Karlsruhe Institute of Technology

Distributed Asynchronous Event Relaying for P2P Matrix

Master Thesis
by

Benjamin Schichtholz

at the Institute of Telematics
Department of Informatics

First Examiner:	Prof. Dr. Martina Zitterbart
Second Examiner:	Prof. Dr. Hannes Hartenstein
Supervisor:	PD Dr.-Ing. Roland Bless

Processing Time: 15. January 2024 - 15. July 2024

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Die veröffentlichte Version dieser Arbeit enthält im Vergleich zur eingereichten Fassung einige Korrekturen in Rechtschreibung und Grammatik.

Karlsruhe, den 15. Juli 2024

Contents

1. Introduction
1.1 Problem Formulation 2
1.2 Outline 2

2. Background
2.1 Matrix 3
2.1.1 Protocol Background 4
2.1.2 DAG and Room State 6
2.1.3 Access Control 8
2.2 Peer-to-Peer Networks 8
2.3 Peer-to-Peer Matrix 9
2.3.1 Motivation 10
2.3.2 Existing Implementations 10

3. Analysis
3.1 Problem Statement 13
3.2 Assumptions on P2P Matrix 16
3.3 Requirements 17
3.4 Solution Approaches 19
3.4.1 Relay Functionality Placement 19
3.4.2 Event Graph Storage 20
3.4.3 Relay Integration into Matrix Protocol 22
3.4.4 Privileges for Room Relay Management 22
3.4.5 Peer-to-Relay Relation 23
3.4.6 Forwarding Events to Relays 24
3.5 Differences to Federated Matrix 26
3.6 Solution Approaches in Related Work 27
3.6.1 Relay Functionality 27
3.6.2 Nostr 27
3.6.3 Scuttlebutt 28
3.6.4 Asynchronous Mobile Peer-to-peer Relay 29
3.6.5 Wesh Protocol 29
3.6.6 Dendrite Relay API 29
3.6.7 Briar 30
3.6.8 Comparison 30
3.7 Summary 33

4. Design
4.1 High-Level Overview . . . 35
4.2 Relay Design . . . 37
4.2.1 Relay Overview . . . 37
4.2.2 AuthDAG . . . 38
4.2.3 Event Cache . . . 39
4.2.4 Relay Functionality . . . 40
4.3 Peer Design . . . 43
4.3.1 Peer Overview . . . 44
4.3.2 Peer Functionality . . . 44
4.4 Use Cases . . . 46
4.4.1 Add Relay Process . . . 46
4.4.2 Room Join . . . 47
4.4.3 Relayless Peers . . . 47
4.5 Summary . . . 48

5. Implementation
5.1 Existing Implementations . . . 49
5.1.1 Analysis of Existing Approaches . . . 49
5.1.2 Dendrite . . . 50
5.1.3 Dendrite-Demo-Pinecone . . . 51
5.2 Overview . . . 52
5.3 Relay Functionality . . . 53
5.3.1 AuthDAG . . . 54
5.3.2 Event Cache . . . 54
5.3.3 Receive Events . . . 56
5.3.4 Receive Resync Request . . . 56
5.4 Peer Functionality . . . 57
5.4.1 Relay Selection . . . 57
5.4.2 Send Event . . . 57
5.4.3 Make Resync Request . . . 58
5.5 Summary . . . 58

6. Evaluation
6.1 Evaluation Setup . . . 59
6.1.1 Overview . . . 59
6.1.2 Traffic Generator . . . 60
6.1.3 Docker Network . . . 62
6.1.4 Logging . . . 62
6.1.5 Configuration Files . . . 63
6.1.6 Experiment Process . . . 64
6.2 Performance Evaluation Traffic . . . 64
6.2.1 Overview . . . 65
6.2.2 Online Margins . . . 65
6.2.3 Applying Online Margins to Dataset . . . 67
6.2.4 Data Selection . . . 68
6.3 Evaluation Overview . . . 69
6.4 Functional Evaluation . . . 71

6.4.1 Asynchronous Delivery: Minimal Setup . . . 72
6.4.2 Asynchronous Delivery in Larger Rooms . . . 73
6.4.3 ASAP Delivery . . . 74
6.4.4 Resync with Since Parameter . . . 74
6.4.5 Resync with Different Relays . . . 75
6.4.6 AuthDAG: Access Control . . . 76
6.5 Performance Evaluation . . . 77
6.5.1 Events Not Delivered to Any Peer . . . 78
6.5.2 Event Delivery Time to any First Peer . . . 80
6.5.3 Online Periods without Receiving Events . . . 83
6.5.4 Reception Time after Returning Online . . . 84
6.6 Summary . . . 86

7. Conclusion and Future Work

Bibliography

A. Appendix
A.1. Evaluation Setup . . . 95
A.2. Functional Evaluation Log Tables . . . 96
A.3. Performance Evaluation Plots . . . 98

1. Introduction

Instant messaging applications are frequently used in modern life [1], not only for private communication, but also for institutions such as companies, governments, NGOs, military organizations and secret services [2], [3]. As the usage of these applications increases, security requirements become more critical. Heavily-frequented platforms and instant messaging applications are targeted by various attackers that try to compromise confidentiality, integrity or availability. Today, end-to-end encryption is a basic requirement for any instant messaging application as it ensures confidentiality of message contents. While integrity can also be achieved by various cryptographic means, e.g., message authentication codes or digital signatures, availability poses a regular threat to services. Architecturally centralized architectures are especially prone to attacks targeting availability, if no additional defense mechanisms are established. To avoid such a single point of failures, some systems are designed as *decentralized* systems. While decentralization can also refer to logical or political decentralization, architectural decentralization inherently has advantages in respect to availability, since the effect of a node's failure can be mitigated by using a different node.

Matrix is an architecturally decentralized protocol used for instant messaging applications. It's federated architecture has similarities with the E-Mail architecture, users exchange messages, known as *events* in Matrix, with a homeserver, and the homeservers in turn forward these messages between each other. A distributed, logically centralized data structure is used to store messages and state related to groups, known as *rooms* in Matrix. This data structure is known as the *DAG*, a directed acyclic graph, inspired by the Git graph, is used to handle inconsistencies between replicas, and also allows resolving these inconsistencies. Despite its decentralized architecture, the current protocol faces challenges regarding centralization. As demonstrated by [4], the convergence of many users on a single large homeserver results in excessive network load on that homeserver, creating a single point of failure and highlighting the lack of scalability in the dissemination mechanism between homeservers.

To address these problems, a next step in the evolution of the Matrix protocol has been proposed: Transforming Matrix into a P2P architecture. First experiments for P2P Matrix have already been conducted, but the P2P version is yet to experience a large-scale deployment, as several open challenges remain, such as multi-homed user accounts across

homeservers, developing an efficient underlying P2P network or establishing trust between peers in a decentralized way.

1.1 Problem Formulation

This thesis seeks to provide a solution to one of the open challenges for P2P Matrix, i.e., *asynchronous delivery*. In the current P2P Matrix implementations, peers are required to be online at the same time in order to exchange events. When a peer attempts to send a message to another peer that is offline, the sending peer must wait and re-send the message later when both peers are online at the same time. A system providing asynchronous delivery would allow peers to exchange events independently from their online state. However, this property requires the event to be stored temporarily somewhere in between the sending and receiving point in time, when both peers are offline. This thesis should discover and design a solution to this problem of providing asynchronous delivery for P2P Matrix.

1.2 Outline

The remainder of this work is structured as follows. Section 2 provides an overview of the Matrix Protocol, P2P networks, and the existing P2P Matrix implementations, which are essential for understanding the content of the subsequent chapters. Section 3 begins with a detailed problem statement, including a formalized definition of asynchronous delivery. It then establishes assumptions on the underlying Matrix P2P network, and presents the functional and qualitative requirements to the solution. Different solutions approaches of related work are discussed, and the differences between relays and homeservers are analysed. Section 4 describes the solution design intended to fulfill the requirements from the previous section. The proof-of-concept implementation is presented in Section 5. Section 6 covers the evaluation setup, the required evaluation traffic, and includes both functional and performance evaluations of the implementation. The functional evaluation verifies the correctness of the solution, whereas the performance evaluation compares different metrics regarding the solution of relay-enhanced P2P Matrix (P2PR) with pure P2P Matrix (P2P).

2. Background

This chapter describes fundamental concepts essential to understanding the problem of the thesis. The first section provides a detailed overview of the Matrix architecture and the Matrix protocol. After that, basic concepts of peer-to-peer (P2P) communication are discussed. The final section presents existing solutions and concepts for P2P Matrix.

2.1 Matrix

Matrix is a decentralized publish-subscribe middleware for real-time communication. The decentralized, interoperable messaging platform has gained a user group of over 100 million users [5] and is deployed by different governments, universities, security services, and military organizations [2]. The mission of Matrix is to provide an open real-time communication layer for the open web, where conversations are replicated, users freely choose software implementing the protocol, and users are able to connect to users of other services via the interoperable Matrix protocol.

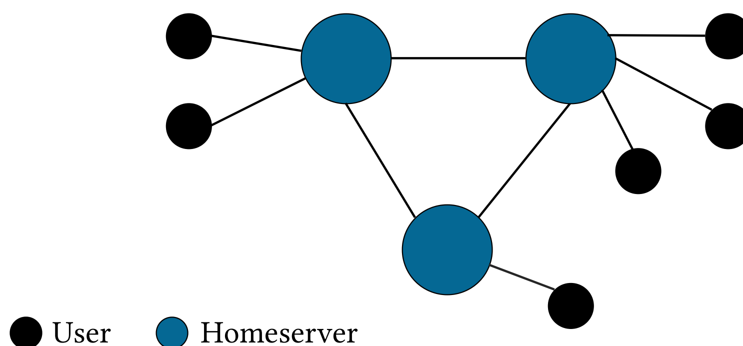

Figure 1: Matrix Federation per Room. The Matrix room is synchronized across three homeservers and has six member users.

The Matrix architecture comprises users and homeservers, as shown in Figure 1. Users can have several devices, on which different clients can be run. To register, Matrix users have to select a homeserver. The domain name of the selected homeserver is part of the user identifier, in the format *@username:domain*. The homeserver also authenticates

the user with a configurable authentication type, e.g., password. After registering, users can become members of rooms. Homeservers store all room information in the *directed acyclic graph* (DAG), which contains the entire room state and messages sent in a room. The DAG is replicated among homeservers holding members of the room and allows homeservers to eventually converge to the same room state (strong eventual consistency). Within a room, users can exchange *events*, that represent a certain user action, e.g., sending messages or changing the room name. Event contents of room events can be configured to be end-to-end-encrypted per room. Matrix employs a level- and attribute-based access control by assigning users to power levels, while sending different events requires different power levels.

Figure 2 depicts the information flow in a Matrix room. Alice sends an event β to her homeserver (1), after which the homeserver appends it to the previous event α (2). A concurrent event γ arrives and the DAG of Alice's homeserver forks into two branches (2), which can be merged later. Alice's homeserver sends the DAG operation (3) to all other homeservers, in this case only to the other homeserver. Bob's homeserver appends the event β to its DAG (4) and forwards it to Bob (5).

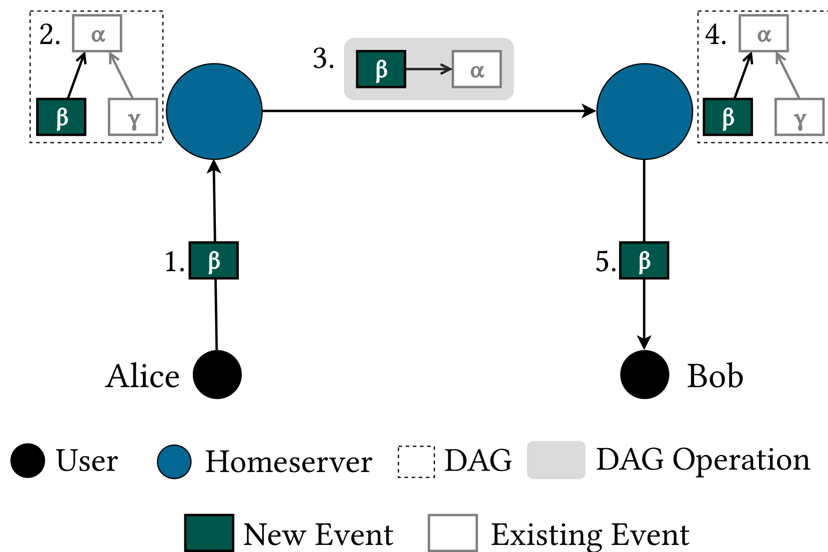

Figure 2: Matrix Event Flow. Alice and Bob are members of the same room. Alice sends an event β in the room.

2.1.1 Protocol Background

The Matrix protocol adheres to the publish-subscribe architecture, that allows scalability in the number of users by decoupling publishers and subscribers along three dimensions: *Space*, *time* and *synchronization* [6]. *Space decoupling* means peers do not have to know each other, which the Matrix protocol achieves by having the user send events to and receive events from only one homeserver. *Time decoupling* means peers do not need to communicate at the same time, which the Matrix protocol achieves by having the homeservers online and storing the full DAG, containing the entire room history. *Synchronization decoupling* means that publishers are not blocked while publishing and subscribers are notified asynchronously. Matrix achieves synchronization decoupling,

because the DAG allows concurrent publishing on events and homeservers notify members on new incoming events. There are different kinds of publish subscribe, which differ on how exchanged information is structured between publishers and subscribers: Content-based, topic-based or type-based [6]. Matrix is a topic-based publish-subscribe system, because events are organized around rooms (i.e., topics), to which users can become members of (i.e., subscribe to).

Matrix is subject to the CAP theorem [7], which captures a fundamental trade-off for distributed systems. It argues that it is impossible for such a service to fulfill three properties at the same time: Consistency, Availability and Partition-tolerance. Only two of these properties can be fulfilled at the same time. If a system is partition-tolerant, the trade-off between consistency and availability remains. However, this does not mean a system designer has to make an absolute choice between consistency and availability. Some mechanisms allow different partitions to temporarily have different states (i.e., inconsistency), but allow an automatic merge of state after the system is not partitioned any longer [8]. Conflict-free replicated data types allow conflict resolution and favor availability under system partition, but forfeit consensus – the strongest notion of consistency. While in state-based CRDTs system nodes exchange states between each other, in operation-based CRDTs system nodes exchange operations. The Matrix DAG has been shown to be an operation-based CRDT and therefore provides a weaker notion of consistency, i.e., strong eventual consistency [9]. Strong eventual consistency allows replicas to eventually converge to the same state if the same updates are eventually applied to each replica [10]. Therefore, Matrix provides availability under system partition, resulting in the possibility of partitions having different states, where the strong eventual consistency property allows these states to be merged at some future point in time.

Distributed systems such as Matrix have a certain timing model, which captures the system behavior regarding time. While some systems require upper bounds on communication and processing delays, others can work without such assumptions. Three different timing models are discussed here: Asynchronous, synchronous, and partially synchronous. Asynchronous systems make no timing assumptions about system nodes and links, there are no upper delays. A system node cannot determine if a message has been lost or if whether is still in transit. To establish an order relation between messages, the concept of logical time is used. Logical time is the time relative to communication activity and can be measured with logical clocks. Logical clocks capture the causal relationship between messages, where a total order can be established between non-concurrent messages, but not between concurrent messages. However, some systems require timing assumptions, as consensus is impossible to solve in asynchronous systems. Synchronous systems can describe three aspects of synchronization: Synchronous computation, synchronous communication and synchronous physical clocks. In synchronous computation, there is a known upper bound on processing delays of system nodes. In synchronous communication, there is a known upper bound on message transmission delays. With synchronous physical clocks, every process is equipped with a local physical clock with a known upper bound on the deviation rate from a global clock, which is an abstraction holding real time. Partially synchronous systems combine asynchronous and synchronous system properties. Upper bounds on physical time hold most of the time, but in some phases the system is asynchronous, where the upper bounds do not hold. Matrix is in the asynchronous timing model, because there are no upper bounds on delays and logical time is used for events. The DAG provides a causal relationship between events, because each event references the previous event(s). This causal rela-

tionship establishes a total order for non-concurrent events, and an arbitrary order for concurrent events.

2.1.2 DAG and Room State

The directed acyclic graph [11] is an append-only, per-room data structure that is managed by homeservers. By replicating the DAG between homeservers, there is no single point of control over the room. Matrix homeservers work together in federation, which means that the DAG is synchronized between homeservers. As the DAG is an operation-based CRDT, homeservers synchronize by exchanging operations on the DAG between each other, as shown in Figure 2. Each node in the DAG is an event and the partial DAG orders these events, resulting in a chronological ordering of events. These events reference previous events, known as parent events. An event can have either zero, one or multiple parents. There is only one event in the DAG that has zero parents, the room creation event. If an event has one parent, the parent event took place before the event. It is also possible for events to have multiple parents. When some events have multiple successor events, i.e., concurrent events, the DAG forks into different branches. Concurrent events occur when a homeserver propagates an event while another event sharing the same parent is still underway, resulting in both events sharing the same parent. The event taking place after all the latest concurrent events merges those together, resulting in those events having multiple parents.

