Opt and Select-CopyFirst is smaller but still noticeable. If link bandwidth is a critical concern, using Select-Opt instead of Select-CopyFirst might be helpful but this is only possible if enough computing resources for Select-Opt are available.



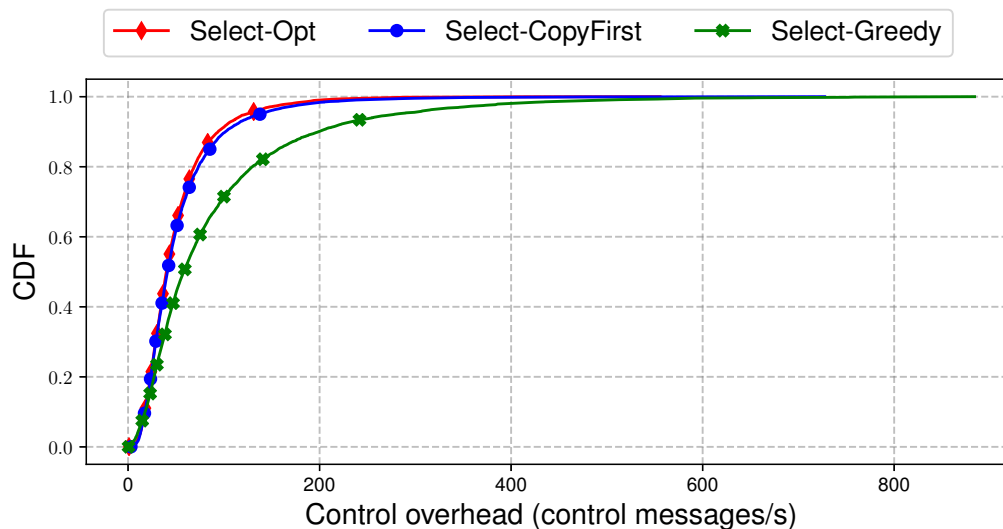
**Figure 15.5:** Link overhead grouped by capacity reduction factor

Fig. 15.5 shows the link overhead for different capacity reduction factors. A value on the x-axis represents all experiments with a specific capacity reduction factor. The corresponding y-value is the 90th percentile of the link overhead for these experiments. It can be seen that the link overhead scales linearly with the capacity reduction factor. This is the expected result because a higher capacity reduction factor leads to a higher number of flow rules that are relocated to a remote switch. This is equivalent to a higher number of packets that must be relocated to a remote switch.

## 15.5 Results for Control Overhead

**Findings:** In 99% of the scenarios in  $Z_{5000}$ , the amount of additional control messages per time slot is below 200 if Select-Opt is used and below 237 if Select-CopyFirst is used. There is only a small difference in terms of control overhead between Select-Opt and Select-CopyFirst and a larger difference between the ILP-based algorithms and Select-Greedy.

The measured control overhead for all scenarios in  $Z_{5000}$  is shown as a cumulative distribution function in Fig. 15.6. The x-axis denotes the calculated control overhead given as the average amount of additional control messages per time slot according to Eq. 15.3. The y-axis represents the fraction of scenarios with a control overhead below the x-value.

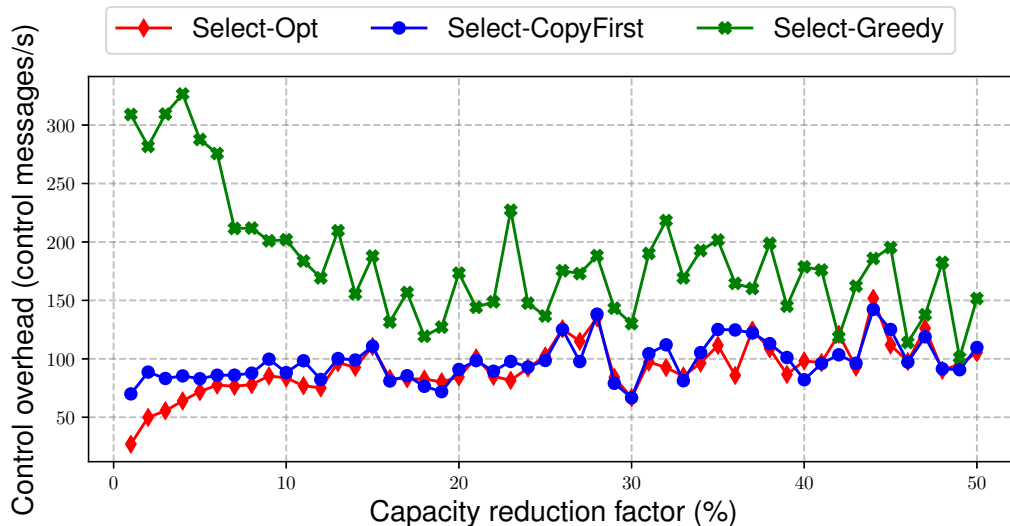


**Figure 15.6:** Control overhead results for Select-Opt, Select-CopyFirst, and Select-Greedy

It can be seen in the figure that Select-Opt (red) and Select-CopyFirst (blue) cause a similar amount of control overhead while the control overhead is increased if Select-Greedy (green) is used. If the delegation templates are calculated with Select-Opt, the amount of additional control messages per time slot is below 40 in 50% of the experiments, below 69 in 80% of the experiments and below 198 in 99% of the experiments. If the delegation templates are calculated with Select-CopyFirst, the amount of additional control messages per time slot is below 42 in 50% of the experiments, below 74 in 80% of the experiments and below 237 in 99% of the experiments. With Select-Greedy, the amount of additional

control messages per time slot is below 58 in 50% of the experiments, below 130 in 80% of the experiments and below 487 in 99% of the experiments.

Similar to the results from the previous section, Select-Greedy performs significantly worse than the two other approaches. The difference between Select-Opt and Select-CopyFirst, on the other hand, is very small.



**Figure 15.7:** Control overhead grouped by capacity reduction factor

Fig. 15.5 shows the control overhead for different capacity reduction factors. A value on the x-axis represents all experiments with a specific capacity reduction factor. The corresponding y-value is the 90th percentile of the control overhead for these experiments. It can be seen that the control overhead converges to a value of approx. 100 control messages per second for increasing capacity reduction factors. This means there is a scenario-independent upper bound on the required control messages per second (approx. 100) that is considered acceptable for current SDN switches.

The results for Select-Opt and Select-CopyFirst are very similar. Select-Greedy performs worse, especially for low capacity reduction factors where a significantly higher number of control messages is required compared to the ILP-based algorithms. This is again caused by the fact that Select-Greedy can only select or unselect delegation templates in one time slot and most delegation templates are used for a long time. This means a higher number of rules is relocated and has to be moved back to the delegation switch when a template is unselected.



# Runtime and Scalability

---

Both, runtime and scalability of the designed algorithms (DT-Select, RS-Alloc) are important for practical application of the flow delegation approach. This chapter investigates the following aspects:

- (1) **Algorithm runtime:** Algorithm runtime is defined as the time in milliseconds required for one optimization period (“sample”) of DT-Select and RS-Alloc. Both, modeling time and solving time are evaluated for the designed algorithms in Sec. 16.1. It is shown that DT-Select and RS-Alloc can be modeled and solved in less than 150ms for 99% of all investigated scenarios on a single CPU core.
- (2) **Number of delegation templates:** The scenarios in  $Z_{100}$  and  $Z_{5000}$  require a comparatively small number of delegation templates (usually  $< 50$ ). It is shown in Sec. 16.2 that the Select-CopyFirst algorithm also works with hundreds of delegation templates without noticeable increase in algorithm runtime.
- (3) **Number of switches:** The scenarios in  $Z_{100}$  and  $Z_{5000}$  only consists of up to 15 switches. It is shown in Sec. 16.3 that RS-Alloc also works with hundreds of switches with a linear increase in algorithm runtime. For capacity reduction factors below 30%, the algorithm runtime is below 500ms in all investigated cases with up to 300 switches (using a single CPU core).

The results presented in this section demonstrate that the flow delegation approach scales linear with three important parameters: individual runtime of the required algorithms, number of delegation templates and number of switches in the topology. This means in summary that a single commodity server with 32 cores can handle a network with hundreds of switches.

## 16.1 Runtime

To analyze runtime characteristics of DT-Select and RS-Alloc, modeling and solving time for each optimization period is measured in milliseconds. Solving time is the time spent inside the Gurobi solver. Modeling time is the time required to prepare the model for the solver and includes translation of the scenario data into lambda notation. The total algorithm runtime is the sum of modeling time and solving time.

### 16.1.1 Experiment Setup

The experiment series used in this section is based on scenario set  $Z_{5000}$  with random capacity reduction factors. This is different from the look-ahead analysis in Sec. A.1.1 which is based on the smaller scenario set with only 100 different scenarios. All experiments are performed with Select-CopyFirst and the recommended parameters specified in Sec. A.4.

Parameter	Description	Used Values
$n_{\text{scenario\_id}}$	Used scenario IDs	All scenario IDs in $Z_{5000}$
$n_{\text{reduction}}$	Capacity reduction factor	Random between 1% and 80%
$n_{\text{dts\_algo}}$	DT-Select algorithm	Select-CopyFirst
$L_{\text{dts}}$	Look-ahead factor DT-Select	3
$L_{\text{rsa}}$	Look-ahead factor RS-Alloc	3
$n_{\text{assignments}}$	Assignments for RS-Alloc	50

**Table 16.1:** *Experiment parameters for Sec. 16.1*

The series is executed on a server with 2 x Intel(R) Xeon(R) Silver 4110 CPUs (32 logical cores, clocked at 2.10 GHz). However, unlike before, the experiment environment is restricted to a single CPU core to avoid the interference effects mentioned in Sec. A.1.2.

### 16.1.2 Results for DT-Select

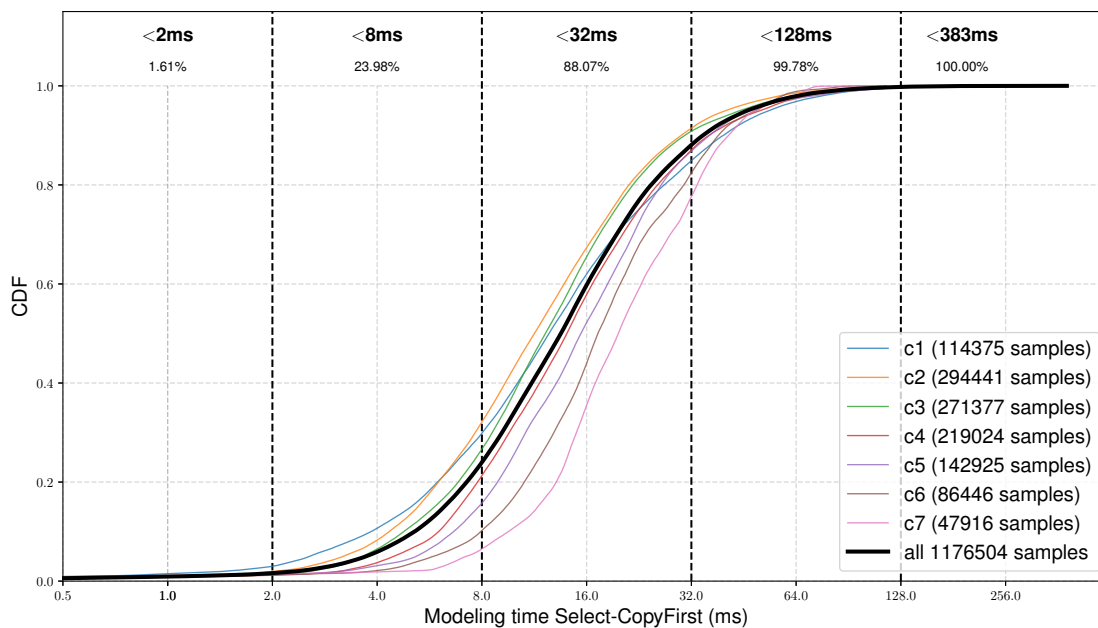
**Findings:** Select-CopyFirst can be modeled and solved in less than 150ms per bottlenecked switch in 99% of the investigated scenarios using a single CPU core. Optimizations can reduce this to values below 50ms.

We already know from the results in Sec. A.1.3 that Select-Opt is too expensive with respect to algorithm runtime. This section will thus focus on the Select-CopyFirst approach.



Fig. 16.1 shows the modeling times for the experiment series as a cumulative distribution function. The x-axis denotes the measured modeling time in milliseconds on a logarithmic scale. The y-axis denotes the fraction of the algorithm samples with a modeling time below the specified x-value. One sample represents the result of one optimization period. The number of samples for an experiment depends on the used scenario and the capacity reduction factor<sup>1</sup>. If 20 time slots are affected, for example, there are  $20+L$  samples for DT-Select.

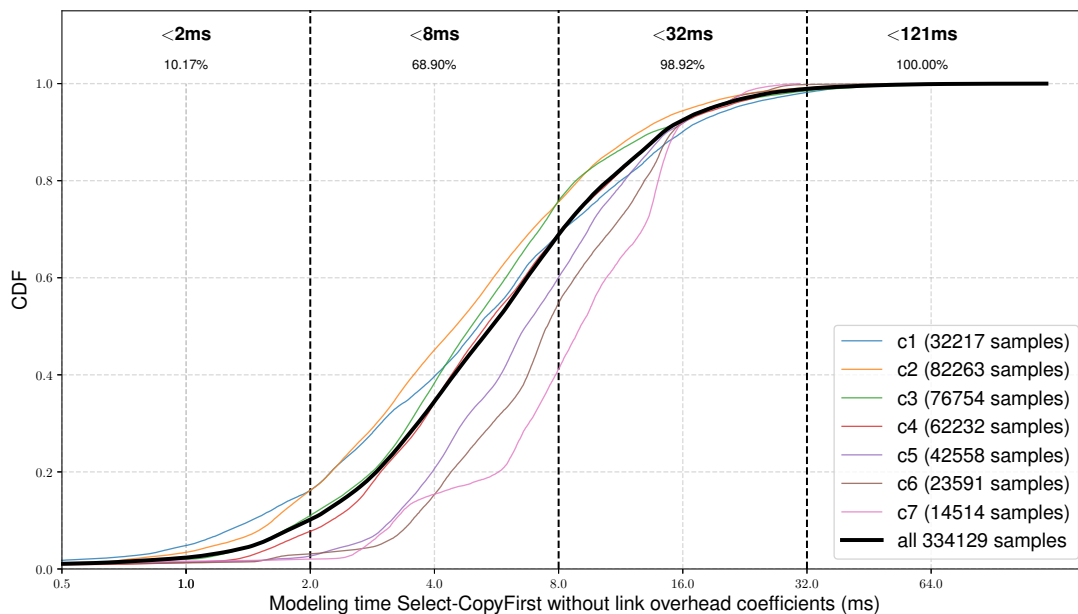
The different colored curves in the plot each represent a subset of the experiments with a certain flow table utilization ratio. This is the ratio between average flow table utilization of all switches in the experiment and the maximum flow table utilization (Def. 12.6). The seven classes between  $c_1$  and  $c_7$  are defined in Table 14.2. The first class  $c_1$  contains all experiments with a flow table utilization ratio between 0% and 20%. Classes  $c_2$  to  $c_6$  contain the experiments with utilization ratio between 20% and 70%. And class  $c_7$  consists of the experiments with a ratio above 70%. The number in brackets behind the class specifies the number of algorithm samples for all experiments in the class. The thicker black line represents the union of all classes, i.e., all samples of all experiments.



**Figure 16.1:** *Select-CopyFirst* modeling time

<sup>1</sup>A higher capacity reduction factor typically means that the bottleneck spans over more time slots

It is shown by the black line in Fig. 16.1 that **99.78% of the 1.176.504 DT-Select samples have a modeling time below 128ms** if a single CPU core is used. This means only 0.22% of the samples require more than 128ms with a worst case modeling time across all samples of 383ms. Note that these are cumulative values and modeling times are lower if the extremer examples are not considered. For 88% of the samples, for example, the modeling step is finished after a maximum of 32ms. And approx. 24% of the samples are finished under 8ms.



**Figure 16.2:** *Select-CopyFirst modeling time without link overhead coefficients*

Nevertheless, a modeling time above 32ms in approx. 12 percent of the cases is not negligible given that DT-Select is executed once per optimization period per switch. However, these number can be optimized in several ways in a real system. For once, the current modeling is done completely in python without optimizations and contains logging/debugging code that could be removed. Using a just-in-time compiler such as PyPy or a more efficient programming language will probably also reduce the modeling time significantly. However, this was not tested. Instead, it was investigated which conceptual part of the modeling step contributes the most to the final measurement result. This investigation showed that more than 50% of the modeling time in the current prototype is used for creating the link overhead cost coefficients. This is expensive because data for each individual flow rule object is extracted. In a real system, however, this data can be provided by some monitoring component and it is not required to calculate

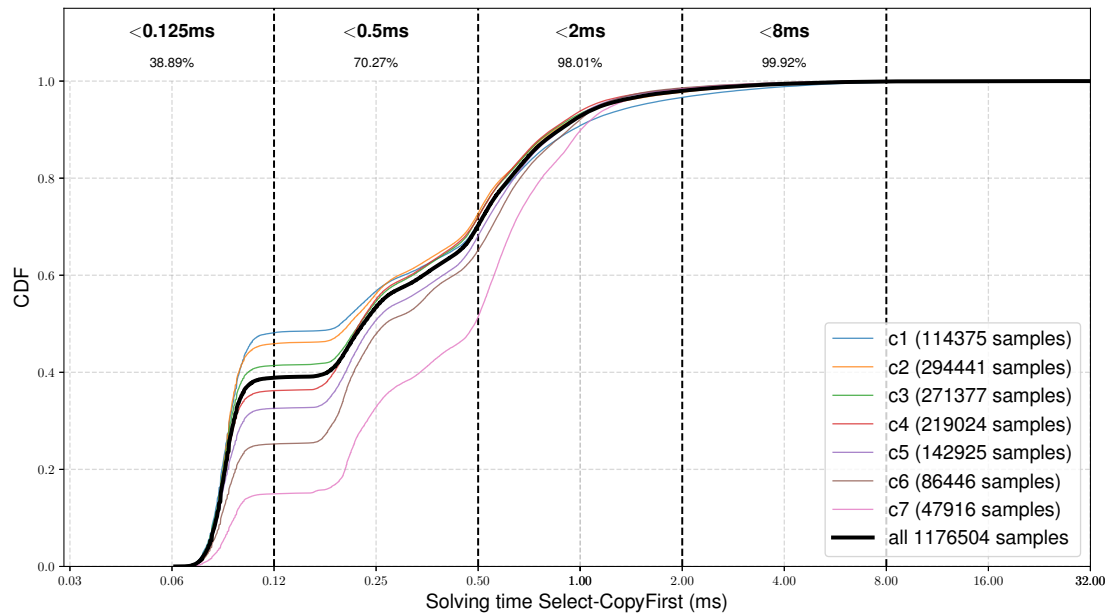
the coefficient from scratch in each optimization period. This and other implementation specific aspects provide potential for optimization.

To quantify this optimization potential, Fig 16.2 shows the modeling times without the link overhead cost coefficients. The figure has the same structure as Fig. 16.1 except that the time required for calculation of the link overhead coefficients is not included. Note that we did not replace or optimize the code, we just set the weight for the link overhead part to 0 in the multi-objective optimization so that these coefficients are not required any more ( $\omega_{DTS}^{\text{Link}} = 0$ ). A real optimization would have to replace this part of the code with an interface to a monitoring system, for example. However, the optimization problem is still usable with  $\omega_{DTS}^{\text{Link}} = 0$  (see results in Sec. A.3.2) and the updated results give a first impression of what can be achieved with a more optimized version of the code. It is shown that **the “optimized” Select-CopyFirst version without link overhead cost coefficients can be modeled in less than 32ms for 98.92% of the samples** which is a significant improvement to before (faster by a factor of 4). The worst case modeling time is reduced from 383ms to 121ms. And 68.9% instead of 24% of the samples require a modeling time of less than 8ms.

Another (less important) aspect that should be discussed here is the impact of the different utilization classes, i.e., the colored curves besides the thicker black curve. Fig. 16.1 shows that scenarios with higher flow table utilization (classes  $c_5$  to  $c_7$ ) seem to have higher modeling times in the lower end of the considered time range, i.e., the fraction of samples with modeling times above 8ms and above 32ms is larger for utilization classes  $c_5$  to  $c_7$ . This observation is accurate but has to be interpreted with caution. Recall that a high flow table utilization ratio means that the bottleneck situation spans over the majority of the time slots while only a small portion of the time slots is affected if the ratio is low. This effect can be observed in the case study where scenarios with different utilization ratio are investigated. Consider the situation with utilization ratio 72.83% in scenario  $z_{166890}$  (Fig. 13.13). Even for small capacity reduction factors such as 10% or 15%, the bottleneck situation basically spans over the whole course of the experiment which means DT-Select is executed for the majority of the 400 time slots. Compare this to the situation in scenario  $z_{155603}$  with utilization ratio 28.4% in Fig. 13.11. In this case – even with a very high capacity reduction factor of 40% –, only a small fraction of the time slots is affected by the bottleneck situation.

This is important here for two reasons: i) the amount of flow rules included in the translation process (lambda notation) is smaller if less time slots are affected over the course of an experiment and ii) the probability that a bottleneck only spans over a small amount of time slots is higher. The latter is especially important given the behavior of the look-ahead mechanism in Select-CopyFirst. As already explained in Sec. A.2.2, delegation templates are selected earlier as necessary because the  $L$  future time slots

are considered as capacity constraints in the optimization problem. It turns out that the coefficients for these “early selections” (i.e., not all considered future time slots have a bottleneck) can be computed faster compared to a situation where all  $L$  future time slots suffer from a bottleneck. This is one of the reasons why the classes with smaller utilization ratio have a higher fraction of samples that can be modeled in 2-8ms.



**Figure 16.3:** *Select-CopyFirst* solving time

The only remaining factor for the algorithm runtime is the time required by the Gurobi solver to calculate the optimal result. The results for the solving time are shown in Fig. 16.3. The structure of the figure is again identical to before. It is shown by the black line that **Gurobi can solve 99.92% of the samples in less than 8ms**. And in 98% of the cases, the solver can find the optimal result in less than 2ms. The strange course of the curves can be explained with different complexity of the individual optimization problems. In approx. 40% of the cases, for example, the solver can simply pre-solve the model which is very fast. For the other cases, the runtime grows with increasing complexity, i.e., with the number of iterations required by the solver.

It is concluded from the above results that i) the current prototype of the Select-CopyFirst algorithm can be calculated in less than 150ms in more than 99.9% of the investigated cases which includes the time for modeling and solving and ii) an implementation with optimizations will be able to reduce the algorithm runtime further. A realistic guess for

the optimization potential is a reduction of the modeling time by at least a factor of 3, i.e., DT-Select can be calculated in less than 50ms.

### 16.1.3 Results for RS-Alloc

**Findings:** RS-Alloc can be modeled and solved in less than 150ms in 99% of the investigated scenarios using a single CPU core.

The previous section investigated the algorithm runtime of the DT-Select step based on required modeling and solving time for one optimization period. This section discusses the same for RS-Alloc. The look-ahead factor is set to  $L = 3$  and 50 allocation assignments are used (to make the problem more difficult).

Fig. 16.4 shows the RS-Alloc modeling times for the experiment series as a cumulative distribution function. The figure has the same structure as the figures in the previous section: the x-axis denotes measured modeling time in milliseconds on a logarithmic scale, the y-axis denotes the fraction of the algorithm samples with a modeling time below the specified x-value and the different colored curves represent subsets of experiments with a certain flow table utilization ratio, arranged in seven classes from  $c_1$  to  $c_7$  as defined in Table 14.2. The thicker black line represents all samples of all experiments.

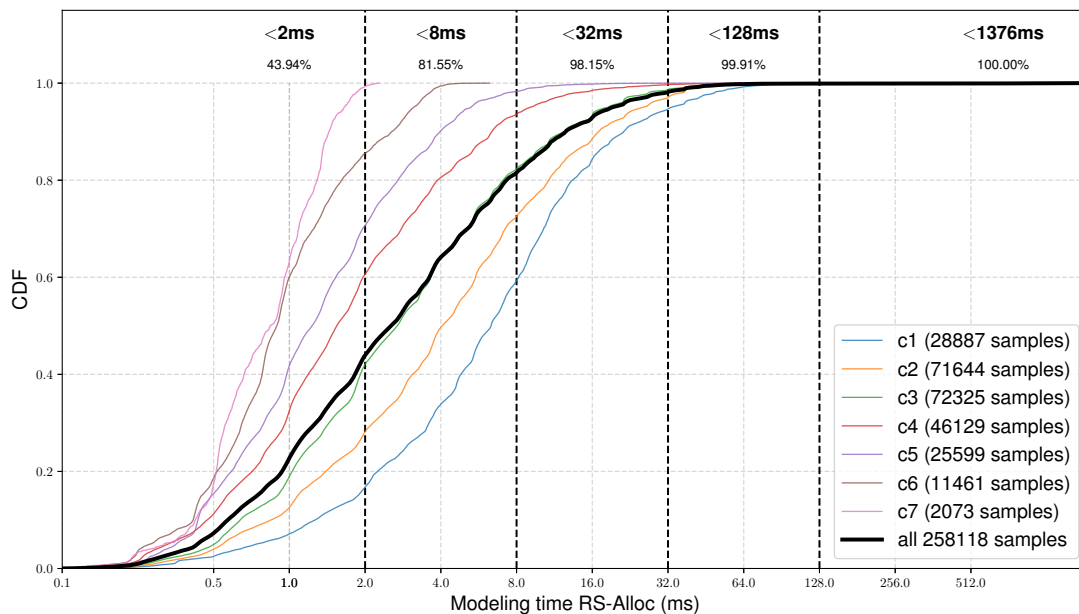
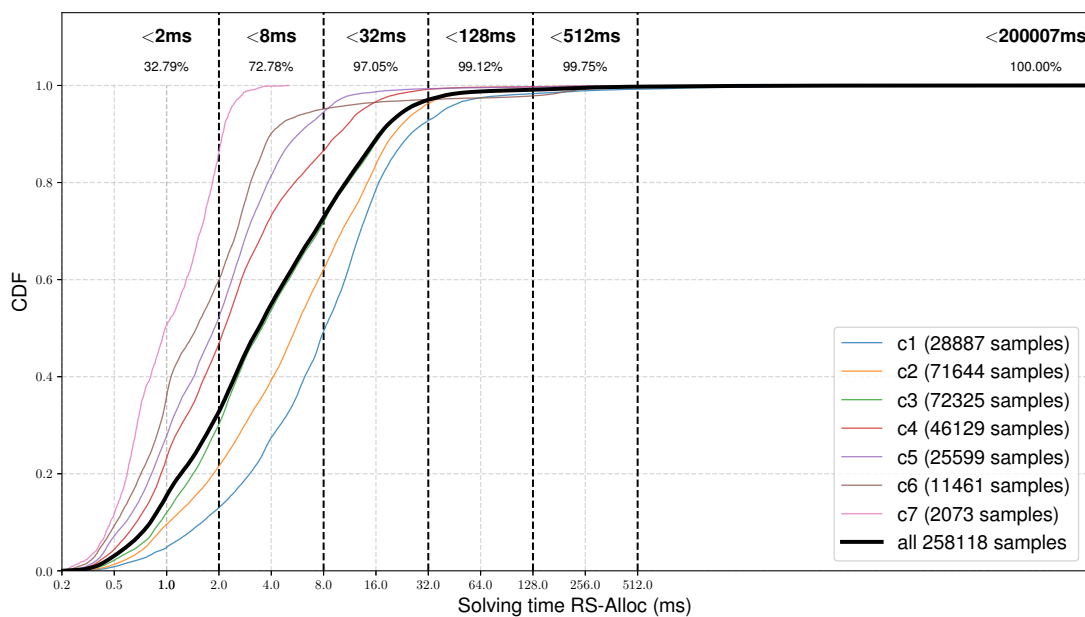


Figure 16.4: RS-Alloc modeling time

The first observation is that the number of samples in case of RS-Alloc is smaller compared to the number of DT-Select samples. This is because DT-Select is executed per bottlenecked switch while RS-Alloc is only executed once after the delegation templates for all bottlenecked switches are calculated. It can be seen in Fig. 16.4 that **the modeling step for RS-Alloc requires less than 128ms in 99.91% of the 258.118 investigated samples using a single CPU core**. Only approx. 0.1% of the samples require more than 128ms with a worst case modeling time of 1376ms across all samples. This is considered acceptable given that RS-Alloc is executed only once per time slot and the modeling step can be easily parallelized if necessary.



**Figure 16.5:** *RS-Alloc solving time*

One interesting observation in Fig. 16.4 is that the different utilization classes have a strong impact on the modeling time. This is expected and linked to the correlation between flow table utilization ratio and available free capacity. Scenarios with high utilization ratio – such as scenarios used for experiments in class  $c_6$  and  $c_7$  – do not have a large amount of free flow table capacity. This means the number of remote switch options in the remote set for the RS-Alloc algorithm is small. In addition, there is a high probability that a remote switch option cannot be used because the number of relocated rules associated with a delegation template is too high (or the free capacity in the remote switch is too low which is the same statement). In result, the number of allocation assignments that can be created for these scenarios is limited to values smaller than the maximum of 50. In most of the extreme cases (class  $c_7$ ), there is only a single

assignment left which is “everything has to be allocated to the backup switch”. It can be concluded that **scenarios with lower flow table utilization ratio and more free capacity require more modeling time.**

The remaining factor for the RS-Alloc algorithm runtime is the time required by the Gurobi solver to calculate the optimal result. The results for RS-Alloc solving time are shown in Fig. 16.5. The figure uses the same structure as before. It is shown by the thicker black line that **Gurobi can solve 99.12% of the samples in less than 128ms.** And in 97% of the cases, the solver can find the optimal result in less than 32ms. The figure also shows that 0.25% of the samples require more than 512ms.

The worst case solving time in this case is 200 seconds which is the pre-configured timeout for RS-Alloc. However, the current setup always uses a fully optimal solution. Because a real system will use anticipated monitoring data, it is uncritical to stop the optimization after a pre-configured gap (towards the optimal solution) and work with non-optimal results if necessary. The timeout can be set to a value of 0.5 seconds to deal with extreme situations. And if the solver cannot provide a result in the specified time, a greedy allocation strategy can be used as a fallback for this time slot.

It is concluded from the above results that the current prototype of the RS-Alloc algorithm can be calculated in less than 200ms in more than 99% of the investigated cases which includes the time for modeling and solving. Note that  $Z_{5000}$  only considers scenarios with up to 15 switches. Results for RS-Alloc with higher switch counts up to 300 are discussed below.

## 16.2 Number of Delegation Templates

An important scalability parameter for DT-Select is the number of delegation templates considered in the optimization problem. The indirect aggregation scheme based on ingress ports introduced in Sec. 9.16 uses a static amount of templates limited by the number of ingress ports. This means the scenarios investigated so far based on  $Z_{100}$  and  $Z_{5000}$  require only a small number of delegation templates ( $< 50$  in most cases). This section investigates modeling and solving times if the number of delegation templates is increased.

### 16.2.1 Experiment Setup

The experiment series used for this section is not based on the scenario generation process. Instead, experiments are created manually to control the number of delegation templates. This is done with a single switch which is sufficient here because DT-Select is executed individually per switch. The experiments are generated with the parameters in Table

16.2 to design scenarios where mitigation is required in almost all time slots. The number of templates is scaled with the number of hosts  $|H|$  attached to the single switch, i.e., the ingress-port based aggregation scheme is still used but with a higher number of ingress ports. Note that the number of required delegation templates in the solution ( $|D_{s,t}^*|$ ) scales with  $|H|$ . This is because less flow rules are associated with a single template if the number of hosts increases while the number of to be relocated flow rules for bottleneck mitigation only depends on the number of host pairs  $n_{\text{pairs}}$ .