There are two kinds of events, message events and state events [11]. *Message events* represent various communication activities, e.g., instant messages. *State events* update the room state. The room state includes the room alias, join rules, per-user membership status and power-level information. The room alias is mostly used as a human-readable room name, that can be used as an alternative to the unique room identifier, and the alias maps to the room identifier. A room can have different join rules, that represent different usage types of a room, i.e., a room with the join rule *public* allows any user to join the room, while a room with the join rule *invite* requires the joining user to be invited in order to join the room. The power-level information includes both the required power levels to perform certain events and user-specific power levels. This authorization information is used in the Matrix access control mechanism, as described in Section 2.1.3. The room state is an abstract key-value lookup table, that contains persistent information related to a room. The key is a tuple of event type and state key. The event type describes which property of the room state is changed, e.g., *m.room.name* changes the room's name. The state key provides additional information with the event type, e.g., the *m.room.member* event type requires the user identifier. Room state values are therefore stored in the mapping $(\text{event type}, \text{state key}) \rightarrow$ value. The Matrix property availability under partition requires homeservers to independently work out the room state. Conflicting states can occur if different events describe the same state.

In order to resolve state conflicts, Matrix uses a state resolution algorithm [12], [13]. Homeservers have to eventually converge to the same room state, so that users have a consistent view of the room. Homeservers have to follow the same state resolution algorithm, so that they can resolve conflicts independently. A consistent room state can be achieved by establishing a total order on state events. The state resolution algorithm follows two principles to establish a total order on conflicting events. Firstly, events sent by users with higher power levels are favored over those with lower power levels. Secondly, earlier events are favored over later ones. The overall goal of the state

resolution algorithm is to generate an unconflicted output from a conflicted input. The DAG is forked when a network is partitioned and homeservers cannot synchronize their states. A conflicted state can occur in two ways. Either equivalent key tuples have different values in each fork, or the value of a key tuple is modified in only one fork. The state resolution algorithm resolves conflicted state by totally ordering state events for authentication events (i.e., power events).

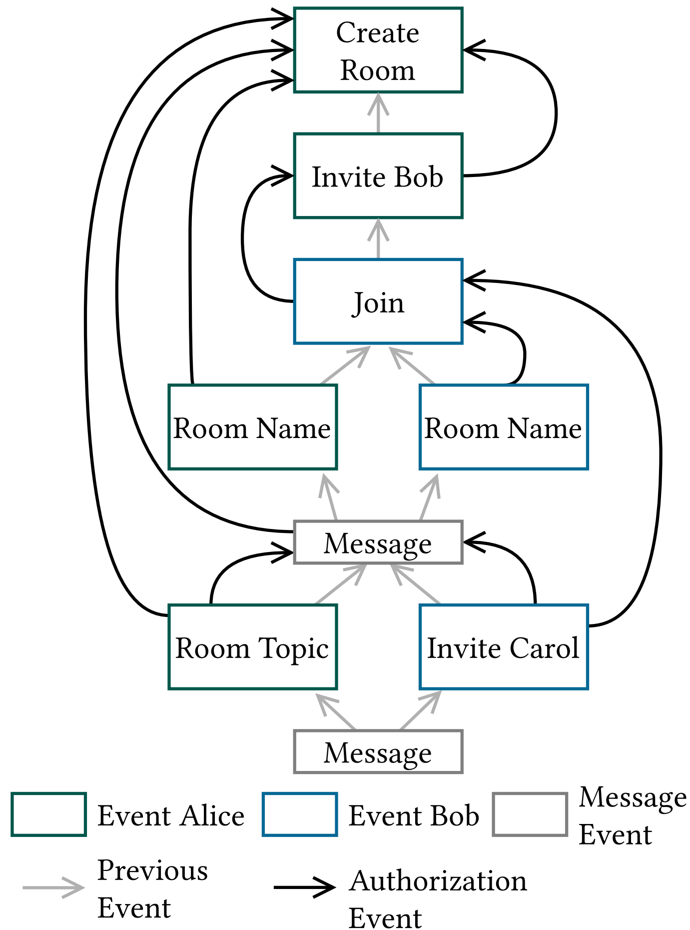

Figure 3: Matrix State Resolution Graph. In the first fork, Alice's room name event is accepted, because her power level is higher than Bob's. In the second fork, both events are accepted, as they change different room states and both are authorized.

Figure 3 shows the state resolution graph, including both edges to previous events as well as edges to authorization events, that are events justifying why an event is allowed to be sent. An auth chain is built by constructing the set of auth events, and adding their auth events recursively [14]. All events have the create event as an authorization event. To make the figure more readable, the authorization event arrows to the create room event are omitted for Bob's events. In the first fork, both Alice and Bob have state events changing an equivalent key, i.e., the room name. Therefore, the two state events are conflicted. The authorization chain for Bob consists of the join event, the invite event and the create room event. Since in both forks equivalent key tuples (Room Name) have different values, the state has to be resolved. Alice's state event is accepted, since her power level is higher than Bob's. Alice's power level is higher than Bob's, because by

default the room creator is assigned a power level of 100, while joining users are assigned a power level of 0 [15]. The second fork has values of key tuples in only one fork. Therefore, the two state events are unconflicted. Alice's auth chain authorizes her to change the room topic, and thus her event is accepted. Bob's auth chain for the invite event is also valid. Both state events of the second fork are accepted and ordered. If one of the events is closer to the last power level event in the respective auth chain, this event precedes the other. In the example, both state events share the same last power level event, i.e., the create room event. In that case, the events are ordered first by their timestamp, then by their event ID.

2.1.3 Access Control

Matrix has an access control mechanism based on an eventually consistent partial order without finality and without a consensus algorithm [16]. The eventual consistent partial order is derived from the DAG, where relevant access control information is stored. Matrix employs a level-based access control, which is enhanced by attributes [16]. Privileges of room members are defined by *power levels*. These power levels define the room hierarchy and can be configured by authorized users, otherwise default values are used [17]. For example, room creators get the power level 100, while other joining users get the power level 0. Unauthorized events, i.e., events where the sending user's power level does not suffice, are declined by homeservers. Power levels are stored in the DAG as state events and have their own event type [18]. The power level event type can be used to assign power levels to users or to configure the required power levels for events. Attributes make the access control mechanism more expressive, e.g., a user membership can have different attributes such as unrelated, knocking, invited, joined or banned. Such membership attributes can be used to enforce policies, e.g., only allow joined members to read events.

Permissions in Matrix can be distinguished into two types, read and write permissions. The read permission determines whether users are allowed to query previously published events. Users are granted the read permission implicitly if they are members of the room. The write permission determines whether users are allowed to send events in a room, resulting in the event being appended to the DAG. Users are granted the write permission on a more fine-grained level compared to the read permission, because granting the permission depends on the configured per-event power level and the user's current power level. For example, only users with a certain power level can invite other users to the room. Overall, power levels apply to the write permission while the read permission only depends on the membership [16].

2.2 Peer-to-Peer Networks

Peer-to-peer (P2P) networks follow a different paradigm than client/server. In the client/server paradigm, servers provide resources and clients use the resources, resulting in a distinct separation of concerns. However, the paradigm has drawbacks. Firstly, the server is a single point of failure. If the server fails, the resources cannot be provided to the clients. Secondly, required resources increase with the number of users, but the server has only limited resources. In P2P networks, a participant (*peer*) acts as both server and client. There is no single point of failure, because if a peer fails, the service

can be provided by another peer. Also P2P is more scalable than the client/server paradigm, because resources provided increase with the number of users.

Figure 4 shows the architectural difference between client/server (a) and P2P (b). While in the client/server architecture clients retrieve resources from a server, in P2P peers act as both server and client. Also, peers can establish connections with each other without requiring a central server as an intermediary.

Whether peers are connected to all others or only a subset of peers, depends on the specific P2P architecture.

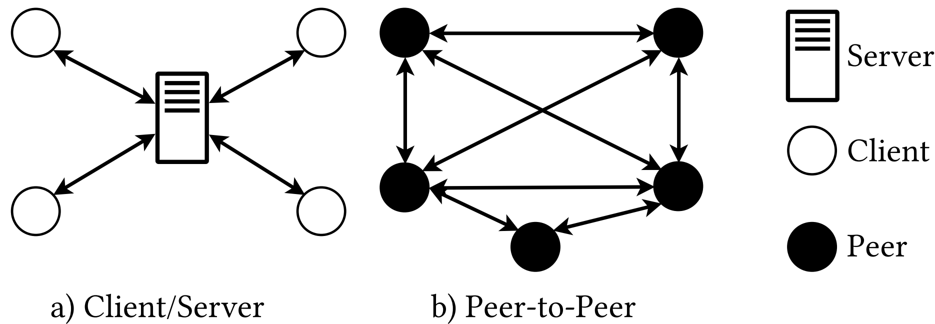

Figure 4: Client/Server vs. Peer-to-Peer

P2P networks are a subset of distributed systems. A distributed system is a *collection of autonomous computing elements that appears to its users as a single coherent system* [19]. Even though network nodes make independent decisions, they have to collaborate in order to appear as a single system. A distributed system is called P2P, if peers share part of their own resources, those resources are necessary to provide the service offered by the network, those resources are accessible by other peers, and peers are both resource providers and resource requesters [20]. Also, peers form a self-organized system that comprises dynamic task distribution between peers and common service provisioning. The fundamental aspect of P2P is that peers combine client and server functionality. There are various kinds of P2P systems, that integrate centralized services into the P2P architecture in different ways. In pure P2P, only peers and no central entities are allowed. Any arbitrarily chosen peer can be removed from the network without diminishing the overall system's availability. An example for a pure P2P system is Gnutella, where peers can retrieve files from other peers by querying files from neighbors, who in turn forward those queries to their neighbors [21]. In hybrid P2P, a central entity is necessary to provide parts of the offered network service. An example for a hybrid P2P system is Napster, where centralized servers index files and monitor the state of each peer [22].

2.3 Peer-to-Peer Matrix

Matrix with homeservers comes with a number of challenges, which arise both from the conceptual level and from observations on the Matrix network. Peer-to-peer Matrix seeks to address these challenges in order provide a more scalable and resilient system. Section 2.3.1 discusses motivation and advantages for P2P Matrix, and which benefits the new architecture provides compared to the federated architecture. Existing implemenations are presented in Section 2.3.2.

2.3.1 Motivation

Homeservers are a single point of failure from a per-user perspective, because a user can only participate in Matrix rooms if the user's homeserver is available. User accounts are bound to the homeserver, as the homeserver's domain name is part of the user identifier. Currently, Matrix does not support user accounts with multiple homeservers, even though solution approaches are being discussed [23]. Also, the user is assumed to trust the homeserver, because the homeserver has access to the user's metadata, such as membership information and traffic patterns. It has been shown in [4], that users are accumulated on few large servers, i.e., 1% of the homeservers with the highest number of users hold 87% of the users that were found in the study. This also leads to network load centralization in the sending process, because events generated on large homeservers have to be distributed to many small homeservers. The load centralization poses the scalability issue in a room of n homeservers, that a homeserver requires $n-1$ connections to other homeservers per room, because the few large homeservers currently establish multiple unicast connections to every other homeserver in the room. If there are few large homeservers with many users, the probability of a user belonging to one of these few homeservers is higher compared to an even user-homeserver distribution. When a room consists of users from only one homeserver, the room DAG is stored only at one homeserver, and is not replicated to other homeservers. The lack of replicating the DAG takes away the advantages of decentralization and results in homeservers being a single point of failure in rooms with one homeserver only. Forwarding events to and between homeservers can lead to inefficient network paths, as the indirection to homeservers is always taken regardless of the shortest path between devices.

P2P Matrix addresses these challenges by removing the need of homeservers and allowing direct connections between devices. Because users are autonomous and do not require a homeserver, they have no single point of failure. Also, the process of joining the Matrix network is simpler, since new users are no longer required to select a homeserver upon registration. The lack of intermediary homeservers also avoids unintentional centralization. The DAG is fully owned by the peers, so that no other nodes can retrieve DAG-related room metadata. Efficient network paths can be established in P2P Matrix with direct connections between devices, even though direct connectivity can be limited, in some cases requiring indirect routes via other peers.

2.3.2 Existing Implementations

There have been various experimental P2P implementations for Matrix, all of which have a peer running a local homeserver in common. Having a peer run a homeserver locally leverages existing progress of the Matrix protocol and minimizes possible protocol changes for P2P Matrix. So far, *Dendrite* [24], a memory-efficient, reliable and scalable homeserver implementation, has been used in experimental P2P setups. Three of the existing P2P implementations are discussed in the following: Matrix-over-libp2p, Matrix-over-Yggdrasil, Pinecone.

Matrix-over-libp2p is a proof-of-concept implementation to connect client-side run homeservers using libp2p, a modular system of protocols, specifications and libraries that enable the development of peer-to-peer network applications [25]. The experiment supports local discovery of peers on the same network using multicast and global discovery of peers over the Internet using a centralized libp2p rendezvous server that acts

as a traffic relay. This implementation has the major disadvantage that all traffic between peers connected through the Internet goes through the same rendezvous server.

Matrix-over-Yggdrasil also supports local discovery of peers, and replaces the libp2p rendezvous server with the global Yggdrasil network. Yggdrasil is a largely self-arranging, mostly self-healing and structured overlay P2P network [26]. A spanning tree is used as overlay topology. All peers with multiple connections to other peers forward messages on behalf of other peers. Peers have an *identifier* and a *coordinate*. The identifier is the hash of the peer's public key and is independent of the node's position in the spanning tree. The coordinate of a peer is a locator and depends on the path from the root node to the peer in the spanning tree. A mapping from identifier to coordinate is stored in a DHT. The root node is chosen based on the public key and periodically sends advertisements, that are forwarded through the whole network. A peer wanting to send a message to another peer has to look up the destination peer's coordinate in the DHT with a given identifier. Then, the peer sends the message to other peers that take it closer to the destination coordinate. Forwarding peers only know the coordinates when forwarding messages. A problem of Yggdrasil is that the network is fragile to topology changes. If a parent node fails, the child coordinate changes as the path to the root is different. Thus, root node failure causes a change of coordinates for all other peers.

Pinecone [27] is the latest P2P Matrix implementation. It is inspired by Yggdrasil and combines two network topologies to build an efficient and scalable overlay network. The two topologies are a spanning tree and a virtual snake. The spanning tree is similar to the Yggdrasil spanning tree, as coordinates describe the peer location in the spanning tree. While using spanning tree routing would allow every peer in the network to reach one another, root node failure would result in temporary disruption because all peers obtain new coordinates. In order to mitigate this disruption, Pinecone introduces the virtual snake. The virtual snake arranges all peers into a line, so that every peer is connected to the peer with the next closest lower public key. Pinecone uses both spanning tree and virtual snake for routing. In the bootstrapping period, neighboring nodes in the virtual snake exchange bootstrap messages to discover each other. Because bootstrap messages are exchanged via the spanning tree, all nodes on the path between neighbors in the virtual snake can add nodes from the bootstrap message to their routing table. Each peer has a routing table containing a list of peers and the public key of the next peer towards the destination, and a forwarding table containing public keys and their corresponding port. Pinecone is used for the two P2P Matrix demos for Android and iOS [28], [29].

3. Analysis

This chapter analyses the problem of asynchronous delivery in Peer-to-Peer (P2P) Matrix. First, Section 3.1 provides a detailed problem analysis, explains the notion of asynchronous and as-soon-as-possible (ASAP) delivery for P2P Matrix, and motivates the use of relays. Then, since P2P Matrix is a work in progress, Section 3.2 specifies assumptions regarding the P2P Matrix architecture, upon which the functionality of asynchronous delivery is built. Section 3.3 derives functional and qualitative requirements from problem statement and scenario definition. Section 3.4 describes and compares different solution approaches, while taking the previously defined requirements into account. To clarify the differences between a relay-enhanced P2P solution and federated Matrix, Section 3.5 compares various aspects between the two architectures. Section 3.6 shows which functionality relays provide in related work and gives a selective overview of existing distributed systems that make use of relays.

3.1 Problem Statement

A problem of P2P Matrix is that peers wanting to exchange events have to be online simultaneously. To have a consistent terminology and to abstract from the concrete P2P network, the *P2P distribution network* describes the network of online peers. While online peers can exchange events via the P2P distribution network, peers that are offline at different times cannot exchange events. If for example, in P2P Matrix a room consists of two users, user a sends an event while user b is offline, when user b comes online later, he/she can only receive the event if user a re-sends the event while being online at the same time as user b. In contrast, Federated Matrix allows users to be online at different times, because homeservers are assumed to be online and store events on behalf of users. Also, Matrix users obtain previously exchanged room events from the homeserver as soon as they come online. Federated Matrix therefore allows events to be delivered *asynchronously* and *ASAP* (as soon as possible). The notions of asynchronous delivery and ASAP delivery are defined in the following.

Asynchronous ASAP Delivery

System Model. Given a set of processes P, that digitally represent Matrix users in the same room, all processes can exchange events using a reliable, instantaneous channel. A global state set *in-transit* consists of all sent events that are in transit. Each process runs a state machine comprising the online state and the two state transitions *send* and *receive*. A process can call *send* to transmit an event to all processes in P, while *receive* is triggered at a process when an event arrives. Both transitions can only be triggered at a process p and time t when precondition is fulfilled, i.e., the state of p is online at t. Each transition is completed at a discrete point in time t. The system model is summarized in the following.