Parameter	Description	Values
$ H $	Number of hosts	25, 50, ..., 475, 500
$ S $	Number of switches	1
$n_{\text{seed}}$	Seed	1, 2, ..., 9, 10
$n_{\text{reduction}}$	Capacity reduction factor	10%, 30%, 50%, 70%
$n_{\text{pairs}}$	Number of host pairs	100.000, 200.000
$n_{\text{iat\_scale}}$	Global scale for $T_{\text{iat}}$	250 seconds
$n_{\text{bneck}}$	Temporal bottlenecks	0
$n_{\text{isr}}$	Inter switch ratio	0%
$n_{\text{hs}}$	Number of hotspots	0
$n_{\text{traffic\_scale}}$	Global traffic scale	100%
$n_{\text{lifetime}}$	Minimum rule lifetime	1 second

**Table 16.2:** Experiment parameters for Sec. 16.2

The generated scenarios are similar to scenario  $z_{166890}$  discussed in the case study, see Fig. 13.13. All experiments are performed with Select-CopyFirst and the recommended parameters specified in Sec. A.4.

### 16.2.2 Results

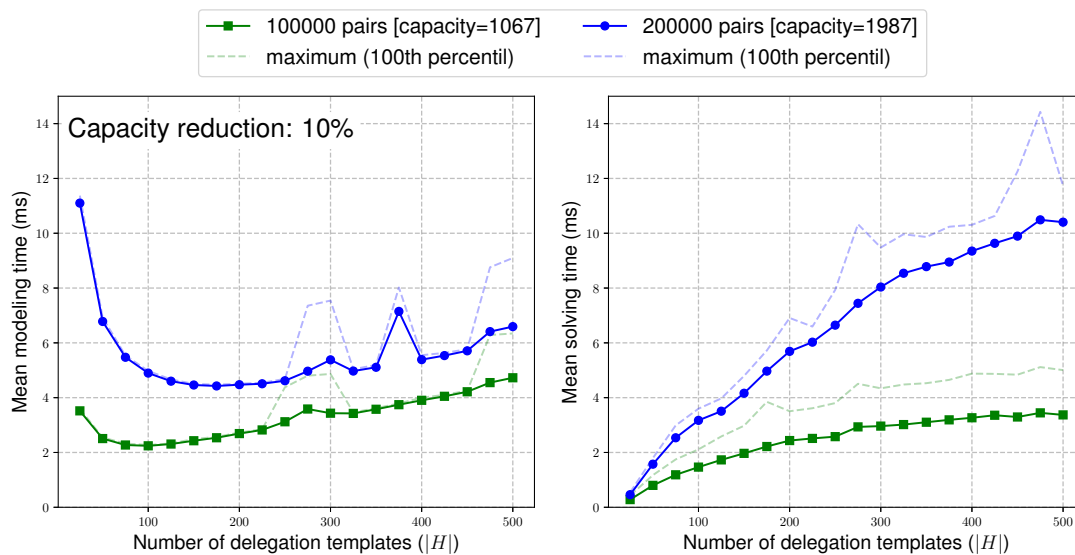
**Findings:** Algorithm runtime increases sub-linearly with the number of considered templates. Select-CopyFirst supports up to 500 delegation templates with modeling and solving times below 20ms in all cases.

The experiment series makes use of four capacity reduction factors between 10% and 70% to change the number of flow rules that need to be relocated. Fig. 16.6 shows the modeling and solving time for a different number of templates and a capacity reduction factor of 10%. The x-axis denotes the number of delegation templates and the y-axis



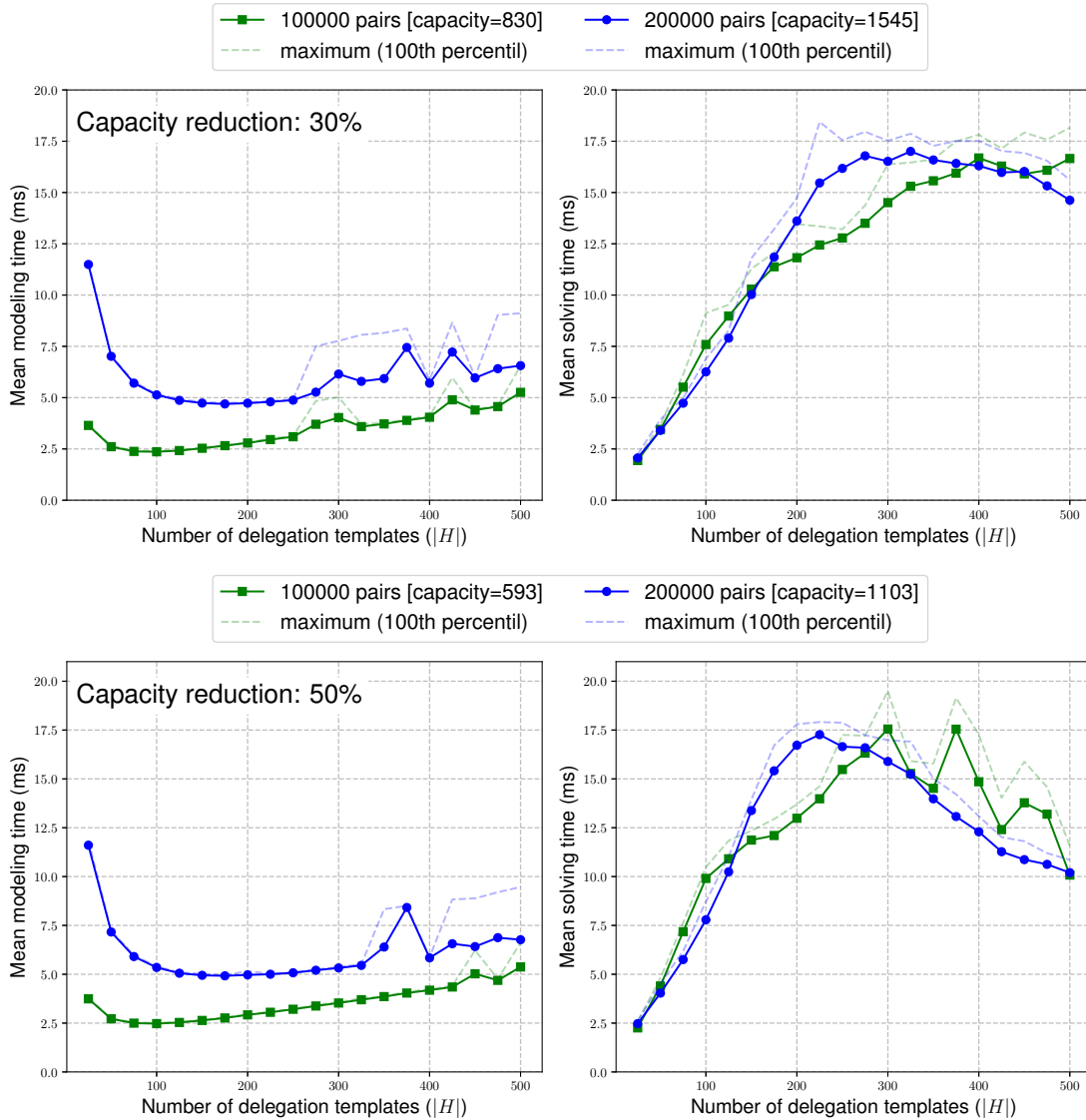
represents the runtime in milliseconds for all DT-Select samples executed with the amount of templates specified on the x-axis. The thick line represents the median (50th percentile), the dashed line the maximum (100th percentile). The two colors represent two different setups: one with  $n_{\text{pairs}} = 100.000$  randomly selected communication pairs (green) and on with  $n_{\text{pairs}} = 200.000$ . These two setups are included because the number of pairs controls the number of rules in the flow table which is important in the following.

Note that there are 10 different experiments for each combination of capacity reduction factor and number of templates (the different  $n_{\text{seed}}$  values). Because each experiment consists of approx. 250 DT-Select samples, each data point in the plot represents approx. 2500 samples. The total amount of considered samples is approx. 50.000 per curve over all 20 different x-values.



**Figure 16.6:** Modeling and solving times of DT-Select for different number of delegation templates (for 10% capacity reduction)

The first important observation for Fig. 16.6 is that – with 10% capacity reduction –, the modeling time never exceeds 12.5ms and the solving time never exceeds 15ms, tested here with up to 500 templates. And very similar results are achieved with higher capacity reduction factors (see figures below). It can be concluded that **neither the solving time nor the modeling time exceeds 20ms in the investigated experiments regardless of the numbers of delegation templates and the capacity reduction factor**. This shows that, from a runtime perspective, DT-Select scales well with the number of considered templates.



**Figure 16.7:** Modeling and solving times of DT-Select for different number of delegation templates and capacity reduction factors (30% in the top, 50% in the bottom)

It can further be seen in the right plot of Fig. 16.6 that the solving time scales sub-linear with the number of templates (decreased slope for higher values). This is expected because the number of templates defines the number of decision variables in the Select-Opt optimization problem. In the left plot, it can be seen that a small amount of templates leads to high modeling times. This is because the flow rules<sup>2</sup> are distributed over the available number of templates. With  $|H|$  templates and  $n_{\text{pairs}}$  flow rules, each template

<sup>2</sup>The number of pairs is equal to the number of flow rules here because the topology consists of a single switch

is associated with approx.  $\frac{n_{\text{pairs}}}{|H|}$  rules over the course of the experiment. Because  $n_{\text{pairs}}$  is static and  $|H|$  is increased from 25 to 500, higher values of  $|H|$  reduce the number of flow rules per template. Because the majority of the computational effort for translation to lambda notation comes from iterating over the flow rule sets in the templates, the modeling time is decreased if the individual templates are associated with less flow rules. It can be seen in Fig. 16.6 and also the following figures that this effect is very strong until the number of templates ( $|H|$ ) reaches 100. At this point in time, the overhead for iterating over the templates is higher than the savings from reduced iterations inside each single template. This is why the modeling time increases from this point on with a sub-linear trend. The outliers in the modeling time plot – e.g., with  $|H| = 300$  – are caused by interference between experiments. This effect is explained in detail in Appendix A.1.2. Also note that the number of different seeds (10) is rather small in this experiment series.

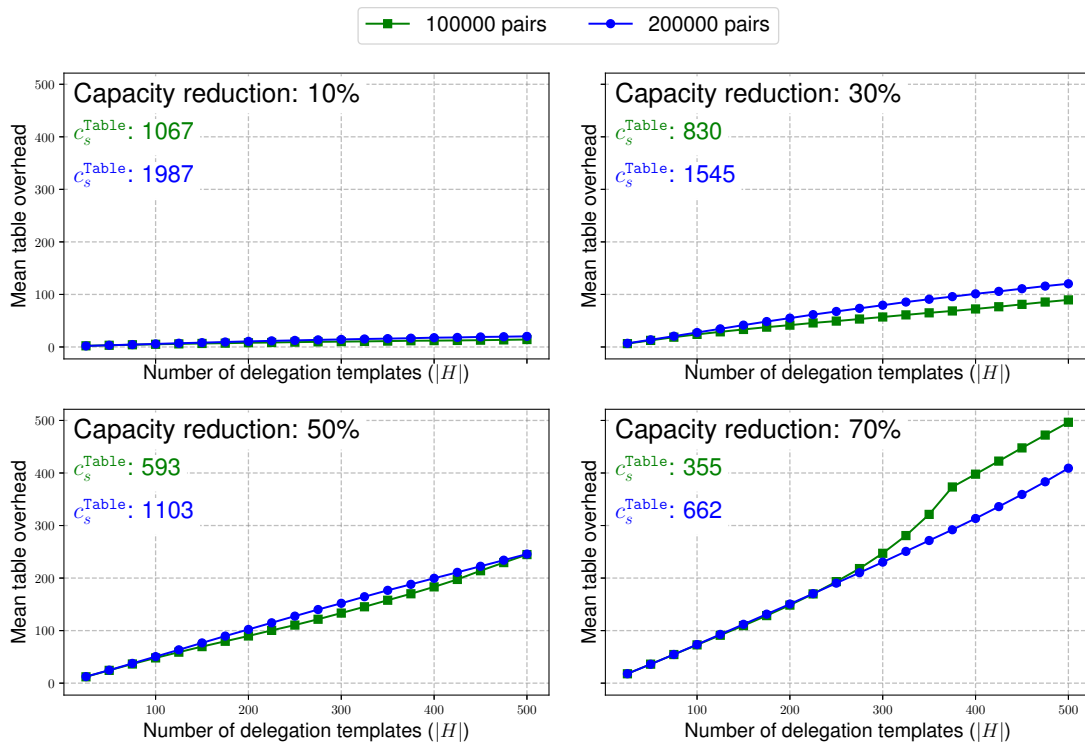
If we compare the results from Fig. 16.6 with 10% capacity reduction and the results from Fig. 16.7 for 30% and 50%, it can be seen that the solving time in the right plot only increases until  $|H|$  reaches a certain value. To explain this effect, it is important to understand that the number of rules to be relocated only depends on the capacity reduction factor and the number of pairs. With 100.000 pairs and a 10% reduction factor, the average capacity of the switch in the experiments is 1067 and 118 rules need to be relocated on average. With 100.000 pairs and a 30% reduction factor, the average capacity is 830 and 355 rules must be relocated.

Note that these numbers do not change for a higher number of templates. Because each template is associated with approx.  $\frac{n_{\text{pairs}}}{|H|}$  flow rules over the course of the experiment, the solver has to select more and more templates to get to the required amount of rules when  $|H|$  increases. Taking this into consideration, it is easy to see that the average number of rules associated with a template  $d$  in one time slot  $t$  is of critical importance. Assume this value is given as  $c_{d,t}$ . A delegation template  $d$  can only be used if  $c_{d,t} \geq 2$ , i.e., there are at least two rules that can be relocated if the template is selected. Otherwise, the overhead added by the aggregation rule will immediately negate the effect of selecting the template. The reduction of solving time observed in Fig. 16.7 can now be explained with two effects:

- (1) Higher values of  $H$  will decrease  $c_{d,t}$  and therefore also the probability that a delegation template has a value  $c_{d,t} \geq 2$ . These templates are not included in the optimization problem in the first place so that the total amount of decision variables will decrease with lower  $H$  which makes the problem simpler and reduces solving time.

- (2) Higher capacity reduction factors will increase the number of rules that need to be relocated and, at the same time, reduce the available capacity of the switch. This puts a hard limit on the maximum number of delegation templates that can be selected by the solver. This makes the problem simpler because the solver is forced to select templates with high  $c_{d,t}$  regardless of the objective function (reduced search space).

The above results indicate that DT-Select can deal with a high number of delegation templates (500) and will automatically reduce the complexity of the optimization problem if only a subset of the templates is usable. However, it is important to keep in mind that there is a hard limit of the number of delegation templates that can be selected<sup>3</sup> in practice. The experiment series defined in this section is a good example for this general limitation.



**Figure 16.8:** Table overhead for different number of templates and capacity reduction factors

<sup>3</sup>Note that it makes a big difference whether a template is “selected” (used in the optimal result) or “considered” (included in the optimization problem as one option). The number of considered templates can be arbitrarily high in theory. The number of selected templates not because of aggregation rule overhead and flow table capacity limitations.

Consider the four plots in Fig. 16.8. The x-axis denotes the number of delegation templates and the y-axis represents the mean table overhead for all DT-Select samples executed with the amount of templates specified on the x-axis. Recall that the table overhead counts the amount of aggregation rules installed in the flow table of the delegation switch, divided by the number of time slots with bottlenecks (Def. 15.1). The four plots represent the different capacity reduction factors used in the experiment series. The two colors represent two setups, one with  $n_{\text{pairs}} = 100.000$  randomly selected communication pairs (green) and one with  $n_{\text{pairs}} = 200.000$ .

It can be seen that the number of used aggregation rules per time slot stays below 20 if the capacity reduction factor is set to 10% (top left). This is a reasonable overhead given that the table capacity  $c_s^{\text{Table}}$  is 1067 rules for the green curve and 1987 for the blue curve. With 30% capacity reduction, the overhead is significantly higher for large values of  $|H|$ . With 200 delegation templates, 40 aggregation rules are required in case of the green curve. With  $|H| = 400$ , this value is already increased to 65 aggregation rules – because  $c_{d,t}$  is reduced and more templates are required. This means that almost 10% of the capacity – 830 in this case – is occupied with aggregation rules. If the capacity reduction factor is increased further, the amount of aggregation rules in the flow table of the delegation switch is also increased.

This can go so far that the number of required aggregation rules exceed the capacity of the flow table. This situation can be observed with the green curve in the bottom right of Fig. 16.8. Because the flow table can store only 355 rules in this example and  $c_{d,t}$  is too low for too many of the templates, the solver can not deal with the situation if  $|H|$  is set to values greater than 250. This means the problem is getting infeasible. In the current prototype, a backup solution is used in such a case that will select additional templates. This backup solution does not consider the capacity of the flow table which explains why the green curve in the bottom right of Fig. 16.8 grows beyond the capacity. The effect can also be observed in the solving time plot for this scenario which is shown for completeness in Fig. 16.9. The additional red curve shows the percentage of the samples that are infeasible.

It is important to mention that the case above is not a realistic scenario. However, it is a good example to demonstrate the conceptual limitations of the approach. In summary, it can be concluded that the Select-CopyFirst algorithm supports hundreds of delegation templates without noticeable increase in algorithm runtime. This could be important, for example, if another aggregation scheme is used where a high number of templates has to be considered. It is further shown in this section that a high number of delegation templates in combination with a small amount of associated rules per template ( $c_{d,t}$ ) leads to more and more table overhead if  $|H|$  is increased and will eventually result in infeasible optimization problems.

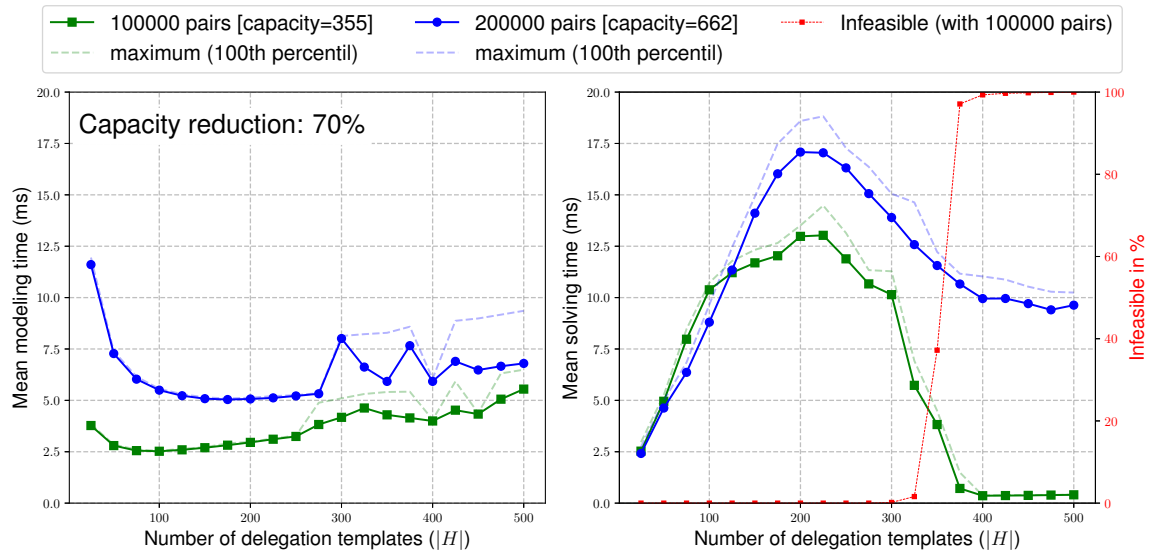


Figure 16.9: Scaling Delegation Templates, 70% capacity reduction

## 16.3 Number of Switches

The scenarios in  $Z_{100}$  and  $Z_{5000}$  represent relatively small networks with up to 15 switches. This section investigates the algorithm runtime of RS-Alloc if the number of switches is increased to higher values (up to 300). Note that DT-Select scales linear with the number of switches by default because delegation template selection is executed individually per switch.

### 16.3.1 Experiment Setup

The experiment series used for this section is not based on the scenario generation process. Instead, experiments are created manually to control the number of switches. Because larger topologies need a larger amount of hosts, the series consists of three “sub-series” where the majority of the parameters is fixed, except for the number of switches, the number hosts and the number of communication pairs. The experiments are generated with the parameters in Table 16.3. The 9 entries in the bottom are the three sub-series.

Parameter	Description	Values
$m$	Topology generation	1,2
$n_{seed}$	Seed	1, 2, ..., 9, 10
$n_{reduction}$	Capacity reduction factor	10%, 30%, 50%, 70%
$n_{iat\_scale}$	Global scale for $T_{iat}$	300 seconds

$n_{\text{bneck}}$	Temporal bottlenecks	1
$n_{\text{bneck\_duration}}$	Bottleneck duration	1
$n_{\text{bneck\_intensity}}$	Bottleneck intensity	20
$n_{\text{isr}}$	Inter switch ratio	70%
$n_{\text{hs}}$	Number of hotspots	0
$n_{\text{traffic\_scale}}$	Global traffic scale	100%
$n_{\text{lifetime}}$	Minimum rule lifetime	3 second
$ S $	Number of switches	10, 15, ..., 95, 100
$ H $	Number of hosts	250
$n_{\text{pairs}}$	Number of host pairs	100.000
$ S $	Number of switches	110, 115, ..., 195, 200
$ H $	Number of hosts	500
$n_{\text{pairs}}$	Number of host pairs	150.000
$ S $	Number of switches	210, 220, ..., 290, 300
$ H $	Number of hosts	1000
$n_{\text{pairs}}$	Number of host pairs	150.000

**Table 16.3:** Experiment parameters for Sec. 16.2

The 11 entries in the top are the same for all three sub-series. The number of switches is scaled from 10 to 300 in steps of 5 or 10. Smaller topologies with up to 100 switches use 250 hosts. Topologies with 100-200 switches use 500 hosts and topologies with more than 200 switches use 1000 hosts. All topologies are created with  $m = 1$  and  $m = 2$  using the Barabasi-Albert model. The number of communication pairs is relatively small (100.000 and 150.000) but this is compensated with a high inter switch ratio. This ensures that 70% of the generated pairs result in flow rules installed over multiple switches. All experiments are performed with Select-CopyFirst and the recommended parameters specified in Sec. A.4.

The experiments were executed on a server with 2 x Intel(R) Xeon(R) Silver 4110 CPUs (32 logical cores, clocked at 2.10 GHz) running 32 experiments in parallel without further management of the resources such as setting processor affinity, i.e., the system was on full load during the evaluation. This can impact the reported runtime values as explained in Sec. A.1.2.

### 16.3.2 Results

**Findings:** Algorithm runtime increases linearly with the number of switches. RS-Alloc supports up to 300 switches with algorithm runtime (modeling + solving) below 5s in all investigated cases using a single CPU core. For smaller and more realistic capacity reduction factors up to 30%, algorithm runtime is below 500ms in all investigated cases.

The experiment series is designed similar to the series used in Sec. 16.2 and makes use of four capacity reduction factors between 10% and 70%. Fig. 16.10 shows the modeling and solving time for a different number of switches and capacity reduction factors. The x-axis denotes the number of switches and the y-axis represents the runtime in milliseconds for all RS-Alloc samples executed with the amount of switches specified on the x-axis. The values for the two plots in the top are given as the median. The values for the two plots in the middle are given as 90th percentiles. And the plots in the bottom show the maximum of the measured values across all samples. The four colored lines represents the four different capacity reduction factors.

It can be seen in Fig. 16.10 that all conducted experiments could be modeled in less than 5 seconds and solved in less than 1 second using a single CPU core – which includes extreme scenarios with 70% capacity reduction. Note that the runtime values are usually better if experiments can not interfere with each other (see Sec. A.1.2). Further note that RS-Alloc is only executed once per time slot and both sub-steps – modeling and solving – can be parallelized if necessary.

It can further be seen that all curves scale linear with the number of switches. In addition, it can be seen that smaller – and more realistic – capacity reduction factors up to 30% can be handled in less than 100ms in 50% of the cases and less than 500ms in 100% of the cases (using one CPU core on a highly loaded system). It is concluded that the RS-Alloc approach can be used in larger networks with (at least) 300 switches.



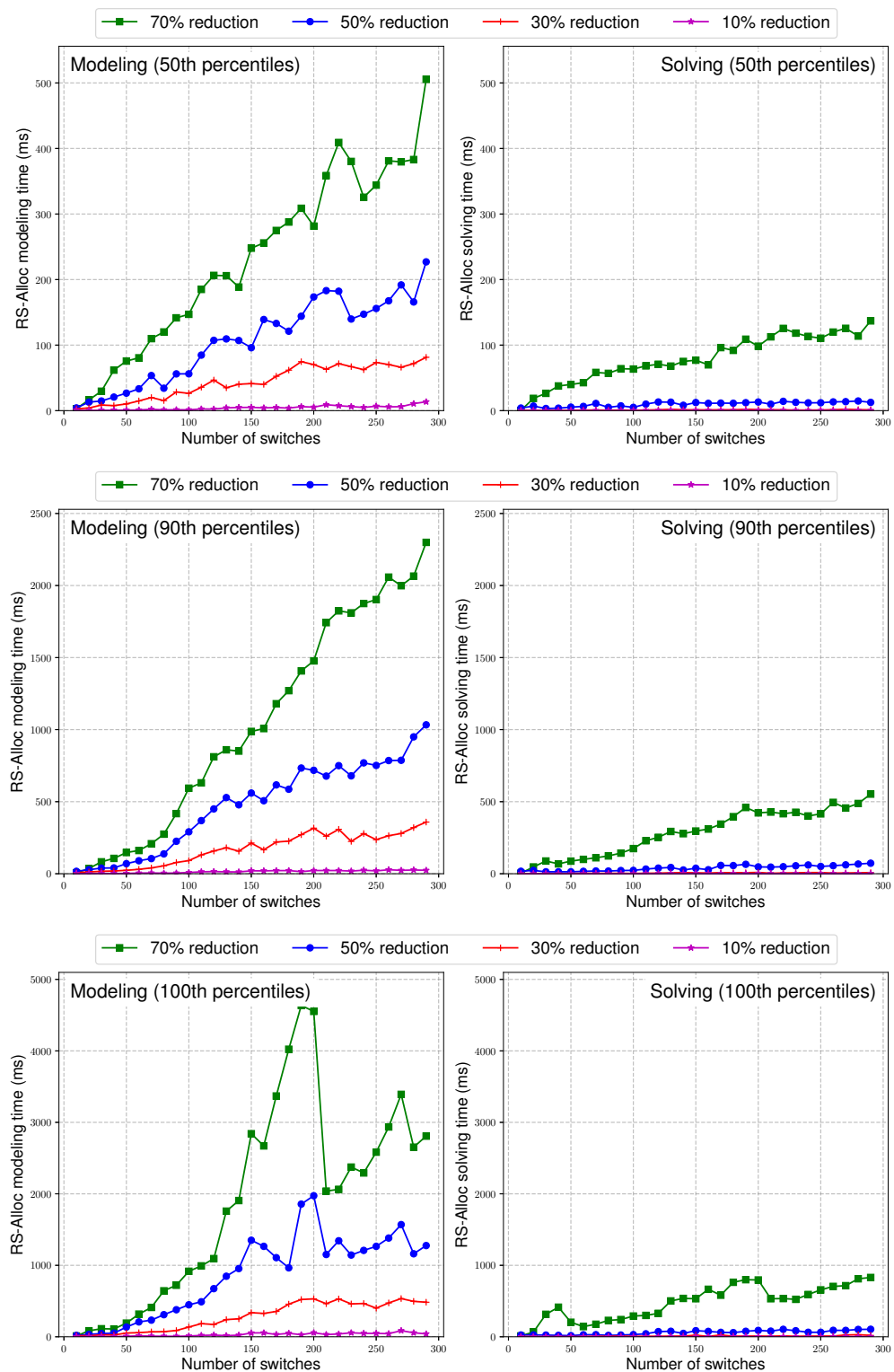


Figure 16.10: Modeling and solving times of RS-Alloc for different number of switches and capacity reduction factors. The values on the y-axis are given in percentiles as specified.



## Part V

# Conclusion and Appendices



# Conclusion and Outlook

---

This thesis introduced **flow delegation**, a novel concept for flow table capacity bottleneck mitigation that can be easily applied to existing software-based networks and enables two powerful new use cases:

- *Bottleneck mitigation with existing hardware*: By using the available spare flow table capacity in the infrastructure, flow delegation can handle situations where the flow table utilization exceeds the maximum capacity – which would normally lead to decreased performance, loss of connectivity, security problems or violation of service agreements. The evaluation shows that flow delegation can deal with flow table utilizations that are **up to 38% above the capacity** by only using available spare resources (median value for 5000 investigated bottleneck scenarios). This means flow delegation can handle bottlenecks with 1380 concurrently installed flow rules when all (!) switches in the infrastructure have a capacity of only 1000 rules.
- *Reduction of operational and capital expenditure*: Network operators can include the mitigation potential of flow delegation into the calculation when new equipment has to be bought and select cheaper and more energy efficient switches with smaller TCAM chips. It is shown that flow delegation allows for a **28% reduction in TCAM size** (median value for 5000 investigated bottleneck scenarios) which can lead to significant savings given that TCAM chips are one of the most expensive components in a modern SDN switch.

In addition to the two above use cases, flow delegation comes with several benefits that cannot be achieved with other existing solutions. The designed system is realized completely in software and can be used without changes to the existing network – in contrast

to approaches that require changes to the infrastructure [Yu+10; Cur+11; Kat+14b] or the network applications [KHK13; KB13; Len+15]. In addition, the operational aspects (i.e., the fact that a flow rule is relocated to another switch) is hidden from the control plane with the help of control message interception. This allows it to simply enable and disable flow delegation on-demand which is well suited for bottlenecks that only occur in parts of the network or within certain time frames. And last but not least, flow delegation has very low requirements in terms of protocol features that must be supported by the controller and the switches: it works with every version of OpenFlow and can be easily adapted to other southbound protocols if necessary.

Flow delegation works as follows: if a bottleneck is detected or anticipated, flow rules from the bottlenecked switch are automatically relocated to a neighboring remote switch with spare capacity. All packets associated with the relocated flow rules are then redirected towards and processed within the remote switch. After processing, the packets are sent back to the bottlenecked switch and forwarded to their original destination. Since all packets are still processed in hardware, the overhead for relocated packets in terms of additional latency is quite small (below 0.1ms). It is shown in the thesis that flow delegation can deal with severe bottleneck situations where the required amount of flow rules exceeds the capacity by more than 50% while only using existing spare capacity. And it is further shown that the price that has to be paid for flow delegation in terms of additional link utilization and control messages is acceptable in the majority of the cases: less than 13 Mbit/s per bottlenecked switch – which is distributed over several links if multiple remote switches are used – and less than 50 additional control messages per second per bottlenecked switch (median values for 5000 investigated bottleneck scenarios).

## 17.1 Summary of Contributions

The thesis makes three main contributions: i) It presents a system design and architecture capable of dealing with the numerous practical challenges associated with flow delegation. ii) It introduces suitable algorithms for flow delegation to efficiently mitigate bottlenecks taking future knowledge and multiple objectives into account. And iii), it studies feasibility, performance, overhead, and scalability of the approach covering various different scenarios. The individual contributions are as follows:

- A modular architecture for flow delegation is presented that consists of five independent building blocks: monitoring system, rule aggregation scheme, delegation algorithm, detour procedure and control message interception. It is discussed in detail how each of these building blocks can be realized.