Global State and Functions

- Set of processes P
- State set *in-transit*, initially: *in-transit* = $\emptyset$
- Execute transition at time t: $[\text{transition}]_t \rightarrow \{\text{true}, \text{false}\}$
- State at time t: $[\text{state}]_t \rightarrow \{\text{true}, \text{false}\}$

Process $p \in P$

state p.online = {true, false}
transition $p.\text{send}(e)$
  Precondition: $p.\text{online} = \text{true}$
  in-transit = *in-transit* $\cup \{e\}$
transition $p.\text{receive}(e)$
  Precondition: $p.\text{online} = \text{true}$
  Precondition: $e \in$ *in-transit*
  get e from *in-transit*

Definition: Asynchronous Delivery
In a system according to the model described, an event e is *asynchronously delivered* at t_1 from a sender s to a receiver r, if the following holds: If both conditions (1) and (2) are true, r receives event e at the point in time after sending, when r is back online. The *asynchronous sending condition* (1) states that at t_0, a sender s sends the event while the receiver r is offline. The *eventual asynchronous online state condition* (2) states that at a later point in time t_1, r is online and s is offline. An event is successfully delivered asynchronously at t_1, if providing both conditions are fulfilled, the *receive statement* (3) is fulfilled, i.e., r receives event e at t_1.

$$\text{asyncDelivery}(e, t_1) :=$$
$$\exists s, r \in P :$$
$$\exists t_0 : [s.\text{send}(e)]_{t_0} \wedge [\neg r.\text{online}]_{t_0} \quad (1)$$
$$t_1 > t_0 \wedge [r.\text{online}]_{t_1} \wedge [\neg s.\text{online}] \quad (2)$$
$$\rightarrow$$
$$[r.\text{receive}(e)]_{t_1} \quad (3)$$

Definition: ASAP Delivery
In a system according to the model described, an event e is delivered *as soon as possible (ASAP)* at t_1, if it provides asynchronous delivery and t_1 (i.e., the time of reception) is a minimum in the set T, containing all timestamps greater than t_0 (i.e., the time of sending), where the receiver is online and the sender is offline.

$$T = \left\{ t_i \mid t_i > t_0 \wedge [r.\text{online}]_{t_i} \wedge [\neg s.\text{online}]_{t_i} \right\}$$

$$\text{asapDelivery}(e, t_1) := \text{asyncDelivery}(e, t_1) \wedge (t_1 = \min(T))$$

Asynchronous ASAP delivery incorporates two fundamental properties. First, sender and receiver do not have to be simultaneously online and can exchange events asynchronously. Second, the event is received directly when the receiver comes online, or (in real-world systems) after a certain propagation delay, i.e., the event is delivered ASAP. Figure 5 shows the relation between ASAP, asynchronous and synchronous delivery. At t_1, the event α is delivered not only asynchronously, but also ASAP, because t_1 is the earliest point in time after sending, where the receiver is online, while the sender is offline. In this example, according to the ASAP Delivery definition, $T = \{t_1, ..., t_2, ...\}$ and $t_1 = \min(T)$. At t_2, the event α is delivered asynchronously, but not ASAP because $t_2 \neq \min(T)$. At t_3, a different event β is delivered synchronously, because sender and receiver are simultaneously online.

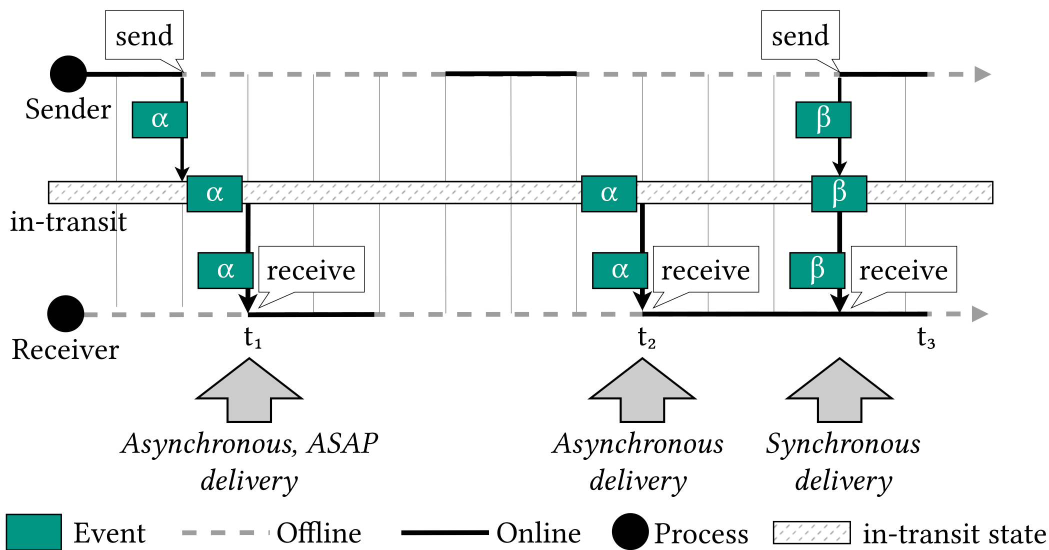

Figure 5: ASAP vs. Asynchronous vs. Synchronous Delivery

P2P Matrix does not inherently provide asynchronous delivery. If all other users in a room are offline, r has to wait beyond t_1 until r and s are online simultaneously and s re-sends the event. Federated Matrix provides asynchronous and ASAP delivery, because homeservers are assumed to be online, store events and deliver them to r asynchronously and as soon as possible.

In order to provide asynchronous, ASAP delivery for P2P Matrix, events have to be buffered during the time of peers being offline, and forwarded as soon as they return online. This buffer-and-forward concept is referred to as *relaying mechanism* and nodes providing this mechanism are referred to as *relays*.

Using relays with a fixed peer-to-relay relation is not a useful option, because it would have similar drawbacks as Federated Matrix, as described in Section 1. From an architectural point of view, there are different options to design such a relaying mechanism.

1. **Peer devices act as relays**
   Devices of peers can perform the relaying mechanism for other peers. This solution has two major drawbacks: (i) The relaying mechanism consumes already limited computational and energy resources of the peers' devices, which makes a difference especially for mobile devices, and (ii) devices have high churn in Matrix, which either decreases the probability of a relaying peer being online or induces a

high network load and a high resource consumption at devices, if events are distributed multiple times among different devices.

2. **Dedicated distributed relays**
Several distributed and dedicated relays, of which at least a subset is assumed to be online, can provide the relaying mechanism without having to require additional resource consumption for peers or making the mechanism itself resilient towards high peer churn. In contrast to the fixed relation between users and homeservers, peers do not directly depend on relays and can still function without relays. Furthermore, peers may switch between multiple relays in order to maintain a level of metadata privacy, as explained below. Relays provide the relaying mechanism, which allows asynchronous delivery of events between peers. If either all relays are unavailable or peers choose to not use them, pure, relay-less P2P Matrix can serve as a fallback mechanism that allows simultaneously online peers to exchange events. Even though peers can in theory choose any relays, per-room policies can constrain which relays are allowed to perform the relaying mechanism on behalf of the room.

Although replication of the event graph to federated homeservers ensures a room's resilience, it has the privacy drawback of disclosing metadata to every homeserver in a room. As a result, metadata accumulates at homeservers over time [30]. Metadata can include room metadata and network-level metadata. Room metadata consists of application-level information about the room, which includes matrix IDs of every room member, homeservers associated to user's rooms and room IDs, as well as the time and sender of room events. Network-level metadata includes information required to route packets through the Internet, such as source and destination IP addresses, and traffic patterns, like the communication time and frequency. By eliminating the use of homeservers, P2P Matrix does not have the disadvantage of servers storing room metadata. However, relays storing and forwarding events on behalf of peers over time re-introduces the risk of metadata disclosure. Therefore, relay-enhanced P2P Matrix should provide a privacy-enhancing solution where relays acquire at least less room metadata compared to homeservers in federated Matrix.

Considering the drawbacks of the first option of peer devices acting as relays, the solution with dedicated distributed relays is pursued further in order to provide asynchronous delivery for P2P Matrix. However, it is conceptually possible for peers to provide the relaying mechanism as an additional service. Finally, this thesis seeks to provide a solution to the following problems for P2P Matrix with dedicated distributed relays: Asynchronous ASAP delivery and metadata privacy.

3.2 Assumptions on P2P Matrix

There is no consensus on the architecture of P2P Matrix, because it is currently on hold and no production-ready version has been released. Several experimental architectures for P2P Matrix have been proposed and discussed in the community, as mentioned in Section 2.3.2. While some problems related to P2P Matrix have already been addressed, such as identities for peers [31], other features are yet to be designed, such as decentralized user accounts [23], end-to-end encryption in a P2P setting, and efficient routing algorithms [30]. This work seeks to provide the feature of asynchronous delivery, while abstracting from a specific P2P implementation for Matrix. However, assumptions on

how P2P Matrix works in general have to be made, in order to include dedicated distributed relays into the P2P architecture.

The assumptions are as follows:

- **Pure P2P Architecture as Basis**
  The architecture consists of peers (and their devices), there are no homeservers. The P2P architecture is extended by distributed relays.
- **P2P Routing Algorithm**
  P2P Matrix is assumed to have a routing mechanism that allows peers to efficiently exchange events with each other. This work abstracts from the specific implementation, regardless of whether the routing algorithm is based on structured, unstructured P2P overlays (e.g., flooding, random walk) or other architectures.
- **Multiple Devices per Peer**
  Peers can have multiple devices, similar to users having multiple devices in federated Matrix.
- **Decentralized Peer Identities**
  Peers have self-managed identities, such as in [32], [31]. Identities can be used on several devices. Peers generate their own cryptographic identities used to sign events.
- **Peer Devices manage Event Graph**
  Peer devices are responsible for managing the event graph comparable to how homeservers manage it in Federated Matrix. This includes storing the event graph, appending new events to it, and performing state resolution.
- **NAT Traversal Problem addressed by P2P Matrix**
  The problem of NAT traversal, where two peers behind NAT gateways require a public node to establish connectivity, is assumed to be addressed by P2P Matrix and is not part of this work.
- **Bootstrapping Problem addressed by P2P Matrix**
  The problem of P2P bootstrapping, where a new peer wants to join the P2P network and requires information to find another peer, is assumed to be addressed by P2P Matrix and is not part of this work. Matrix already provides invitations for non-members of a room, which include the required information for a new peer to join rooms. These invitations are exchanged out-of-band (e.g., via e-mail).
- **Heterogeneity of Devices Comparable to the Federated Architecture**
  P2P Matrix is assumed to work for the same range of devices as those in the federated architecture, e.g., desktop computers, laptops, tablets, smartphones.
- **End-to-end Encryption**
  P2P Matrix is assumed to provide end-to-end encryption for non-public rooms of all sizes. This includes a key exchange mechanism required for the room members to exchange symmetric keys or group keys. Therefore, this work assumes confidentiality and integrity of event contents in non-public rooms.

3.3 Requirements

This section presents the requirements for a solution to the problem described in Section 3.1, while taking into account the assumptions made regarding the architecture of P2P Matrix described in Section 3.2. The requirements are divided into functional and qualitative requirements.

The functional requirements are:

- **Asynchronous Delivery among Peers**
  When a sender publishes an event in a room, every room member should be able to eventually receive this event, even if the sender goes offline directly after publishing.
- **ASAP Asynchronous Delivery**
  When a sender publishes an event in a room, every room member should receive this event as soon as it comes online after the point in time of publishing.
- **Relay Discovery for Peers**
  The discovery mechanism allows peers to retrieve a set of relay candidates. Peers can also obtain metrics relevant for relay selection (e.g., latency, propagation delay, data rate, geographic information).
- **Relay Selection for Peers**
  A peer can select one or more relays from the set of relay candidates and can base the selection decision on different criteria, such as subscription similarities, latency etc. To fulfill the goal of privacy and to distribute the load, peers should be able to select different relays over time. Peers should also be able to establish multiple simultaneous connections to different relays.
- **Smooth Relay Failure**
  As the later described qualitative requirement "Independence of Relay" suggests, the system should also work without relays. The transition from the relay-based to the fallback system of pure P2P Matrix without relays should work fast and without manual interventions, e.g., configuration changes.

The qualitative requirements are:

- **Comparable Performance to Federated Matrix**
  The event propagation delay between sender and receiver should not be significantly higher than in federated Matrix. This applies to the cases where all peers are simultaneously online, and the asynchronous case, where relays are used to buffer and forward events.
- **Metadata Privacy**
  Even though events between peers are end-to-end encrypted, relays obtain metadata when storing and forwarding events on behalf of peers. Which metadata can be obtained by relays in P2P Matrix or by homeservers in federated Matrix, is described in Section 3.1. The solution should provide enhanced metadata privacy to the degree of relays acquiring less per-user metadata than homeservers do in federated Matrix.
- **Reduce Relay State**
  Relays should store as little state as possible to forward events. The event itself and some relay-relay associations have to be stored in order to provide asynchronous delivery functionality. However, stateful peer-relay connections or storing the state of room memberships of peers should be avoided.
- **Authorization of Relays and Peers**
  Relays should authorize other room relays and peers, in order to prevent malicious peers/relays injecting unauthorized events into the room. A relay accepting unauthorized events can degrade the performance of forwarding benign events, resulting in peers experiencing higher delivery delays.

- **Decentralized Relay Architecture**
  No manual configuration should be necessary at the relays in order to establish connectivity and exchange events between each other.
- **Network Load Distribution among Relays**
  The solution should be able to distribute the network load among the dedicated relays, in order to avoid load centralization where a high share of events is directed towards one of the dedicated relays.
- **Independence of Relay Usage**
  If no relays are available, even though P2P Matrix cannot provide the functionality of asynchronous delivery, it should still work among online peers. Therefore, the performance of P2P Matrix should not be impaired by the relays.
- **Deployability**
  The solution should make as few changes as possible to the existing Matrix Specification. Also, it should support integration into a hybrid system consisting of P2P and federated Matrix, where peers can exchange events between users behind homeservers.
- **No Redundant Functionality**
  Existing Matrix functionality should be applied in the solution, i.e., no redundant or parallel solutions should be developed when applying Matrix functionality would be more suitable.

3.4 Solution Approaches

This chapter presents different solution approaches for the problem of asynchronous, low-latency, privacy-enhancing P2P Matrix. The approaches are assessed while considering the requirements defined in Section 3.3.

3.4.1 Relay Functionality Placement

The relaying mechanism can be incorporated into architectures in two ways. As discussed in Section 3.1, either a subset of peers or dedicated nodes can act as relays. These two approaches are compared in more detail in the following.

Peers as Relays
The first option is for a subset of peers, which are sometimes called "super-peers", to act as relays. Super-peers are often used as relays, when the relay functionality consists of exchanging requests or information between peers [33]. Having peers provide the relaying mechanism on behalf of other peers requires the ability to deal with high churn in P2P Matrix, because mobile devices in particular switch between on- and offline state regularly. If the sender and all relaying peers go offline before the receiving peer comes online, the system does not provide asynchronous delivery. To improve the hit-rate of events reaching the receiving node when it comes online, the event would be redundantly disseminated multiple times to different peers (depending on the P2P distribution network), which has two disadvantages. Firstly, sending redundant events generates additional network traffic and requires higher resource consumption at the peers compared to sending only the necessary number of events directed to the destinations. Secondly, to improve the probability of a relaying peer buffering a certain event being online even further, events could be disseminated to non-members of the room, which exposes room metadata. Consequently, a decision needs to be made between a higher robustness (i.e.,

probability of event reaching receiver in time) with higher overall network load, and lower robustness with a lower overall network load. The trade-off between robustness under churn and duplicate delivery [34] is a general challenge in P2P publish-subscribe systems. The original Skype VoIP P2P application made use of super-peers, where a publicly addressable Skype client could relay traffic between other endpoints [35].

Dedicated Relays
The second option is to deploy relays as separate infrastructure components. This is especially useful for networks with high churn, devices with limited resources or devices not publicly accessible because of NAT gateways or restrictive firewalls. These relays are assumed to stay online and thus have lower churn than devices of peers, providing a higher rate of availability, which allows a higher robustness under churn compared to a pure P2P architecture. Several distributed relays can be used in order to provide the functionality of asynchronous, low-latency delivery in a flexible, scalable and efficient way. This solution has the advantage of consuming less resources for peer devices compared to peers acting as relays. To avoid the disadvantages of the fixed relation between users and homeservers in the federated architecture, peers should be able to use different distributed relays over time. Also, it would be conceptually possible for peers to establish connections to multiple relays at the same time, in order to retrieve the latest events from the relay storing the latest events. This however, would induce additional resource consumption at the peer.

3.4.2 Event Graph Storage

In federated Matrix, homeservers store the complete event graph of a room. If end-to-end encryption is enabled, the homeserver cannot decrypt event contents, but has access to metadata of all room events. Thus, if relays store the complete event graph of a room, they also have access to metadata of all room events.

Alternatively, relays could only buffer and forward events without storing any other room-related data. On the other hand, relays cannot determine whether a peer is authorized to read or send events. Without knowing read permissions, both member and non-member peers can request cached events from the relay. Even though non-member peers cannot decrypt event contents, they can acquire event and room metadata. Without knowing sending permissions, peers could send arbitrary events to rooms in which they are not members. This can lead to two different degradation of service attacks. First, non-member peers can fill the relay cache with events and thereby consume most of the relay's storage resources, so that the relay has to drop events of member peers. Although benign room members could also fill the relay cache with a large number of events, the Matrix protocol provides a mechanism to ban members from the room. Second, non-member peers can utilize relays as traffic multipliers, because sending one event to the relay results in the event being distributed to all room relays and peers. The traffic multiplication would enable non-member peers to consume resources both in the network and at other relays and peers at a low cost. However, the degradation of service attacks are only present if the asynchronous delivery function is provided by relays, pure P2P Matrix can still provide availability since a fallback from relay-enhanced P2P Matrix to pure P2P Matrix should be possible.

The Authorization DAG (AuthDAG) is a reduced event graph and contains only authorization-relevant state events of a room. It can be created by extracting all policy and permission events from the DAG. As in the DAG, the extracted events have causal rela-

tions among each other, i.e., events in the AuthDAG reference previous events. To perform authorization checks such as “user is member of room and has read permission” or “user has required power level for event”, the relay has to establish a total order on all state events in order to make a deterministic authorization decision. This total ordering on state events is performed with the state resolution algorithm, as described in Section 2.1.2. Storing the AuthDAG at relays allows access control without disclosing all metadata to relays. Relays use the AuthDAG to determine if a peer is authorized to read or send events, thereby reducing attack vectors of non-member peers. Also, it prevents relays from storing all room metadata, which would establish drawbacks similar to those of homeservers in Federated Matrix.

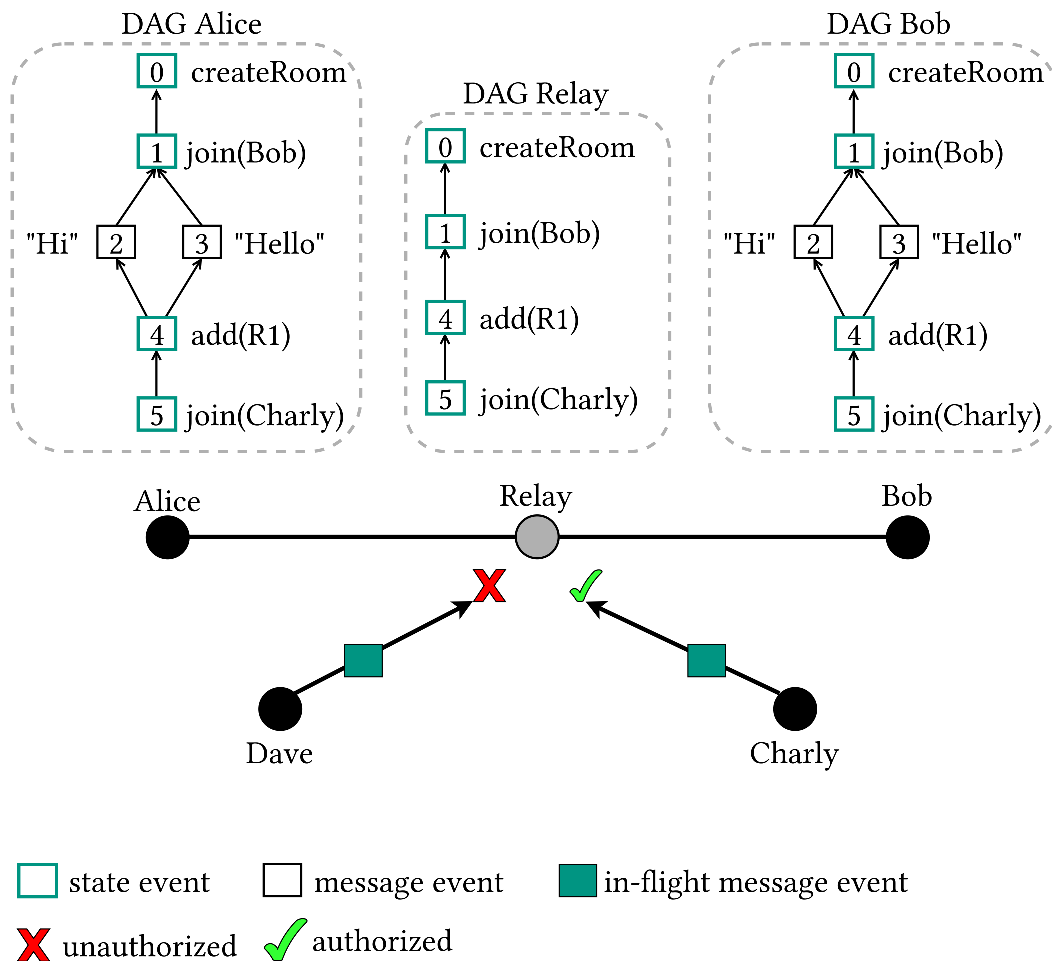

Figure 6: Authorization enforcement by a relay with an Authorization DAG, a reduced event graph containing only authorization room events.

An example AuthDAG is shown in Figure 6. Both peers Alice and Bob store the complete DAG, while the relay stores only the AuthDAG. The AuthDAG allows the relay to determine whether senders of incoming events have the necessary permissions, and also whether requesting peers have the permission to read events. In the example, we assume that both Charly and Dave send a message event to the relay. Assuming that only joined members have the permission to send message events in the room, the relay can search for the join event in the AuthDAG. Because a join event exists for Charly, the

relay accepts his event. Because there is no join event for Dave and he is therefore not a room member, his event is rejected.

3.4.3 Relay Integration into Matrix Protocol

Relays can be integrated into the Matrix Protocol in various ways. Two approaches are compared: Stand-alone relay integration, and relay integration into DAG.

Stand-alone relay integration means that relays are managed independently from the Matrix protocol. Therefore, relays are not necessarily bound to rooms, but can be managed in a custom and efficient way. A stand-alone relay integration allows efficient implementations of relay lookup, selection and load distribution. However, this solution (similar to any other relaying solution) requires some form of access control in order to avoid unauthorized access to events. The major drawback of this approach is that an access control mechanism would be used parallel to the existing Matrix access control mechanism. Although this would allow fine-grained privilege control over relay management, it leads to redundant functionality, because Matrix already has a distributed access control mechanism, as described in Section 2.1.3.

Integrating relays into the Matrix protocol by managing them in the DAG would utilize existing Matrix functionality. Integrating relays into the event graph means that they are managed per-room, i.e., each room has a dedicated own set of relays. This solution uses existing Matrix functionality, such as access control, replication and strong eventual consistency. As relays are managed in the DAG, they are part of the room state. Access control is possible in the sense that peers authorize each other for managing relays, i.e., only authorized peers can add or remove room relays. Information as to which relay is part of the room, is distributed to all peers, because the Matrix DAG is an already replicated data structure. Because the DAG is not only a replicated data structure, but a CRDT (conflict-free replicated data type), it allows strong eventual consistency, so that all peers eventually converge to the same view of room relays. Relays can either be managed by a dedicated state event, or as room members. Introducing a state event for relays would differentiate relays and room members on a conceptual level, and is a semantically cleaner solution than managing relays and peers in the same set of room members.

Having relays as room members would require an additional mechanism to differentiate them from peers. While managing relays in a state event allows fine-grained control over relay management privileges, it requires a Matrix specification change due to the new state event. In contrast, managing relays as room members does not require a specification change. However, relays are not fully capable room members, they should only receive, but do not pro-actively send room events. They merely buffer received events and make them available to other peers upon request. In confidential rooms, peers can choose not to share keys with the relays, who therefore would only have access to event metadata, but could not decrypt events.

3.4.4 Privileges for Room Relay Management

If there are multiple room relays, relays have to know which other relays are part of the room, in order to avoid sharing event metadata with unrelated relays. Federation between homeservers in Matrix is similar, as homeservers only forward events to those homeservers that have member users. In P2P Matrix, if relays are managed per-room,

authorized peers have control over which relays can be used in the room. Relays themselves should not be able to include additional relays to a room, since their main function is to store and forward events on behalf of offline peers. Figure 7 shows an abstraction of the relay mapping privileges. The room relay set stores the relays belonging of the room. Various implementation possibilities exist for storing this information, such as an event in the DAG, a DHT or centralized storage. Regardless of where the room relay set is stored, only peers should have the permission to add and remove relays from the set. As depicted, this permission can be tied to power levels, allowing only a subset of peers to manage relays. Relays have read access to the room relay set and thus can disseminate events to all other room relays.

Managing relays as state events allows a fine-grained privilege configuration, as required power levels can be defined for any state event. In the case of relays being managed as room members, the required power level for membership state events would also be the power level for managing relays. In this case, if a user has the required power level to invite users, he/she also has the power level to invite relays. A dedicated access control for managing relays also allows a fine-grained privilege configuration, which nevertheless is configured separately from Matrix power levels.

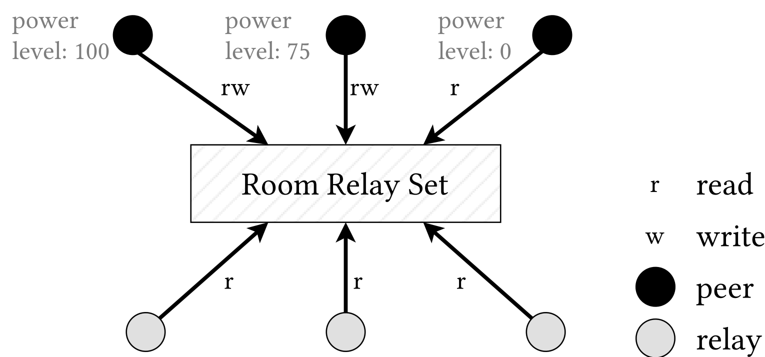

Figure 7: Room-Relay Abstraction for Access Control between Relays.

3.4.5 Peer-to-Relay Relation

There are several options for how many relays a peer can be assigned to over time. The relation between peers and relays should not only apply to a discrete point in time, but also for longer periods. Consequently, on this abstraction layer, a peer using multiple relays is equivalent to a peer switching between relays over time, or peer devices selecting distinct relays.

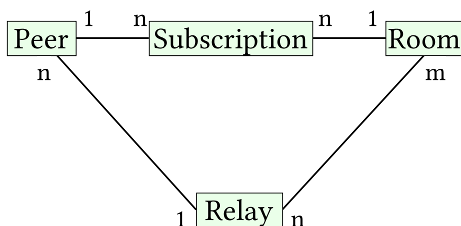

Figure 8: Peers use one relay

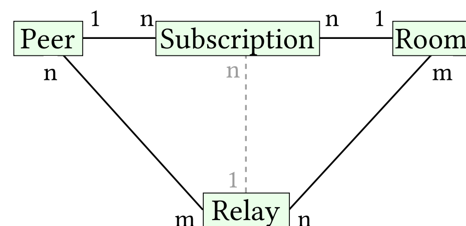

Figure 9: Peers use multiple relays

The simplest peer-to-relay relation is to have one relay per peer (Figure 8) for the total lifetime of the peer. This would have several drawbacks. Similar to federated Matrix, where all per-user metadata is stored at a homeserver, with this relation all per-peer metadata is stored at one relay. This comes into conflict with the privacy requirement, because the relation does not provide higher metadata privacy compared to federated Matrix. Accepting a relay-using peer to join a room includes an implicit consent of the room members to allow the joining peer's relay to have access to the room's future metadata. Consequently, the one-to-one peer-relay relation allows no fine-grained control over which relay is permitted in which room, because the room relays are implicitly made part of the room if their peer is a room member. Also, using the same relay regardless of the latency and bandwidth between device and relay can result in inefficient paths.

The next option is to have multiple relays per peer (Figure 9), where each subscription can be assigned to a relay (dashed grey relation), but also different devices can use different relays. The complexity of sending and retrieving events is higher compared to having one relay per peer, as the peer has to exchange events with multiple relays. The per-peer metadata can be distributed, if different relays are selected across the peer's subscriptions and devices. Also, a per-device selection of peers can improve the latency between the device and the selected relay, because each device can choose the relay with the lowest latency and highest bandwidth. In contrast, Federated Matrix does not provide this improvement, as each user device has to use the same homeserver. Multiple relays per peer avoid the disadvantage of having to add relays to a room whenever a peer joins, that is the case when each peer has only one relay. Having multiple relays per peer allows a more fine-grained and explicit control over room relays, because the decision as to which relays may be used in a room, can be constrained by authorized peers, e.g., room admins. Consequently, instead of peer-centered control over relays, the multiple-relay solution allows room-centered control over relays.

3.4.6 Forwarding Events to Relays

Peers can use different mechanisms to forward events toward relays. Assuming there are several relays in a room, there are two extremes of mechanisms ensuring that events sent by peers reach all relays. First, peers can either forward events to each relay in a multi-unicast manner, as depicted in Figure 10.

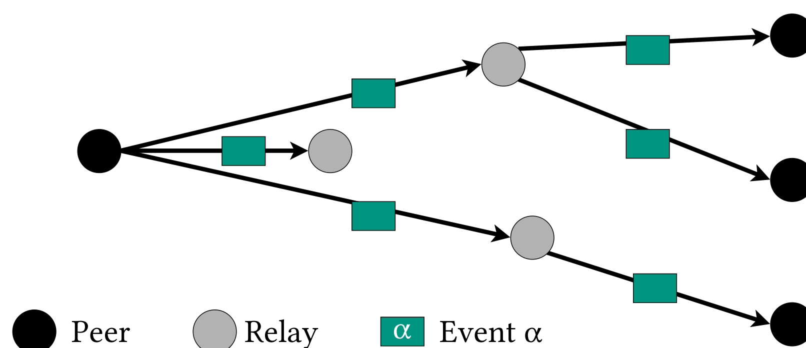

Figure 10: Peer multi-unicast. The peer sends the event to each relay, after which the relays forward the events to their peers.

Second, peers could forward to only one relay, which in turn broadcasts the event to all other relays, as depicted in Figure 11.

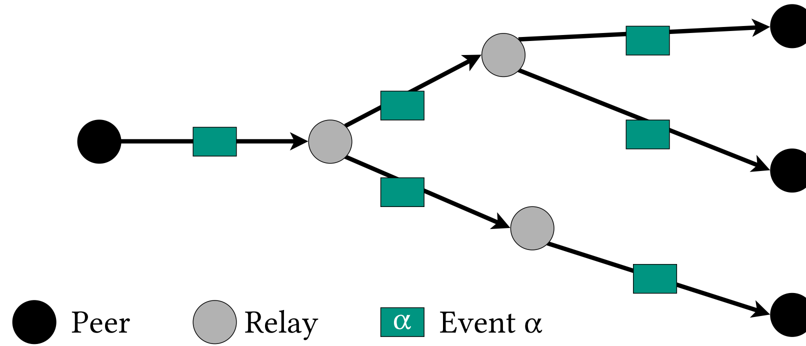

Figure 11: Inter-relay forwarding. The peer sends the event to a relay, which then forwards the event to the other relays.

There may also be other mechanisms operating on a spectrum between the two extremes, e.g., peers disseminating events to relays via an application level multicast mechanism. The peer dissemination complexity is the complexity term of a peer in respect to the number of relays the peer has to forward an event to. The two extremes show different tradeoffs, which are discussed in the following.

1. *Information disclosure to relays vs. peer dissemination complexity*
   When a peer sends a room event, there are two options of how the event can be disseminated to all room relays. The first option is having the peer forward the event to each relay (Figure 10), incurring a peer dissemination complexity of $O(n)$, but not disclosing room memberships of relays to other relays. The second option is having the peer forward the event to only one relay and the relay disseminating the event to all other relays (Figure 11), incurring a peer dissemination complexity of $O(1)$, but disclosing room memberships of relays to other relays, in contrast to the first option.
2. *Relay Complexity vs. Peer Complexity*
   If a sender forwards an event to only one relay, incurring a peer dissemination complexity of $O(1)$, the relay would have to disseminate the event to all other relays, if also the receiving peer should be able to retrieve the event from any relay. The relay dissemination complexity would be $O(n)$, n being the number of relays. In the opposite scenario, peers would forward events to and retrieve events from all relays ($O(n)$, while relays would only have to connect to one peer ($O(1)$).
3. *Sender Complexity vs. Receiver Complexity*
   Assuming that relays do not interconnect with each other, either sending peers can disseminate events to all relays and receiving peers retrieve events from only one relay, or sending peers forward events to only one relay and receiving peers have to find the relay, which the event was forwarded to. Consequently, there is a choice between a sender complexity of $O(n)$ with a receiver complexity of $O(1)$, and a sender complexity of $O(1)$ with a receiver complexity of $O(n)$.
4. *Efficient Dissemination vs. Reliable Retrieval*
   In order to reduce the dissemination effort, the sending peer may distribute the event to only a few relays, resulting in only a subset of relays storing the event. If the receiver reduces the retrieval effort, the probability of the event not being stored by one of the contacted relays rises, i.e., the event retrieval becomes less

reliable. Thus, a solution must weigh the aspects of efficient event dissemination and reliable event retrieval against each other.

3.5 Differences to Federated Matrix

It is important to highlight the differences between P2P Matrix with relays and Federated Matrix, in order to avoid the relay-based solution introducing similar drawbacks as Federated Matrix. To make the systems comparable, *designated nodes* refer to either relays or homeservers, while users can stand for both peers or users (Federated Matrix). Table 1 summarizes these differences.

	P2P Matrix with Relays	**Federated Matrix**
Per user single point of failure	No	Yes
Room availability depends on designated node availability	No	Yes
Changing designated node possible	Yes	No
Designated nodes per user	$1 \leq \|N\| \leq \|R\|$	$\|N\| = 1$
Event graph storage location	Peers	Homeservers
User Metadata	Distributed	Centralized

Table 1: Differences between P2P Matrix with Relays and Federated Matrix

$R :=$ Set of all relays
$N :=$ Set of designated nodes

While in P2P Matrix with relays, peers online at the same time can exchange events via the P2P distribution network without the relays having to be online, in Federated Matrix the homeserver is a single point of failure for a user, because a homeserver failure disconnects the Matrix user from the Matrix network. In both settings, there are designated nodes that forward events on the user's behalf. However, if all room relays are unavailable, simultaneously online peers can still exchange events, whereas if all homeservers of a room are unavailable, no events can be exchanged in the room. In P2P Matrix with relays, the number of designated nodes is subject to different factors (e.g., number of distinct relays per peer across different rooms), but generally the number is limited by the total number of relays and can be 1, if a peer never switches relays and all other rooms of the peer use the same relay. In Federated Matrix, every user has only one designated node, the homeserver. P2P Matrix requires peers to store the event graph for a room, regardless if relays are used or not, while homeservers store the event graph in Federated Matrix. Finally, the distribution of metadata disclosed to the designated nodes is compared. In Federated Matrix, the homeserver holds the complete amount of metadata per peer, that can be derived from sent and received events. In P2P Matrix with relays, the user metadata can be distributed if different relays are used per room. In Federated Matrix, user metadata is stored centrally at the user's homeserver.