- Practical feasibility of the designed system is shown with two independent proof-of-concept implementations evaluated in a real (emulated) OpenFlow network.
- A novel rule aggregation concept is introduced to calculate so-called delegation templates independently of the flow rules installed in the flow table. These delegation templates are much easier to handle than the output of comparable approaches such as dependency graph algorithms [Kat+14a] because different delegation templates are conflict-free by design. Note that this is not only useful for flow delegation but could also be applied in the context of software offloading [BP12; Kat+14b; MDS17], flow rule distribution [Yu+10; KHK13] or other related areas.
- A novel flow delegation algorithm is designed to mitigate bottlenecks. The underlying problem is decomposed into two sub-problems (DT-Select, RS-Alloc) which are then modeled with the help of integer linear programming. The thesis presents single-period and multi-period versions for both problems that can deal with multiple and potentially conflicting objectives. The multi-period versions consider future network situations to proactively mitigate anticipated bottlenecks.
- Fast and scalable heuristics for the flow delegation algorithm are developed which operate in the millisecond range ( $< 150ms$ ) on a single CPU core – fast enough to be used in practice. A single commodity server with 32 cores can handle a network with hundreds of switches.
- A comprehensive evaluation is conducted to investigate feasibility, performance, overhead, and scalability of the approach. It is shown that the approach is primarily limited by two factors: the available free flow table capacity and the available free link bandwidth. Other factors such as additional rules required by the approach (e.g., aggregation rules) or additional control messages are uncritical in most cases.

The code for the prototypes, the algorithms and the evaluation is publicly available. All data sets used for the evaluation are also available to support reproducibility.

## 17.2 Perspectives for Future Work

It was already mentioned in [BZ16] that the flow delegation approach is not restricted to flow table capacity bottlenecks<sup>1</sup>. It could also be used, for example, to cope with hardware heterogeneity and missing data plane features. Due to the feature-richness of the OpenFlow specification and diversity in the switch vendor market, not all features are

---

<sup>1</sup>Some ideas in this direction were already investigated, for example in the area multi table processing [Gei+17; Gei+19] (joint work with the University of Würzburg and Nokia Bell Labs) and model driven networking [Lop+18] (joint work with the Federal University of Pernambuco)

automatically available on every switch. Now assume the controller uses a certain feature that is only supported by a subset of the switches. Flow delegation can relocate flow rules that use this feature to the supporting switches. Another area where flow delegation may be applicable is load balancing of switch CPU resources, e.g., for processing packet\_in control messages or monitoring requests which can be expensive for large flow tables. If one of the switches suffers from a high CPU utilization, processing could be delegated to another switch with less CPU utilization.

Another area for future work is combination of flow delegation with other approaches. With support from the controller and/or the network applications, for example, it could be possible to combine flow delegation with solutions for flow table compression such as [LMT10; MLT12] to handle bottlenecks that cannot be mitigated by one approach alone. It could also be a valid strategy to combine flow delegation with software offloading [BP12; Kat+14b; MDS17].

Application of machine learning is also a promising direction for further investigation. Reinforcement learning could be used to control whether delegation templates are selected or not based on the current situation in the network<sup>2</sup>. The expectation is that this may potentially lead to better decisions tailored to the learned behavior in a particular environment. Another idea in the context of machine learning is automatic parameterization of the algorithms. In the current approach, a generic set of parameters is used for all scenarios. Experiments showed, however, that different parameters can lead to different results. It is thus expected that flow delegation would benefit from automatic parameter tuning which is a classical task for machine learning.

Finally, there are several options for optimizations with respect to the five designed building blocks of the flow delegation architecture which may merit further investigation. One is extending the detour procedure and the RS-Alloc algorithm so that flow rules can be relocated to every remote switch with free capacity and not only those that are directly connected to the delegation switch. It could also be possible to develop a pre-processing step similar to the approach presented in Sec. 11.3 to improve the performance of Select-Opt. Another idea is to further investigate generalized control message interception that was here only addressed in the context of flow delegation but has many other applications – for example in the area of verification and security [Khu+13; Kaz+13; Hu+14] or network debugging [Han+12; Wun+11; WBZ16].

---

<sup>2</sup>Some promising steps in this direction were already performed in a master thesis by Jichao He entitled “Flow Delegation with Reinforcement Learning”. This thesis applied Q-learning and Deep Q Networks (DQNs) to the flow delegation problem.



# Appendices



# Parameter Study

---

The developed algorithms need to be parameterized with a look-ahead factor (represents the amount of future time slots to consider), with the number of allocation assignments in case of RS-Alloc and with weights to control the different objectives. These parameters have a significant impact on flow delegation performance and overhead. It is explained in the following, how the default parameters used for all experiments in this thesis were determined.

- Sec. A.1 investigates the impact of look-ahead factor  $L$  on algorithm runtime
- Sec. A.2 investigates the impact of look-ahead factor  $L$  on overhead
- Sec. A.3 investigates the weights for multi-objective optimizations
- Sec. A.4 presents the determined parameters

## A.1 Impact of Look-ahead Factor on Algorithm Runtime

The look-ahead factor  $L$  determines the amount of future time slots considered in the periodic optimization. Higher look-ahead factors increase the amount of coefficients in the optimization problem which leads to higher modeling and solving times but can also reduce the overhead because the algorithms can better anticipate how the situation will develop in the near future and react accordingly. This section will investigate how the runtime of the two ILP-based algorithms scales with factor  $L$ . The subsequent section will then investigate the improvement in terms of reduced overhead achieved by the look-ahead mechanism.

### A.1.1 Experiment Setup

Two different experiment series are designed for the look-ahead analysis, one for DT-Select and one for RS-Alloc. Both are based on scenario set  $Z_{100}$  with random capacity reduction factors. This is done because it is expected that high look-ahead factors will result in potentially high runtimes. This is also the reason why all experiments are aborted after a timeout of 1600 seconds to save evaluation resources. The timeout was chosen so that each optimization period gets at least 4s of processing power from one logical CPU core with some spare time for the evaluation overhead.

	Parameter	Description	Used Values
(for both)	$n_{\text{scenario\_id}}$	Used scenario IDs	All scenario IDs in $Z_{100}$
	$n_{\text{reduction}}$	Capacity reduction factor	Random between 1% and 80%
DT-Select	$n_{\text{dts\_algo}}$	DT-Select algorithm	Select-Opt, Select-CopyFirst
	$L_{\text{dts}}$	Look-ahead factor DT-Select	1, ..., 9
RS-Alloc	$n_{\text{dts\_algo}}$	DT-Select algorithm	Select-CopyFirst
	$L_{\text{dts}}$	Look-ahead factor DT-Select	3
	$L_{\text{rsa}}$	Look-ahead factor RS-Alloc	1, ..., 9
	$n_{\text{assignments}}$	Assignments for RS-Alloc	5, 20, 50, 100, 200

**Table A.1:** Experiment parameters for Sec. A.1

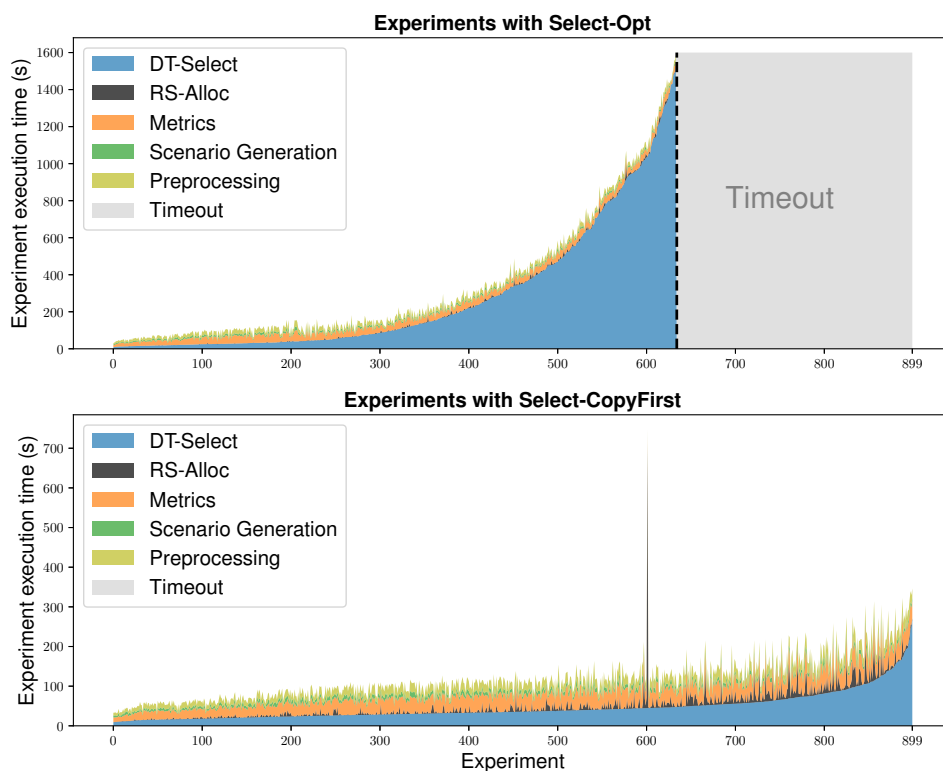
The first experiment series investigates the look-ahead factor for DT-Select. In this case, each scenario in  $Z_{100}$  is executed with look ahead factors between  $L_{\text{dts}} = 1$  to  $L_{\text{dts}} = 9$ . This is done for Select-Opt and Select-CopyFirst which results in 2 x 900 experiments. The second experiment series investigates the look-ahead factor for RS-Alloc. This is done with Select-CopyFirst and  $L_{\text{dts}} = 3$ . The look-ahead factor for RS-Alloc is varied between  $L_{\text{rsa}} = 1$  and  $L_{\text{rsa}} = 9$ . In addition, the maximum number of allocation assignments is varied between 1 and 200. This results in 5 x 900 experiments.

Both series were executed on a server with 2 x Intel(R) Xeon(R) Silver 4110 CPUs (8 physical cores per CPU, 32 logical cores in total, clocked at 2.10 GHz) equipped with 96 GBytes of physical memory<sup>1</sup>.

<sup>1</sup>The high amount of RAM is not required. Maximum RAM utilization of the system was approx. 8 GBytes with 32 experiments running in parallel.

### A.1.2 Reported Runtime Values

For correct interpretation of the reported values for modeling and solving time in this section, it is necessary to understand that 32 experiments were executed in parallel without further management of the resources such as setting processor affinity. The system used for evaluation was on full load during the experiment. This is important here because the experiments did not only run DT-Select and RS-Alloc but also various other tasks such as pre-processing, scenario generation and calculation of metrics. Fig. A.1 shows the DT-Select experiment series with additional details regarding the executed tasks.



**Figure A.1:** Experiment execution times for DT-Select look-ahead analysis

The top shows all experiments that use Select-Opt as DT-Select algorithm and the bottom shows all experiments that use Select-CopyFirst. There are 900 experiments for each algorithm, 9 for each scenario in  $Z_{100}$ . The x-axis shows the experiment index between 0 and 899 sorted by total time used for calculating DT-Select. The y-axis shows the total experiment execution time in seconds. The different colors represent different tasks in the evaluation environment. Note that the timings in the tasks are aggregated, i.e.,

the blue area contains all individual runs of DT-Select for the experiment which can be hundreds or thousands.

It can be seen from the figure that, in case of Select-CopyFirst, a considerable amount of the execution time is used for pre-processing (setup of the evaluation environment) and calculating metrics for the result set. These steps depend heavily on I/O and can slow down other processes which unfortunately also includes the Gurobi solver. Because the different processes are not coordinated in any way, the measured execution time for DT-Select and RS-Alloc can be increased due to interference with other processes running on the server.

There are two additional observations. First, it can be seen in Fig. A.1 that the majority of the time is used for DT-Select (blue area). This is expected because delegation template selection is executed for every switch. The effort for RS-Alloc (black area), on the other hand, is almost negligible with respect to overall resource consumption. The small black spike in the bottom plot at the 600 second mark represents either an unknown error or an extreme case of interference<sup>2</sup>. The second observation is that Select-Opt can only calculate 634 out of 900 experiments without running into the 1600 second timeout. This is because the number of assignments grows exponentially with  $L$  which cannot be solved in a reasonable amount of time. A single experiment with  $L = 9$  and Select-Opt can take more than 24h hours to be modeled and solved. It is important to understand that the experiments with timeout are missing in the results below. This only affects Select-Opt (see Fig. A.1) and means that the reported runtime results would be much worse if the experiments with timeout could be included. Because the goal is to show that Select-Opt does *not* scale and the results for Select-CopyFirst are not affected, this is uncritical.

### A.1.3 Results for DT-Select

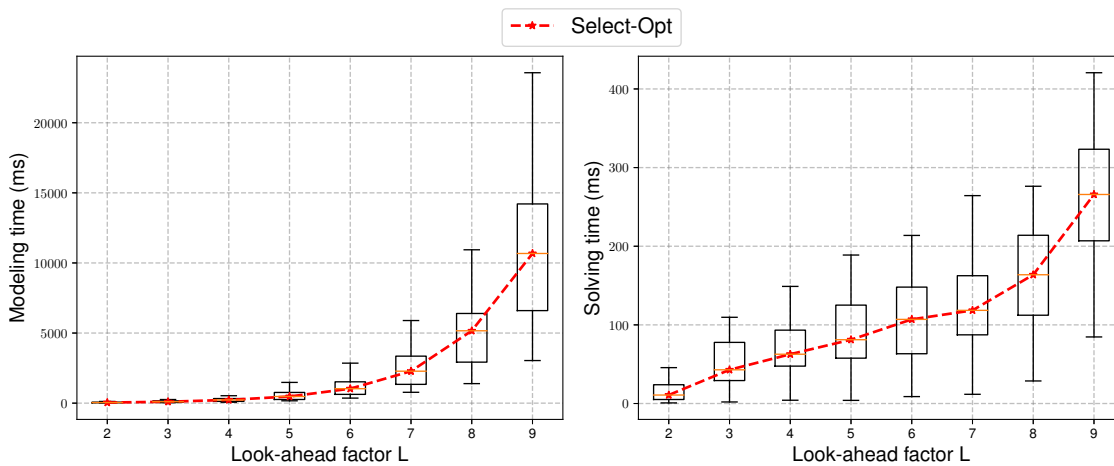
**Findings:** The runtime (= sum of modeling and solving time) of Select-CopyFirst scales linear with  $L$  and stays below 0.5 seconds for  $L < 18$ . The solving time is constant while the modeling time grows linearly for larger  $L$ . For Select-Opt, the modeling time grows exponentially with  $L$ , i.e., this approach should not be used in practice, at least not with look-ahead factors  $L > 3$ .

To analyze the impact of the look-ahead factor, the modeling and solving time for each optimization period is measured in milliseconds. Solving time is the time spent inside the

<sup>2</sup>The experiment with the spike is based on scenario ID 392105 with look-ahead factor  $L = 6$ . This experiment was executed again multiple times after the series where the RS-Alloc part was always measured as approx. 7 seconds instead of 660 seconds.

Gurobi solver. Modeling time is the time required to prepare the model for the solver, i.e., primarily the translation of the scenario data into lambda notation. Time to calculate the metrics and other tasks of the evaluation environment are not included in the results presented below. One optimization period includes the last time slot and up to  $L$  future time slots.

The results for Select-Opt are presented in Fig. A.2 and the results for Select-CopyFirst are presented in Fig. A.3. The x-axis denotes the look-ahead factor  $L$ . The y-axis denotes the modeling or solving time for one optimization period using the look-ahead factor specified as x-value. The data for each look-ahead factor consists all samples of the executed algorithm. It is displayed in box plot form showing minimum, maximum, median as well as 25th and 75th percentiles.



**Figure A.2:** *Select-Opt modeling and solving times for different look-ahead factors*

The runtime advantage of Select-CopyFirst in the second figure over Select-Opt in the first figure is clearly visible: when looking at higher look-ahead factors, the difference is up to two orders of magnitude for the modeling time and up to one order of magnitude for the solving time. The differences for lower factors are shown in the table below. It contains the median values for  $L = 2$  to  $L = 6$  in milliseconds for modeling and solving time of both algorithms, i.e., for 50% of the investigated experiments, the measured times are lower than the value specified in the table.

		L=2	L=3	L=4	L=5	L = 6
Modeling Time	Select-Opt	40.64	101.94	230.00	474.29	1031.42
	Select-CopyFirst	21.34	31.95	46.92	55.39	70.09

Solving Time	Select-Opt	10.78	42.86	62.67	81.11	106.98
	Select-CopyFirst	2.13	1.87	2.10	1.83	1.87

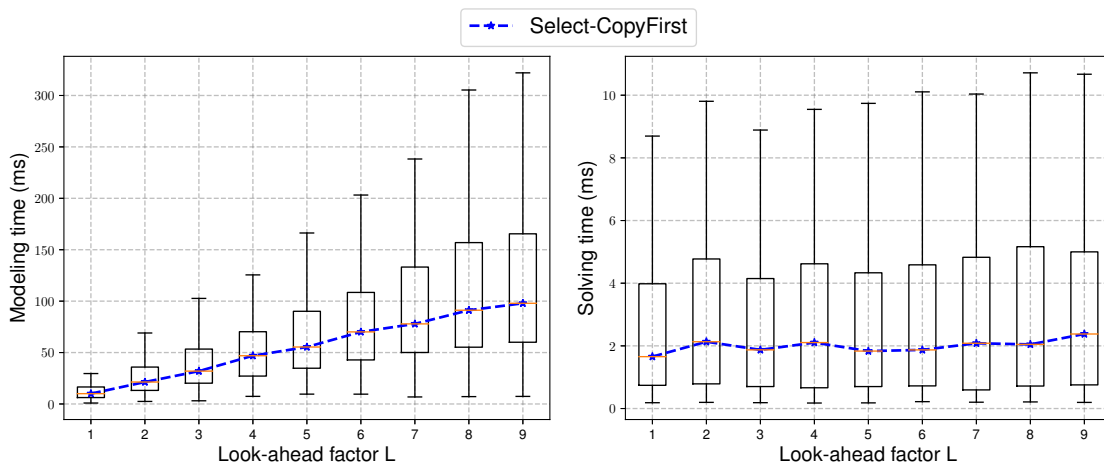
In 50% of the experiments, **modeling for Select-CopyFirst is 2 to 14 times faster than modeling with Select-Opt** if the look-ahead factor is small<sup>3</sup>: 21.34ms with Select-CopyFirst vs 40.46ms with Select-Opt in case of  $L = 2$  (factor 1.9) and 70.09ms vs 1031.42ms in case of  $L = 6$  (factor 14.7). Solving of the created linear program is faster by a factor of 20 with  $L = 2$  – 2.13ms with Select-CopyFirst vs 42.86ms with Select-Opt – up to a factor of 57 with  $L = 6$  (1.87ms vs 106.98ms). It can be seen, however, that both algorithms are primarily limited by the modeling effort and not the solving effort. Fig. A.2 shows that the modeling time for Select-Opt grows exponentially with  $L$  while the modeling time for Select-CopyFirst in Fig. A.3 grows linearly with  $L$ . The trend for the solving time is approx. linear in case of Select-Opt and constant in case of Select-CopyFirst. Fig. A.4 shows how the trend continues with larger values of  $L$  for Select-CopyFirst. Larger values of  $L = 9$  for Select-Opt can not be computed in reasonable time.

The above results limit the algorithms and look-ahead factors that can be used in practice. Select-Opt is conceptually limited by the modeling time. Because the input size grows exponentially, it is difficult to optimize this step. And even with  $L = 4$ , the modeling time in the current prototype is already at 230ms in 50% of the experiments. The 99th percentile with  $L = 4$  is at 815ms. Further recall that these are only the values for experiments that did not run into a timeout. There are also scenarios with potentially much higher modeling times not included here. In result, **Select-Opt is too expensive in terms of the required modeling time** given that this algorithm is executed individually for each switch once per optimization period – even for very small look-ahead factors such as  $L = 3$  or  $L = 4$ . This is the main reason why the Select-CopyFirst heuristic was designed in the first place.

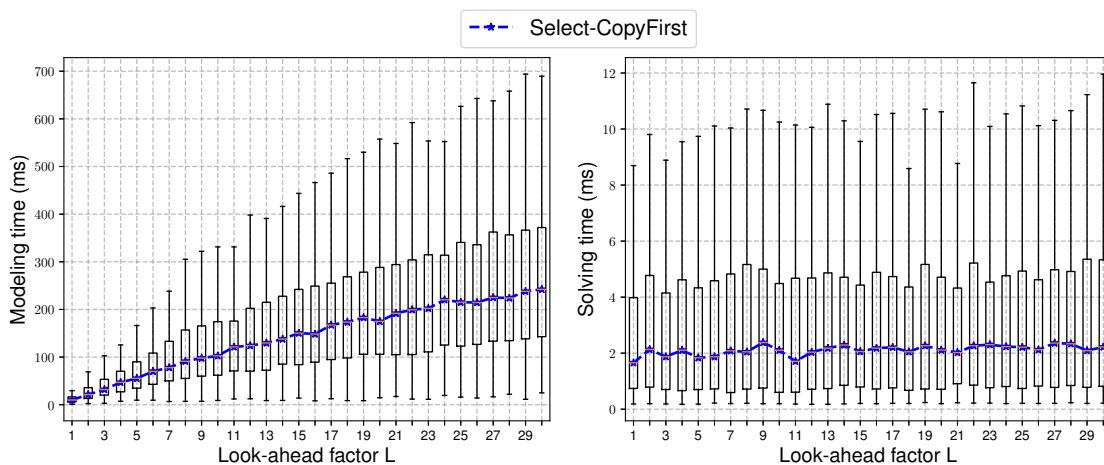
In case of Select-CopyFirst, higher look-ahead factors can be used with a linear increase of the modeling time. The solving time does not increase for larger  $L$  – which is expected given that the heuristic makes only a decision for the first time slot and the number decision variables in the optimization problem stays the same for all  $L$ . The number of coefficients in the optimization problem, however, does scale with  $L$  so it is expected that there is a small linear trend that is not visible for  $L = 1$  to  $L = 30$  (larger values could not be tested with meaningful results because of the limited number of time slots in the experiments). It can be concluded that **Select-CopyFirst has no immanent runtime limitation** and the selection of a suitable look-ahead factor mainly depends on

<sup>3</sup>Note that high look-ahead factors  $L > 6$  also require a mechanism that can predict  $L$  steps into the future





**Figure A.3:** *Select-CopyFirst* modeling and solving times for different look-ahead factors



**Figure A.4:** *Select-CopyFirst* modeling and solving times for  $L = 1$  to  $L = 30$

the available processing resources, the optimizations that can be realized with respect to the modeling time and – most important – the expected benefits or drawbacks from using higher look-ahead factors. It is shown in Sec. A.2 that  $L$  for *Select-CopyFirst* is limited primarily by the fact that the overhead is increased if  $L$  is set to values above 6.

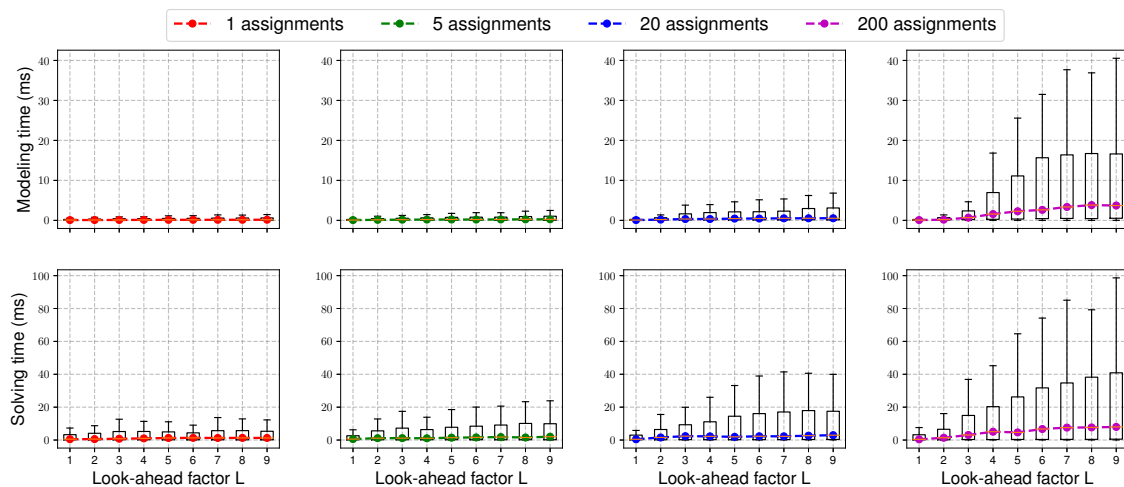
Closing remark: Fig. A.2 does not specify values for  $L = 1$ . This is because all experiments with *Select-Opt* and  $L = 1$  result in an infeasible optimization problem. This is not a conceptual limitation and could be optimized. However, this was not done here due to

time constraints (requires a complete re-design of the history mechanism) and because  $L = 1$  represents flow delegation without look-ahead which is not the focus of this work.

### A.1.4 Results for RS-Alloc

**Findings:** From a runtime scalability perspective, RS-Alloc can support all tested look-ahead factors up to  $L = 9$  with up to 200 allocation assignments. Modeling and solving time scale primarily with the number of allocation assignments in a linear way.

The look-ahead factor used with RS-Alloc can be determined independently from the look-ahead factor used with DT-Select, i.e., the two factors are not automatically restricted to the lower one. And because the RS-Alloc look-ahead factor is directly linked to the size of the allocation assignments, these two aspects are investigated here together. Similar as before, the modeling and solving time for each optimization period is measured in milliseconds. Solving time is again the time spent inside the Gurobi solver and modeling time is the time required to prepare the model for the solver. One optimization period includes the last time slot and up to  $L$  future time slots.



**Figure A.5:** RS-Alloc modeling and solving time for different look-ahead values and number of assignments.

The results obtained with the second experiment series from Sec. A.1.1 are shown in Fig. A.5. The plots are arranged in four columns, each of which represent a maximum number of allocation assignments (different colors). The first row shows modeling times, the second row solving times. The two axis in the individual plots are designed in the

same way as before. The x-axis denotes the look-ahead factor  $L$ . The y-axis denotes the modeling or solving time for one optimization period using the look-ahead factor specified as x-value. The data for each look-ahead factor consists all samples of the executed algorithm. It is displayed in box plot form showing minimum, maximum, median as well as 25th and 75th percentiles.

It can be seen that RS-Alloc is faster and scales better with  $L$  than DT-Select with respect to both, modeling and solving times. All combinations of look-ahead factors and number of assignments stay below 50ms in 50% of the experiments and below 400ms in 100% of the experiments (the outliers are not shown here because of readability). The impact of the used number of allocation assignments is much higher than the impact of using higher look-ahead factors. In conclusion: selection of the two aforementioned parameters is not restricted by runtime scalability and can be done primarily with respect to the achieved overhead reduction (next section).

## A.2 Impact of Look-ahead Factor on Overhead

The previous section did analyze the impact of the look-ahead factor on algorithm runtime. This section will now investigate the impact of higher look-ahead factors on overhead caused by DT-Select and RS-Alloc.

### A.2.1 Experiment Setup

The results in this section are based on the experiment series described in Sec. A.1.1. The parameters are listed in Table A.5. No further changes are required.

### A.2.2 Results for DT-Select

**Findings:** The look-ahead factor for Select-CopyFirst should be set to values between  $L = 3$  and  $L = 5$ . These factors lead to feasible problems in all conducted experiments and show a good trade-off between runtime and overhead:  $L = 3$  is best for runtime,  $L = 5$  is best for control overhead. In this thesis,  $L = 3$  is used as the default value. Values larger than  $L = 6$  will only increase overhead as well as runtime and should not be used.

Before the overhead for different look-ahead factors is investigated, it is important to mention that small values below  $L = 3$  can lead to infeasible optimization problems. This effect is explained in Sec. 10.2.6 and occurs when a bottleneck can only be mitigated if delegation templates are selected anticipatory. Experiments with  $Z_{5000}$  show that this

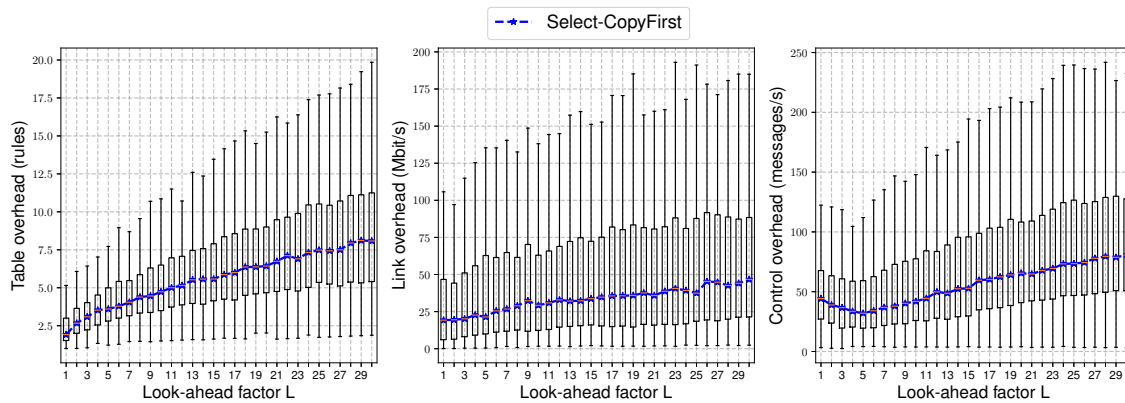
effect occurs in 201 out of 8940 bottlenecked switches with  $L = 1$ . There are no cases with  $L > 1$  in  $Z_{5000}$ , but the effect also occurs in experiments with  $L = 2$  if the capacity reduction factor is set to high values of  $> 80\%$ . The effect never occurred with  $L = 3$  in any of the conducted experiments. It is therefore recommended to use at least a look-ahead factor of  $L = 3$ .

	Description	Formula
<b>Table overhead</b> (Def. 15.1)	Represents the average value of required aggregation rules per time slot. Calculated as the amount of aggregation rules installed in the flow table of the delegation switch divided by the number of time slots with bottlenecks.	$\frac{1}{ T^{\text{Bneck}} } * \sum_{t \in T^{\text{Bneck}}}  D_{s,t}^* $
<b>Link overhead</b> (Def. 15.2)	Represents the average bandwidth required in Mbits/s between a delegation switch and one or more remote switches per time slot. Calculated as the bits relocated by all installed aggregation rules, divided by the number of time slots with bottlenecks.	$\frac{1}{ T^{\text{Bneck}} } * \sum_{t \in T^{\text{Bneck}}} \sum_{d \in D_{s,t}^*} w_{d,t}^{\text{Link}}$
<b>Control overhead</b> (Def. 15.3)	Represents the average amount of additional control messages required per time slot to realize flow delegation. Calculated the additional control messages sent by the flow delegation system, divided by the number of time slots with bottlenecks.	$\frac{1}{ T^{\text{Bneck}} } * \sum_{t \in T^{\text{Bneck}}} \sum_{d \in D_{s,t}^*} w_{d,t}^{\text{Ctrl}}$

**Table A.3:** DT-Select overhead definitions

Further note that the remainder of this section only investigates the results for Select-CopyFirst because it was shown above in Sec. A.1 that Select-Opt is too expensive in terms of required modeling and solving time. The overhead results for different  $L$  are shown in Fig. A.6. The three columns represent the three kinds of overhead introduced for DT-Select in Chapter 15. The used overhead definitions are summarized in Table A.3. The x-axis in the individual plot represents the selected look-ahead factor  $L$ . The y-axis represents the calculated overhead for 900 experiments and is presented in box plot form.