3.6 Solution Approaches in Related Work

This section provides an overview of relay features in related work and discusses existing relay-based solutions for distributed systems. After discussing different motivations for including relays into network architectures, different networking protocols that make use of relays are presented. Finally, the presented protocols are compared in regard to functionality potentially relevant for this work.

3.6.1 Relay Functionality

Relays are used in various distributed systems, such as P2P networks, Publish/Subscribe architectures, IoT networks. In related work, various reasons are stated to include relays in architectures:

- *Asynchronous delivery*
  While in P2P networks peers have to be online at the same time in order to exchange messages with each other, a backbone of stable nodes (i.e., relays) can store and forward messages on behalf of peers, who can obtain these asynchronously [36].
- *Connectivity establishment*
  Some devices cannot establish direct connections due to NAT gateways and restrictive firewalls [35], [37]. They therefore require a publicly addressable node (i.e., a relay) in order to provide interconnection [25], [33].
- *Latency optimization*
  In IoT scenarios, relays or brokers are installed at the edge and therefore near the IoT devices in order to reduce latency compared cloud-based approaches[38].
- *Anonymity*
  The delivery service Tor [39] routes traffic on the path from sender to receiver between multiple relays, where each relay only knows its predecessor and successor. Since whole traffic flows are encrypted in layers at each relay ("Onion Encryption"), an adversary who can observe only a fraction of network traffic is not able to determine the original sender and receiver of a message. Tor therefore provides sender- and receiver anonymity against the described adversary.

In this work, the reason to include relays in the P2P Matrix architecture is to provide asynchronous delivery. Relays are not used for connectivity establishment, since NAT traversal is assumed to be provided by P2P Matrix, as described in Section 3.2. Although the relay-based solution is required to provide low latency (Section 3.3), latency optimization is not the motivating reason to deploy relays. Also, relays for P2P Matrix do not inherently provide anonymity, although a Tor-like network could be used for both the P2P network and as group communication mechanism between relays.

3.6.2 Nostr

Nostr, which is an acronym for "Notes and other stuff transmitted by relays" is a decentralized, relay-based social media protocol [40]. Figure 12 shows the architecture, which consists of users, clients and relays. Users can have several clients, that can publish events to one or more relays, to which other clients can subscribe.

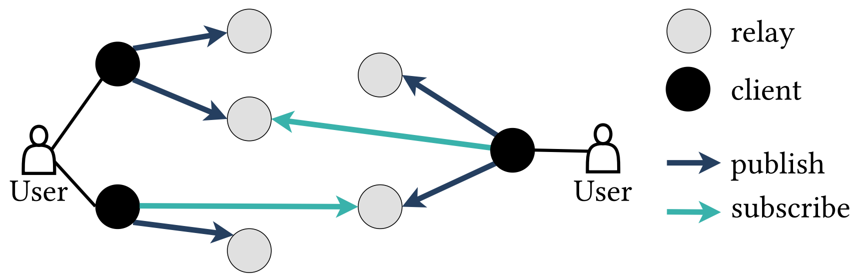

Figure 12: Nostr architecture

A Nostr user has a public/private key pair, where the user has complete ownership over his keys. Events are signed by the sender with the private key, while receivers validate events with the public key of the sender. Relays store events on behalf of senders and broadcast them to all other subscribed clients. Users can connect, publish to and subscribe from an arbitrary number of relays over time. A subscription (i.e., following another user) means that a user's client queries relays for posts from the followee's public key.

The client-relay protocol includes three types of messages: EVENT messages for publishing events, REQ to request subscriptions and CLOSE to stop existing subscriptions [41]. Relays only connect directly to clients, as Nostr does not include a delivery protocol between relays. Nostr Implementation Possibilities (NIPs) provide a mechanism to coordinate additional protocol features and make the protocol extensible. Aside from NIP-01, which describes the basic Nostr protocol, all other NIPs are optional. In order to find relays belonging to a given public key of a user, NIP-05 specifies how Nostr keys can be mapped to DNS-based Internet identifiers.

3.6.3 Scuttlebutt

Secure Scuttlebutt (SBB) is a peer-to-peer protocol for building decentralized applications, most prominently decentralized social networks [42]. Users can have multiple identities, which consist of public/private key pairs. However, an identity is tied to a single device, because if two devices use the same identity, some messages cannot be propagated by other peers [43]. SSB utilizes append-only logs that are an immutable data-structure containing all messages posted by a user.

While regular peers store their own append-only log and are able to publish or subscribe messages, there are two kinds of peers providing additional functionality: *rooms* and *pubs*. Rooms are publicly accessible peers that can provide connectivity between two other peers, unable to establish a direct connection due to NAT gateways or restrictive firewalls. Pubs can also provide connectivity between two other peers, but additionally can serve as rendezvous point for new users to find existing users. Pubs therefore store a peer's log and make it available for other peers even if the publishing peer is offline, therefore providing asynchronous delivery of log entries. A peer can distribute its log to many different pubs. Both rooms and pubs are, aside from their additional functionality, regular peers that usually deployed on servers, so that they are online most of the time.

Peers can discover each other by either broadcasting regular advertisements in the local network, receiving invite codes that point to a pub hosting the append-log of the inviting peer, or a peer posting a pub advertisement in its append-only log, which is

only feasible if the peer already knows some other peers, to which it can distribute the advertisement.

3.6.4 Asynchronous Mobile Peer-to-peer Relay

The paper extends a mobile P2P network with relays providing asynchronous delivery between devices, and focuses on encryption mechanisms between devices and relays [36]. Relays cache end-to-end encrypted messages until the receiving peer comes online, thereby providing a reliable delivery channel. Relays are lightweight and require no preconditions for peer-relay delivery, such as a peer having to register at a relay. Peers wanting to establish a connection to relays have to perform a proof-of-work handshake, where the difficulty increases proportionally to the relay load. This proof-of-work handshake reduces the incentive for DDoS-attacks and can also protect against a flash crowd, where many legitimate peers try to establish connections with the relay at the same time. The set of all relays is organized in a global address space, which peers can use to find relays for sending or retrieving messages. The protocol is flexible, because any peer can use any relay to send or retrieve messages. However, the protocol does not provide a mechanism on how two peers agree on a relay, it instead assumes that applications implementing the protocol provide such a solution. Because messages are end-to-end encrypted, compromised relays only expose metadata, but not the content of messages.

3.6.5 Wesh Protocol

The Wesh protocol [44] provides end-to-end encrypted, distributed and asynchronous delivery in a P2P setting. Both one-to-one and group delivery are possible. Rendezvous points are used to establish connections between two devices and can be used for peers to register themselves, after which other peers can request the list of registered peers. Rendezvous points can either be used to add peers to the list of contacts or to exchange messages in a group (more than two peers). Wesh rendezvous points can be based either on DHTs, other decentralized servers or on local records, which are shared through direct local transports. These local transports allow direct delivery between close devices via Bluetooth Low Energy (BLE) or Wi-Fi direct. As in Matrix, messages are stored in a Directed Acyclic Graph (DAG), allowing forks and providing strong eventual consistency.

For asynchronous delivery, Wesh provides four different solutions: Bot account, linked device, replication device and replication server. A bot account runs on a server and can only be added to groups with more than two participants. A linked device is simply an additional device for a user account, to which the DAGs can be replicated. A replication device is linked to an account but does not have access to account secrets and thus cannot encrypt or decrypt messages. A replication server is owned by a third party, is not linked to an account and therefore cannot encrypt or decrypt messages, but can be added to groups in order to provide asynchronous delivery.

3.6.6 Dendrite Relay API

Dendrite is a Matrix homeserver implementation that intends to be efficient, reliable and scalable [24]. It has been used for the “Pinecone” Matrix P2P experiment, where lightweight Dendrite homeservers run on peers [45]. For this P2P setting, Dendrite includes

a Relay API [46], which allows peers (i.e., homeservers running within a peer) to act as a relays.

Relays store and forward events, thereby providing asynchronous delivery between devices. Each peer can be associated to multiple relays. Because the relay is associated only to receiving peers, the sender has to know at least one of the receiver's relays. However, the current specification does not include a process to find a receiver's relays, instead they have to be manually configured in the local database. It also does not support relays exchanging events between each other, whereby peers have to agree on the same relay in order to exchange events asynchronously. The relay is used only after a configurable number of failed connection attempts between two peers. The relay API provides an endpoint for sending and another for retrieving events. With the sending event endpoint, a peer can send an event directed to a given receiving peer to the relay. The retrieving event endpoint allows peers to pull events destined to their address from the relay.

3.6.7 Briar

Briar [47] is a peer-to-peer messaging app that focuses on privacy rather than asynchronous delivery. Peers can establish direct connections over different transports [48], e.g., Bluetooth, Wi-Fi or the Tor network. Briar defines its own message data format, the Binary Data Format. It also uses a custom transport layer security protocol, which is optimized for delay-tolerant networks. Even though privacy is a main focus of Briar, the base protocol does not provide anonymity, unlinkability or unobservability. However, since the protocol is transport-agnostic, it can be extended by using Tor to provide a stronger notion of privacy. Despite it not being the primary focus of Briar, the Mailbox extension allows asynchronous delivery between peers. Briar Mailbox [49] is an app that can run on an additional peer device and provides store- and forward functionality providing this additional device stays online. Therefore, users are in full control of the relaying functionality and have to provide the devices themselves.

3.6.8 Comparison

This section compares the different relaying solution approaches in related work, according to five different properties. The properties are *Peer Authorization*, *Relay Discovery*, and *Relay Selection*. They are discussed in the following for each relaying solution and summarized in Table 2. As all of the compared solutions provide asynchronous delivery, provided that the relaying node is online during the offline time of sending and receiving peers, this aspect is not compared.

Peer Authorization

Peer Authorization means that the node providing the relaying functionality ensures that only benign nodes can read and receive messages directed to them. The authorization of relays is not evaluated, because the selected protocols do not specify inter-relay communication protocols. This property relates to the requirement *Metadata Privacy*, because differentiating between authorized an unauthorized peers makes the Metadata available only to benign peers, e.g., peers within a certain group.

Property/System	Nostr	SSB	Async. Mobile P2P	Wesh	Dendrite Relay API	Briar
Peer Authorization	✗	✗	(✓)	✓	(✓)	✓
Relay Discovery	(✓)	(✓)	✗	✓	✗	✓
Relay Selection	✓	✓	n/a	(✓)	✓	✗

Table 2: Comparison between relaying solution approaches in related work.

✓ Property applies to system
(✓) Property partially applies to system
✗ Property does not apply to system
n/a Property is not applicable in system

In Nostr, relays are controlled by the peers and only provide their store- and forward functionality on behalf of senders to receivers that explicitly made a subscription to the sender peer. Since any peer can send messages to or request events from any relay, the relay does not implement any authorization mechanisms for either sending or receiving peers Nostr makes this design decision in order to keep the protocol as simple as possible, but allows malicious relays to send events to the relay, even though peers can perform access control of relayed messages themselves. As Nostr is an protocol for social media applications, message contents are not encrypted and can be read by any subscribers. Consequently, the aspect of metadata privacy is not a major concern for Nostr, in contrast to encrypted messaging applications. Like Nostr, SSB relays also do not authorize sending or receiving peers, as the peers enforce authorization rules themselves.

The Asynchronous Mobile P2P architecture has a global address space for relay nodes, where any peer can contact any relay to request content. Therefore relay does not perform authorization checks for receiving peers, even though receiving peers that haven't exchanged a symmetric key with the sender cannot decrypt the end-to-end-encrypted content. For sending peers however, relays do perform authorization by requiring the sender to sign each event with its private key, while the sender public key is known to the relay. Therefore, in order for the peer to be authorized to send a message, a relay must have successfully authenticated the sending peer by verifying the signature of the session key. Because the Asynchronous Mobile P2P architecture partially employs authorization of peers, because only the authorization of sending peers is determined.

Each of the four Wesh relaying solutions require the relay to be added to the group, thereby providing cryptographic keys and signatures for the group, with which the relay can determine whether a sender or receiver are part of the group. As a result, Wesh provides sender and receiver authorization. In the Dendrite Relay API, transactions holding multiple events are directed to specific peers are stored at the relay. The sent event is always directed to only one specific receiver, thereby authorizing receiving peers, while any sending peer can send any events to the relay, resulting in the Dendrite Relay API partially fulfilling the authorization property. In Briar, peers can set up their own Mailbox providing relaying functionality on the receiving side. Therefore, relays in Briar forward all events only to the configured receiver. As the peer's relay is linked to a user account, the relays also perform authorization of sending peers.

Relay Discovery

The property *Relay Discovery* is fulfilled, if the system includes an automatic mechanism that allows peers to discover relays storing events for a given group or subset of peers, in order to receive events from the relay. The ability of senders discovering available relays is not evaluated, because the solutions require the sender to manually configure the relay to be used for sending. This property relates to the requirement *Relay Discovery for Peers*, which specifies that peers should be able to obtain a list of available relay candidates.

Nostr partially provides relay discovery, as NIP-05 allows peers linking their public keys to domain names. Requests directed to this domain name return a list of relays used for sending, while the base protocol does not inherently support finding another peer's relays. In SSB, peers can publish their relays (pubs/rooms) to their own append-only log. However, other peers require access to the append-only log, in order to discover the relay. This initial discovery of another peer's relays is only possible when both seeking peer and the peer publishing a relay are online simultaneously. Because SSB provides no solution to this problem of relay bootstrapping, it only partially fulfills the relay discovery property. Asynchronous Mobile P2P does not provide an automatic relay discovery mechanism, and defers this functionality to a higher application layer.

In Wesh, members can discover another peer's relay, because the relaying solutions require either adding a device to an existing account, having a dedicated account for a relay, or adding a replication server to the group. Receiving peers can then request messages from these relaying devices. The Dendrite Relay API does not support automatic relay discovery, as peers have to configured relays that should be used manually in the local database. To use relay functionality in Briar, a user is required to manually install the Mailbox application to a local device. After the Mailbox application is configured, other peers can find the Mailbox device with the same mechanisms as they find regular Briar devices, i.e., via Wi-Fi, Bluetooth or the Tor network.

Relay Selection

The system provides *relay selection*, if a peer can choose between a set of relays for each group, in order to send or receive messages. This set may be dependent on the number of relays a sender or a group of senders uses. Because the compared solutions require the sending peer to manually add the relay used for sending, this property is not evaluated. This property relates to the requirement *Relay Selection for Peers*, which specifies that peers can select relays from a list of different relay candidates.

In Nostr, receiving peers can select between multiple relays, depending on how many relays the senders for each subscription use. The Nostr architecture encourages using different relays to utilize the advantages of decentralization. Similar to Nostr, SSB also allows a selection of relays, because sending peers can assign multiple relays (pubs) to their user account. The Asynchronous Mobile P2P relay architecture does not specify the number of relays a sending peer can user, and defers this functionality to higher-level applications of the protocol. In Wesh, sending peers can set up an arbitrary number of relays, allowing a relay selection for the receiving peers. Sending peers in the Dendrite Relay API can use different relays for sending different events. Unlike Nostr, SSB and Wesb, peers in the Dendrite Relay API do not automatically publish a list of used relays. The configured relay relation is inverse to the other protocols, as the sending peer has to find the receiver's relay. Therefore, the receiver is free to choose any relays.

In Briar, only single relays (Mailboxes) are supported per peer. Therefore, peers cannot select relays for each group, and are limited to using one relay.

Conclusion

Authorizing peers is an important property for relay-enhanced P2P Matrix, because of the *Metadata Privacy* requirement. In Matrix, only members of the room should be able to obtain metadata for the room messages. As a consequence, existing authorization mechanisms, such as Wesh's mechanism of having the relay added explicitly to the group, could provide useful solutions. A Relay Discovery mechanism enables peers to find the relays of other peers, so that messages to those other peers can be delivered asynchronously. Relays can be discovered by having peers publish a list of their selected relays or by adding relays to groups. Since metadata privacy is a requirement of P2P Matrix with relays, changing relays over time would allow distributing the Metadata to several relays. In related work, relay selection can be designed by having peers choose one or more relays themselves and storing the list of selected relays. In the case that relays do not interconnect with each other, a prerequisite for messages to be delivered asynchronously via a relay is that the sending and receiving peer's selected relay sets must have at least one relay in common. In group-based systems, having multiple relays per rooms, from which peers can choose their per-room relays, may present a suitable solution.

3.7 Summary

This chapter gave a more detailed insight into the problem of asynchronous and as-soon-as-possible delivery, by formalizing the two aspects in an abstract model, that can be applied to several classes of messaging systems. Assumptions on a P2P system for Matrix were specified, in order to have a foundation, upon which the relay extension can be built. Both functional and qualitative requirements to a solution were specified. Solution approaches to different problem aspects were analysed and compared. After that, the differences between P2P Matrix with relays and Federated Matrix were discussed, so that the relay-based solution does not re-introduce existing drawbacks of Federated Matrix. The last section analysed different architectures making use of relays for asynchronous delivery, and compared them regarding three properties a relay-based solution for P2P Matrix requires.

4. Design

This chapter describes the design of P2P Matrix with relays for asynchronous delivery. Section 4.1 provides a general, high-level overview of the architecture of P2P Matrix with relays. The design of relays is described in Section 4.2, and the design of peers in the setting of P2P Matrix with relays is described in Section 4.3. Section 4.4 provides use cases, that show how the solution works in different scenarios.

4.1 High-Level Overview

Dedicated relays extend P2P Matrix in order to provide asynchronous delivery for peers. Relays are managed per room, i.e., every room can have a set of relays. Member peers with a sufficient power level manage relays as room members. This approach utilizes the membership concept as existing Matrix functionality. Peers can send events to these relays, from which other peers can asynchronously retrieve events. Although relays have access to event metadata, they cannot decrypt contents of end-to-end encrypted room messages. To avoid metadata accumulation, relays do not store the full room history, but only a subset of the DAG. Relays disseminate events between each other, so that peers can retrieve events from any relay in the room relay set. Relays contain two essential components, the *Event Cache* and the *AuthDAG*. Events are stored in per-room event caches until a configurable retention time expires, when these events are deleted, which avoids accumulation of metadata at relays over time. The AuthDAG is a reduced DAG consisting of a room's authorization events and allows the relay to check, whether other peers or relays are authorized to send or receive events in a room.

Peers can independently decide whether to use relays or not. Also, P2P Matrix is not dependent on relays. The relay-based solution is therefore optional and does introduce additional constraints to P2P Matrix, e.g., if all room relays fail, simultaneously online peers are still able to exchange messages over the P2P distribution network. The solution is agnostic of both underlying P2P architecture of Matrix and the group communication mechanism used between relays. Any P2P architecture fulfilling the assumptions described in Section 3.2 is compatible with the relay-based solution. Abstracting from the

group communication mechanism between relays that are part of a room allows further development on a more efficient event dissemination mechanism.

Figure 13 shows how event dissemination works within a room. A sending peer sends the event to both online peers as well as to a chosen relay. The Peer Distribution Network is an abstraction of the specific P2P network implementation and represents the dissemination mechanism of events to online nodes. Events are sent to online peers via the Peer Distribution Network, because online peers can exchange events directly without the relay indirection, and because peers that do not use relays also have to receive events. After receiving the event and successful authorization checks using the AuthDAG, the relay caches the received event and disseminates it to all other relays. At a later stage, a peer that was offline at the time of sending can asynchronously retrieve the event from one of the relays by making a resynchronization (*resync*) request. The relay responds to the resync request with a set of events, containing the originally sent event.

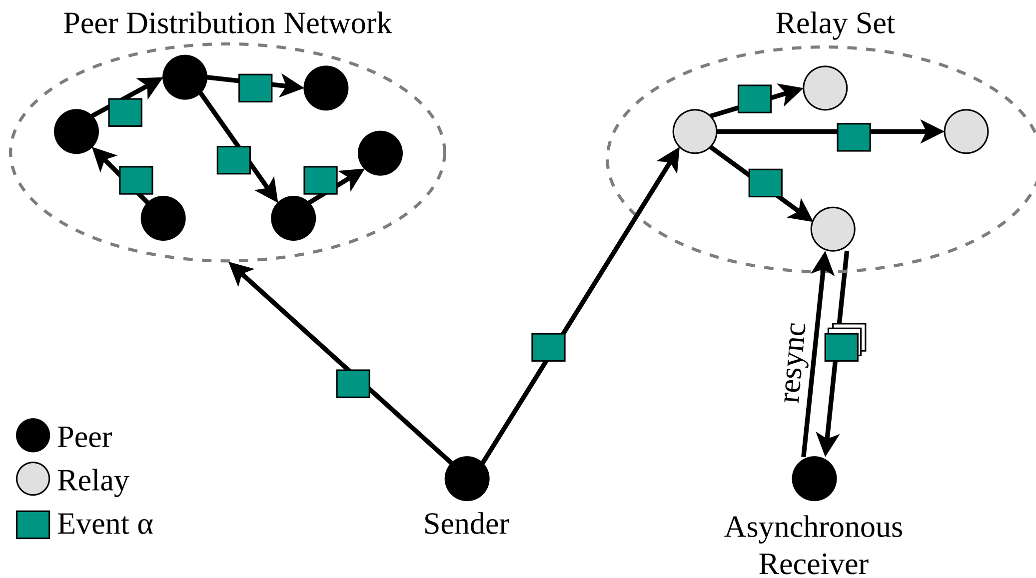

Figure 13: Architecture Overview. A peer sends an event to a set of online peers and to a relay from the relay set. After the point in time of sending, a previously offline peer comes online, resynchronizes with a relay from the relay set and receives the event.

Example. Figure 14 shows a Matrix room with a sender, a receiver and a single room relay. Because all other peers in the room are offline, the peer sends the event to the relay. The sender sends the event e to the relay at t_0. In rooms with multiple relays, the relay forwards the event to all other relays. Since only one room relay exists in this case, the event is not forwarded to any other relay. While at t_0, the receiver is offline, the receiver comes back online at t_1, and requests all events since the last time the receiver was online (t_s). After checking the sending peer's room membership, the relay then delivers all events E that occurred after t_s to the receiver, assuming that the relay received and cached those events previously. The receiver receives the event e originally sent by the sender, because it is in the delivered set E. The example shows that the relay solution provides both asynchronous and ASAP delivery, as described in Section 3.1.

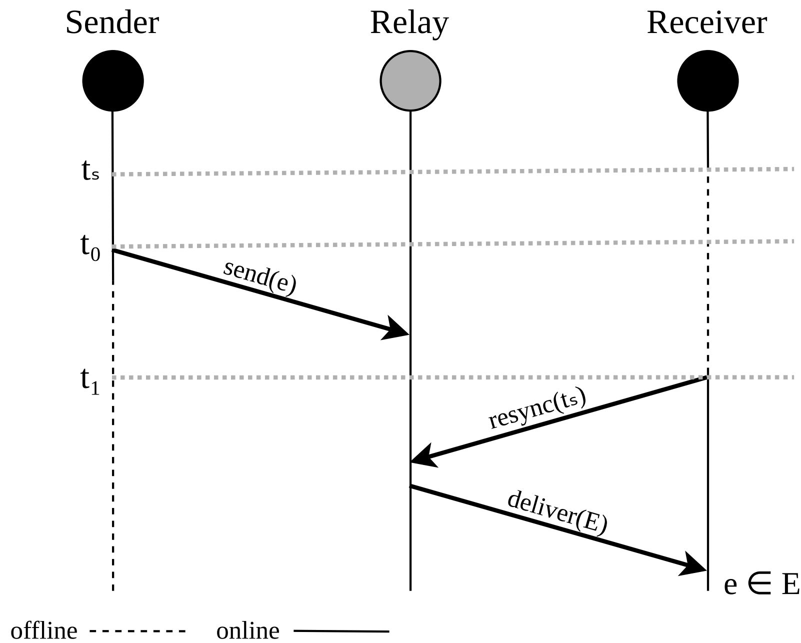

Figure 14: Asynchronous Delivery Example. Sender, receiver and relay are all room members. The sender sends the event e at t_0 to the relay, who e adds it to the set of cached events E. When returning online, the receiver sends a resync request with a since timestamp t_s. The receiver receives e as part of the returned event set E.

4.2 Relay Design

This section describes the design of relays for P2P Matrix in detail. It discusses the different relay components, the interfaces the relay provides for peers, and then describes the relay functionality.

4.2.1 Relay Overview

Relays include two essential components, the Event Cache and the AuthDAG. Figure 15 summarizes the relay components and how relays interact with peers and other relays. The points on the circle boundary, where arrows point to, represent relay API endpoints. A relay can either receive events from peers/other relays or resynchronization requests from peers. A relay responds to a resynchronization request with a set of events, after having checked the requester's authorization. After receiving events from peers or relays, a relay performs authorization checks and forwards the event to the other room relays.

Relays provide three API endpoints that can be used by other peers/relays:

- *Send* – Peer or relay sends event e to relay, which then caches event e.
- *Resync* – Peer requests events from relay cache given a *since*-parameter. All events after *since* are sent to the peer.
- *Get* – Peer requests event with given ID.

Following an API endpoint request, the relay interacts with the AuthDAG and Event Cache for different purposes. Because these API endpoints require a form of authentication, the AuthDAG contains the room state relevant for authenticating peers. While

for the *Send* endpoint, the relay determines the peer's privileges for sent event, the endpoints *Resync* and *Get* merely require checking the requesting peer's room membership. When processing a request to the *Send* endpoint, after checking the peer's authentication, the relay stores the event in the Event Cache. For the endpoints *Resync* and *Get*, the relay returns a range of requested events from the Event Cache.

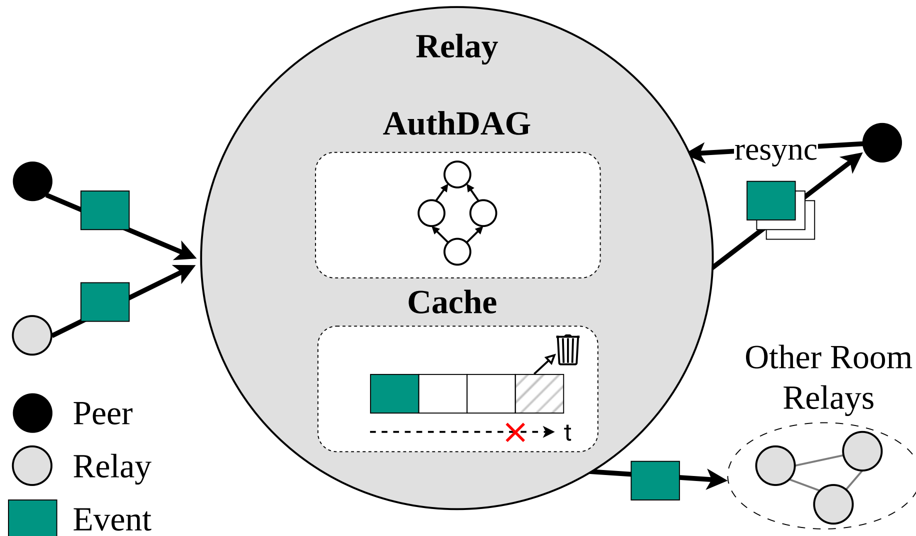

Figure 15: Relay Functionality Overview with the two main components *AuthDAG* and *Cache*. Relays receive either events from peers/relays, or resync requests from peers. After receiving an event from a peer, the relay forwards it to the other room relays.

4.2.2 AuthDAG

The AuthDAG is a reduced DAG containing control events and therefore provides necessary information for a relay to perform access control. As discussed in Section 2., state events are events that update metadata state of a room, e.g., room name. *Control events* are a subset of state events that include power levels, join rules and membership status, i.e., all state events relevant for authentication. As the AuthDAG consists of these control events, the relay can perform access control on peers and relays. For example, a relay receiving a resync-request from a peer can check whether the membership status being *join* for the corresponding peer is part of the resolved state, that can be determined with the state resolution algorithm [12].

When a relay joins a room after having received and accepted an invitation from a peer, it requests the AuthDAG from one of the online peers. After having received the AuthDAG, the relay must obtain subsequent state events in order to be up-to-date on the room's current state. In order to obtain the latest room state continuously, the relay first checks whether the received event is a control event. If so, the control event is appended to the AuthDAG. However, the relays do not necessarily receive every control event. Because online peers exchange events directly, those events are not directly sent to the relay. To allow a relay to proactively retrieve missing state events, it can request these events from other online peers or relays.

The access control mechanism at the relay protects the room from attacks when processing sent events and resync-requests. In the context of processing sent events, relays

protect the room from non-member peers or relays that could perform degradation of service attacks against the P2P Matrix network, as described in Section 3.4.2. In the context of processing resync-requests, relays protect the room from unauthorized disclosure of metadata , i.e., non-member peers requesting room events and acquiring event metadata of the room.

4.2.3 Event Cache

A relay uses the Event Cache to buffer events for the duration of a peer-configurable event retention period. On a conceptual level, the Event Cache consists of a set of *timestamp-event* tuples. When a relay receives an event, it associates a UTC timestamp to the event and stores it in the corresponding room Event Cache. The reason why relays and not peers generate the UTC timestamp is to avoid malicious peers producing out-of-order tuples with artificial timestamps. Even though tuples of events with modified timestamps do not impair the relay's functionality, this could pose a performance degradation, as relays would have to re-order the out-of-order tuples. Events can be filtered from the Event Cache by selecting events with a greater timestamp. This filtering by timestamp directly serves the *Resync* endpoint, where a peer requests all events later than the given *since* timestamp. On a conceptual level, the Event Cache can be seen as a circular buffer. Because the relay's storage capacity is limited, the newest events overwrite the oldest events, if the circular buffer is full.

The retention period is the maximum time an event is stored at the relay. Events are not stored indefinitely in the cache to avoid accumulating metadata. For each received event, a timer is started with the timeout being the a retention period. After the retention period expires, the event is deleted from the cache. The retention period is configured in a dedicated state event, allowing only peers with a sufficient power level to set the retention time. The retention period in the new state event is applied to all relays in the room, discouraging relays with longer retention times storing a larger number of events, which could lead to only the room relays with the longest retention period being used. Applying the retention period to all relays is a more uniform approach, and can prevent heavily imbalanced relay usage.

Setting an optimal retention time is challenging, as both too high and also too low retention times have their drawbacks. If the retention time is high, metadata accumulates at the relay and the event is cached longer than necessary, i.e., because the event has already reached all peers. This would remove the important difference of relays to homeservers, namely that relays do not store the whole state of the room and forget old events over time. If the retention time is too low, the relay fails to provide the core functionality of asynchronous delivery, because the relay removes the event before it can be requested by peers. The optimal retention time balancing the trade-off is subject to various factors, such as number of room relays, number of room peers, distribution of peer online times, interarrival times etc. Taking all of the factors into account in order to determine an optimal retention time represents a difficult multi-objective optimization problem. This work does not further pursue this optimization problem and relies on the peers to configure the retention time.

4.2.4 Relay Functionality

As a relay can in principle provide and respond to the defined API endpoints in many different ways, an abstraction that concentrates on the relays interface and the exchanged data types is introduced. This abstraction is named the *Matrix Broker*. The Matrix Broker uses the types *event* and *authevent*, which both correspond to PDUs (Persistent Data Units), as defined in the Matrix Specification [50]. While an *event* can represent any event, an *authevent* represents a state event with authorization modifications, e.g., joining a room, banning a user, etc. On an abstract level, an event contains the sender, content, the hashes of the previous events, and the hashes of the auth events, that would authorize the event in the room. Which state is modified for auth events is stored in the variable *type*. The Matrix Broker defines three functions: Receiving events, receiving resynchronization requests given a timestamp, and receiving event requests for single events, as shown in Listing 1. The RECEIVE function returns whether the event has been processed successfully, while the RESYNC function returns the set of events with a UTC timestamp larger than the provided *since* timestamp. The GET function returns the event with the requested event ID.

Listing 1: Matrix Broker Interface

```
1:  type event {
2:     sender: user, content: string, prev: set<hash>, auth: set<hash>
3:  }
4:  type authevent extends event {
5:     type: string
6:  }
7:
8:  interface MatrixBroker
9:     function RECEIVE (e: event): boolean
10:    function RESYNC (time: since, peer: user, auth: authevent): set<event>
11:    function GET (id: int): event
```

The Matrix Relay implements the Matrix Broker, as shown in Listing 2. The RECEIVE function updates the cache with new events and the AuthDAG with new authorization events. The RESYNC function returns the set of events with UTC timestamps larger than the provided *since* timestamp. The GET function returns the event with the provided event ID, which is the hash of event. The isValid function returns whether the provided event passes checks regarding event format, hash, and authorization. Finally, the isMember function returns true if after state resolution in the AuthDAG the membership state of the provided peer is *join*.

Listing 2: Relay Functions

```
1:  class Relay implements MatrixBroker:
2:      Cache: state set<(time, event)>
3:      AuthDAG: state set<authevent>
4:      RoomRelays: state set<user>
5:
6:      function RECEIVE (e: event): boolean
7:      function RESYNC (time: since, peer: user, auth: authevent): set<event>
8:      function GET (id: int): event
9:      function ISVALID (e: event): boolean
10:     function ISMEMBER (peer: user, auth: authevent): boolean
```

When a relay processes an event with the **RECEIVE** function (Listing 3), it first performs an event validity check, which is comparable to checks performed on receipt of a PDU as described in the Matrix Specification [50]. This validity check is performed separately in the ISVALID function (Listing 4). The function only processes the event, if the precondition of the event being valid is true. If so, the relay associates the current UTC-timestamp to it and adds it to the set of cached events. Next, if the event came from a peer, i.e., if the event sender is not in the set of room relays, the relay distributes the event to the other room relays. Events received from relays are assumed to have already been distributed to other relays. Finally, the RECEIVE function returns whether the sending of the event was successful.