It can be seen in Fig. A.6 that **higher look-ahead factors result in higher table overhead and higher link overhead**. This observation is directly linked to the design of the heuristic. Recall that Select-CopyFirst only makes a decision for the first time slot and “copies” this decision to all future time slots. Because future time slots are still considered



**Figure A.6:** Overhead with *Select-CopyFirst* and different look-ahead values

as capacity constraints, delegation templates are selected earlier as necessary if  $L$  is high. Assume a situation where the flow table utilization is increased by 100 rules per time slot. Further assume the flow table capacity of the switch is 550 rules. The following table shows when the approach has to start the mitigation (i.e., select the first template) based on the applied look-ahead factor  $L$ , denoted here with “x”.

Anticipated flow table utilization	$t_1$	$t_2$	$t_3$	$t_4$	$t_5$	$t_6$
	100	200	300	400	500	600
$L=1$	0	0	0	0	x	x
$L=2$	0	0	0	x	x	x
$L=3$	0	0	x	x	x	x
$L=4$	0	x	x	x	x	x
$L=5$	x	x	x	x	x	x

If the look ahead is set to  $L = 1$ , the first template has to be selected in  $t_5$  because the bottleneck in  $t_6$  would otherwise not be mitigated. With  $L = 2$ , the first template has to be selected in  $t_4$ . Not necessarily because it is not sufficient to start the mitigation in  $t_5$  but because the utilization constraints now include time slot  $t_6$ . This pattern continues. With  $L = 5$ , the algorithm has to select the first template in  $t_1$ . This is the fundamental difference in flexibility between *Select-Opt* and *Select-CopyFirst*. Compare the situation for the two algorithms with  $L = 5$  in time slot  $t_1$ :

- Select-Opt can easily select assignments in such a way that no delegation template is selected in  $t_1$  to  $t_4$ . This is possible because the assignments consists of all possible selections for all considered time slots.
- Select-CopyFirst does not have this flexibility. It HAS to come up with a solution in  $t_1$  so that the capacity for all considered  $L$  future time slots is below 550 rules. Because the decision is copied from the first time slot, the algorithm is forced to select at least one delegation template in  $t_1$  – it can not “delay” the selection for a couple of time slots to save overhead.

It is easy to follow that this “earlier selection” leads to more selected delegation templates over time which means the table overhead is increased. This effect obviously gets stronger if  $L$  is higher which is clearly visible in the leftmost plot in Fig. A.6. For  $L = 20$ , Select-CopyFirst will start the mitigation 20 time slots before the actual bottleneck. The same effect also explains why the link overhead is increased (second plot in the figure).

Early selection of templates, however, is not necessarily a bad thing. One positive side effect is that Select-CopyFirst – similar to Select-Greedy – tends to stick with a once selected delegation template without loosing the flexibility to change from one set of selected templates to another one. This is the reason why Select-CopyFirst is strong in terms of control overhead. In some scenarios, Select-CopyFirst can even achieve a lower control overhead than Select-Opt. To understand this effect, it is important to understand where the control overhead comes from:

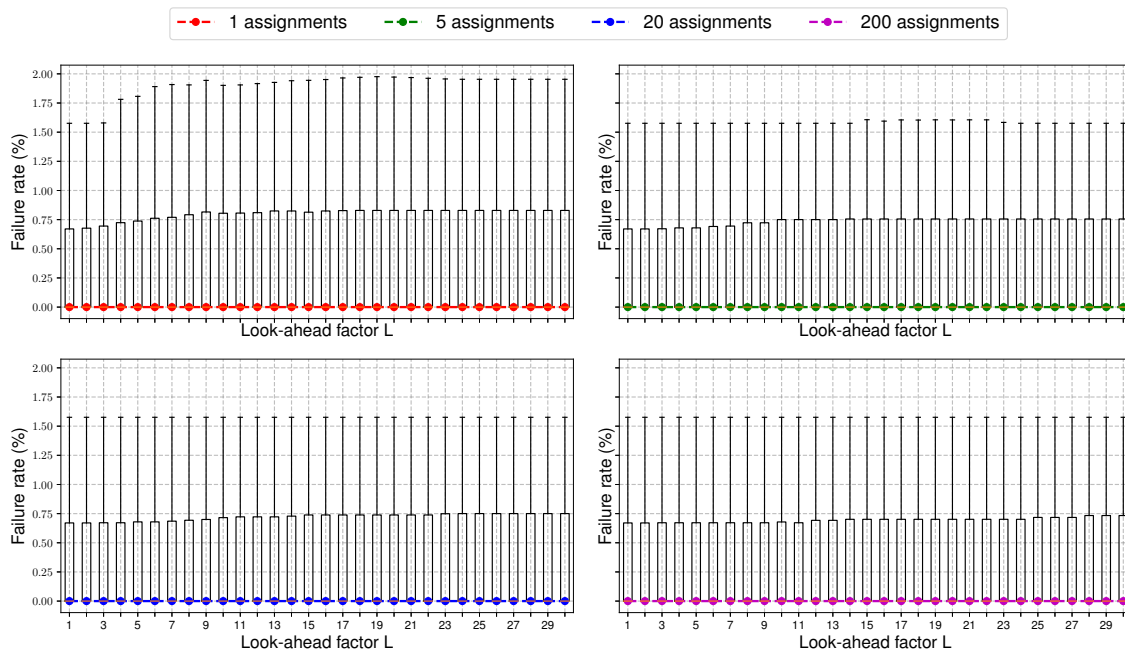
- First recall that, if a delegation template is unselected, the relocated rules associated with this template have to be moved back from the remote switch to the delegation switch which requires control messages and thus contributes to the control overhead.
- Further recall that only the rules after installation of the aggregation rule are relocated. If a delegation template is selected in multiple consecutive time slots, the “new” flow rules are automatically relocated as well and the amount of relocated rules associated with this template grows. Consequentially, the control overhead is higher if such a “grown” template is unselected because more rules need to be moved back to the delegation switch.

Select-CopyFirst frequently changes the selection if  $L$  is set to small values such as 1 and 2 which leads to high control overhead. At the same time, if  $L$  is set too high ( $> 6$ ), the number of relocated rules associated with the selected “grown” templates also causes high control overhead if these templates are unselected (which is inevitable at some point in time). In conclusion, **the best results with Select-CopyFirst in terms of control overhead are achieved for look-ahead factors between  $L=3$  and  $L=6$** . This effect can be observed in the rightmost plot in Fig. A.6

## A.2.3 Results for RS-Alloc

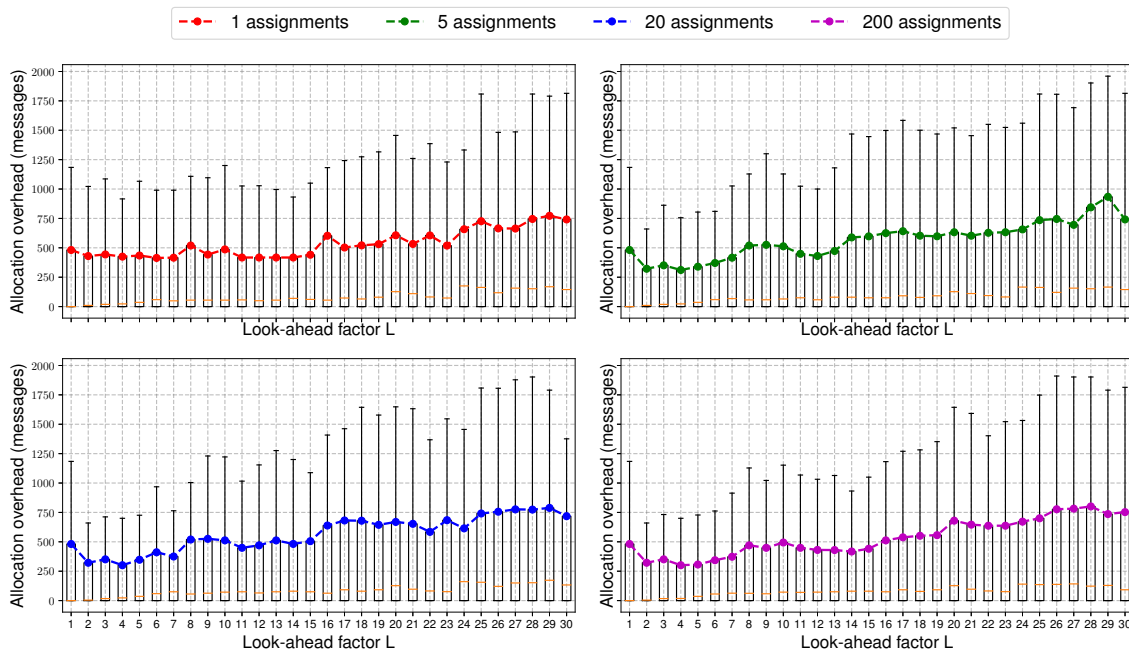
**Findings:** The look-ahead factor for RS-Alloc should be set to values between  $L = 2$  and  $L = 7$ . Higher values result in increased allocation overhead. The number of allocation assignments should be set to a value of at least  $n_{\text{assignments}} = 5$ . In this thesis,  $L = 3$  and  $n_{\text{assignments}} = 50$  are used as default values.

This section investigates the impact of look-ahead factor  $L$  and number of allocation assignments  $n_{\text{assignments}}$  on failure rate and allocation overhead. The failure rate is introduced in Def. 12.4 and represents the amount of flow rules allocated to the backup switch over the course of an experiment. And the allocation overhead represents additional control messages that are required because RS-Alloc changed the allocation of a delegation template to another remote switch or to the backup switch.



**Figure A.7:** Impact on failure rate

The failure rate for different look-ahead factors and number of assignments is shown in Fig. A.7. It can be seen that the median is at 0% for all combinations of look-ahead factors and number of allocation assignments. If a single assignment is used, the failure rates are higher in general. Note that a difference of 0.3% in the failure rate makes a big difference in practice and  $n_{\text{assignments}} = 1$  should not be used. There is no noticeable difference between the remaining combinations, however, at least not with respect to the failure rate.



**Figure A.8:** *Impact on allocation overhead*

The situation for the allocation overhead is different. The results in Fig. A.8 show that smaller look-ahead factors result in lower allocation overhead. The lowest overhead values are achieved with look-ahead factors between 2 and 7. Because the results for  $n_{\text{assignments}} = 5$  (top right),  $n_{\text{assignments}} = 20$  (bottom left) and  $n_{\text{assignments}} = 200$  (bottom right) are again very similar, it can be concluded that RS-Alloc achieves the best results if the look-ahead factor is between 2 and 7 and the number of assignments is at least 5.

### A.3 Weights for Multi-Objective Optimization

DT-Select and RS-Alloc are defined as multi-objective optimization problems and the individual objectives need to be parameterized with weights. The weights balance the different overhead cost coefficients against each other, i.e., specify, for example, how important it is to reduce the number of relocated bits compared to the additional number of control messages. The weights for DT-Select are specified as  $\omega_{\text{DTS}}$  and the weights for RS-Alloc are specified as  $\omega_{\text{RSA}}$ .

This section presents a set of ready-to-use default weights that achieve a reasonable trade-off between the different kinds of overhead if no further assumptions are made with respect to the network scenario. The weights are tuned using scenario set  $Z_{100}$  and work well in all scenarios investigated in this thesis.



### A.3.1 Experiment Setup

Two experiment series are designed for the investigation of the weights, one for DT-Select and one for RS-Alloc. Both are based on scenario set  $Z_{100}$  with random capacity reduction factors. The first experiment series investigates the impact of the three weights for DT-Select. In this case, each scenario in  $Z_{100}$  is executed with all combinations of  $\omega_{DTS}^{Table}$ ,  $\omega_{DTS}^{Link}$ , and  $\omega_{DTS}^{Ctrl}$  using values from  $\omega_{DTS} = 0$  to  $\omega_{DTS} = 8$ . This results in  $9 \times 9 \times 9 = 729$  combinations per experiment and  $729 \times 100 = 72.900$  experiments in total.

	Parameter	Description	Used Values
used for both series	$n_{scenario\_id}$	Used scenario IDs	All scenario IDs in $Z_{100}$
	$n_{reduction}$	Capacity reduction factor	Random between 1% and 80%
	$n_{dts\_algo}$	DT-Select algorithm	Select-CopyFirst
	$L_{dts}$	Look-ahead for DT-Select	3
	$L_{rsa}$	Look-ahead for RS-Alloc	3
	$n_{assignments}$	Assignments for RS-Alloc	50
DT-Select (1st series)	$\omega_{DTS}^{Table}$	Weight for DT-Select	0, 1, ..., 7, 8
	$\omega_{DTS}^{Link}$	Weight for DT-Select	0, 1, ..., 7, 8
	$\omega_{DTS}^{Ctrl}$	Weight for DT-Select	0, 1, ..., 7, 8
RS-Alloc (2nd series)	$\omega_{DTS}^{Table}$	Weight for DT-Select	6
	$\omega_{DTS}^{Link}$	Weight for DT-Select	2
	$\omega_{DTS}^{Ctrl}$	Weight for DT-Select	1
	$\omega_{RSA}^{Table}$	Weight for RS-Alloc	0, 1, ..., 7, 8
	$\omega_{RSA}^{Link}$	Weight for RS-Alloc	0, 1, ..., 7, 8
	$\omega_{RSA}^{Ctrl}$	Weight for RS-Alloc	0, 1, ..., 7, 8

**Table A.5:** Experiment parameters for Sec. A.1

The second experiment series investigates the weights for RS-Alloc. The setup is identical to the first series except that the  $\omega_{RSA}$  weights are varied, i.e., the series also consists of 72.900 experiment. The remaining parameters are set according to the finding from previous sections. All experiments use the Select-CopyFirst algorithm with look-ahead factor  $L_{dts} = 3$ . The look-ahead factor for RS-Alloc is also set to  $L_{rsa} = 3$  and the maximum number of allocation assignments is set to 50.

### A.3.2 Results for DT-Select

**Findings:** The general recommendation for the DT-Select weights is:  $\omega_{\text{DTS}}^{\text{Table}} = 6$ ,  $\omega_{\text{DTS}}^{\text{Link}} = 2$ , and  $\omega_{\text{DTS}}^{\text{Ctrl}} = 1$ . In other words: table and link overhead coefficients should be used with at least a 3:1 ratio and link overhead and control overhead coefficients should be used with at least a 2:1 ratio.

This section investigates the DT-Select overhead if different  $\omega_{\text{DTS}}$ -weights are applied. As a first step, normalized overhead definitions are required because the raw overhead values can be several orders of magnitude apart. This makes it difficult to systematically test the different combinations. Table A.6 explains how the three kinds of overhead for DT-Select from Chapter 15 are normalized. This normalization ensures that all three kinds of overhead are always specified as a percentage value between 0% and 100%.

	Description	Formula
<b>Normalized table overhead</b>	Similar as table overhead in Def. 15.1 but divided by a normalization factor that represents the maximum amount of possible aggregation rules	$\left( \frac{\sum_{t \in T^{\text{Bneck}}} D_{s,t}^*}{\sum_{t \in T^{\text{Bneck}}}  D_{s,t} } \right) * 100$
<b>Normalized Link overhead</b>	Similar as link overhead in Def. 15.2 but divided by a normalization factor that represents the total amount of bits processed over the course of the experiment.	$\left( \frac{\sum_{t \in T^{\text{Bneck}}} \sum_{d \in D_{s,t}^*} w_{d,t}^{\text{Link}}}{\sum_{t \in T^{\text{Bneck}}} \sum_{f \in F_{s,t}} \delta_{f,t}} \right) * 100$
<b>Normalized Control overhead</b>	Similar as control overhead in Def. 15.3 but divided by a normalization factor that represents the amount of install control messages required without flow delegation over the course of the experiment.	$\left( \frac{\sum_{t \in T^{\text{Bneck}}} \sum_{d \in D_{s,t}^*} w_{d,t}^{\text{Ctrl}}}{ F_s } \right) * 100$

**Table A.6:** Normalized overhead definitions for DT-Select

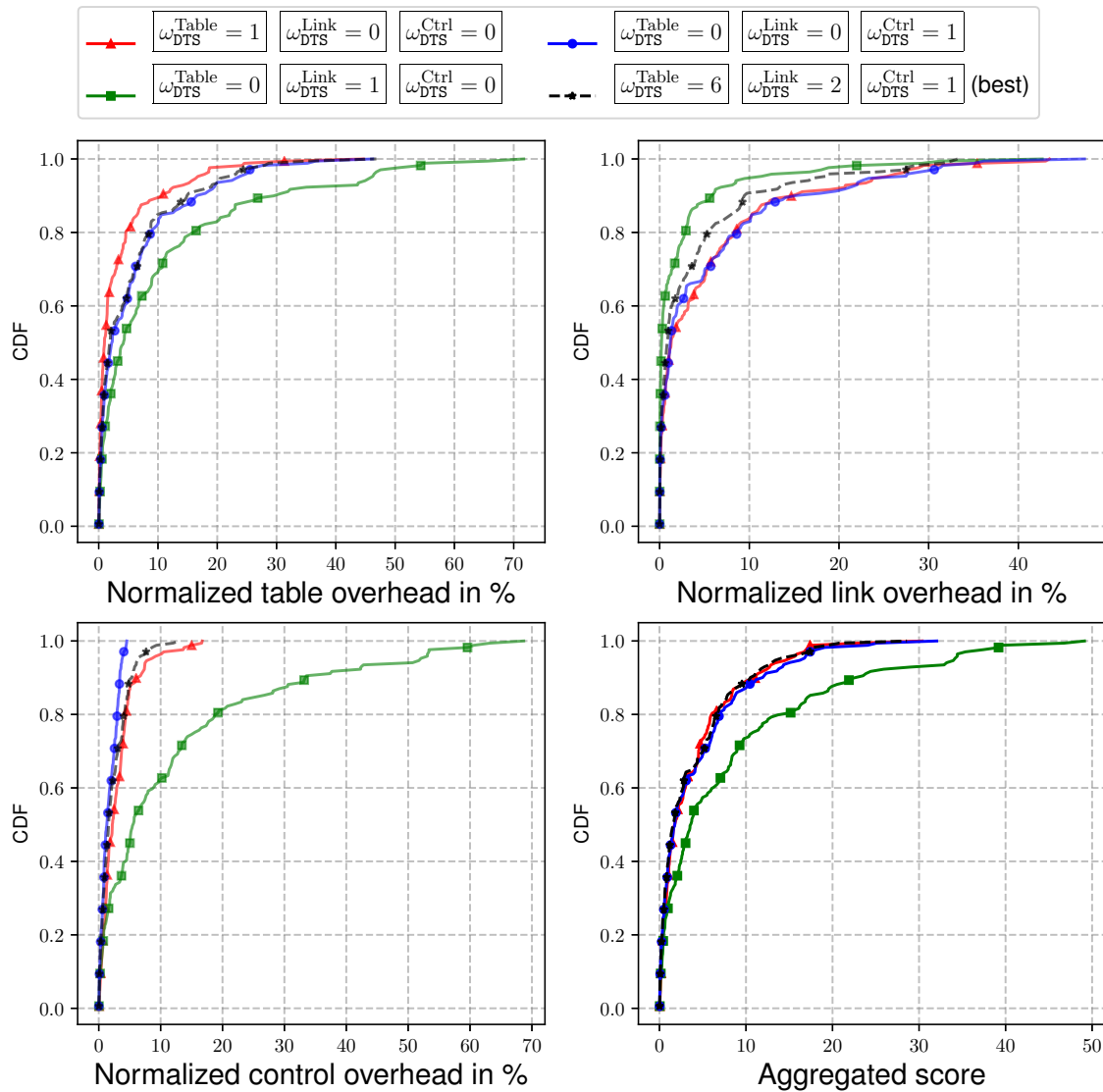
The results of the first experiment series for DT-Select are shown in Fig. A.9. The figure consists of four parts, three of which represent the different kinds of overhead in normalized form according to Table A.6. The fourth part in the bottom right represents an **aggregated score** calculated as the sum of the three other overhead values divided by 3, i.e., the three kinds of overhead have the same weight. The colors represent four selected parameterizations out of the 729 available combinations:

- The red curve shows the cumulative distribution functions for 100 experiments executed with  $\omega_{DTS}^{\text{Table}} = 1$ ,  $\omega_{DTS}^{\text{Link}} = 0$ , and  $\omega_{DTS}^{\text{Ctrl}} = 0$ . This means only the table overhead cost coefficients are included in the objective function.
- The green curve shows the cumulative distribution functions for 100 experiments executed with  $\omega_{DTS}^{\text{Table}} = 0$ ,  $\omega_{DTS}^{\text{Link}} = 1$ , and  $\omega_{DTS}^{\text{Ctrl}} = 0$ . This means only the link overhead cost coefficients are included in the objective function.
- The blue curve shows the cumulative distribution functions for 100 experiments executed with  $\omega_{DTS}^{\text{Table}} = 0$ ,  $\omega_{DTS}^{\text{Link}} = 0$ , and  $\omega_{DTS}^{\text{Ctrl}} = 1$ . This means only the control overhead cost coefficients are included in the objective function.
- The black curve finally represents the combination of weights with the highest aggregated score summed up over all 100 experiments. In this case, all three overhead parts are included in the objective functions, weighted with  $\omega_{DTS}^{\text{Table}} = 6$ ,  $\omega_{DTS}^{\text{Link}} = 2$ , and  $\omega_{DTS}^{\text{Ctrl}} = 1$ .

The four above example parameterizations are shown here because they demonstrate the improvement of a multi-objective approach compared to using only a single objective. Take the green curve as an example. If only the link overhead cost coefficients are included – and the coefficients for table overhead and control overhead are ignored –, the link overhead is minimized. This is clearly visible in the top right of Fig. A.9 where the 100 experiments summarized by the green curve achieve significantly smaller link overhead values. However, at the same time, the control overhead shown in the bottom left is significantly larger for the green curve, because this parameterization will always (and only) prefer delegation templates that lead to a minimization of the link overhead and ignore everything else. Switching from one set of selected delegation templates in time slot  $t_x$  to a completely new one in  $t_{x+1}$ , however, requires a large amount of control messages which contributes to the control overhead (we already saw this effect in Sec. A.2.2).

Similarly, the red and the blue curve can also achieve lower overhead in one specific area. The red curve minimizes the amount of required aggregation rules (table overhead, see plot in the top left) and the blue curve minimizes the amount of required additional control messages (control overhead, see plot in the bottom left). In practice, however, all three overhead parts are important. The black curve shows the best aggregated score with multi-objective optimization if all three parts are weighted equally. It is easy to see that the black curve with **parameterization**  $\omega_{DTS}^{\text{Table}} = 6$ ,  $\omega_{DTS}^{\text{Link}} = 2$ ,  $\omega_{DTS}^{\text{Ctrl}} = 1$  **achieves a good trade-off between table overhead, link overhead and control overhead.**

It can further be concluded that the red curve (only using table overhead coefficients) and the blue curve (only using control overhead coefficients) show similar behavior. This



**Figure A.9:** Evaluation of different DT-Select weights. Lower values are better.

means these two parameterizations can be used as a stand-alone objective function with acceptable results if multi-objective optimization is too expensive. The link overhead coefficients, on the other hand, are too aggressive when used alone, i.e., they have to be used together with at least one of the other cost coefficients in order to avoid an excessive amount of control messages.

### A.3.3 Results for RS-Alloc

**Findings:** The general recommendation for the RS-Alloc weights is:  $\omega_{\text{RSA}}^{\text{Table}} = 5$ ,  $\omega_{\text{RSA}}^{\text{Link}} = 0$ , and  $\omega_{\text{RSA}}^{\text{Ctrl}} = 1$ .

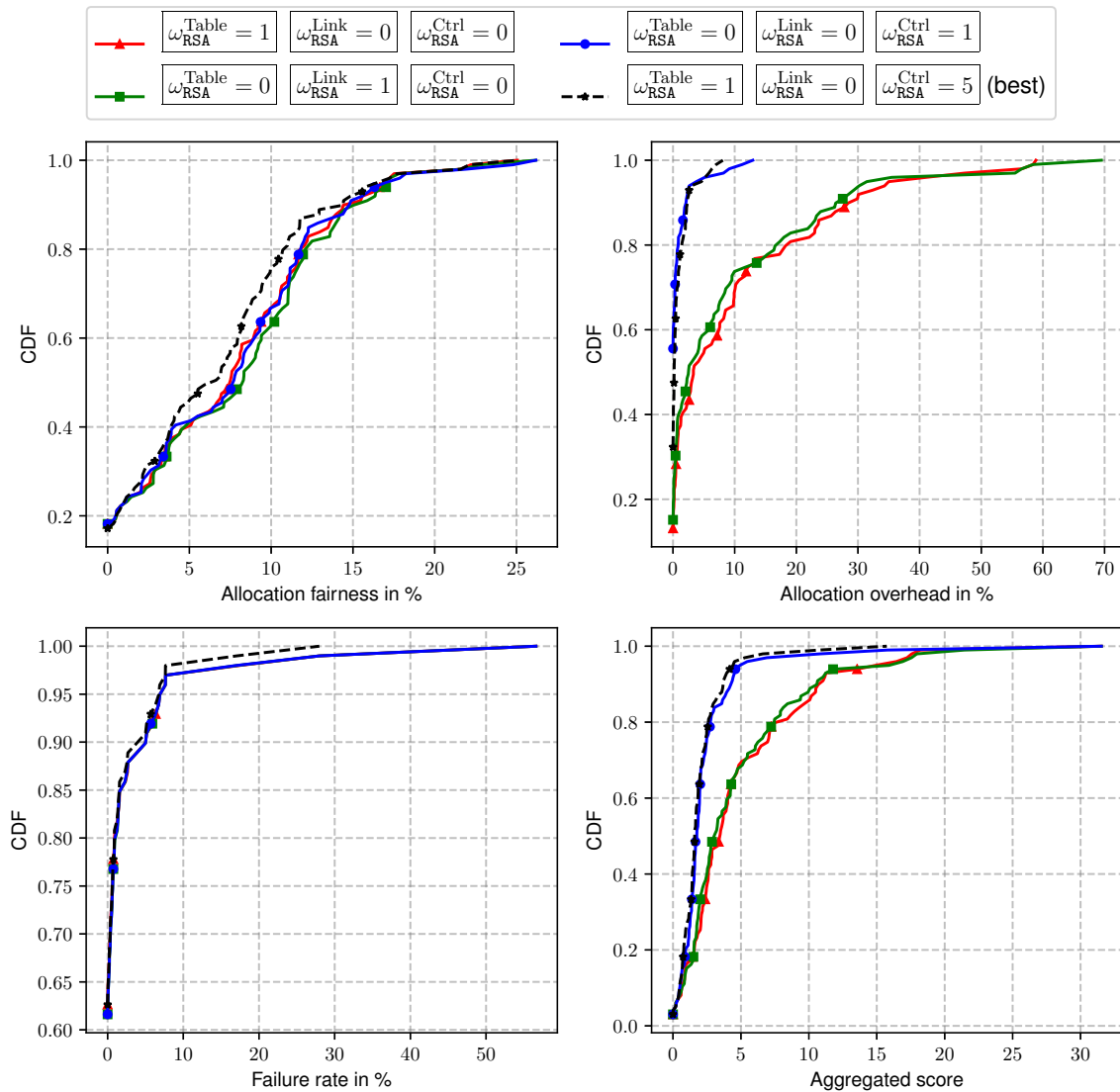
In case of RS-Alloc, we are mainly interested in failure rate, allocation overhead and allocation fairness if different  $\omega_{\text{DTS}}$ -weights are applied. Failure rate is defined in Def. 12.4 and determines the fraction of the flow rules not handled by a remote switch over the course of an experiment. This value is already given as a percentage value and no further changes are required.

**Allocation overhead** represents additional control messages that are required because RS-Alloc changed the allocation of a delegation template to another remote switch or to the backup switch. Allocation overhead is just another name for control overhead that was chosen to make sure that the control overhead caused by RS-Alloc can be properly distinguished from the control overhead caused by DT-Select. Note that RS-Alloc has to work with the delegation templates calculated by DT-Select. As a result, it has no impact on link or table overhead which is determined fully by the selected delegation templates. The only overhead directly associated with RS-Alloc is allocation overhead.

**Allocation fairness** determines the ability of RS-Alloc to balance available free flow table capacity of multiple remote switches. It is quantified with a metric that measures the difference in available free flow table capacity of all potential remote switch options  $R_{s,t}$  of a bottlenecked switch  $s$  in time slot  $t$ . Lower values mean the free flow table capacity of the switches in  $R_{s,t}$  is closer at each other. The formula for allocation fairness used in this work is defined as follows:

$$\left( \frac{1}{|T^{\text{Bneck}}|} \sum_{t \in T^{\text{Bneck}}} \frac{\frac{1}{|S|} \sum_{s \in S} \left( \frac{\sum_{r \in R_{s,t}} c_r^{\text{Table}} - u_{r,t}^{\text{Table}}}{|R_t|} \right)}{\max \left( \left\{ u_{s,t}^{\text{Table}} \mid s \in S, t \in T \right\} \right)} \right) * 100 \quad (\text{A.1})$$

Different versions of this formula and also other approaches with Jain's fairness index and a minimum mean squared error formulation were tested, all with very similar results (included in the data set but not used in the plots below). The difference value for one time slot is divided by the maximum flow table utilization that occurred over the course of the experiment and averaged over all bottlenecked time slots to get a percentage value between 0% and 100%.



**Figure A.10:** Evaluation of different RS-Alloc weights. Lower values are better.

It is important to mention that allocation fairness has to be explicitly included in the optimization problem. This is done here by adding a small amount of dummy assignments for each switch. These dummy assignments improve the objective but block a certain amount of flow rules in the assigned switch if selected by the solver. Because the cost for selecting a dummy depends on the free flow table capacity and is further scaled with  $\omega_{\text{RSA}}^{\text{Table}}$ , allocation of delegation templates can be balanced between the remote switches which results in more “fairness” with respect to resource allocation – because the dummy

assignments are more likely to be selected which will reserve capacity<sup>4</sup>. It is important to mention, however, that this is only an example for a very simple resource allocation strategy. It could also be valid strategy to reduce the number of remote switches, for example.

The results for the RS-Alloc experiment series with different  $\omega_{\text{DTS}}$ -weights are shown in Fig. A.10. The figure consists of four parts: Allocation fairness is shown in the top left, allocation overhead in the top right and the failure rate in the bottom left. The last part in the bottom right shows the aggregated score of the three other values. The colors represent four selected parameterizations out of the 729 available combinations, very similar to the discussion in the previous section:

- The red curve shows the cumulative distribution functions for 100 experiments executed with  $\omega_{\text{RSA}}^{\text{Table}} = 1$ ,  $\omega_{\text{RSA}}^{\text{Link}} = 0$ , and  $\omega_{\text{RSA}}^{\text{Ctrl}} = 0$ . This means only the available free flow table capacity at the remote switch and allocation fairness is taken into account.
- The green curve shows the cumulative distribution functions for 100 experiments executed with  $\omega_{\text{RSA}}^{\text{Table}} = 0$ ,  $\omega_{\text{RSA}}^{\text{Link}} = 1$ , and  $\omega_{\text{RSA}}^{\text{Ctrl}} = 0$ . This means only the available free link bandwidth between delegation switch and remote switch is taken into account.
- The blue curve shows the cumulative distribution functions for 100 experiments executed with  $\omega_{\text{RSA}}^{\text{Table}} = 0$ ,  $\omega_{\text{RSA}}^{\text{Link}} = 0$ , and  $\omega_{\text{RSA}}^{\text{Ctrl}} = 1$ . This means only the allocation overhead is taken into account.
- The black curve finally represents the combination of weights with the highest aggregated score summed up over all 100 experiments. The aggregated score is again the weighted sum of the three other parts listed above. In this case, the best result was achieved for  $\omega_{\text{RSA}}^{\text{Table}} = 1$ ,  $\omega_{\text{RSA}}^{\text{Link}} = 0$ , and  $\omega_{\text{RSA}}^{\text{Ctrl}} = 5$ .