Listing 3: Receive Event

```
1:  function RECEIVE(e: event)
2:      v ← ISVALID
3:      if v then
4:          timestamp ← now(UTC)
5:          Cache ← Cache ∪ (timestamp, e)
6:          ▷ If event is from peer, distribute to all other room relays
7:          if ¬(e.sender ∈ RoomRelays) then
8:              ∀r ∈ RoomRelays : deliver(e, r)
9:      return v
```

The **ISVALID** function (Listing 4) validates the event format, checks the event hash and checks event authorization and is called by the RECEIVE function. The relay initially checks if the event is in the correct format (VALIDEVENTFORMAT), i.e., it has to comply with room version event format [50]. After that, the relay determines whether the sending peer or relay is authorized to send events in the room. To check the authorization, the relay requires the complete AuthDAG at least up to the received event *e*. The COMPLETEAUTHCHAIN function verifies whether an auth chain can be calculated for a given event. The auth chain of an event is the set that can be constructed from the event's auth events and recursively adding each of its auth events. The event authorization can be determined, only if the complete auth chain is available. If events from the auth chain are missing, they are re-requested from other peers/relays in a best-effort manner with the GETMISSINGEVENTS function. The process of retrieving missing events is limited by a maximum number of tries and timeouts in a best-effort approach, because peers/relays

storing the missing events might not be online. This best-effort approach provides an upper bound on the response time of the isValid function. Best-effort means that getMissingEvents returns either the missing events after before a timeout expires, or the empty set, e.g., if all other room relays and peers are offline. Therefore, the AuthDAG is either extended with the missing events in m or remains unchanged in the case of getMissingEvents returning the empty set (AuthDAG = AuthDAG ∪ ∅). After that, the actual authorization check for the event given the AuthDAG is performed. Because the previously mentioned getMissingEvents function may return the empty set, the AuthDAG used in the isAuthorized function might still miss events. In this case, the relay cannot perform the authorization check, assumes that the peer of the event is unauthorized and the isAuthorized function returns false. Within the isAuthorized function, if a full auth chain can be calculated and the AuthDAG includes state conflicts, the state resolution algorithm is used to resolve these conflicts. If the peer is authorized and the event e in the isValid function is a state event, the AuthDAG has to be extended with the event. An event is considered valid, if the event is in the correct format, the hash is correct and the peer/relay is authorized to send the event.

Listing 4: Event Validity Check

```
1:  function isValid(e: event)
2:      ▷ Check if event format complies with specification
3:      fmt ← validEventFormat(e)
4:
5:      ▷ If events are missing in auth chain for event, get them
6:      if ¬completeAuthChain(AuthDAG, e, e.auth) then
7:          m ← getMissingEvents(AuthDAG, e)
8:          AuthDAG ← AuthDAG ∪ m
9:
10:     ▷ Perform authorization check with AuthDAG
11:     auth ← isAuthorized(AuthDAG, e)
12:
13:     ▷ Append new event to AuthDAG if it is a state event
14:     if auth ∧ isStateEvent(e) then
15:         AuthDAG ← AuthDAG ∪ e
16:     return fmt ∧ hsh ∧ auth
```

The **isMember** function (Listing 5) checks whether a peer is a member of the room, given the join event. More precisely, it checks whether the membership state of the peer, that the join event references, equals *join* after state resolution. First, the validity of the event is determined with the isValid function. Since this check also re-requests the newest events of the AuthDAG, the isMember function assumes that the AuthDAG has the latest state of any online relay or peer. The provided event must be in the AuthDAG, it should correspond to a state event that changes the peer's membership to *join* and the event sender should equal the provided peer . Because the provided event might not be the latest event modifying the peer's membership state, the final check determines whether after resolving the room state with the complete AuthDAG, the peer's membership remains *join*. If the AuthDAG includes a ban event that still remains after state resolution and is the most recent membership modification, the isMember function fails.

Listing 5: Membership check

```
1: function isMember(peer: user, auth: authevent)
2:     ▷ Check event validity
3:     if ¬isValid(auth) then
4:         return false
5:
6:     ▷ Check if provided event is in AuthDAG and if it is a correct join event
7:     if auth ∉ AuthDAG ∨ auth.type ≠ join ∨ auth.sender ≠ peer then
8:         return false
9:
10:    ▷ Membership state of peer after state resolution should be join (Not banned)
11:    return memberState(peer, auth, AuthDAG) == join
```

The **RESYNC** function (Listing 6) processes resync requests from peers that request a set of events later than the given timestamp. The requesting peer also provides an authorization event, which should be the peers's join event and allows the relay to check the peer's membership, i.e., if the peer has the necessary permission to receive room events. Because the peers want to retrieve only a subset of cached events, they provide the *since* parameter. The relay returns all events with a timestamp larger than the *since* parameter.

Listing 6: Receive Resync Request

```
1: function resync(since: time, peer: user, auth: authevent)
2:     if ¬isMember(peer, auth) then
3:         return ∅
4:     return {e | (e, t) ∈ Cache, t > since}
```

The **GET** endpoint (Listing 7) allows peers or relays to request an event with a specific ID. As in the RESYNC function, the relay checks the peer's membership state with the *isMember* function. If peers/relays are missing only one specific event and they know the id, this endpoint can be used. This endpoint is advantageous in cases where peers/relays receive an event that references an unknown auth event, but already have all other events, as they can retrieve this event by providing the specific ID.

Listing 7: Receive Get Request

```
1: function get(id: int, peer: user, auth: authevent)
2:     if ¬isMember(peer, auth) then
3:         return ∅
4:     return {e | (e, t) ∈ Cache, e.id = id}
```

4.3 Peer Design

This section describes the design of peers in the setting of P2P Matrix with relays in detail. It discusses the required changes to peers in existing P2P Matrix designs, focuses on the interaction with relays, how requests are generated and how relay responses are processed.

4.3.1 Peer Overview

Peers include both the functionality of pure P2P peers, where only peers exchange events, and peers for P2P Matrix with relays, that extends the pure P2P peer with relay functionality.

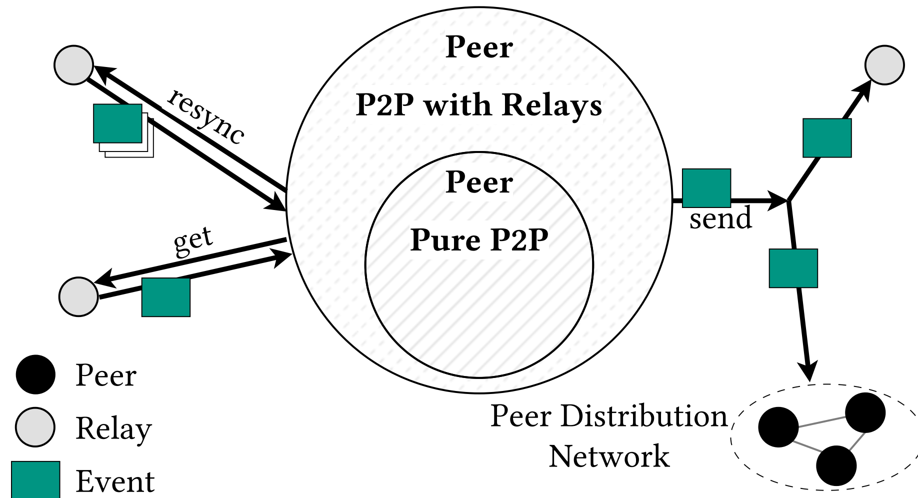

Figure 16: Peer Functionality Overview. A peer combines the functionality of a peer in pure P2P (small circle) with the relay-related functionality (large circle). A peer can resync, get specific events from relays, or send events to the relay and to online peers.

When a peer sends an event in P2P Matrix with relays, the event is sent both to a room relay and to the peer distribution network, which is the network of online peers. With the resync request, a peer can receive a subset of room events from the selected relay. In practice, this subset most often consists of those events exchanged when the requesting peer is online. With the get request, a peer can retrieve a specific event from the relay, given the event ID.

The requests a peer makes additionally to the requests of pure P2P peers are:

- *Send* – Peer sends event to both relay and online peers in the peer distribution network.
- *Resync* – Peer requests events from relay cache given a *since*-parameter.
- *Get* – Peer requests event with given ID.

4.3.2 Peer Functionality

The existing peer functionality, as described in Section 4.3.1, is extended with functionality regarding room relays, which is summarized in Listing 8. A peer stores the set of the room relay's in the state set *RoomRelays*. In practice, a peer can determine the room relays via the DAG, because room relays are room members with an additional attribute and membership events are stored in the DAG. The current relay represents the selected relay, although in practice a peer can use multiple relays, e.g., per device or per subscription. Because P2P Matrix should work also without relays, peers can decide whether to use the relay of a room (*useRelay*). Because this design is based on the one-

room model, there is only one *useRelay* state. In a multi-room model, a peer can choose which rooms a relay can be used for.

A peer has three functions that correspond to an interaction with the relay. The function SEND distributes the event to the online peers via the P2P distribution network and to the relay, if *useRelay* is true. The function RESYNC requests events newer than the provided *since* timestamp and requires the peer to provide an authorization event to prove his/her membership. The function GET requests an event with an given ID from the relay and also requires an authorization event.

Listing 8: Peer Functions

```
1:  type event {
2:     sender: user, content: string, prev: set<hash>, auth: set<hash>
3:  }
4:  type authevent extends event {
5:     type: string
6:  }
7:
8:
9:  class Peer:
10:    Self: const user
11:    RoomRelays: state set<user>
12:    CurrentRelay ∈ RoomRelays: state user
13:    UseRelay: state boolean
14:
15:    function SEND (e: event): boolean
16:    function RESYNC (time: since, auth: authevent): boolean
17:    function GET (id: int, auth: authevent): boolean
```

The **SEND** function (Listing 9) allows a peer to send an event into a room. The first step for the peer is to distribute the event to online peers. This direct peer distribution is necessary because even with relays, online peers should exchange events directly, in order allow P2P Matrix to work when the relays are unavailable or peers choose not to use relays. The distribution to online peers is considered successful, if the event has been sent to at least one of the online peers, and is stored in the *success* variable. If a peer chooses not to use a relay, has not selected a relay, or the current room has no relays, the successful delivery depends on the previous distribution to online peers (*success* variable). If however room relays exist, a relay is selected, the online peer distribution was successful and the relay responds with a success, the SEND function returns true.

Listing 9: Send Event

```
1:  function SEND(e: event)
2:        success ← distributeToOnlinePeers(e)
3:        if ¬UseRelay ∨ CurrentRelay = null ∨ ¬ROOMHASRELAYS() then
4:              return success
5:        return CurrentRelay.RECEIVE(e) ∧ success
```

The **RESYNC** function (Listing 10) allows a peer to resynchronize events. The *resync* function cannot make the request to a relay, if the peer chooses not to use a relay, the room has no relays, or a relay has not been selected. In this case, the peer tries to resynchronize with currently online peers. The SYNCPEERS returns true, if the peer resynchronized with at least one other peer. If all of these preconditions are true, the peer requests all events newer than the provided *since* timestamp and provides the *authEvent*, allowing the relay to perform authorization checks.

Listing 10: Resynchronize events

```
1: function RESYNC(since: time, auth: authevent)
2:     if ¬useRelay ∨ CurrentRelay = null then
3:         ▷ Synchronize with online peers
4:         return SYNCPEERS()
5:     return CurrentRelay.RESYNC(since, Self, auth)
```

The **GET** function (Listing 11) allows a peer to request an event with a specific ID from a relay. The *get* function cannot make the request to a relay, if the peer chooses not to use a relay, the room has no relays, or a relay has not been selected. If all of these preconditions are true, the peer requests the event with a given ID and provides the *authEvent*, allowing the relay to perform authorization checks. This function is especially helpful, when after receiving events from the RESYNC function, only a few events are missing in the peer's DAG. Because all events reference previous events, the peer knows the missing event IDs.

Listing 11: Get event with given ID from relay

```
1: function GET(id: int, e: authevent)
2:     if ¬useRelay ∨ RoomRelays = ∅ ∨ CurrentRelay = null then
3:         ▷ Get event from online peer
4:         return GETFROMPEER(id, e))
5:     return CurrentRelay.GET(id, Self, auth)
```

4.4 Use Cases

This section presents three different use cases, that show how the design works in detail and which corner-cases have to be considered.

4.4.1 Add Relay Process

Because relays are managed as room members, a relay is added to a room as a regular peer. The process of a relay joining a room depends on the room's join rules, as described in Section 2.1.2. While in a public room, any relay can join the room, this use case focuses on private rooms, where relays have to be invited by peers in order to join the room.

In the first step, a peer sends an invitation to a relay to join the room. The peer also publishes an invitation event in the room, whereby all online member peers are notified that a relay has been invited. The relay joins the room by publishing a join event into the room. The peers enforce the access control mechanism when processing the join event, as only peers with a required power level are permitted to invite other peers. Whether

the peers add the join event to their DAG depends on the room state, which is calculated by the state resolution algorithm, as described in Section 2.1.3. Having joined, the relay obtains state events from other online peers, which is necessary to construct the AuthDAG. In this step, at least one online peer is required for the relay to request events from. Even if the peer is malicious and the peer's DAG includes altered state events, benign peers eventually coming online later and re-syncing with a relay only accept authorized events, because they also have the DAG, which holds room state necessary for enforcing authorization rules. After that, peers can use the relay-related functions, as described in Section 4.3.1. Managing relays as room members therefore utilizes existing access control in Matrix, and allows only those peers to add relays to a room that have the necessary power level. For relays to construct the AuthDAG, at least one peer in the room must be online.

4.4.2 Room Join

Even though relays provide asynchronous delivery of room events for joined peers, they do not provide asynchronous room joins. An asynchronous room join is the process of a peer becoming member of the room and obtaining the room DAG without the other peers having to be simultaneously online. In the case of invite-only rooms, an asynchronous room join can also require an invitation from an existing peer. P2P Matrix with relays does not support joining rooms asynchronously, because the joining peer cannot retrieve the complete room DAG from the relay — only the cached events. However, as the relay stores the AuthDAG and the cache, the peer could in principle obtain the AuthDAG and existing events from the cache. With the AuthDAG, the peer obtains membership information and knows from which other peers the DAG can be obtained. While a room join with relays does not enable the peer to build the complete DAG, the peer can obtain a subset of events that exist in the cache, and thus, a subset of previous room events.

4.4.3 Relayless Peers

Peers can either choose to use relays for asynchronous delivery, or they can choose not to use them. When a peer does not use a relay and therefore does not send events to the relay, those events cannot be asynchronously retrieved from the relay with the resync-request. Figure 17 illustrates a room where two peers use the relay, but another peer does not use the relay. At t_0, a relay-using peer sends event a to both the other online peer and the relay, while the other peer, who is not using the relay, sends event b to only the other peer. When at $t_1, t_1 > t_0$ both sending peers are online and a third peer comes online and resynchronizes with the relay, only event a is returned, not event b. For the event b to be available asynchronously, peers would have to forward events on behalf of non-relay-using peers the relay. This would require the peers to agree on the forwarding peer in order to reduce messaging overhead. Such a forwarding mechanism is not realized in the design in order to keep the functionality simple.

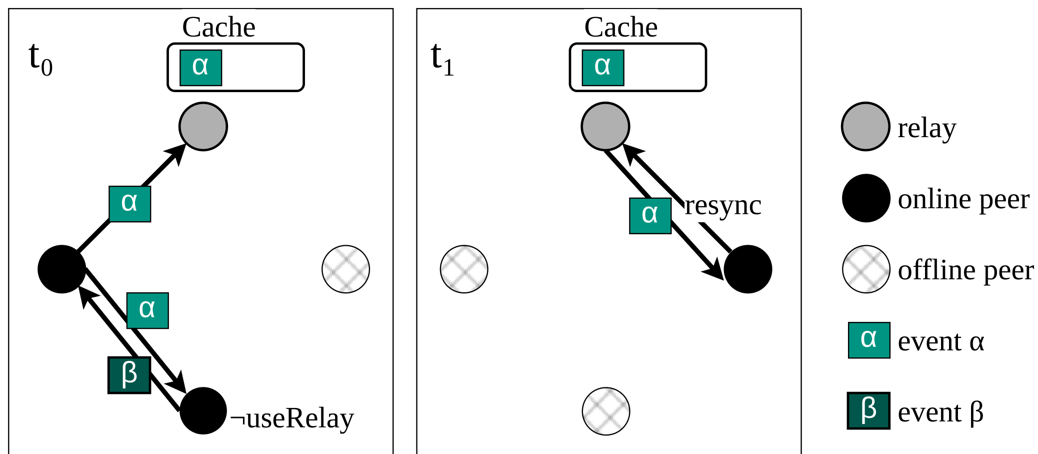

Figure 17: Relayless Peers. The event β, sent by a relayless peer (¬useRelay) at t_0, is not returned by the relay, responding to a resync-request made by a previously offline peer at t_0.

4.5 Summary

The design chapter first outlines the high-level system architecture for P2P Matrix with relays, and then focuses on the functionality of relays and peers. The high-level system architecture consists of a set of online peers connected via the Peer Distribution Network, a set of offline peers and the relay set consisting of all room relays. The two main relay components allowing asynchronous delivery (Event Cache) and access control (AuthDAG) are described. The functionality of relays is explained by separately specifying the individual functions. The functions corresponding to the relay's API endpoints are specified in the Matrix Broker interface. The necessary modifications to peers for relay-enhanced P2P Matrix are also shown by specifying each peer functions. Finally, three different use cases for P2P Matrix with relays are discussed, each of which provide insights into how the solution handles various edge cases.

5. Implementation

This chapter provides an overview of the implementation of relays in P2P Matrix. First, existing implementations relevant for the P2P Matrix relay extension are presented in Section 5.1. Then, a general overview over the modified components is given in Section 5.2. Finally, the implementation of relays (Section 5.3) and peers (Section 5.4) is discussed.

5.1 Existing Implementations

This section presents existing implementations relevant for the P2P Matrix relay extension. Implementation possibilities for P2P Matrix with relays, based on existing Matrix software, are analysed in Section 5.1.1. Because *Dendrite* is selected as the peer's locally run homeserver, both the homeserver itself (Section 5.1.2) and the deployment with the P2P network *Pinecone* are presented (Section 5.1.3).