It can be seen that the blue curve associated with allocation overhead plays a crucial role when selecting weights for RS-Alloc. This is expected because allocation overhead is the only part directly associated with remote switch allocation. Setting  $\omega_{\text{RSA}}^{\text{Ctrl}}$  to 0 – as it is done in the two other example parameterizations (red and green)– leads to significantly more allocation overhead. This effect can be observed in the top right plot in Fig. A.10. Using only  $\omega_{\text{RSA}}^{\text{Ctrl}} = 1$  and ignoring the two other weights, however, is a viable option. This can be used if multi-objective optimization is too expensive. It can further be seen that the green and the red curve show a similar trend. This is due to the fact that RS-Alloc has little influence on both, table utilization and link bandwidth because the algorithm works with delegation templates calculated by DT-Select.

<sup>4</sup>The implementation of this feature can be found under “balancing assignments” in the RS-Alloc solver, see [https://github.com/kit-tm/fdeval/blob/master/engine/solve\\_rsa.py](https://github.com/kit-tm/fdeval/blob/master/engine/solve_rsa.py)

The black curve representing the best combination of weights basically follows the blue curve. The only exception is that using non-zero  $\omega_{RSA}^{Table}$  weights improves allocation fairness as shown in the top left. In conclusion, the **recommended parameterization for RS-Alloc is  $\omega_{RSA}^{Table} = 1$ ,  $\omega_{RSA}^{Link} = 0$ , and  $\omega_{RSA}^{Ctrl} = 5$** . It is expected that a non-zero  $\omega_{RSA}^{Link}$  weight will have a more noticeable effect if a resource allocation strategy for link bandwidth is included which is basically identical to the allocation fairness mechanism that balances the free flow table capacity. Allocation strategies for RS-Alloc in general are an interesting topic for future research.

## A.4 Used Parameters

The following table lists the used parameters for DT-Select and RS-Alloc. Note that better results may be achieved for individual scenarios if other parameterizations are used. However, this and advanced mechanisms such as automatic parameter tuning with the help of machine learning (for example) are considered future work and not discussed here further.

Parameter	Description	Used Values	
$n_{dts\_algo}$	DT-Select algorithm	Select-CopyFirst	Sec. A.2.2
$L_{dts}$	Look-ahead factor for DT-Select	3	
$L_{rsa}$	Look-ahead factor for RS-Alloc	3	Sec. A.2.3
$n_{assignments}$	Assignments for RS-Alloc	50	
$\omega_{DTS}^{Table}$	Weight for DT-Select	6	Sec. A.3.2
$\omega_{DTS}^{Link}$	Weight for DT-Select	2	
$\omega_{DTS}^{Ctrl}$	Weight for DT-Select	1	
$\omega_{RSA}^{Table}$	Weight for RS-Alloc	5	Sec. A.3.3
$\omega_{RSA}^{Link}$	Weight for RS-Alloc	0	
$\omega_{RSA}^{Ctrl}$	Weight for RS-Alloc	1	

**Table A.7:** Recommended parameterization for DT-Select and RS-Alloc



# Scenario Set Characteristics

---

The scenario-based evaluation uses two scenario sets: one large scenario set with 5,000 scenarios called  $Z_{5000}$  and a smaller set with 100 scenarios called  $Z_{100}$ . This section presents additional characteristics with respect to these two important sets.

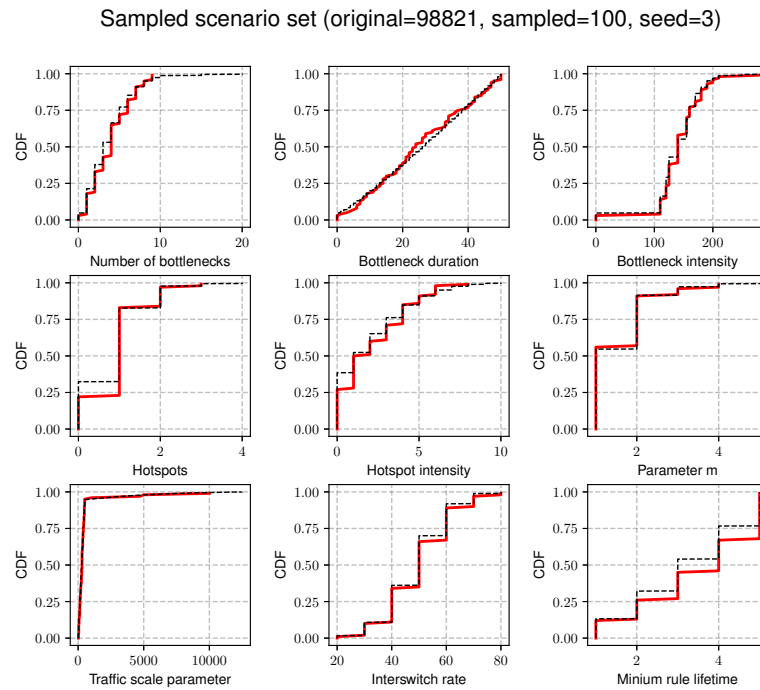
## B.1 Parameter Distribution

Fig. B.1 and Fig. B.2 show CDFs for additional parameters in scenario set  $Z_{100}$  and  $Z_{5000}$ , compared to the data provided in Fig. 12.11 and Fig. 12.10. The dashed black line shows the parameter distribution in the original scenario set with 98,821 scenarios. The red line shows the parameter distribution in the smaller scenario set with 100 and 5,000 scenarios.

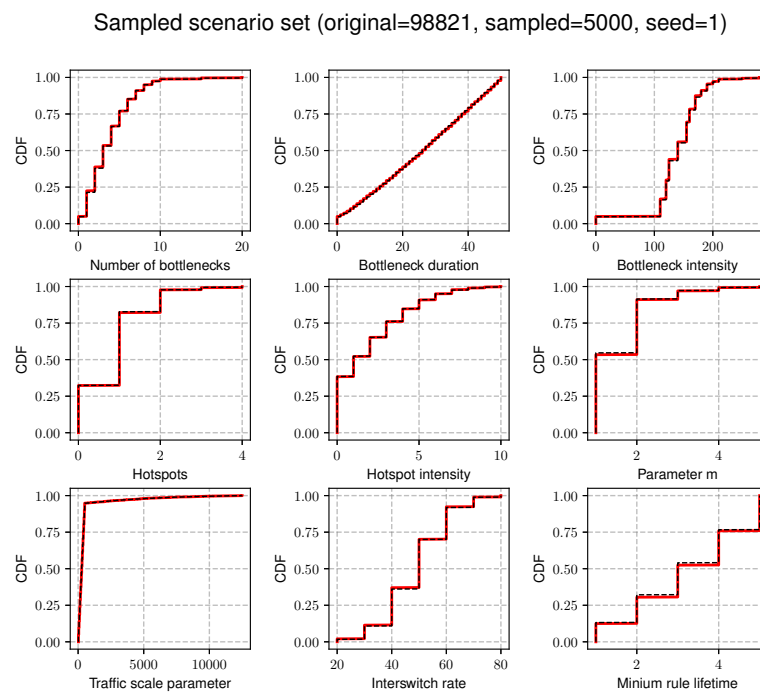
## B.2 Correlation between Important Characteristics

This section further characterizes the investigated scenarios in  $Z_{5000}$ . Fig. B.3 shows the correlation between capacity reduction factor, failure rate and flow table utilization ratio if the accepted maximum failure rate is set to 0%. The x-axis represents the capacity reduction factor between 1% and 80%. The y-axis denotes the fraction of experiments that actually have a capacity reduction factor that is equal or higher to the specified x-value. The different curves represent the seven flow table utilization ratio classes from Table 14.2.

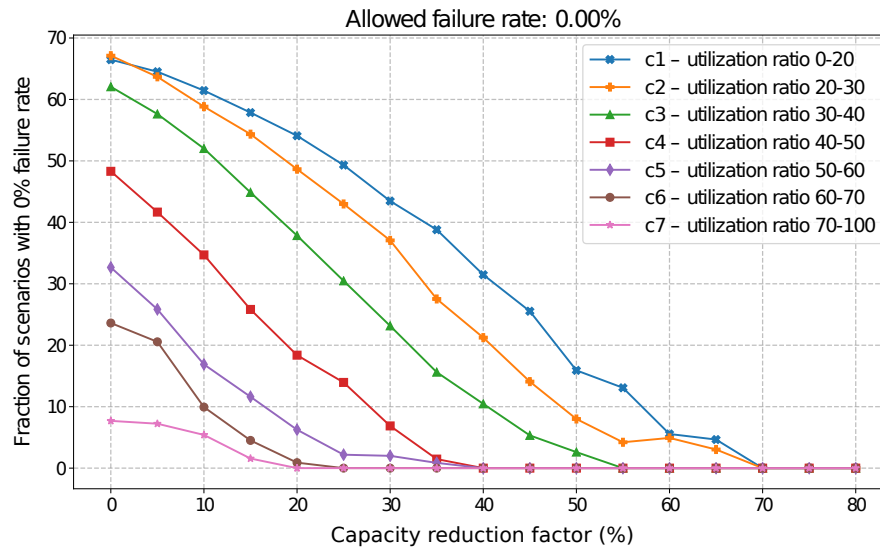
Take class  $c_1$  in Fig. B.3 – the blue line – as an example for how to read the plots. We know from Table 14.2 that 66.48% of the experiment in class  $c_1$  were able to achieve a



**Figure B.1:** CDFs for parameter characterization of scenario set  $Z_{100}$



**Figure B.2:** CDFs for parameter characterization of scenario set  $Z_{5000}$



**Figure B.3:** Correlation between capacity reduction factor, failure rate and flow table utilization ratio in  $Z_{5000}$  with a maximum accepted failure rate of 0%

failure rate of 0%. In Fig. B.3, we can see this value at  $x = 0$  because all experiments have at least a capacity reduction factor of 1%. The fraction of experiments with 0% failure rate then decreases for increased values of  $x$ . 61.45% of the experiments in  $c_1$  achieved at least a capacity reduction of 10%. 57.86% achieved a reduction of 15%. This trend continues, as shown in the plot: the fraction of experiments without failures decreases with increasing capacity reduction. Only 25.53% can achieve a reduction of 45%. 4.65% can achieve 65%. And no experiment in  $c_1$  is able to achieve a capacity reduction of 70% or higher.

When comparing the different classes, it is easy to see that there is a strong correlation between utilization ratio and the capacity reduction that can be achieved with a 0% failure rate. The lower the utilization ratio, the higher the achievable capacity reduction. Take  $x=40\%$  as an example. With  $c_1$ , 31.48% of the experiments can achieve a reduction of 40%. For  $c_2$ , the fraction of experiments with a 0% failure rate for 40% reduction is only 21.24% which is considerable lower.  $c_3$  achieves 40% for 10.46% of its experiments. And none of the higher classes ( $c_4$  to  $c_7$ ) has an experiment that gets as high as 40% reduction. Class  $c_4$  and  $c_5$  are capped at  $x=35\%$  (0.85% of the scenarios in case of  $c_5$ ). Class  $c_6$  and  $c_7$  are capped at  $x=15\%$ .

The results for a maximum accepted failure rate of 0.1% and 1% are shown Fig. B.4 and Fig. B.5. The overall trend is the same while the fraction of scenarios is increased with higher maximum accepted failure rate.

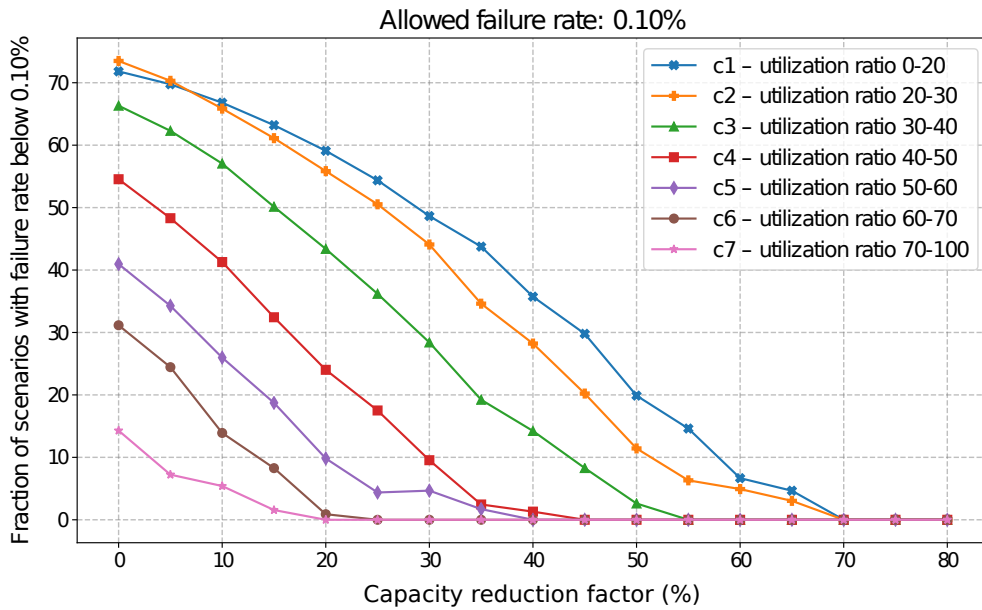


Figure B.4: Correlation between capacity reduction factor, failure rate and flow table utilization ratio in  $Z_{5000}$  with a maximum accepted failure rate of 0.1%

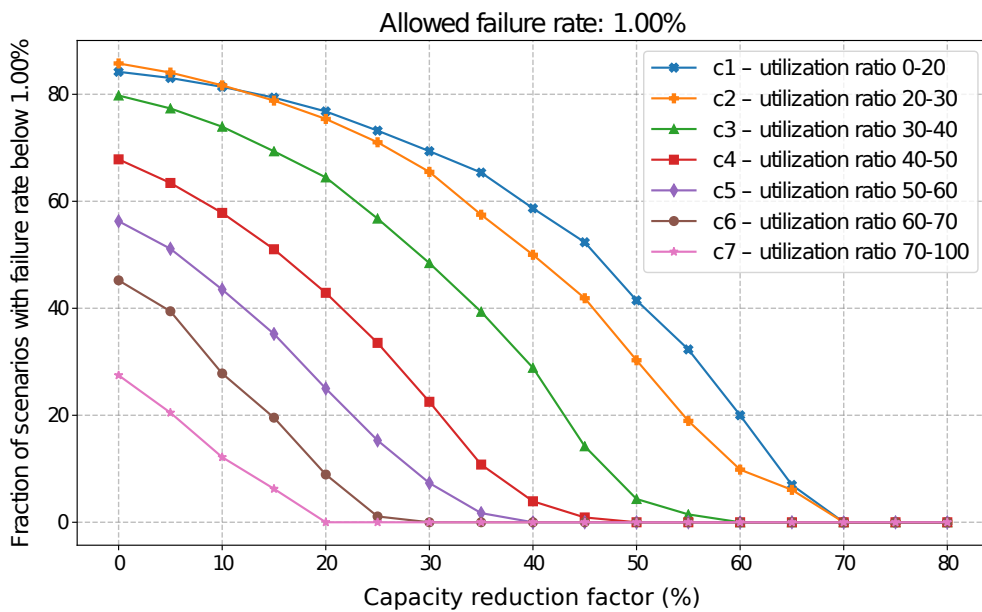


Figure B.5: Correlation between capacity reduction factor, failure rate and flow table utilization ratio in  $Z_{5000}$  with a maximum accepted failure rate of 1%

## B.3 Examples for Bottlenecked Situations

Fig. B.6 to Fig. B.9 show different bottleneck situations in scenario set  $Z_{100}$ . Note that scenarios can have more than one bottleneck switch, that is why there are more bottleneck situations than scenarios. The small numbers in the rectangles are the scenario IDs and the id of the bottlenecked switch (the number after the period).

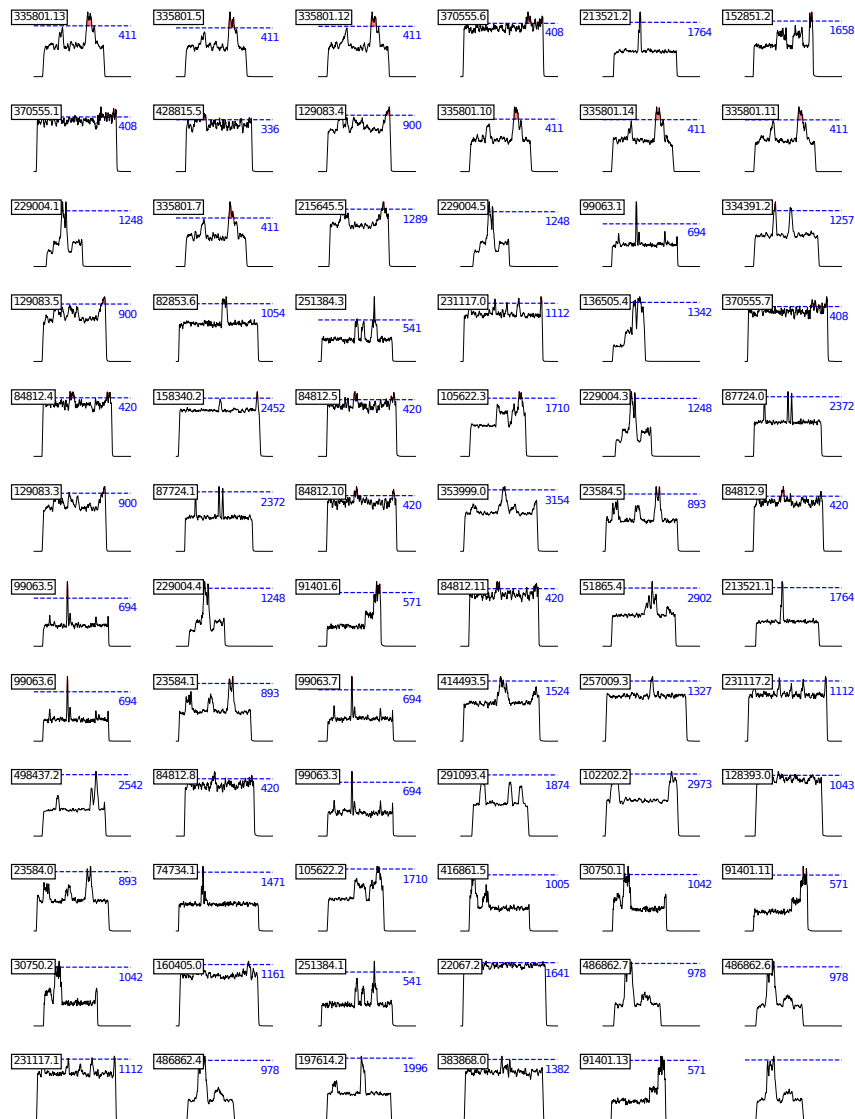


Figure B.6: Bottleneck Scenarios Example 1

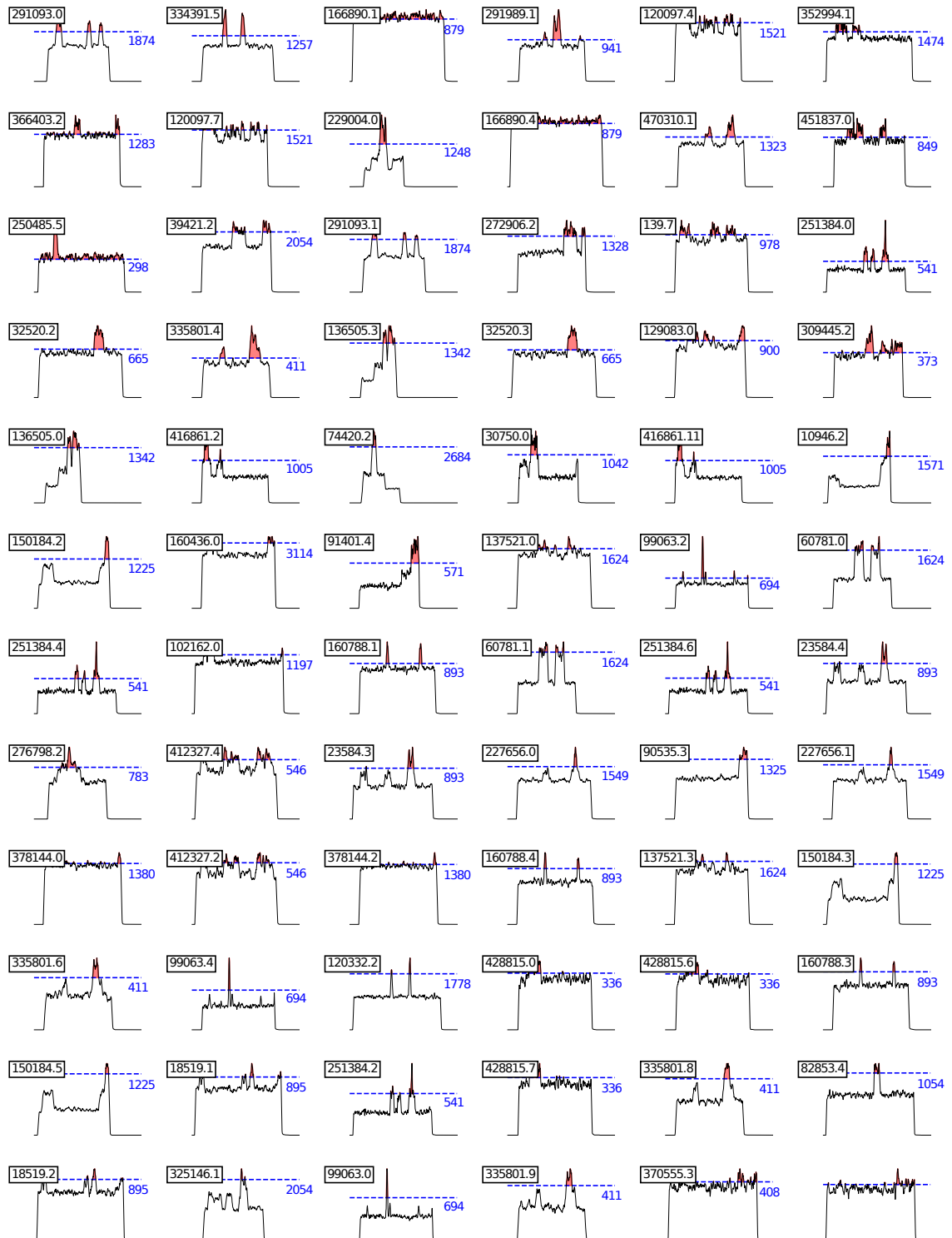


Figure B.7: Bottleneck Scenarios Example 2

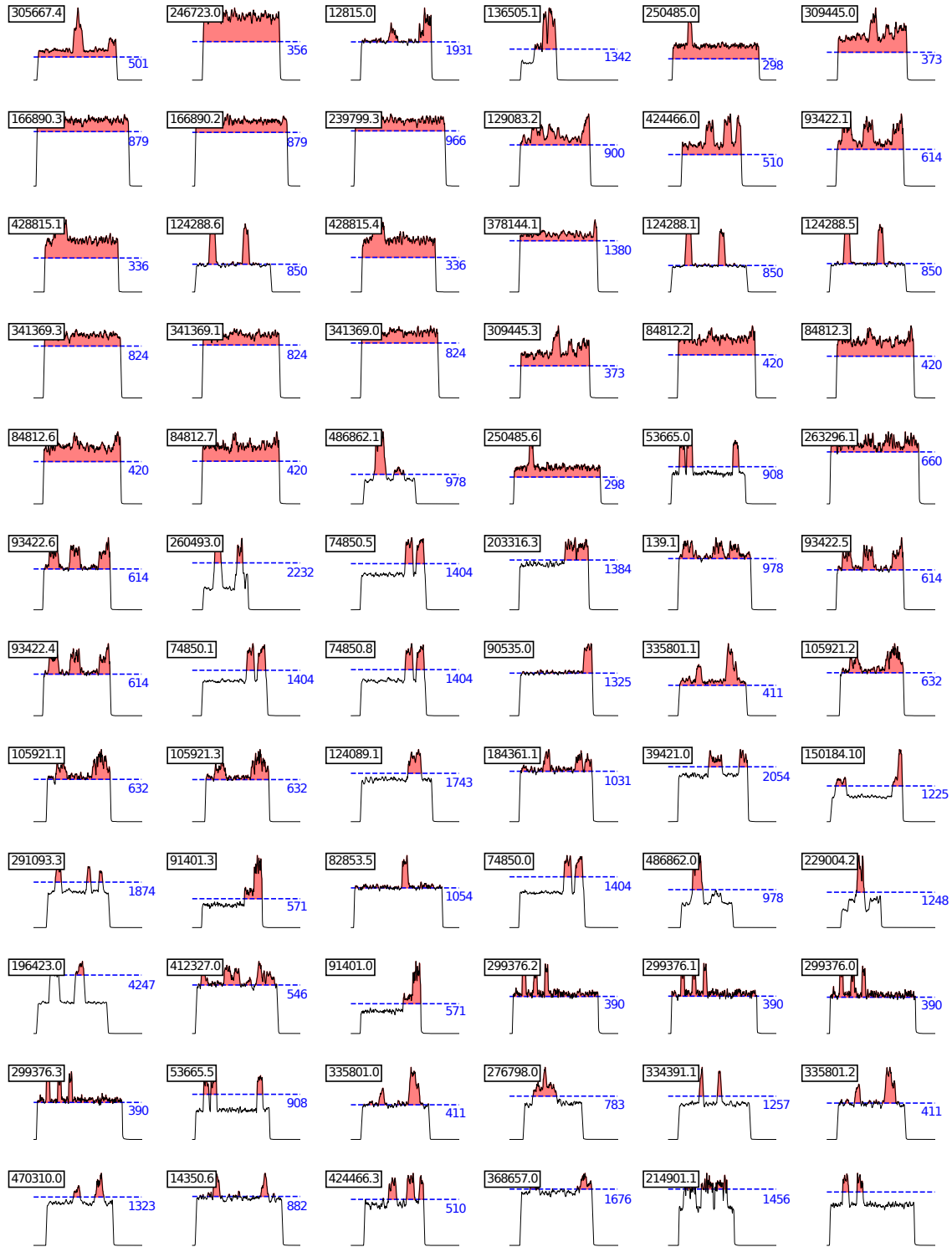


Figure B.8: Bottleneck Scenarios Example 3

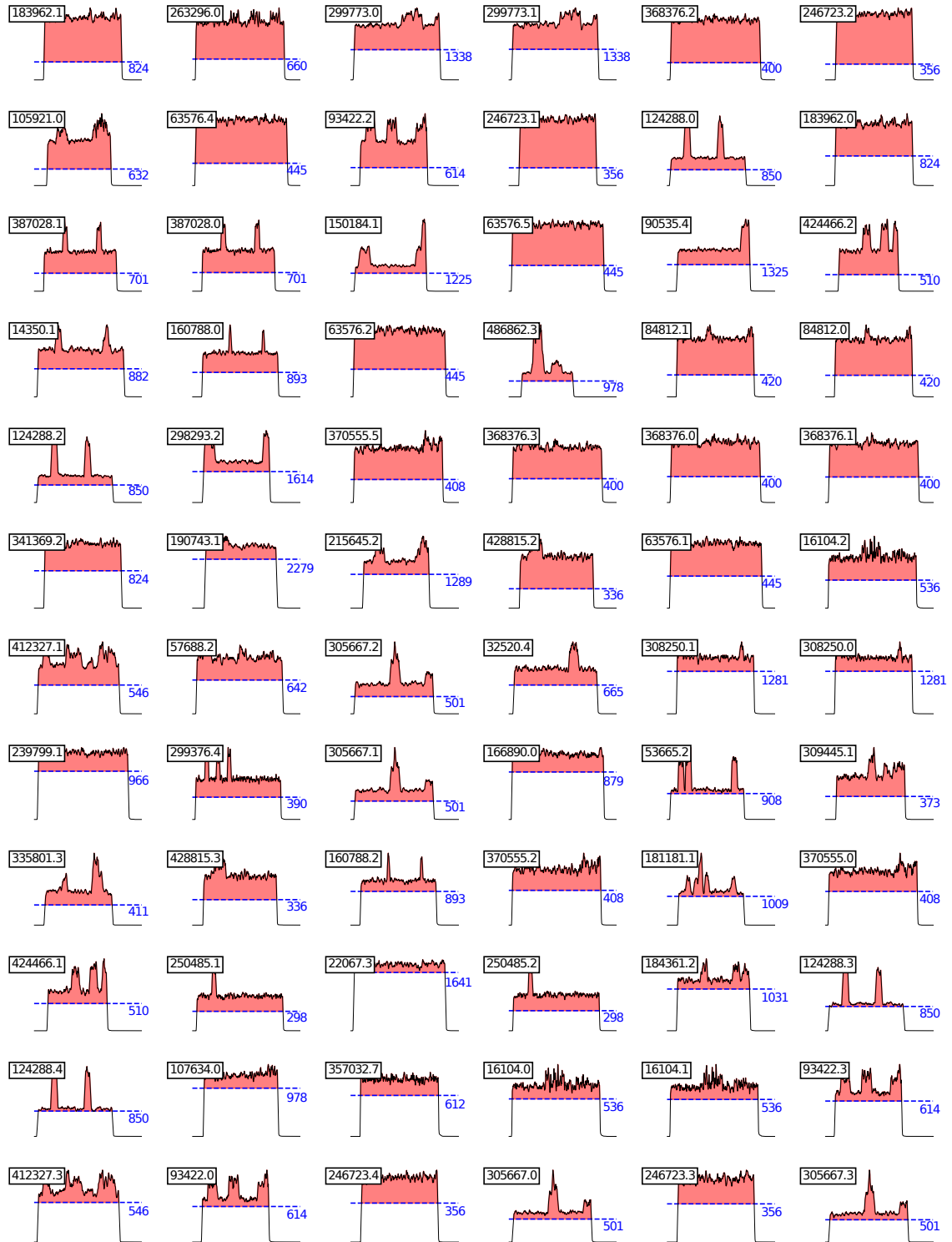


Figure B.9: Bottleneck Scenarios Example 4

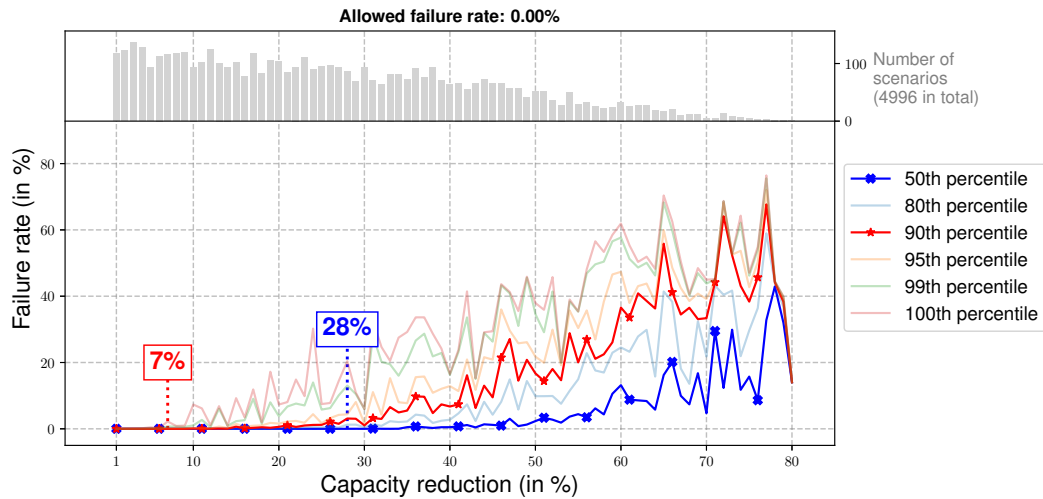


## Appendix C

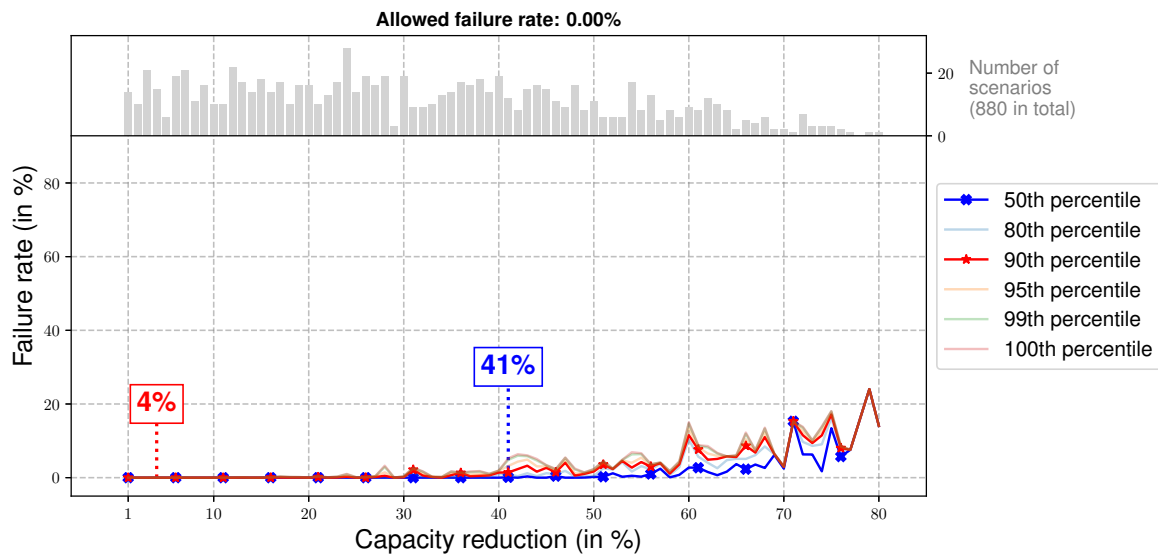
# Additional Performance Results

The following two sections show the flow delegation performance for all scenarios in  $Z_{5000}$  with different maximum accepted failure rates and different flow table utilization ratios.

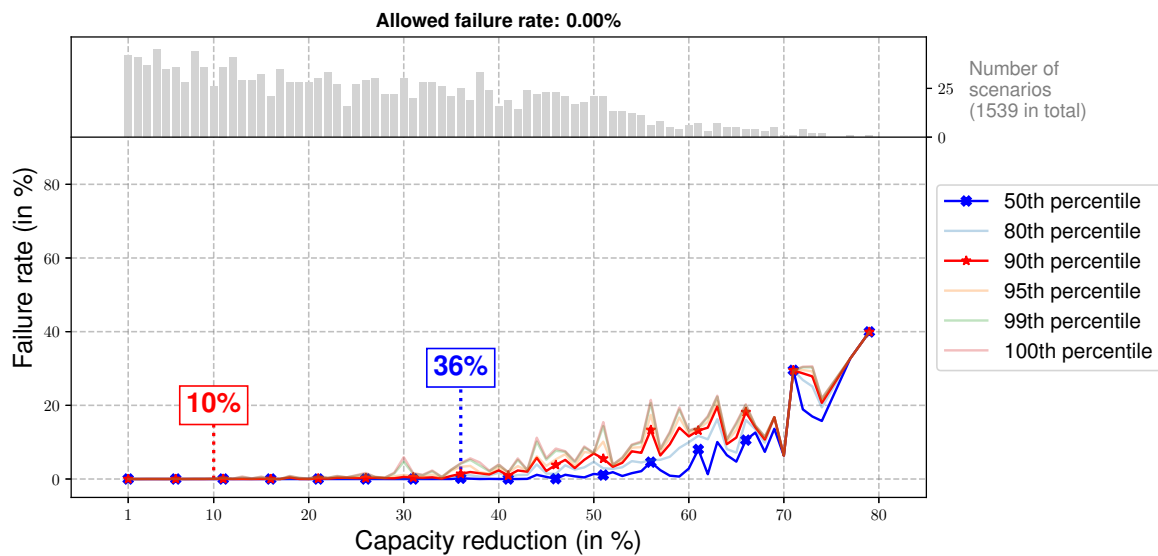
### C.1 Performance without Failures



**Figure C.1:** Flow delegation performance for all scenarios in  $Z_{5000}$  given a 0% accepted maximum failure rate



**Figure C.2:** Flow delegation performance for scenarios in  $Z_{5000}$  with flow table utilization ratio class **c1** (0-20%) and a 0% accepted maximum failure rate



**Figure C.3:** Flow delegation performance for scenarios in  $Z_{5000}$  with flow table utilization ratio class **c2** (20-30%) and a 0% accepted maximum failure rate

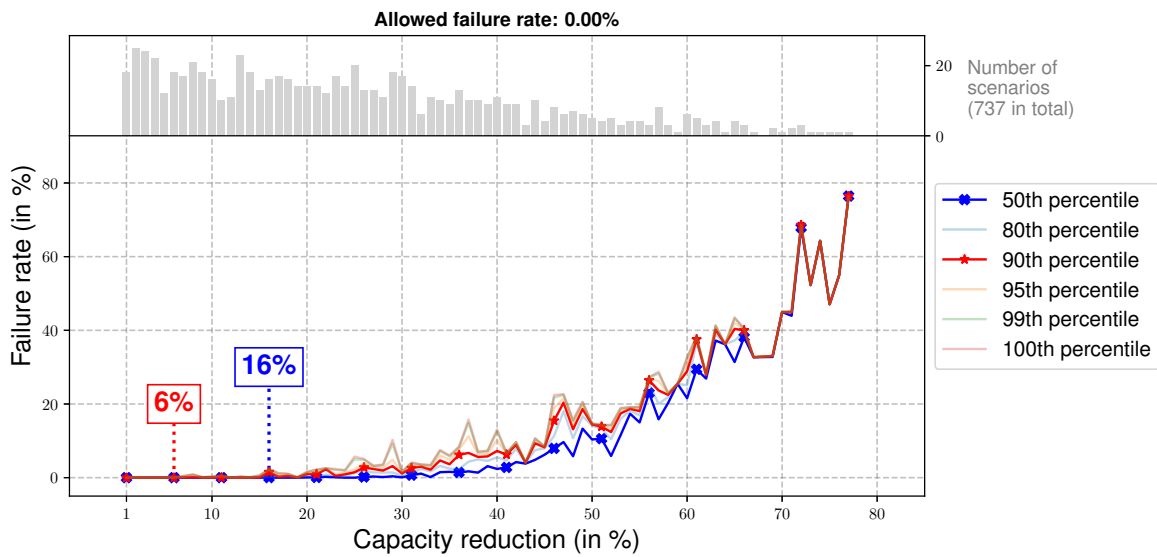


Figure C.4: Flow delegation performance for scenarios in  $Z_{5000}$  with flow table utilization ratio class *c4* (40-50%) and a 0% accepted maximum failure rate

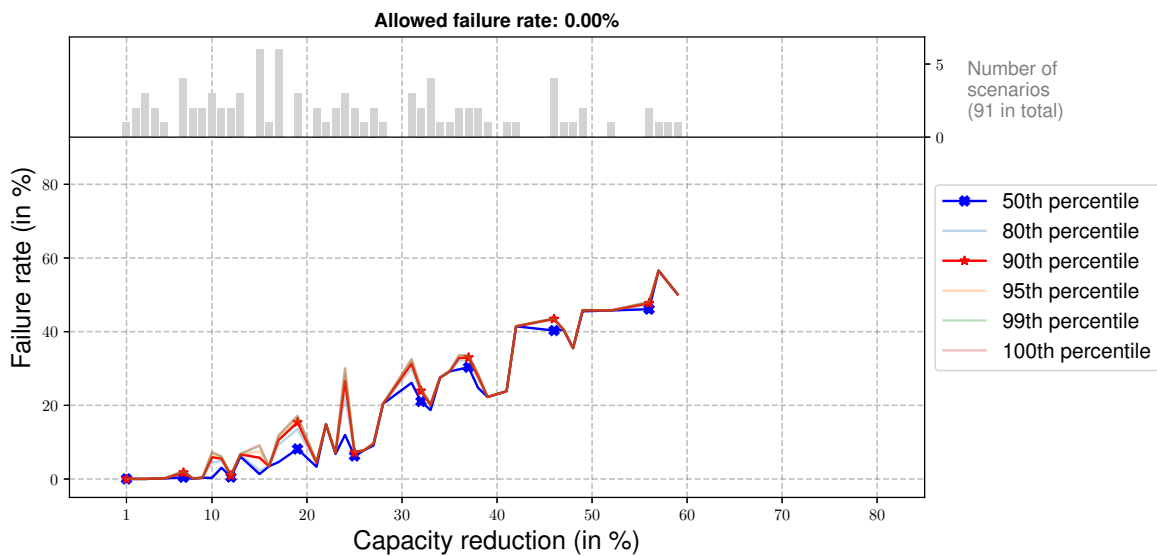
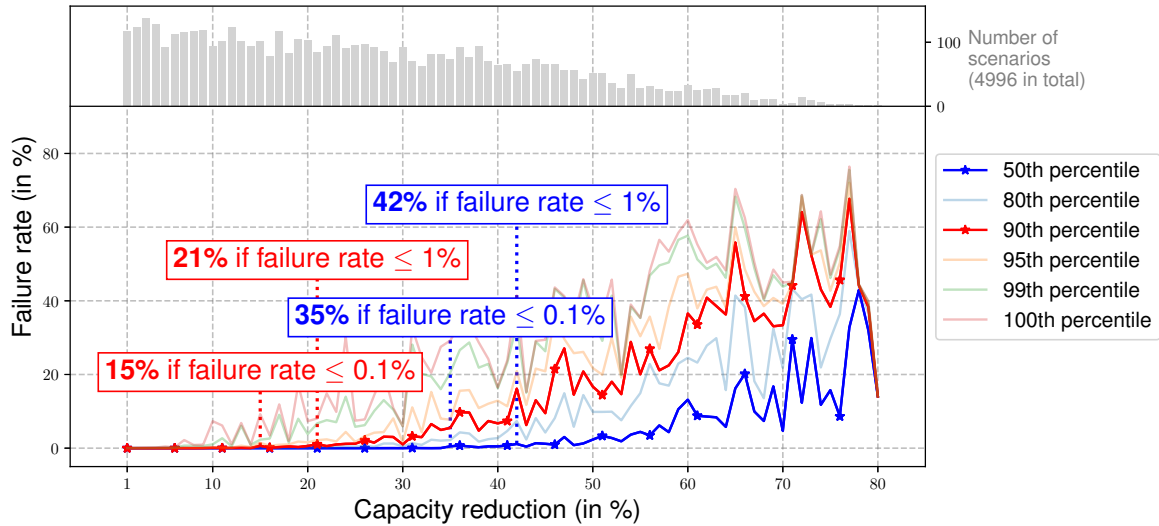
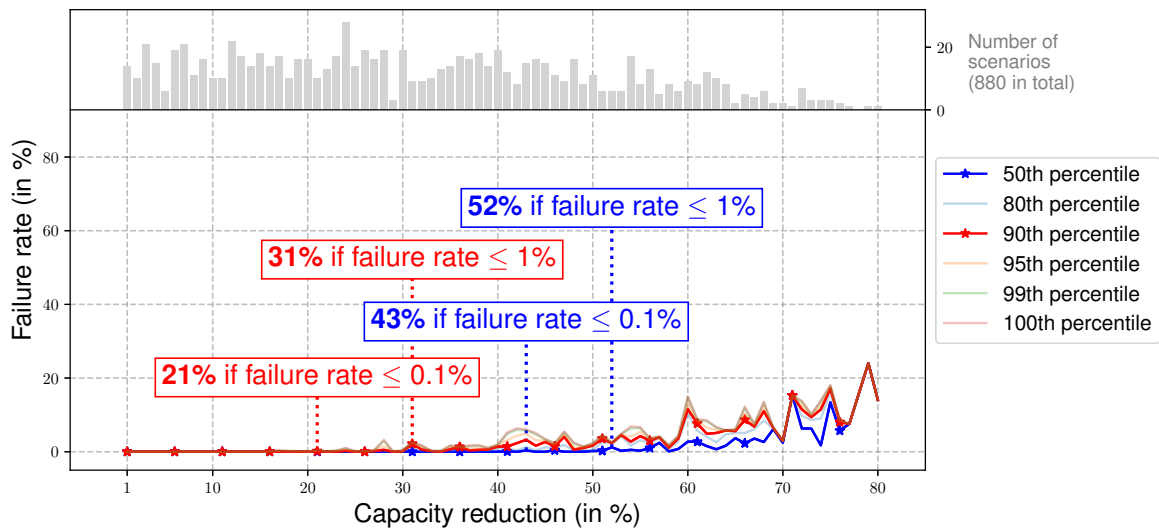


Figure C.5: Flow delegation performance for scenarios in  $Z_{5000}$  with flow table utilization ratio class *c7* (70-100%) and a 0% accepted maximum failure rate

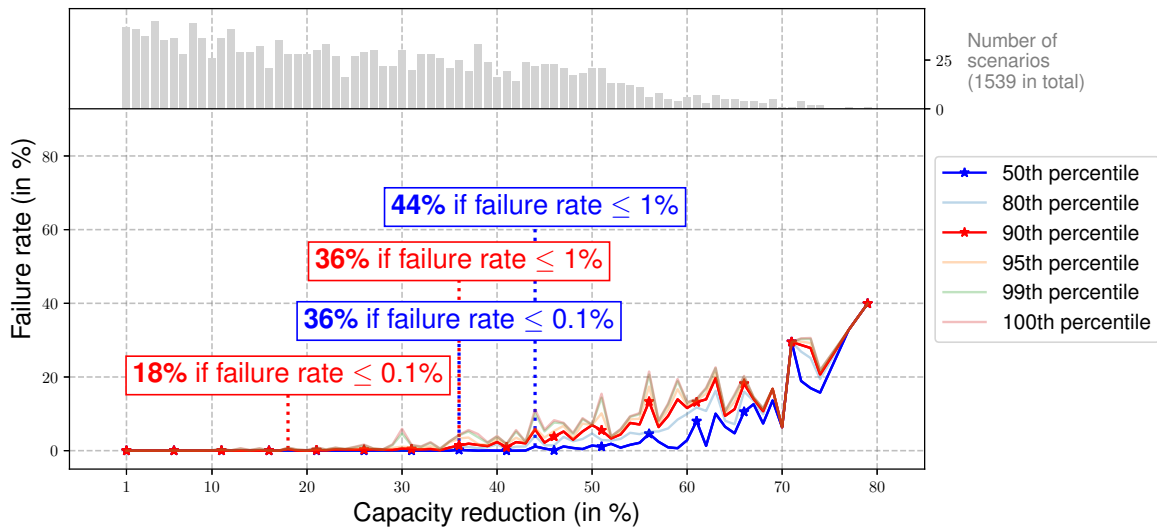
### C.2 Performance with small Failure Rates



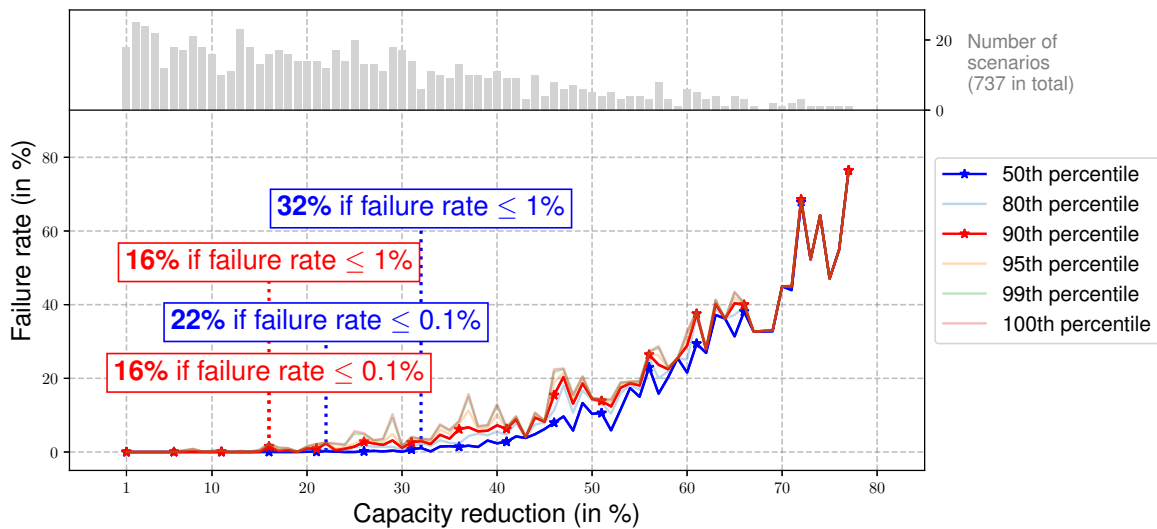
**Figure C.6:** Flow delegation performance for all scenarios in  $Z_{5000}$  and a accepted maximum failure rate  $>0\%$



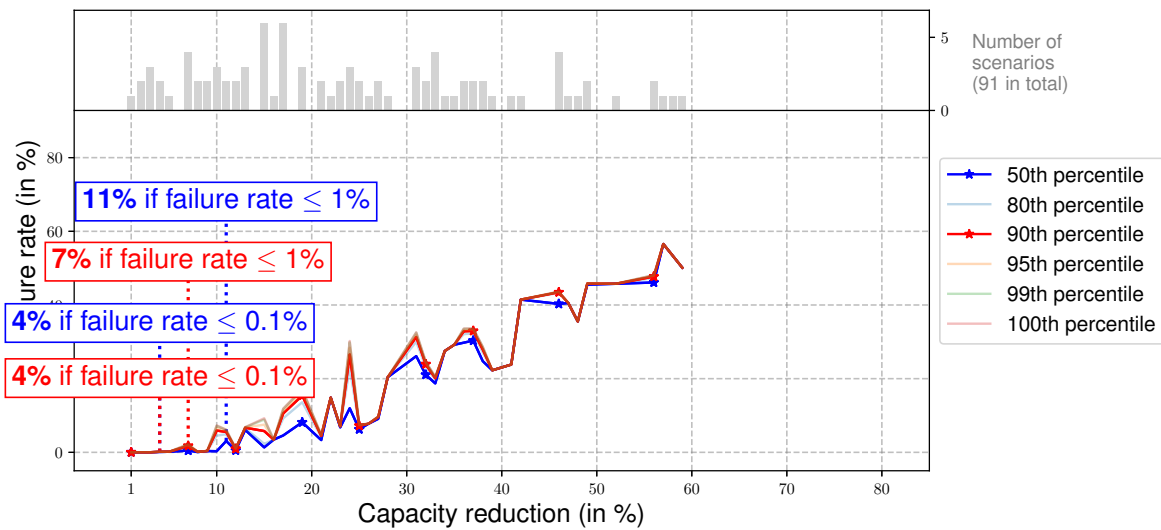
**Figure C.7:** Flow delegation performance for scenarios in  $Z_{5000}$  with flow table utilization ratio class *c1* (0-20%) and a accepted maximum failure rate  $>0\%$



**Figure C.8:** Flow delegation performance for scenarios in  $Z_{5000}$  with flow table utilization ratio class *c2* (20-30%) and a accepted maximum failure rate  $>0\%$



**Figure C.9:** Flow delegation performance for scenarios in  $Z_{5000}$  with flow table utilization ratio class *c4* (40-50%) and a accepted maximum failure rate  $>0\%$



**Figure C.10:** Flow delegation performance for scenarios in  $Z_{5000}$  with flow table utilization ratio class **c7** (70-100%) and a accepted maximum failure rate  $>0\%$

## Appendix D

---

# Reproducibility

---

Reproducibility of the results was an important requirement for all experiments conducted in this thesis. Code, data sets, and scripts that were used to create the results are available online and referenced here.

## D.1 Code

The code for the prototypical implementations used in Chapter 8 and the evaluation environment used in chapters 12 to 16 can be found on GitLab/GitHub, see table below.

	Middleware Prototype (2016)	Proxy Prototype (2019)	Evaluation Environment (2020)
Language	Python 2.7	Java 8	Python 3.6.8
Contributors	Robert Bauer, Ioannis Papamanoglou	Robert Bauer, Simon Herter, Addis Dittebrandt	Robert Bauer
Code access	<a href="https://git.scc.kit.edu/work/sdn-pbce">https://git.scc.kit.edu/work/sdn-pbce</a>	<a href="https://github.com/kit-tm/gcmi">https://github.com/kit-tm/gcmi</a> , <a href="https://github.com/kit-tm/gcmi-exp">https://github.com/kit-tm/gcmi-exp</a>	<a href="https://github.com/kit-tm/fdeval">https://github.com/kit-tm/fdeval</a>

**Table D.1:** *Code used in this thesis*

## D.2 Datasets

Download instructions for all datasets can be found here: <https://doi.org/10.5445/IR/1000120288> [Bau20]. Details are listed in the read-me file in the Github repository (<https://github.com/kit-tm/fdeval>). Each dataset comes as an sqlite3 database that contains the input parameters and the results of the experiments. Please refer to the scripts in <https://github.com/kit-tm/fdeval/tree/master/plotter> for details on how to use the datasets. Good starting points for this are `agg_01_info.py` and `agg_01_d021_scale_switches.py`.

The following table summarizes all datasets that were used in the evaluation part of the thesis in chapters 12 to 16 and in the appendices. The first column denotes the dataset identifier. The last column points to all figures that were created with the dataset. The most important datasets are D5000 and D100 which contain the results for the two scenario sets  $Z_{5000}$  and  $Z_{100}$ .

	Description of the dataset	Used for figures
D1	Randomly generated dataset with 500.000 different scenarios used in the scenario generation process. The scenario sets $Z_{100}$ and $Z_{5000}$ were sampled from this dataset.	12.9 to 12.11, B.1, B.2
D2	Consists of handcrafted scenarios with different scenario generation and bottleneck parameters. Only used for examples in Chapter 12, not for performance or scalability evaluations.	12.4 to 12.8
D3	Consists of handcrafted scenarios with different topology parameters. Only used for examples in Chapter 12, not for performance or scalability evaluations.	12.3
D20	Consists of handcrafted scenarios with different amounts of considered delegation templates. Used for the scalability evaluation in Chapter 16.	16.6 to 16.9
D21	Consists of handcrafted scenarios with different amounts of switches in the topology. Used for the scalability evaluation in Chapter 16.	16.10
D30	Contains the scenarios from scenario set $Z_{100}$ with different values for the DT-Select look-ahead factor ( $L=1$ to $L=9$ ). Used for the parameter study in Appendix A.	A.1 to A.3
D31	Same as dataset D30 above but the DT-Select look-ahead factor is varied between 1 and 50. Used for the parameter study in Appendix A.	A.4, A.6



D40	Contains the scenarios from scenario set $Z_{100}$ with different values for the RS-alloc look-ahead factor ( $L=1$ to $L=9$ ) and allocation assignments (5,20,50,100,200). Used for the parameter study in Appendix A.	A.5
D41	Same as dataset D40 above but the RS-Alloc look-ahead factor is varied between 1 and 30. Used for the parameter study in Appendix A.	A.7, A.8
D50	Contains the scenarios from scenario set $Z_{100}$ with different weights for the DT-Select algorithms. Used for the parameter study in Appendix A.	A.9
D60	Contains the scenarios from scenario set $Z_{100}$ with different weights for the RS-alloc algorithm. Used for the parameter study in Appendix A.	A.10
D100	Contains the scenarios from scenario set $Z_{100}$ with all capacity reduction factors between 1% and 80% (80 experiments per scenario). Used primarily for the performance evaluation in Chapter 14.	13.1, 14.2, 14.4, 14.6, B.6 to B.9
D110	Contains four selected scenarios from $Z_{100}$ with capacity reduction factors between 1% and 80%. Used for the case study in Chapter 13. Is redundant to D100 but the dataset is much smaller which can be helpful.	13.2 bis 13.13, 15.1
D5000	Main dataset of the thesis. Contains the scenarios from scenario set $Z_{5000}$ where each scenario is assigned a random capacity reduction factor between 1% and 80%. Contains results for all three DT-Select algorithms. Used primarily for the performance evaluation in Chapter 14.	14.1, 14.3, 14.5, 14.7 to 14.15, 15.2 to 15.7, B.3 to B.5, C.1 to C.10
D5050	Contains the scenarios from scenario set $Z_{5000}$ where each scenario is assigned a random capacity reduction factor between 1% and 80%. Was only executed with Select-CopyFirst and the execution was restricted to a single CPU core. Used for the algorithm runtime evaluation in Chapter 16.	16.1 to 16.5

**Table D.2:** *Datasets used in this thesis*

### D.3 Dataset Details

This section presents some selected statistics of the used datasets. The table below lists 14 parameters, seven in the top and seven in the bottom (was splitted in two parts due to space constraints). The parameters are:

- **Size (MB):** Size of the dataset in MBytes
- **Exp.:** Total number of experiments in the dataset
- **T/O:** Number of experiments that were aborted because of a timeout
- **Scenarios:** Number of different scenarios in the dataset
- **Switches:** Minimum and maximum number of switches in the topology
- **Hosts:** Minimum and maximum number of hosts in the topology
- **Pairs:** Minimum and maximum number of selected communication pairs
- **Reduction:** Minimum and maximum capacity reduction factor
- **Bneck:** Minimum and maximum number of temporal bottlenecks
- **HS:** Minimum and maximum number of hotspots in the topology
- **m:** Minimum and maximum parameter for the Barabasi-Albert model
- **ISR:** Minimum and maximum inter switch ratio
- **Traffic Scale:** Minimum and maximum traffic scale parameter
- **DT-Select:** Used DT-Select algorithms

Dataset	Size (MB)	Exp.	T/O	Scenarios	Switches	Hosts	Pairs
D1	1356.86	500000	3164	500000	2-15	10-300	25000-249999
D2	4807.54	64800	0	10	6	100	100000
D3	48.72	1458	0	6	4-12	100-200	150000
D20	1197.71	1600	0	10	1	25-500	100000-200000
D21	330.00	3120	0	10	10-300	250-1000	100000-150000
D30	354.37	1800	266	100	2-15	13-244	44322-246377
D31	1236.34	8488	0	100	2-15	13-244	44322-246377
D40	1096.76	5400	0	100	2-15	13-244	44322-246377

D41	699.18	15000	12	100	2-15	13-244	44322-246377
D50	15943.81	70713	52	97	2-15	14-300	37339-245537
D60	3506.99	73000	0	100	2-15	13-244	44322-246377
D100	2273.61	8000	93	100	2-15	14-300	37339-245537
D110	135.66	320	0	4	5-13	56-203	83758-231821
D5000	2984.41	15000	905	5000	2-15	10-299	26129-249948
D5050	1062.96	5000	5	5000	2-15	10-299	26129-249948
Dataset	Reduction	Bneck	HS	m	ISR	Traffic Scale	DT-Select
D1	1-80	0-20	0-4	1-5	20-80	25-12500	None
D2	1	0-5	0-2	1-3	20-75	0	None
D3	1	0	0	1-3	0	0	None
D20	10-70	0	0	1	0	0	CopyFirst
D21	10-70	1	0	1-2	75	100	CopyFirst
D30	1-73	0-9	0-3	1-5	20-70	25-5250	Opt, CopyFirst
D31	1-73	0-9	0-3	1-5	20-70	25-5250	CopyFirst
D40	1-73	0-9	0-3	1-5	20-70	25-5250	CopyFirst
D41	1-73	0-9	0-3	1-5	20-70	25-5250	CopyFirst
D50	1-69	0-10	0-4	1-5	20-70	25-8750	CopyFirst
D60	1-73	0-9	0-3	1-5	20-70	25-5250	CopyFirst
D100	1-80	0-10	0-4	1-5	20-70	25-8750	CopyFirst
D110	1-80	0-4	0-1	1-2	40-70	75-500	CopyFirst
D5000	1-80	0-20	0-4	1-5	20-80	25-12500	All three
D5050	1-80	0-20	0-4	1-5	20-80	25-12500	CopyFirst

**Table D.3:** Selected dataset statistics

## D.4 Scripts

This section lists the scripts that were used to extract the presented results from the datasets. The first column denotes the result (pointer to a figure in the thesis). The second column denotes the used dataset. The third column denotes the script that was used for analysis. The scripts can be found here: <https://github.com/kit-tm/fdeval/tree/master/plotter>. The list is sorted in ascending order by the first column so

that it is easy to find the relevant combination of dataset and script for each figure in the thesis.

Figures	Dataset	Used script
12.3	D3	agg_01_d003_topology_examples.py
12.4 to 12.8	D2	agg_01_d002_scenario_examples.py
12.9 to 12.11, B.1, B.2	D1	agg_01_d001_scenario_selection.py
13.1	D100	agg_01_d100_case_study_selection.py
13.2 to 13.13	D110	agg_01_d110_case_study_examples.py
14.1, 14.3, 14.5, 14.7, 14.8	D5000	agg_01_d100_d5000_performance.py
14.2, 14.4, 14.6	D100	agg_01_d100_d5000_performance.py
14.9 to 14.15	D5000	agg_01_d5000_over_underutil.py
15.1	D110	agg_01_d110_case_study_examples.py
15.2 to 15.7	D5000	agg_01_d5000_overhead.py
16.1 to 16.5	D5050	agg_01_d5050_runtime.py
16.6 to 16.9	D20	agg_01_d020_scale_templates.py
16.10	D21	agg_01_d021_scale_switches.py
A.1	D30	agg_01_d030_execution_time_analysis.py
A.2, A.3	D30	agg_01_d030_lookahead_dts_runtime.py
A.4, A.6	D31	agg_01_d031_lookahead_dts_overhead.py
A.5	D40	agg_01_d040_lookahead_rsa_runtime.py
A.7, A.8	D41	agg_01_d041_lookahead_rsa_overhead.py
A.9	D50	agg_01_d050_weights_dts.py
A.10	D60	agg_01_d060_weights_rsa.py
B.3 to B.5	D5000	agg_01_d100_d5000_performance.py
B.6 to B.9	D100	agg_01_d100_bottlenecked_situations.py
C.1 to C.10	D5000	agg_01_d100_d5000_performance.py

**Table D.4:** *Scripts used for the analysis*

## Appendix E

# Terminology

The following table gives an overview of the most important parameters and variables used in this work, ordered by category.

Infrastructure		
$S$	Set of all switches in the infrastructure	
$s$	A single switch $s \in S$ , usually refers to a delegation switch (with bottleneck)	Def. 2.2
$r$	A single switch $r \in S$ , always refers to a remote switch (with free capacity)	Def. 2.2
$y_{s \rightarrow r}$	Binary variable, set to 1 if a direct physical link exists between $s$ and $r$	Sec. 4.1
$c_s^{\text{Table}}$	Flow table capacity of switch $s$	Def. 2.11
$c_{s \rightarrow r}^{\text{Link}}$	Capacity of link between switch $s$ and switch $r$	
$u_{r,t}^{\text{Table}}$	Flow table utilization of switch $r$ in time slot $t$ (always given without flow delegation, i.e., aggregation, backflow and remote rules are not considered)	Def. 2.12
$u_{s \rightarrow r,t}^{\text{Link}}$	Link utilization between $s$ and $r$ in time slot $t$ in bits/s (always given without flow delegation, i.e., traffic from aggregation and remote rules is not considered)	Sec. 4.2.3
Flow rules		
$F$	Generic set of flow rules	
$M$	Generic set of matches	
$F_{s,t}$	Set of flow rules associated with switch $s$ in time slot $t$ (represents flow table of $s$ )	
$M_{s,t}$	Set of all matches contained in $F_{s,t}$	
$M_{s,t}^C$	Set of matches representing the cover set of an aggregation match $\vec{m}_{\text{agg}}$ with respect to a set of matches $M$	Def. 3.4
$F_{s,t}^{\text{CS}}$	Set of flow rules representing a conflict free cover set	Def. 3.6
$f$	A single flow rule $f = \langle \vec{m}, \vec{a}, \text{prio} \rangle$	Def. 2.8

$f_{\text{agg}}$	A single aggregation rule $f^{\text{agg}} = \langle \vec{m}_{\text{agg}}, \vec{a}_{\text{agg}}, \text{prio}_{\text{agg}} \rangle$	Def. 3.2
$\vec{m}$	Match of a single flow rule, $\vec{m} = \langle \langle k_1, v_1 \rangle, \dots, \langle k_n, v_n \rangle \rangle^T$	Def. 2.6
$\vec{m}_{\text{agg}}$	Aggregation match that can be used to create an aggregation rule	Def. 3.1
$k_i$	Packet header field used in a match	Def. 2.3
$v_i$	Value taken by a packet header field, can be a wildcard (*)	Def. 2.6
$\vec{a}$	Action of a single flow rule, $\vec{a} = (a_1(), \dots, a_m())^T$	Def. 2.7
$\text{prio}$	Priority value of a single flow rule	Def. 2.8
$\text{prio}_{\text{agg}}$	Priority value of an aggregation rule (aggregation priority)	Def. 3.2
$\lambda_{f,t}^a$	Binary variable, set to 1 if flow rule $f$ is active in time slot $t$	Def. 4.2
$\lambda_{f,t}^i$	Binary variable, set to 1 if flow rule $f$ was installed in time slot $t$	Def. 4.2
$\phi_{f,s}$	Binary variable, set to 1 if $f$ has a forwarding action that forwards packets to $s$	Sec. 4.2.3
$\delta_{f,t}$	Processed bits by flow rule $f$ in time slot $t$ in bits/s	Sec. 4.2.3
<b>Time slots (see Sec. 2.1.1)</b>		
$T$	Set of consecutive time slots $T = \{t_1, \dots, t_m\}$	
$t$	One specific time slot (window between time slot $t_{-1}$ and time slot $t$ )	
$t_0$	Time slot prior to $t_1$ , i.e., $t_0 \notin T$	
$t_1$	First time slot in set $T$	
$t_m$	Last time slot in set $T$	
$t_x$	$x$ -th time slot in $T$	
$t_{-x}$	$x$ -th time slot prior to $t$ , i.e., $t_3$ if $t = t_4$ and $x = 1$	
$t_{+x}$	$x$ -th time slot after $t$ , i.e., $t_5$ if $t = t_4$ and $x = 1$	
$q, v$	$q$ and $v$ also represent time slots, used as iterator variables	
<b>Delegation templates</b>		
$D_s$	Set of delegation templates calculated for delegation switch $s$ ; The notation without time slot can be used because indirect rule aggregation results in the same delegation templates in each time slot (with respect to the aggregation match $\vec{m}_d$ )	
$D_{s,t}$	Set of delegation templates for switch $s$ in time slot $t$ (input for single period DT-Select problem, see Def. 9.2)	Sec. 9.4.5
$D_{s,t}^*$	Set of selected delegation templates for switch $s$ in time slot $t$ (output of single period DT-Select problem, see Def. 9.2)	Sec. 9.4.5
$D_{s,T}$	Sets of delegation templates for switch $s$ for set $T$ (input for multi period DT-Select problem, see Def. 9.3)	Sec. 9.4.6
$D_{s,T}^*$	Sets of selected delegation templates for switch $s$ for set $T$ (output of multi period DT-Select problem, see Def. 9.3)	Sec. 9.4.6
$D_t^{**}$	Set of selected and allocated delegation templates for switch $s$ in time slot $t$ (output of single period RS-Alloc problem, see Def. 9.7)	Sec. 9.5.5

$D_T^{**}$	Sets of selected and allocated delegation templates for switch $s$ for set $T$ (output of multi period RS-Alloc problem, see Def. 9.8)	Sec. 9.5.6
$d$	One single delegation template $d = \langle \vec{m}_d, F_{d,t} \rangle$	Def. 3.7
$\vec{m}_d$	Aggregation match associated with a delegation template $d$	Def. 3.7
$F_{d,t}$	Set of flow rules representing the conflict-free cover set of delegation template $d$ in time slot $t$	Def. 3.7
<b>DT-Select</b>		
$X_{d,t}$	Binary decision variable. Delegation template $d$ is selected in time slot $t$	Def. 9.1
$u_{d,t}^{\text{Table}}$	Utilization of delegation template $d$ in time slot $t$ . Amount of flow rules that can be relocated to the remote switch minus aggregation and backflow rules (overhead)	Sec. 9.4.3
$w_{d,t}$	Combined cost coefficient (includes all other DT-Select cost coefficients)	Sec. 9.4.4
$w_{d,t}^{\text{Table}}$	Rule overhead of delegation template $d$ in time slot $t$ , i.e., amount of aggregation rules in the flow table	Sec. 9.4.4.1
$w_{d,t}^{\text{Link}}$	Link overhead of delegation template $d$ in time slot $t$ , i.e., bits/s that must be relocated	Sec. 9.4.4.2
$w_{d,t}^{\text{Ctrl}}$	Control message overhead if delegation template $d$ is selected in time slot $t$	Sec. 9.4.4.3
<b>DT-Select with assignments (Select-Opt)</b>		
$L$	Look ahead factor. A limited future horizon of $L$ time slots is considered in the multi period problem (same parameter for DT-Select and RS-Alloc)	Sec. 10.2.1.2
$\boxed{0} \boxed{1}$	Represents a delegation template selection that is encoded in an assignment, i.e., this is not a decision made by a decision variable but a fixed value inside an assignment (see $a_t$ below)	Sec. 10.2.1.1
$A_{d,T}$	Assignment set $A_{d,T} = \{A_1, A_2, \dots\}$ for a single delegation template, $ A_{d,T}  = 2^{ T }$	Def. 10.1
$A_i$	Single assignment $A_i = \langle a_1, \dots, a_m \rangle$ for DT-Select with $a_t \in \{\boxed{0}, \boxed{1}\}$ , $m =  T $	Def. 10.1
$a_t$	Value of assignment $A_i$ in time slot $t$ (selected = $\boxed{1}$ , not selected = $\boxed{0}$ ). Follows the time slot notation, i.e., $a_{t+1}$ represents the value of assignment $A_i$ in time slot $t+1$ (can also be used with $t_1, t_m$ etc)	Def. 10.1
$H_s$	History $H_s = \langle H_s^X, H_s^F \rangle$ of the last optimization period for switch $s$ (required for periodic optimization, see Sec. 10.2.1.2)	Def. 10.2
$H_d^X$	Binary variable. Set to 1 if delegation template $d$ was selected in the last optimization period (in time slot $t_0$ )	Def. 10.2
$H_d^F$	Set of all active flow rules that are relocated to the remote switch in the last optimization period (in time slot $t_0$ )	Def. 10.2
$X_{d,A}$	Binary decision variable for Select-Opt. Assignment $A$ is selected	Def. 10.3
$u_{d,t,A}^{\text{Table}}$	Utilization of delegation template $d$ in time slot $t$ for assignment $A$	Sec. 10.2.3
$w_{d,A}^{\text{Table}}$	Rule overhead cost coefficient for delegation template $d$ and assignment $A$	Sec. 10.2.4.1
$w_{d,A}^{\text{Link}}$	Link overhead cost coefficient for delegation template $d$ and assignment $A$	Sec. 10.2.4.2
$w_{d,A}^{\text{Ctrl}}$	Control message overhead for delegation template $d$ and assignment $A$	Sec. 10.2.4.3

$w_{d,t,A}^{\overline{0}\overline{1}}$	Example of a helper variable used for calculating the three above cost coefficients. Represents the cost for a case with two consecutive time slots $t$ and $t+1$ where $a_t$ (the value of the assignment in time slot $t$ ) is set to $\overline{0}$ and $a_{t+1}$ is set to $\overline{1}$ . The first index is usually colored grey because the cost is associated with the second time slot (in this case when switching from $\overline{0}$ to $\overline{1}$ ).	Sec. 10.2.4
<b>DT-Select with restricted assignments (Select-CopyFirst)</b>		
$X_d^{\overline{0}}$	First binary decision variable for Select-CopyFirst. $H_d^X = \overline{0}$ and template $d$ is selected in time slot $t_1$	Def. 10.5
$X_d^{\overline{1}}$	Second binary decision variable for Select-CopyFirst. $H_d^X = \overline{1}$ and template $d$ is selected in time slot $t_1$	Def. 10.5
$u_{d,t}^{\overline{0}\overline{1}}$	Utilization for delegation template $d$ for case with $H_d^X = \overline{0}$ and $X_d^{\overline{0}} = 1$ (example, the other utilization coefficients are defined similarly)	Sec. 10.3.3
$w_d^{\overline{0}\overline{1}}$	Cost coefficient for delegation template $d$ for case with $H_d^X = \overline{0}$ and $X_d^{\overline{0}} = 1$ (example, each of the three cost coefficients for rule overhead, link overhead and control message overhead is defined with four of these variables)	Sec. 10.3.4
<b>RS-Alloc</b>		
$J_t$	All allocation jobs (of all switches) in time slot $t$	Def. 9.4
$s_j$	Delegation switch associated with allocation job $j$ in time slot $t$	Def. 9.4
$d_{j,t}$	Delegation template associated with allocation job $j$	Def. 9.4
$F_{j,t}$	Conflict-free cover set associated with $j$ in time slot $t$ (from $d_{j,t}$ )	Def. 9.4
$\vec{m}_j$	Aggregation match associated with $j$ (from $d_{j,t}$ )	Def. 9.4
$R_{j,t}$	Allocation set for allocation job $j$ in time slot $t$ . Contains all possible remote switch options (and a backup switch $s_B$ )	Def. 9.5
$s_B$	Backup switch with infinite resources to assure the problem is always feasible	Def. 9.5
$Y_{j \rightarrow r,t}$	Binary decision variable for RS-Alloc. Remote switch $r$ is allocated to job $j$ in time slot $t$	Def. 9.6
$u_{j,t}^{\text{Table}}$	Utilization coefficient. Flow table utilization demand of allocation job in time slot $t$	Sec. 9.5.3
$u_{j,t}^{\text{Link}}$	Utilization coefficient. Link bandwidth demand of allocation job in time slot $t$	Sec. 9.5.3
$w_{j \rightarrow r,t}$	Combined cost coefficient (includes all other RS-Alloc cost coefficients)	Sec. 9.5.4
$w_{j \rightarrow r,t}^{\text{Table}}$	Cost coefficient. Free flow table capacity at remote switch $r$ in time slot $t$	Sec. 9.5.4.1
$w_{j \rightarrow r,t}^{\text{Link}}$	Cost coefficient. Free link bandwidth between $s_j$ and $r$ in time slot $t$	Sec. 9.5.4.2
$w_{j \rightarrow r,t}^{\text{Ctrl}}$	Cost coefficient. Control messages necessary if $r$ is allocated to $j$ in time slot $t$	Sec. 9.5.4.3
$w_{r,t}^{\text{Static}}$	Cost coefficient. Static cost for using remote switch $r$ in time slot $t$	Sec. 9.5.4.4
$\omega_{\text{RSA}}$	Weights to balance the different cost coefficients against each other	Sec. 9.5.5
<b>RS-Alloc with assignments (Alloc-Opt)</b>		
$T_j$	Allocation job $j$ is active in all time slots in $T_j$ , i.e., $J_t$ contains a delegation template with aggregation match $\vec{m}_j$ for delegation switch $s_j$ in all time slots in $T_j$	Def. 11.1
$J_T$	Contains all allocation jobs that are active in one of the time slots in $T$	Def. 11.2



$A_{j,T_j}$	Allocation assignment set for allocation job $j$ and time slots $T_j$	Def. 11.3
$A_i$	Single allocation assignment $A_i = \{a_1, \dots, a_m\}$ for RS-Alloc with $a_t \in R_{j,t}$ , $m =  T $	Def. 11.3
$a_t$	Value of assignment $A_i$ in time slot $t$ (the allocated remote switch for delegation template $d_{j,t}$ ). Follows the time slot notation, i.e., $a_{t+1}$ represents the value of assignment $A_i$ in time slot $t+1$ (can also be used with $t_1, t_m$ etc)	Def. 11.3
$\alpha_{j \rightarrow r,t}$	Binary variable, set to 1 if remote switch $r$ is used in time slot $t$ of allocation assignment $A$ (required because $a_t$ is not binary)	Def. 11.6
$H_j$	History $H_j = \langle H_j^X, H_j^F, H_j^r \rangle$ of the last optimization period for allocation job $j$ (required for periodic optimization, see Sec. 11.2.1.3)	Def. 11.4
$H_j^X$	Binary variable. Set to 1 if allocation job $j$ was active in the last optimization period (in time slot $t_0$ )	Def. 11.4
$H_j^F$	Set of all active flow rules that are relocated to the remote switch in the last optimization period (in time slot $t_0$ )	Def. 11.4
$H_j^r$	The remote switch that was allocated to $d_{j,t_0}$ in the last optimization period (in time slot $t_0$ )	Def. 11.4
$Y_{j,A}$	Binary decision variable for Alloc-Opt. Allocation assignment $A$ is selected	Def. 11.5
$u_{j,t}^{\text{Table}}$	Utilization coefficient. Flow table utilization demand of allocation job in time slot $t$ (same variable, but updated for the multi period problem)	Sec. 11.2.3
$u_{j,t}^{\text{Link}}$	Utilization coefficient. Link bandwidth demand of allocation job in time slot $t$ (same variable, but updated for the multi period problem)	Sec. 11.2.3
$w_{j,A}$	Combined cost coefficient (includes all other RS-Alloc cost coefficients based on assignments)	Sec. 11.2.4
$w_{j,A}^{\text{Table}}$	Cost coefficient. Free flow table capacity at remote switch $a_t$ in the $t$ -th time slot of allocation assignment $A$ for all $t \in T_j$ ( $w_{j \rightarrow r,t}^{\text{Table}}$ defined for allocation assignments)	Sec. 11.2.4
$w_{j,A}^{\text{Link}}$	Cost coefficient. Free link bandwidth between $s_j$ and remote switch $a_t$ in the $t$ -th time slot of allocation assignment $A$ for all $t \in T_j$ ( $w_{j \rightarrow r,t}^{\text{Link}}$ defined for allocation assignments)	Sec. 11.2.4
$w_{j,A}^{\text{Ctrl}}$	Cost coefficient. Control messages necessary if $r$ is allocated to the remote switch $a_t$ in the $t$ -th time slot of allocation assignment $A$ for all $t \in T_j$ . This includes cost if the remote switch is changed within the assignment, i.e., two or more different remote switches are used ( $w_{j \rightarrow r,t}^{\text{Ctrl}}$ defined for allocation assignments)	Sec. 11.2.4
$w_A^{\text{Static}}$	Cost coefficient. Static cost for using remote switch $a_t$ in the $t$ -th time slot of allocation assignment $A$ for all $t \in T_j$ ( $w_{r,t}^{\text{Static}}$ defined for allocation assignments)	Sec. 11.2.4

**Table E.1:** *Used terminology*



# Bibliography

---

- [AB02] R. Albert and A.-L. Barabási. “Statistical mechanics of complex networks.” In: *Rev. Mod. Phys.* 74 (1 Jan. 2002), pp. 47–97. DOI: 10.1103/RevModPhys.74.47. URL: <https://link.aps.org/doi/10.1103/RevModPhys.74.47>.
- [Aga+14] K. Agarwal et al. “Shadow MACs: Scalable Label-switching for Commodity Ethernet.” In: *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*. HotSDN ’14. Chicago, Illinois, USA: ACM, 2014, pp. 157–162. ISBN: 978-1-4503-2989-7. DOI: 10.1145/2620728.2620758. URL: <http://doi.acm.org/10.1145/2620728.2620758>.
- [Alv+17] R. Alvizu et al. “Comprehensive Survey on T-SDN: Software-Defined Networking for Transport Networks.” In: *IEEE Communications Surveys Tutorials* 19.4 (Fourthquarter 2017), pp. 2232–2283. DOI: 10.1109/COMST.2017.2715220.
- [AMA19] M. Alsaeedi, M. M. Mohamad, and A. A. Al-Roubaiey. “Toward Adaptive and Scalable OpenFlow-SDN Flow Control: A Survey.” In: *IEEE Access* 7 (2019), pp. 107346–107379. DOI: 10.1109/ACCESS.2019.2932422.
- [And+14] C. J. Anderson et al. “NetKAT: Semantic Foundations for Networks.” In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’14. San Diego, California, USA: ACM, 2014, pp. 113–126. ISBN: 978-1-4503-2544-8. DOI: 10.1145/2535838.2535862. URL: <http://doi.acm.org/10.1145/2535838.2535862>.
- [AO15] B. Assefa and O. Ozkasap. “State-of-the-art energy efficiency approaches in software defined networking.” In: *ICN 2015* (Jan. 2015).

- [AÖ19] B. G. Assefa and Ö. Özkasap. “A survey of energy efficiency in SDN: Software-based methods and optimization models.” In: *Journal of Network and Computer Applications* 137 (2019), pp. 127–143. ISSN: 1084-8045. DOI: <https://doi.org/10.1016/j.jnca.2019.04.001>. URL: <http://www.sciencedirect.com/science/article/pii/S1084804519301134>.
- [AP12] A. B. and Alberto Dainotti and A. Pescapè. “A tool for the generation of realistic network workload for emerging networking scenarios.” In: *Computer Networks* 56.15 (2012), pp. 3531–3547.
- [Ars+18] I. Arsovski et al. “1.4Gsearch/s 2-Mb/mm<sup>2</sup> TCAM Using Two-Phase-Pre-Charge ML Sensing and Power-Grid Pre-Conditioning to Reduce Ldi/dt Power-Supply Noise by 5%.” In: *IEEE Journal of Solid-State Circuits* 53.1 (Jan. 2018), pp. 155–163. ISSN: 0018-9200. DOI: [10.1109/JSSC.2017.2739178](https://doi.org/10.1109/JSSC.2017.2739178).
- [ARS16] M. Agiwal, A. Roy, and N. Saxena. “Next Generation 5G Wireless Networks: A Comprehensive Survey.” In: *IEEE Communications Surveys Tutorials* 18.3 (thirdquarter 2016), pp. 1617–1655. DOI: [10.1109/COMST.2016.2532458](https://doi.org/10.1109/COMST.2016.2532458).
- [BAM10a] T. Benson, A. Akella, and D. A. Maltz. “Network Traffic Characteristics of Data Centers in the Wild.” In: *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*. IMC ’10. Melbourne, Australia: ACM, 2010, pp. 267–280. ISBN: 978-1-4503-0483-2. DOI: [10.1145/1879141.1879175](https://doi.org/10.1145/1879141.1879175). URL: <http://doi.acm.org/10.1145/1879141.1879175>.
- [BAM10b] T. Benson, A. Akella, and D. A. Maltz. “Network Traffic Characteristics of Data Centers in the Wild.” In: *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*. IMC ’10. Melbourne, Australia: ACM, 2010, pp. 267–280. ISBN: 978-1-4503-0483-2. DOI: [10.1145/1879141.1879175](https://doi.org/10.1145/1879141.1879175). URL: <http://doi.acm.org/10.1145/1879141.1879175>.
- [Bau20] R. Bauer. *Data Set Collection for Flow Delegation*. 2020. DOI: [10.5445/IR/1000120288](https://doi.org/10.5445/IR/1000120288).
- [BD17] R. Bauer and A. Dittebrandt. “Towards resource-aware flow delegation for software-defined networks.” In: *2017 International Conference on Networked Systems (NetSys)*. Mar. 2017, pp. 1–6. DOI: [10.1109/NetSys.2017.7903966](https://doi.org/10.1109/NetSys.2017.7903966).
- [BDZ19] R. Bauer, A. Dittebrandt, and M. Zitterbart. “GCMI: A Generic Approach for SDN Control Message Interception.” In: *NetSoft 2019*. July 2019.
- [Bei+15] A. Beifus et al. “A Study of Networking Software Induced Latency.” In: *International Conference and Workshops on Networked Systems (NetSys)*. IEEE. 2015.

- 
- [BGW10] S. Borst, V. Gupta, and A. Walid. “Distributed Caching Algorithms for Content Distribution Networks.” In: *2010 Proceedings IEEE INFOCOM*. Mar. 2010, pp. 1–9. DOI: 10.1109/INFCOM.2010.5461964.
- [BH12] A. Bremner-Barr and D. Hendler. “Space-Efficient TCAM-Based Classification Using Gray Coding.” In: *IEEE Transactions on Computers* 61.1 (Jan. 2012), pp. 18–30. DOI: 10.1109/TC.2010.267.
- [BK16] N. Bizanis and F. A. Kuipers. “SDN and Virtualization Solutions for the Internet of Things: A Survey.” In: *IEEE Access* 4 (2016), pp. 5591–5606. DOI: 10.1109/ACCESS.2016.2607786.
- [BM14] W. Braun and M. Menth. “Wildcard Compression of Inter-Domain Routing Tables for OpenFlow-Based Software-Defined Networking.” In: *2014 Third European Workshop on Software Defined Networks*. Sept. 2014, pp. 25–30. DOI: 10.1109/EWSN.2014.23.
- [BM15] R. Bifulco and A. Matsiuk. “Towards Scalable SDN Switches: Enabling Faster Flow Table Entries Installation.” In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. SIGCOMM ’15. London, United Kingdom: ACM, 2015, pp. 343–344. ISBN: 978-1-4503-3542-3. DOI: 10.1145/2785956.2790008. URL: <http://doi.acm.org/10.1145/2785956.2790008>.
- [BMJ19] S. Bera, S. Misra, and A. Jamalipour. “FlowStat: Adaptive Flow-Rule Placement for Per-Flow Statistics in SDN.” In: *IEEE Journal on Selected Areas in Communications* 37.3 (Mar. 2019), pp. 530–539. DOI: 10.1109/JSAC.2019.2894239.
- [BOE17] A. C. Baktir, A. Ozgovde, and C. Ersoy. “How Can Edge Computing Benefit From Software-Defined Networking: A Survey, Use Cases, and Future Directions.” In: *IEEE Communications Surveys Tutorials* 19.4 (Fourthquarter 2017), pp. 2359–2391. DOI: 10.1109/COMST.2017.2717482.
- [BP12] Z. Bozakov and P. Papadimitriou. “AutoSlice: Automated and Scalable Slicing for Software-defined Networks.” In: *Proceedings of the 2012 ACM Conference on CoNEXT Student Workshop*. CoNEXT Student ’12. Nice, France: ACM, 2012, pp. 3–4. ISBN: 978-1-4503-1779-5. DOI: 10.1145/2413247.2413251. URL: <http://doi.acm.org/10.1145/2413247.2413251>.
- [Bu+18] K. Bu et al. “FlowCloak: Defeating Middlebox-Bypass Attacks in Software-Defined Networking.” In: *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*. Apr. 2018, pp. 396–404. DOI: 10.1109/INFOCOM.2018.8486230.

- [BZ16] R. Bauer and M. Zitterbart. “Port Based Capacity Extensions (PBCEs): Improving SDNs Flow Table Scalability.” In: *2016 28th International Teletraffic Congress (ITC 28)*. Vol. 01. Sept. 2016, pp. 225–233. DOI: 10.1109/ITC-28.2016.139.
- [Car18] Carmelo Cascone. *P4 Runtime support in ONOS*. Feb. 2018. URL: <https://wiki.onosproject.org/display/ONOS/%20%5Cnewline%20P4+Runtime+support+in+ONOS%7D>.
- [Che+18] T. Cheng et al. “An In-Switch Rule Caching and Replacement Algorithm in Software Defined Networks.” In: *2018 IEEE International Conference on Communications (ICC)*. May 2018, pp. 1–6. DOI: 10.1109/ICC.2018.8422992.
- [Cho+19] A. Chowdhary et al. “SDNSOC: Object Oriented SDN Framework.” In: *Proceedings of the ACM International Workshop on Security in Software Defined Networks &#38; Network Function Virtualization*. SDN-NFVSec ’19. Richardson, Texas, USA: ACM, 2019, pp. 7–12. ISBN: 978-1-4503-6179-8. DOI: 10.1145/3309194.3309196. URL: <http://doi.acm.org/10.1145/3309194.3309196>.
- [CLC16] R. Challa, Y. Lee, and H. Choo. “Intelligent eviction strategy for efficient flow table management in OpenFlow Switches.” In: *2016 IEEE NetSoft Conference and Workshops (NetSoft)*. June 2016, pp. 312–318. DOI: 10.1109/NETSOFT.2016.7502427.
- [Coh+14a] R. Cohen et al. “On the effect of forwarding table size on SDN network utilization.” In: *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*. Apr. 2014, pp. 1734–1742.
- [Coh+14b] R. Cohen et al. “On the effect of forwarding table size on SDN network utilization.” In: *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*. Apr. 2014, pp. 1734–1742. DOI: 10.1109/INFOCOM.2014.6848111.
- [Cos+17] L. C. Costa et al. “Performance evaluation of OpenFlow data planes.” In: *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. May 2017, pp. 470–475. DOI: 10.23919/INM.2017.7987314.
- [Cur+11] A. R. Curtis et al. “DevoFlow: Scaling Flow Management for High-performance Networks.” In: *Proceedings of the ACM SIGCOMM 2011 Conference*. SIGCOMM ’11. Toronto, Ontario, Canada: ACM, 2011, pp. 254–265. ISBN: 978-1-4503-0797-0. DOI: 10.1145/2018436.2018466. URL: <http://doi.acm.org/10.1145/2018436.2018466>.

- 
- [CYP18] C. Chuang, Y. Yu, and A. Pang. “Flow-Aware Routing and Forwarding for SDN Scalability in Wireless Data Centers.” In: *IEEE Transactions on Network and Service Management* 15.4 (Dec. 2018), pp. 1676–1691. DOI: 10.1109/TNSM.2018.2865166.
- [CYY16] L. Cui, F. R. Yu, and Q. Yan. “When big data meets software-defined networking: SDN for big data and big data for SDN.” In: *IEEE Network* 30.1 (Jan. 2016), pp. 58–65. DOI: 10.1109/MNET.2016.7389832.
- [Din+17] X. Ding et al. “Unified nvTCAM and sTCAM Architecture for Improving Packet Matching Performance.” In: *Proceedings of the 18th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*. LCTES 2017. Barcelona, Spain: ACM, 2017, pp. 91–100. ISBN: 978-1-4503-5030-3. DOI: 10.1145/3078633.3081034. URL: <http://doi.acm.org/10.1145/3078633.3081034>.
- [Don+06] Q. Dong et al. “Packet Classifiers in Ternary CAMs Can Be Smaller.” In: *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS ’06/Performance ’06. Saint Malo, France: ACM, 2006, pp. 311–322. ISBN: 1-59593-319-0. DOI: 10.1145/1140277.1140313. URL: <http://doi.acm.org/10.1145/1140277.1140313>.
- [Emm+18] P. Emmerich et al. “Throughput and Latency of Virtual Switching with Open vSwitch: A Quantitative Analysis.” In: *Journal of Network and Systems Management* 26.2 (2018).
- [Enn06] R. Enns. *NETCONF Configuration Protocol*. RFC 4741 (Proposed Standard). RFC. Obsoleted by RFC 6241. Fremont, CA, USA: RFC Editor, Dec. 2006. DOI: 10.17487/RFC4741. URL: <https://www.rfc-editor.org/rfc/rfc4741.txt>.
- [Fos+10] N. Foster et al. “Frenetic: A High-level Language for OpenFlow Networks.” In: *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow*. PRESTO ’10. Philadelphia, Pennsylvania: ACM, 2010, 6:1–6:6. ISBN: 978-1-4503-0467-2. DOI: 10.1145/1921151.1921160. URL: <http://doi.acm.org/10.1145/1921151.1921160>.
- [FRZ14] N. Feamster, J. Rexford, and E. Zegura. “The Road to SDN: An Intellectual History of Programmable Networks.” In: *SIGCOMM Comput. Commun. Rev.* 44.2 (Apr. 2014), pp. 87–98. ISSN: 0146-4833. DOI: 10.1145/2602204.2602219. URL: <http://doi.acm.org/10.1145/2602204.2602219>.

- [Gao+09] W. Gao et al. “Multicasting in Delay Tolerant Networks: A Social Network Perspective.” In: *Proceedings of the Tenth ACM International Symposium on Mobile Ad Hoc Networking and Computing*. MobiHoc '09. New Orleans, LA, USA: ACM, 2009, pp. 299–308. ISBN: 978-1-60558-624-3. DOI: 10.1145/1530748.1530790. URL: <http://doi.acm.org/10.1145/1530748.1530790>.
- [Geb+12] S. Gebert et al. “Internet Access Traffic Measurement and Analysis.” In: *Proceedings of the 4th International Conference on Traffic Monitoring and Analysis*. TMA'12. Vienna, Austria: Springer-Verlag, 2012, pp. 29–42. ISBN: 9783642285332. DOI: 10.1007/978-3-642-28534-9\_3. URL: [https://doi.org/10.1007/978-3-642-28534-9\\_3](https://doi.org/10.1007/978-3-642-28534-9_3).
- [Gei+17] S. Geissler et al. “TableVisor 2.0: Towards Full-Featured, Scalable and Hardware-Independent Multi Table Processing.” In: *NetSoft 2017*. Feb. 2017.
- [Gei+19] S. Geissler et al. “The Power of Composition: Abstracting a Multi-Device SDN Data Path Through a Single API.” In: *IEEE Transactions on Network and Service Management* (2019), pp. 1–1. ISSN: 2373-7379. DOI: 10.1109/TNSM.2019.2951834.
- [GJ78] M. R. Garey and D. S. Johnson. ““ Strong ” NP-Completeness Results: Motivation, Examples, and Implications.” In: *J. ACM* 25.3 (July 1978), pp. 499–508. ISSN: 0004-5411. DOI: 10.1145/322077.322090. URL: <http://doi.acm.org/10.1145/322077.322090>.
- [GNY18] K. Gao, T. Nojima, and Y. R. Yang. “Trident: Toward a Unified SDN Programming Framework with Automatic Updates.” In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '18. Budapest, Hungary: ACM, 2018, pp. 386–401. ISBN: 978-1-4503-5567-4. DOI: 10.1145/3230543.3230562. URL: <http://doi.acm.org/10.1145/3230543.3230562>.
- [Guo+15] Z. Guo et al. “JumpFlow: Reducing flow table usage in software-defined networks.” In: *Computer Networks* 92 (2015). Software Defined Networks and Virtualization, pp. 300–315. ISSN: 1389-1286. DOI: <https://doi.org/10.1016/j.comnet.2015.09.030>. URL: <http://www.sciencedirect.com/science/article/pii/S1389128615003503>.
- [Han+12] N. Handigol et al. “Where is the Debugger for My Software-defined Network?” In: *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*. HotSDN '12. Helsinki, Finland: ACM, 2012, pp. 55–60. ISBN: 978-1-4503-1477-0. DOI: 10.1145/2342441.2342453.



- 
- [He+18] C.-H. He et al. “A Zero Flow Entry Expiration Timeout P4 Switch.” In: *Proceedings of the Symposium on SDN Research*. SOSR ’18. Los Angeles, CA, USA: ACM, 2018, 19:1–19:2. ISBN: 978-1-4503-5664-0. DOI: 10.1145/3185467.3190785. URL: <http://doi.acm.org/10.1145/3185467.3190785>.
- [Hel+10] B. Heller et al. “ElasticTree: Saving Energy in Data Center Networks.” In: *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*. NSDI’10. San Jose, California: USENIX Association, 2010, pp. 17–17. URL: <http://dl.acm.org/citation.cfm?id=1855711.1855728>.
- [HLS10] B. Han, J. Leblet, and G. Simon. “Hard multidimensional multiple choice knapsack problems, an empirical study.” In: *Computers & Operations Research* 37.1 (2010), pp. 172–181. ISSN: 0305-0548. DOI: <https://doi.org/10.1016/j.cor.2009.04.006>. URL: <http://www.sciencedirect.com/science/article/pii/S0305054809001166>.
- [HM18] I. Haque and M. Moyeen. “Revive: A Reliable Software Defined Data Plane Failure Recovery Scheme.” In: *2018 14th International Conference on Network and Service Management (CNSM)*. Nov. 2018, pp. 268–274.
- [Hon+18] C.-Y. Hong et al. “B4 and After: Managing Hierarchy, Partitioning, and Asymmetry for Availability and Scale in Google’s Software-defined WAN.” In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM ’18. Budapest, Hungary: ACM, 2018, pp. 74–87. ISBN: 978-1-4503-5567-4. DOI: 10.1145/3230543.3230545. URL: <http://doi.acm.org/10.1145/3230543.3230545>.
- [Hu+14] H. Hu et al. “FLOWGUARD: Building Robust Firewalls for Software-defined Networks.” In: *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*. HotSDN ’14. Chicago, Illinois, USA: ACM, 2014, pp. 97–102. ISBN: 978-1-4503-2989-7. DOI: 10.1145/2620728.2620749.
- [Hua+16a] H. Huang et al. “Cost Minimization for Rule Caching in Software Defined Networking.” In: *IEEE Transactions on Parallel and Distributed Systems* 27.4 (Apr. 2016), pp. 1007–1016. DOI: 10.1109/TPDS.2015.2431684.
- [Hua+16b] J. Huang et al. “Heterogeneous Flow Table Distribution in Software-Defined Networks.” In: *IEEE Transactions on Emerging Topics in Computing* 4.2 (Apr. 2016), pp. 252–261. DOI: 10.1109/TETC.2015.2457333.

- [Hun+18] C. Hung et al. “Heterogeneous Flow Table Integration for Capacity Enhancement in Software-Defined Networks.” In: *2018 International Conference on Computing, Networking and Communications (ICNC)*. Mar. 2018, pp. 832–836. DOI: 10.1109/ICCNC.2018.8390421.
- [ITR15] ITRS. *International Technology Roadmap for Semiconductors 2015 Edition - MORE MOORE*. Tech. rep. ITRS, 2015.
- [Jai+13] S. Jain et al. “B4: Experience with a Globally Deployed Software Defined WAN.” In: *Proceedings of the ACM SIGCOMM Conference*. Hong Kong, China, 2013. URL: <http://cseweb.ucsd.edu/~vahdat/papers/b4-sigcomm13.pdf>.
- [Jan+19] S. Jan et al. “Intelligent Dynamic Timeout for Efficient Flow Table Management in Software Defined Satellite Network.” In: *Wireless and Satellite Systems*. Ed. by M. Jia, Q. Guo, and W. Meng. Cham: Springer International Publishing, 2019, pp. 59–68. ISBN: 978-3-030-19153-5.
- [Jar+14] M. Jarschel et al. “Interfaces, attributes, and use cases: A compass for SDN.” In: *IEEE Communications Magazine* 52.6 (June 2014), pp. 210–217. ISSN: 0163-6804. DOI: 10.1109/MCOM.2014.6829966.
- [Jin+15] X. Jin et al. “CoVisor: A Compositional Hypervisor for Software-defined Networks.” In: *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*. NSDI’15. Oakland, CA: USENIX Association, 2015, pp. 87–101. ISBN: 978-1-931971-218. URL: <http://dl.acm.org/citation.cfm?id=2789770.2789777>.
- [JMD14] Y. Jarraya, T. Madi, and M. Debbabi. “A Survey and a Layered Taxonomy of Software-Defined Networking.” In: *Communications Surveys Tutorials, IEEE* 16.4 (Fourthquarter 2014), pp. 1955–1980. ISSN: 1553-877X. DOI: 10.1109/COMST.2014.2320094.
- [JRB18] P. Jurkiewicz, G. Rzym, and P. Boryło. *How Many Mice Make an Elephant? Modelling Flow Length and Size Distribution of Internet Traffic*. 2018. arXiv: 1809.03486 [cs.NI].
- [JRW14] X. Jin, J. Rexford, and D. Walker. “Incremental Update for a Compositional SDN Hypervisor.” In: *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*. HotSDN ’14. Chicago, Illinois, USA: ACM, 2014, pp. 187–192. ISBN: 978-1-4503-2989-7. DOI: 10.1145/2620728.2620731. URL: <http://doi.acm.org/10.1145/2620728.2620731>.

- 
- [Kan+09] S. Kandula et al. “The Nature of Data Center Traffic: Measurements & Analysis.” In: *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement*. IMC '09. Chicago, Illinois, USA: Association for Computing Machinery, 2009, pp. 202–208. ISBN: 9781605587714. DOI: 10.1145/1644893.1644918. URL: <https://doi.org/10.1145/1644893.1644918>.
- [Kan+13] N. Kang et al. “Optimizing the “One Big Switch” Abstraction in Software-defined Networks.” In: *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*. CoNEXT '13. Santa Barbara, California, USA: ACM, 2013, pp. 13–24. ISBN: 978-1-4503-2101-3. DOI: 10.1145/2535372.2535373. URL: <http://doi.acm.org/10.1145/2535372.2535373>.
- [Kar72] R. M. Karp. “Reducibility among Combinatorial Problems.” In: *Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations, held March 20–22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, and sponsored by the Office of Naval Research, Mathematics Program, IBM World Trade Corporation, and the IBM Research Mathematical Sciences Department*. Ed. by R. E. Miller, J. W. Thatcher, and J. D. Bohlinger. Boston, MA: Springer US, 1972, pp. 85–103. ISBN: 978-1-4684-2001-2. DOI: 10.1007/978-1-4684-2001-2\_9. URL: [https://doi.org/10.1007/978-1-4684-2001-2\\_9](https://doi.org/10.1007/978-1-4684-2001-2_9).
- [Kat+14a] N. P. Katta et al. *Rule-Caching Algorithms for Software-Defined Networks*. Tech. rep. 2014.
- [Kat+14b] N. Katta et al. “Infinite CacheFlow in Software-defined Networks.” In: *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*. HotSDN '14. Chicago, Illinois, USA: ACM, 2014, pp. 175–180. ISBN: 978-1-4503-2989-7. DOI: 10.1145/2620728.2620734. URL: <http://doi.acm.org/10.1145/2620728.2620734>.
- [Kaz+13] P. Kazemian et al. “Real Time Network Policy Checking Using Header Space Analysis.” In: *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*. nsdi'13. Lombard, IL: USENIX Association, 2013, pp. 99–112.
- [KB13] K. Kannan and S. Banerjee. “Compact TCAM: Flow Entry Compaction in TCAM for Power Aware SDN.” English. In: *Distributed Computing and Networking*. Ed. by D. Frey et al. Vol. 7730. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 439–444. ISBN: 978-3-642-35667-4. DOI: 10.1007/978-3-642-35668-1\_32. URL: [http://dx.doi.org/10.1007/978-3-642-35668-1\\_32](http://dx.doi.org/10.1007/978-3-642-35668-1_32).

- [KB14] K. Kannan and S. Banerjee. “FlowMaster: Early Eviction of Dead Flow on SDN Switches.” In: *Distributed Computing and Networking*. Ed. by M. Chatterjee et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 484–498. ISBN: 978-3-642-45249-9.
- [KD17] M. Karakus and A. Durresi. “A survey: Control plane scalability issues and approaches in Software-Defined Networking (SDN).” In: *Computer Networks* 112 (2017), pp. 279–293. ISSN: 1389-1286. DOI: <https://doi.org/10.1016/j.comnet.2016.11.017>. URL: <http://www.sciencedirect.com/science/article/pii/S138912861630411X>.
- [KHK13] Y. Kanizo, D. Hay, and I. Keslassy. “Palette: Distributing tables in software-defined networks.” In: *INFOCOM, 2013 Proceedings IEEE*. Apr. 2013, pp. 545–549. DOI: 10.1109/INFCOM.2013.6566832.
- [Khu+13] A. Khurshid et al. “VeriFlow: Verifying Network-wide Invariants in Real Time.” In: *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*. nsdi’13. Lombard, IL: USENIX Association, 2013, pp. 15–28.
- [Kim+10] W. Kim et al. “Automated and Scalable QoS Control for Network Convergence.” In: *Proceedings of the 2010 Internet Network Management Conference on Research on Enterprise Networking*. INM/WREN’10. San Jose, CA: USENIX Association, 2010, pp. 1–1. URL: <http://dl.acm.org/citation.cfm?id=1863133.1863134>.
- [Kos+17] T. Kosugiyama et al. “A flow aggregation method based on end-to-end delay in SDN.” In: *2017 IEEE International Conference on Communications (ICC)*. May 2017, pp. 1–6. DOI: 10.1109/ICC.2017.7996341.
- [KPK14] M. Kuzniar, P. Peresini, and D. Kostic. *What you need to know about SDN control and data planes*. Tech. rep. 2014.
- [Kre+15] D. Kreutz et al. “Software-Defined Networking: A Comprehensive Survey.” In: *Proceedings of the IEEE* 103.1 (Jan. 2015), pp. 14–76. ISSN: 0018-9219. DOI: 10.1109/JPROC.2014.2371999.
- [Lat+19] Z. Latif et al. *A Comprehensive Survey of Interface Protocols for Software Defined Networks*. 2019. arXiv: 1902.07913 [cs.NI].
- [LC15] Y. Li and M. Chen. “Software-Defined Network Function Virtualization: A Survey.” In: *IEEE Access* 3 (2015), pp. 2542–2553. DOI: 10.1109/ACCESS.2015.2499271.
- [Len+15] B. Leng et al. “A mechanism for reducing flow tables in software defined network.” In: *2015 IEEE International Conference on Communications (ICC)*. June 2015, pp. 5302–5307. DOI: 10.1109/ICC.2015.7249166.

- 
- [LHM10] B. Lantz, B. Heller, and N. McKeown. “A Network in a Laptop: Rapid Prototyping for Software-defined Networks.” In: *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. Hotnets-IX. Monterey, California: ACM, 2010, 19:1–19:6. ISBN: 978-1-4503-0409-2. DOI: 10.1145/1868447.1868466. URL: <http://doi.acm.org/10.1145/1868447.1868466>.
- [Li+15] H. Li et al. “FDRC: Flow-driven rule caching optimization in software defined networking.” In: *2015 IEEE International Conference on Communications (ICC)*. June 2015, pp. 5777–5782. DOI: 10.1109/ICC.2015.7249243.
- [Li+19a] D. Li et al. “Estimating SDN traffic matrix based on online adaptive information gain maximization method.” In: *Peer-to-Peer Networking and Applications* 12.2 (Mar. 2019), pp. 465–480. ISSN: 1936-6450. DOI: 10.1007/s12083-018-0646-0. URL: <https://doi.org/10.1007/s12083-018-0646-0>.
- [Li+19b] Q. Li et al. “HQTimer: A Hybrid Q-Learning-Based Timeout Mechanism in Software-Defined Networks.” In: *IEEE Transactions on Network and Service Management* 16.1 (Mar. 2019), pp. 153–166. DOI: 10.1109/TNSM.2018.2890754.
- [Li+19c] R. Li et al. “A Tale of Two (Flow) Tables: Demystifying Rule Caching in OpenFlow Switches.” In: *Proceedings of the 48th International Conference on Parallel Processing*. ICPP 2019. Kyoto, Japan: ACM, 2019, 59:1–59:10. ISBN: 978-1-4503-6295-5. DOI: 10.1145/3337821.3337896. URL: <http://doi.acm.org/10.1145/3337821.3337896>.
- [LMT10] A. X. Liu, C. R. Meiners, and E. Tornø. “TCAM Razor: A Systematic Approach Towards Minimizing Packet Classifiers in TCAMs.” In: *IEEE/ACM Trans. Netw.* 18.2 (Apr. 2010), pp. 490–500. ISSN: 1063-6692. DOI: 10.1109/TNET.2009.2030188. URL: <http://dx.doi.org/10.1109/TNET.2009.2030188>.
- [Lop+18] F. A. Lopes et al. “Model-based flow delegation for improving SDN infrastructure compatibility.” In: *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*. Apr. 2018, pp. 1–9. DOI: 10.1109/NOMS.2018.8406230.
- [LPW16] Linqi Guo, J. Pang, and A. Walid. “Dynamic Service Function Chaining in SDN-enabled networks with middleboxes.” In: *2016 IEEE 24th International Conference on Network Protocols (ICNP)*. Nov. 2016, pp. 1–10. DOI: 10.1109/ICNP.2016.7784431.

- [Luo+16] L. Luo et al. "Achieving Fast and Lightweight SDN Updates with Segment Routing." In: *2016 IEEE Global Communications Conference (GLOBECOM)*. Dec. 2016, pp. 1–6. DOI: 10.1109/GLOCOM.2016.7841562.
- [LYL15] S. Luo, H. Yu, and L. Li. "Practical flow table aggregation in SDN." In: *Computer Networks* 92 (2015), pp. 72–88. ISSN: 1389-1286. DOI: <https://doi.org/10.1016/j.comnet.2015.09.016>. URL: <http://www.sciencedirect.com/science/article/pii/S1389128615003278>.
- [Maq+15] Q. Maqbool et al. "Virtual TCAM for Data Center switches." In: *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*. Nov. 2015, pp. 61–66. DOI: 10.1109/NFV-SDN.2015.7387407.
- [MDS17] A. Marsico, R. Doriguzzi-Corin, and D. Siracusa. "An effective swapping mechanism to overcome the memory limitation of SDN devices." In: *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. May 2017, pp. 247–254. DOI: 10.23919/INM.2017.7987286.
- [Mic18] Micah Singleton (theverge.com). *SpaceX's Falcon Heavy launch was YouTube's second biggest live stream ever*. 2018.
- [MK17] O. Michel and E. Keller. "SDN in wide-area networks: A survey." In: *2017 Fourth International Conference on Software Defined Systems (SDS)*. May 2017, pp. 37–42. DOI: 10.1109/SDS.2017.7939138.
- [MLT12] C. R. Meiners, A. X. Liu, and E. Tornø. "Bit Weaving: A Non-Prefix Approach to Compressing Packet Classifiers in TCAMs." In: *IEEE/ACM Transactions on Networking* 20.2 (Apr. 2012), pp. 488–500. ISSN: 1063-6692. DOI: 10.1109/TNET.2011.2165323.
- [Moy98] J. Moy. *OSPF Version 2*. RFC 2328 (Internet Standard). RFC. Updated by RFCs 5709, 6549, 6845, 6860, 7474, 8042. Fremont, CA, USA: RFC Editor, Apr. 1998. DOI: 10.17487/RFC2328. URL: <https://www.rfc-editor.org/rfc/rfc2328.txt>.
- [MTG18] P. M. Mohan, T. Truong-Huu, and M. Gurusamy. "Towards resilient in-band control path routing with malicious switch detection in SDN." In: *2018 10th International Conference on Communication Systems Networks (COM-SNETS)*. Jan. 2018, pp. 9–16. DOI: 10.1109/COMSNETS.2018.8328174.
- [MY09] R. McGeer and P. Yalagandula. "Minimizing Rulesets for TCAM Implementation." In: *IEEE INFOCOM 2009*. Apr. 2009, pp. 1314–1322. DOI: 10.1109/INFCOM.2009.5062046.

- 
- [Nar+12] R. Narayanan et al. “Macroflows and Microflows: Enabling Rapid Network Innovation through a Split SDN Data Plane.” In: *2012 European Workshop on Software Defined Networking*. Oct. 2012, pp. 79–84. DOI: 10.1109/EWSDN.2012.16.
- [Ngu+14] X.-N. Nguyen et al. “Optimizing Rules Placement in OpenFlow Networks: Trading Routing for Better Efficiency.” In: *HotSDN 2014 - Proceedings of the ACM SIGCOMM 2014 Workshop on Hot Topics in Software Defined Networking* (Aug. 2014). DOI: 10.1145/2620728.2620753.
- [Ngu+16] X. Nguyen et al. “Rules Placement Problem in OpenFlow Networks: A Survey.” In: *IEEE Communications Surveys Tutorials* 18.2 (Secondquarter 2016), pp. 1273–1286. DOI: 10.1109/COMST.2015.2506984.
- [OA18] R. Ogasawara and M. Arai. “A SAT-Based Approach for SDN Rule Table Distribution.” In: *2018 IEEE 23rd Pacific Rim International Symposium on Dependable Computing (PRDC)*. Dec. 2018, pp. 191–192. DOI: 10.1109/PRDC.2018.00034.
- [OCF19] L. Ochoa-Aday, C. Cervelló-Pastor, and A. Fernández-Fernández. “Self-healing and SDN: bridging the gap.” In: *Digital Communications and Networks* (2019). ISSN: 2352-8648. DOI: <https://doi.org/10.1016/j.dcan.2019.08.008>. URL: <http://www.sciencedirect.com/science/article/pii/S2352864818302827>.
- [ODL19] ODL: OpenDaylight Foundation. *OpenDaylight: a modular open platform for customizing and automating networks of any size and scale*. <https://www.opendaylight.org/>. Accessed: 2019-10-30. 2019.
- [ONF09] ONF. The Open Networking Foundation. *OpenFlow Switch Specification, Version 1.0.0*. Dec. 2009.
- [ONF12] ONF. The Open Networking Foundation. *OpenFlow Switch Specification, Version 1.3.0*. June 2012.
- [ONF15a] ONF. The Open Networking Foundation. *OpenFlow Switch Specification, Version 1.5.1*. Mar. 2015.
- [ONF15b] ONF. The Open Networking Foundation. *The Benefits of Multiple Flow Tables and TTPs*. Feb. 2015.
- [P4o19] P4.org API Working Group. *P4Runtime Specification*. Oct. 2019. URL: [%7Bhttps://p4.org/p4runtime/spec/v1.0.0/P4Runtime-Spec.html%7D](https://p4.org/p4runtime/spec/v1.0.0/P4Runtime-Spec.html%7D).
- [Pap81] C. H. Papadimitriou. “On the Complexity of Integer Programming.” In: *J. ACM* 28.4 (Oct. 1981), pp. 765–768. ISSN: 0004-5411. DOI: 10.1145/322276.322287. URL: <http://doi.acm.org/10.1145/322276.322287>.

- [PD13] B. Pfaff and B. Davie. *The Open vSwitch Database Management Protocol*. RFC 7047 (Informational). RFC. Fremont, CA, USA: RFC Editor, Dec. 2013. DOI: 10.17487/RFC7047. URL: <https://www.rfc-editor.org/rfc/rfc7047.txt>.
- [Pei+18] j. Pei et al. “Resource Aware Routing for Service Function Chains in SDN and NFV-Enabled Network.” In: *IEEE Transactions on Services Computing* (2018), pp. 1–1. DOI: 10.1109/TSC.2018.2849712.
- [Pet14] Peter Kafka (allthingsd.com). *Felix Baumgartner’s Crazy Space Parachute Jump Is Live Web Video’s Biggest Event Ever*. 2014.
- [Pha+19] T. V. Phan et al. *Q-DATA: Enhanced Traffic Flow Monitoring in Software-Defined Networks applying Q-learning*. 2019. arXiv: 1909.01544 [cs.NI].
- [Rif+15] M. Rifai et al. “Too Many SDN Rules? Compress Them with MINNIE.” In: *2015 IEEE Global Communications Conference (GLOBECOM)*. Dec. 2015, pp. 1–7. DOI: 10.1109/GLOCOM.2015.7417661.
- [RK10] O. Rottenstreich and I. Keslassy. “Worst-Case TCAM Rule Expansion.” In: *2010 Proceedings IEEE INFOCOM*. Mar. 2010, pp. 1–5.
- [Ryg+17] P. Rygielski et al. “Performance Analysis of SDN Switches with Hardware and Software Flow Tables.” In: *Proceedings of the 10th EAI International Conference on Performance Evaluation Methodologies and Tools on 10th EAI International Conference on Performance Evaluation Methodologies and Tools*. VALUETOOLS’16. Taormina, Italy: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2017, pp. 80–87. ISBN: 978-1-63190-141-6. DOI: 10.4108/eai.25-10-2016.2266540. URL: <https://doi.org/10.4108/eai.25-10-2016.2266540>.
- [Ryu19] Ryu SDN Framework Community. *Ryu: Component-based Software Defined Networking Framework*. <http://osrg.github.io/ryu/>. Accessed: 2015-03-09. 2019.
- [SBM18] N. Saha, S. Bera, and S. Misra. “Sway: Traffic-Aware QoS Routing in Software-Defined IoT.” In: *IEEE Transactions on Emerging Topics in Computing* (2018), pp. 1–1. DOI: 10.1109/TETC.2018.2847296.
- [SC16] J. P. Sheu and Y. C. Chuo. “Wildcard Rules Caching and Cache Replacement Algorithms in Software-Defined Networking.” In: *IEEE Transactions on Network and Service Management* 13.1 (Mar. 2016), pp. 19–29. ISSN: 1932-4537. DOI: 10.1109/TNSM.2016.2530687.



- 
- [Sha+16] M. Shahbaz et al. “PISCES: A Programmable, Protocol-Independent Software Switch.” In: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM ’16. Florianopolis, Brazil, 2016. ISBN: 978-1-4503-4193-6. DOI: 10.1145/2934872.2934886.
- [She+09] R. Sherwood et al. *FlowVisor: A Network Virtualization Layer*. Tech. rep. 2009.
- [Sho+13] H. Shojaei et al. “A Fast and Scalable Multidimensional Multiple-choice Knapsack Heuristic.” In: *ACM Trans. Des. Autom. Electron. Syst.* 18.4 (Oct. 2013), 51:1–51:32. ISSN: 1084-4309. DOI: 10.1145/2541012.2541014. URL: <http://doi.acm.org/10.1145/2541012.2541014>.
- [SK75] H. M. Salkin and C. A. D. Kluyver. “The knapsack problem: A survey.” In: *Naval Research Logistics (NRL)* 22.1 (Mar. 1975), pp. 127–144. ISSN: 1931-9193. DOI: 10.1002/nav.3800220110. URL: <https://doi.org/10.1002/nav.3800220110>.
- [SLC18] J. Sheu, W. Lin, and G. Chang. “Efficient TCAM Rules Distribution Algorithms in Software-Defined Networking.” In: *IEEE Transactions on Network and Service Management* 15.2 (June 2018), pp. 854–865. DOI: 10.1109/TNSM.2018.2825026.
- [SLX10] Y. Shang, D. Li, and M. Xu. “Energy-aware Routing in Data Center Network.” In: *Proceedings of the First ACM SIGCOMM Workshop on Green Networking*. Green Networking ’10. New Delhi, India: ACM, 2010, pp. 1–8. ISBN: 978-1-4503-0196-1. DOI: 10.1145/1851290.1851292. URL: <http://doi.acm.org/10.1145/1851290.1851292>.
- [TCL18] Y. Tian, W. Chen, and C. Lea. “An SDN-Based Traffic Matrix Estimation Framework.” In: *IEEE Transactions on Network and Service Management* 15.4 (Dec. 2018), pp. 1435–1445. DOI: 10.1109/TNSM.2018.2867998.
- [Tro+16] C. Trois et al. “A Survey on SDN Programming Languages: Toward a Taxonomy.” In: *IEEE Communications Surveys Tutorials* 18.4 (Fourthquarter 2016), pp. 2687–2712. DOI: 10.1109/COMST.2016.2553778.
- [Van+19] A. Van Bemten et al. “Empirical Predictability Study of SDN Switches.” In: *Proceedings of the 15<sup>th</sup> ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. Cambridge, UK, 2019.
- [Vis+14] A. Vishnoi et al. “Effective Switch Memory Management in OpenFlow Networks.” In: *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*. DEBS ’14. Mumbai, India: ACM, 2014, pp. 177–188. ISBN: 978-1-4503-2737-4. DOI: 10.1145/2611286.2611301. URL: <http://doi.acm.org/10.1145/2611286.2611301>.

- [Wan+15] B. Wang et al. “A survey on data center networking for cloud computing.” In: *Computer Networks* 91 (2015), pp. 528–547. ISSN: 1389-1286. DOI: <https://doi.org/10.1016/j.comnet.2015.08.040>. URL: <http://www.sciencedirect.com/science/article/pii/S138912861500300X>.
- [Wan+17] D. Wang et al. “Balancer: A Traffic-Aware Hybrid Rule Allocation Scheme in Software Defined Networks.” In: *2017 26th International Conference on Computer Communication and Networks (ICCCN)*. July 2017, pp. 1–9. DOI: 10.1109/ICCCN.2017.8038395.
- [Wan+19a] J. Wang et al. “FlowTracer: An Effective Flow Trajectory Detection Solution Based on Probabilistic Packet Tagging in SDN-Enabled Networks.” In: *IEEE Transactions on Network and Service Management* (2019), pp. 1–1. DOI: 10.1109/TNSM.2019.2936598.
- [Wan+19b] X. Wang et al. “FlowMap: A Fine-Grained Flow Measurement Approach for Data-Center Networks.” In: *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*. May 2019, pp. 1–7. DOI: 10.1109/ICC.2019.8761390.
- [WBZ16] Y. Wang, J. Bi, and K. Zhang. “A tool for tracing network data plane via SDN/OpenFlow.” In: *Science China Information Sciences* 60.2 (Nov. 2016), p. 022304. ISSN: 1869-1919. DOI: 10.1007/s11432-015-1057-7.
- [Wun+11] A. Wundsam et al. “OFRewind: Enabling Record and Replay Troubleshooting for Networks.” In: *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*. USENIXATC’11. Portland, OR: USENIX Association, 2011, pp. 29–29.
- [WYW19] W. Wang, Y. Yang, and E. Wang. “A Distributed Hierarchical Heavy Hitter Detection Method in Software-Defined Networking.” In: *IEEE Access* 7 (2019), pp. 55367–55381. DOI: 10.1109/ACCESS.2019.2905526.
- [Xu+17] T. Xu et al. “Mitigating the Table-Overflow Attack in Software-Defined Networking.” In: *IEEE Transactions on Network and Service Management* 14.4 (Dec. 2017), pp. 1086–1097. ISSN: 2373-7379. DOI: 10.1109/TNSM.2017.2758796.
- [xu+18] J. xu et al. “Proactive Mitigation to Table-Overflow in Software-Defined Networking.” In: *2018 IEEE Symposium on Computers and Communications (ISCC)*. June 2018, pp. 00719–00725. DOI: 10.1109/ISCC.2018.8538670.
- [XW17] Xianfeng Li and Wencong Xie. “CRAFT: A Cache Reduction Architecture for Flow Tables in Software-Defined Networks.” In: *2017 IEEE Symposium on Computers and Communications (ISCC)*. July 2017, pp. 967–972. DOI: 10.1109/ISCC.2017.8024651.

- 
- [Yan+14] B. Yan et al. “CAB: A Reactive Wildcard Rule Caching System for Software-defined Networks.” In: *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*. HotSDN '14. Chicago, Illinois, USA: ACM, 2014, pp. 163–168. ISBN: 978-1-4503-2989-7. DOI: 10.1145/2620728.2620732. URL: <http://doi.acm.org/10.1145/2620728.2620732>.
- [Yan+16] Yang Liu et al. “A dynamic adaptive timeout approach for SDN switch.” In: *2016 2nd IEEE International Conference on Computer and Communications (ICCC)*. Oct. 2016, pp. 2577–2582. DOI: 10.1109/CompComm.2016.7925164.
- [Yan+18a] B. Yan et al. “Adaptive Wildcard Rule Cache Management for Software-Defined Networks.” In: *IEEE/ACM Trans. Netw.* 26.2 (Apr. 2018), pp. 962–975. ISSN: 1063-6692. DOI: 10.1109/TNET.2018.2815983. URL: <https://doi.org/10.1109/TNET.2018.2815983>.
- [Yan+18b] C. Yang et al. “CNOR: A Non-Overlapping Wildcard Rule Caching System for Software-Defined Networks.” In: *2018 IEEE Symposium on Computers and Communications (ISCC)*. June 2018, pp. 707–712. DOI: 10.1109/ISCC.2018.8538522.
- [YR18] H. Yang and G. F. Riley. “Machine Learning Based Proactive Flow Entry Deletion for OpenFlow.” In: *2018 IEEE International Conference on Communications (ICC)*. May 2018, pp. 1–6. DOI: 10.1109/ICC.2018.8422626.
- [YTG13] S. Yeganeh, A. Tootoonchian, and Y. Ganjali. “On scalability of software-defined networking.” In: *Communications Magazine, IEEE* 51.2 (Feb. 2013), pp. 136–141. ISSN: 0163-6804. DOI: 10.1109/MCOM.2013.6461198.
- [Yu+10] M. Yu et al. “Scalable Flow-based Networking with DIFANE.” In: *Proceedings of the ACM SIGCOMM 2010 Conference*. SIGCOMM '10. New Delhi, India: ACM, 2010, pp. 351–362. ISBN: 978-1-4503-0201-2. DOI: 10.1145/1851182.1851224. URL: <http://doi.acm.org/10.1145/1851182.1851224>.
- [Yu+16] C. Yu et al. “Characterizing Rule Compression Mechanisms in Software-Defined Networks.” In: *Passive and Active Measurement*. Ed. by T. Karagiannis and X. Dimitropoulos. Cham: Springer International Publishing, 2016, pp. 302–315. ISBN: 978-3-319-30505-9.
- [Zha+14] L. Zhang et al. “AHTM: Achieving efficient flow table utilization in Software Defined Networks.” In: *2014 IEEE Global Communications Conference*. Dec. 2014, pp. 1897–1902. DOI: 10.1109/GLOCOM.2014.7037085.

- 
- [Zha+15] L. Zhang et al. “TimeoutX: An Adaptive Flow Table Management Method in Software Defined Networks.” In: *2015 IEEE Global Communications Conference (GLOBECOM)*. Dec. 2015, pp. 1–6. DOI: 10.1109/GLOCOM.2015.7417563.
- [Zha+19] Z. Zhang et al. “How Advantageous Is It? An Analytical Study of Controller-Assisted Path Construction in Distributed SDN.” In: *IEEE/ACM Transactions on Networking* 27.4 (Aug. 2019), pp. 1643–1656. ISSN: 1558-2566. DOI: 10.1109/TNET.2019.2924616.
- [Zhu+15] H. Zhu et al. “Intelligent timeout master: Dynamic timeout for SDN-based data centers.” In: *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. May 2015, pp. 734–737. DOI: 10.1109/INM.2015.7140363.