5.1.1 Analysis of Existing Approaches

As outlined in Section 2.3.2, the current P2P Matrix implementations utilize the existing Matrix protocol for P2P, such that peers consist of Matrix clients and Matrix homeservers run on local devices. Thereby, required client-modifications for P2P Matrix are minimized, as they communicate with the locally run homeserver via the Client-Server API, which they already do in Federated Matrix. Additionally, retaining the separation between the client and server functionality in P2P Matrix allows using different clients for the same server. Consequently, the development of the P2P functionality for Matrix can focus on modifying homeservers. The alternative approach of extending clients with the P2P functionality would require implementing much functionality already provided by homeservers, e.g., the DAG, state resolution and API endpoints.

There are three existing implementations for P2P Matrix: Matrix-over-libp2p, Matrix-over-Yggdrasil, and Pinecone, as described in Section 2.3.2. All of these implementations use the homeserver *Dendrite*, because it is memory-efficient and has 100% Server-Server API parity [24] with *Synapse*, the reference homeserver implementation. The memory-efficiency enables the deployment of P2P Matrix on mobile devices, where Dendrite runs

as a service in the background, alongside a Matrix client (e.g., Element). In summary, Dendrite has already been successfully used in P2P Matrix scenarios, has been deployed on mobile devices alongside the Element client, and the P2P Dendrite implementations have been made publicly available. A more detailed description of the Dendrite homeserver can be found in Section 5.1.2.

The Dendrite homeserver is chosen as the peer implementation due to the discussed advantages of Dendrite in P2P Matrix scenarios, and because P2P deployments already exists, in contrast to other homeservers, such as Synapse [51], Conduit [52] or Construct [53]. Therefore, Dendrite is used as a basis for implementing the additional relay-related functionality for peers. The P2P Matrix implementation *Pinecone* is chosen as underlying P2P network protocol, because it mitigates the shortcomings of the Yggdrasil network, as described in Section 2.3.2, and because a working deployment of Matrix peers running Pinecone has been made publicly available, in contrast to the Matrix-over-libp2p implementation.

Relays could either be based on an existing homeserver, or they could be written from scratch. Using a homeserver implementation has the advantage of utilizing existing functionality that could be re-used for relays, such as API endpoints, DAG functionality, that can be used for the AuthDAG implementation, and a communication protocol between homeservers, that can be utilized for both relay-peer and relay-relay communication. Therefore, relays are based on a homeserver implementation. Again, in principle any homeserver could be chosen. Dendrite is selected as a basis for the relay implementation, as it is an API-compliant homeserver, and provides a scalable and modular architecture, allowing the relay functionality to represent a new component.

5.1.2 Dendrite

Dendrite [24] is a second-generation homeserver written in Go and was developed as an alternative to Synapse [51], the original reference homeserver implementation. As already mentioned, Dendrite makes improvements regarding efficiency and scalability compared to Synapse, as it has a smaller memory footprint with a better baseline performance, can run on multiple machines and scales to large homeserver deployments [24]. The compliance with the Matrix specification is tested by both the same test suite as Synapse and a new go test suite is added. The architecture is structured into different microservice components that all can independently scale horizontally and have a dedicated database. Therefore, the microservice architecture allows independent component upgrades, schema updates etc. The microservice architecture is an important improvement compared to Synapse, because it does not have the scalability problems that come with having a single database for the homeserver. Even though components are independent in principle, Dendrite also has a monolithic setup, where the components are assembled to one single binary [54]. Dendrite includes different top-level modes that combine the microservice components for different use cases, three of which are:

1. *Dendrite*: Regular homeserver mode for Federated Matrix.
2. *Dendrite Demo Yggdrasil*: Homeserver as peer in the Yggdrasil [26] network.
3. *Dendrite Demo Pinecone*: Homeserver as peer in the Pinecone [27] network. Detailed description in Section 5.1.3.

While the top-level mode *Dendrite* is used to deploy homeservers in Federated Matrix, the other two modes apply the different P2P network implementations *Yggdrasil* and

Pinecone respectively, as described in Section 2.3.2, to the homeserver functionality. Matrix functionality needed by homeservers in general such as state resolution, federation requests, or parsing JSON data is implemented in a separate library, the gomatrixserverlib [55]. Dendrite uses this library in various components. These different Dendrite components are described as follows:

- *Room Server*
  Models homeserver rooms and all operations related to the DAG, making use of the gomatrixserverlib for common homeserver functionality, such as state resolution.
- *Sync*
  Implements the synchronization functionality from the Client-Server API in the Matrix Specification. With the sync endpoint, clients can request a range of events from the homeserver for a given room.
- *Federation*
  Implements the Server-Server API from the Matrix Specification, describing the communication protocol between homeservers.
- *Client*
  Implements the Client-Server API and provides the communication protocol between clients and homeservers.
- *User*
  Models user-related processes for the local homeserver, such as account management, login and registration.
- *Media*
  Implements the media repository functionality from the Client-Server API, where users can upload files to the homeserver.
- *AppService*
  Implements the Application Service API from the Matrix specification, which enables modular extensions to the homeserver functionality, such as protocol bridges that allow Matrix users to exchange messages with other messaging platforms, e.g., WhatsApp or Signal.

5.1.3 Dendrite-Demo-Pinecone

Dendrite is especially useful for P2P Matrix, because it can be run efficiently on client devices. This allows a P2P Matrix architecture without having to change the Matrix Protocol, as both homeservers and clients are run on peer devices, whereas peers exchange events by having their locally run homeservers exchange events. Two existing Matrix demos [29], [28] run both a Dendrite homeserver and the Element client on a mobile iOS or Android device, while the local homeserver runs as a peer in a Matrix Pinecone network.

The Dendrite Pinecone demo runs a homeserver as a Pinecone peer, that can discover and connect to other peers, forming a logical Pinecone overlay network. Any Matrix client can connect to the peer's local homeserver, create an account and log in to a homeserver. The demo also includes store- and forward relays, which however have to be manually configured in the peer's database and only store and forward transactions directed to single peers without any additional access control. Because a Docker configuration file is provided, the Dendrite Pinecone demo can also be run as a Docker container, allowing reproducible deployments and further development of multiple containers running in a custom network.

The Dendrite Pinecone implementation uses the Dendrite monolithic setup, which includes all homeserver components, and additionally the P2P monolith, which includes Pinecone and P2P-related functionality. Public and private cryptographic keys are generated for the Pinecone peer, as the public key serves as routing identifier, while the private key is used to sign Pinecone announcements or bootstrap messages. The P2P monolith comprises all components necessary for a Pinecone peer, such as a Pinecone router, a connection manager, and a multicast discovery mechanism. When starting the peer, the connection manager triggers periodic multicast discovery messages, whereby other peers within a local network can be found. These multicast discovery messages are implemented both in IPv4 and IPv6, and use UDP on the transport layer. Also, HTTP servers are started that handle Matrix requests from either clients or other peers. Finally, when a peer is added, and a relay has already been manually configured in the database, all transactions directed to the current peer's user are requested from the relay.

5.2 Overview

This section provides an overview on the changes made to peers of the existing P2P Matrix demo, and the implementation of relays to P2P Matrix.

In addition to the three Dendrite modes, as described in Section 5.1.2, the implementation introduces a new Dendrite mode: *Dendrite P2P Relays*. This new mode assembles all necessary components for a relay-enhanced P2P Dendrite demo with Pinecone peers. The mode can be used to run either a peer or a relay, which is configurable by a flag. Peers and relays share one mode, because they share a considerable amount of functionality and separating them into two different modes would result in a large amount of redundant functionality.

Apart from the new mode, the implementation also introduces a new component, the *Room Relay*. This component is named Room Relay in order to stress that relays are configured per-room, and also to differentiate it from the existing Relay component (as described in Section 3.6.6), where relays are statically configured in the database and forward transactions to single peers without additional authorization mechanisms. The Room Relay API provides *resync* functionality and includes both making resync requests to relays for peers, and processing resync requests from peers for relays. Figure 18 depicts the changes made to the respective components. The only new component *Room Relay API* depends on the User API, the Federation API, and the RoomServer API. These three components are also modified, in order to support both relay-related functionality for peers and the integration of relays into the existing microservice architecture.

While Section 4.2 models relays in a one-room model, the implementation supports multiple rooms with different sets of relays. Section 5.3 describes the implementation of the relay functions, which is based on the design from Section 4.2.4, while Section 5.4 describes the implementation of the peer functions, which is based on the design from Section 4.3.2.

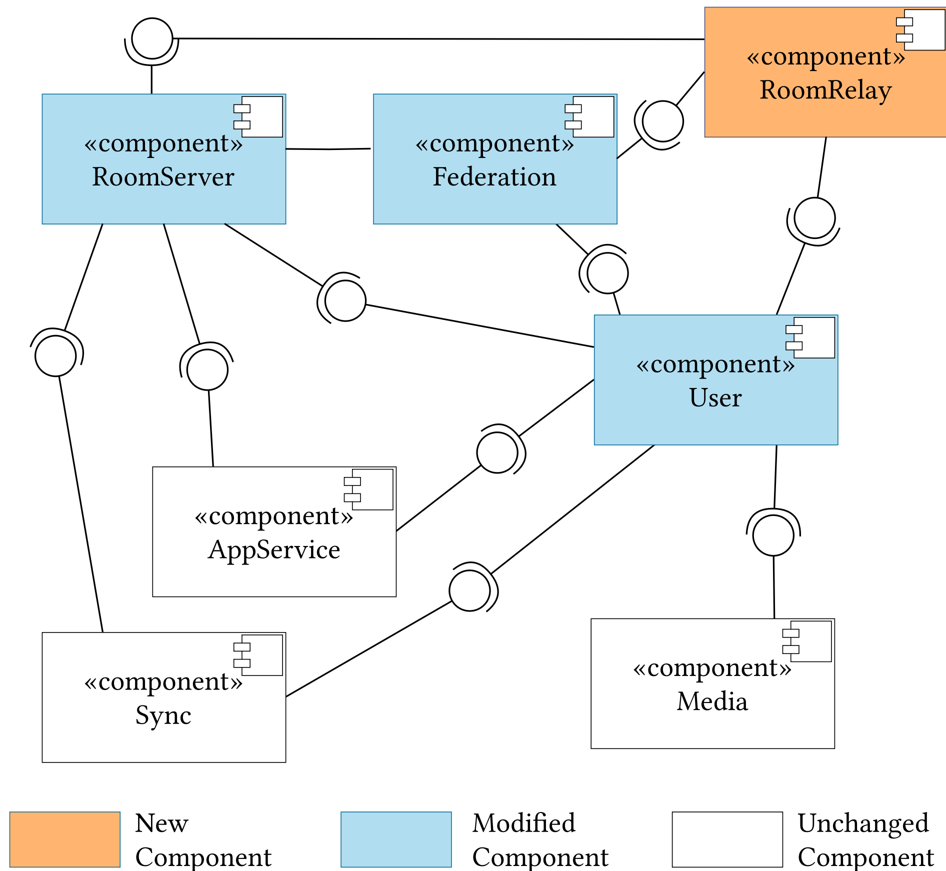

Figure 18: Changes made to Dendrite Components. For simplicity reasons, the unchanged component *Client API* is not depicted. Aside from dependencies for the new component *RoomRelay API*, no inter-component dependencies are changed.

5.3 Relay Functionality

The relay implementation is based on the Dendrite Pinecone Demo, described in Section 5.1.3. There are several reasons for this implementation decision:

- The Demo provides a working P2P Matrix Setup, where peers can exchange events. By using this demo, no additional work is necessary to implement P2P-specific transport functionality. Consequently, within the Dendrite Pinecone Demo, relays can be regarded as modified peers.
- The Demo uses Dendrite and therefore includes all homeserver functionality. Relays use parts of the homeserver-related functionality, in order to avoid developing redundant functionality.
- By utilizing parts of the existing implementation, the relays are built upon an already tested, validated and maintained code base.
- Re-using existing code allows focusing on the core functionality of relays, i.e., Cache, AuthDAG, inter-relay event forwarding and providing resync functionality.

The implementation of relay functionality is described in the same order as in Section 4.2, beginning with the AuthDAG and Cache, and continuing with each conceptual function provided by relays.

5.3.1 AuthDAG

In Dendrite, the *Room Server API* is responsible for all operations concerning the DAG. A peer's internal component implementing the Room Server API and handling all room-related operations is referred to as the *room server*. For incoming events, the room server has the sub-component *inputer* that consumes input streams and processes events of all rooms. The inputer queues incoming events of all rooms, and then assigns each event to a worker for the event's respective room. The room server's *processRoomEvent* function performs event validity (e.g., validate JSON, check if room exists) and authorization checks (e.g, check if event is allowed based on current room state), and stores events that pass all checks in the DAG.

Since the `processRoomEvent` function already implements the authorization checks from the *isValid* function Section 4.2.4 (Listing 4), it is extended by the relay's AuthDAG functionality. Two changes are made to this function:

1. Relays only store state events persistently in the DAG, thereby forming the AuthDAG. As only state events, and no message events are stored, the *previous event* field may point to an event hash that is not stored in the DAG. However, because only the *auth event* relation is relevant for authorization checks, storing only state events in the DAG (i.e., the AuthDAG) for authorization purposes is possible. While homeservers in Federated Matrix store message events in the DAG to establish the causal order relation, relays in P2P Matrix do not have to provide this relation, as peers already establish this relation by storing the DAG.
2. Events passing all authorization checks are cached. Separate caches exist for each room. If an event should be cached, but no cache exists yet for the room, the relay creates a new room cache and stores the event in it. Relays cache all authorized events, including state and message events.

5.3.2 Event Cache

The *Event Cache* provides an interface for the relays to globally store and retrieve PDUs. As Dendrite already includes a set of internal caches, this set is extended by the Event Cache. A conceptual overview on the Event Cache is depicted in Figure 19. On the highest level, the global cache index maps room identifiers to a pointer to the start of the room-specific cache, as a relay stores a separate cache for the rooms it is member in. The entries of a room cache are stored in a slice, which is a resizable go datatype referencing an underlying array. The room cache stores tuples of UTC timestamps and PDUs in a slice. Therefore, the cache is a non-persistent storage, so that a relay shutting down results in the loss of cached data. However, this is acceptable for the proof-of-concept implementation, where relays are assumed to be online.

Three cache operations are provided:

- *Store PDU*: Store the PDU with a UTC timestamp in the given room's cache. This operation is called in the process of receiving events, when events pass all validity and authorization checks.
- *Get PDUs by UTC timestamp*: Return subset of events in cache with timestamp later than given timestamp. This operation is called when processing resync-requests, after having determined the requesting peer's room membership.
- *Get PDU by ID*: Return event with given ID. This operation is called when an authorized peer requests an event with a given ID from the relay.

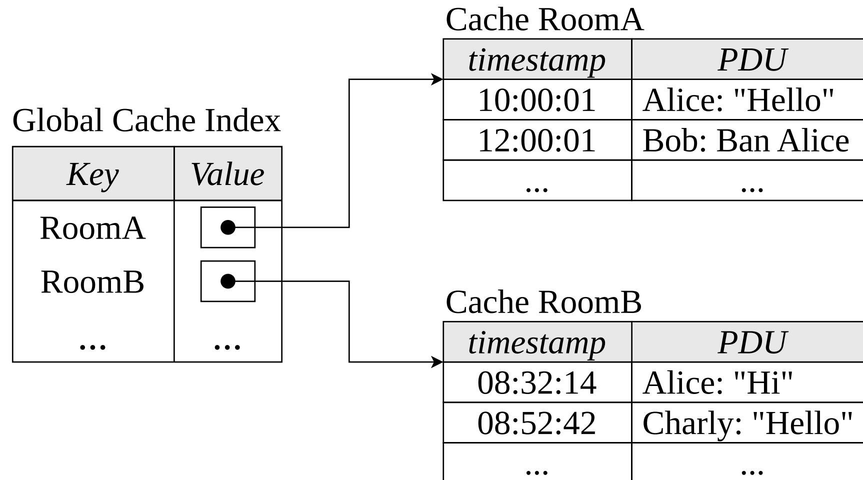

Figure 19: Implementation of the Event Cache. The global cache index maps each room to a pointer to the room cache, where tuples of timestamps and PDUs are stored.

Considering performance, the cache operations of the cache datatype can be optimized. The operations *Store PDU* and *Get PDU by UTC timestamp* are taken into account here, since they are assumed to be the most frequent cache operations. The data type could be optimized in two respects: Efficient insertion of elements (*Store PDU*), and efficient retrieval of multiple elements (*Get PDU by UTC timestamp*).

For unsorted linked lists and array lists, the insertion complexity is $O(1)$, assuming the insertion is performed at the first element for linked lists, and at the end for array lists. In both cases, the retrieval complexity is $O(n)$, n denoting the number of cache entries. In the worst case, all n cache entries have to be compared to the given timestamp. For sorted linked lists or array lists, the insertion complexity is $O(n)$, because the new element has to be inserted in the correct position, so that the data type remains sorted. The retrieval complexity is $O(\log(n) + m)$, since with a sorted data structure a binary search can be performed to search for the closest element to the timestamp, after which the next m elements following the closest element are returned. The self-balancing binary search tree stores the elements in an already binary sorted and also balanced data structure, allowing not only a retrieval complexity of $O(\log(n) + m)$, but also an insertion complexity of $O(\log(n))$. As this data type has the most efficient insertion and retrieval mechanism, this is the most efficient data type for the room relay cache. The comparison of the data type's insertion and retrieval complexity is summarized in Table 3.

Data Type	**Insertion Complexity**	**Retrieval Complexity**
Unsorted {Linked,Array-}List	$O(1)$	$O(n)$
Sorted {Linked,Array-}List	$O(n)$	$O(\log(n) + m)$
Self-Balancing Binary Search Tree	$O(\log(n))$	$O(\log(n) + m)$

Table 3: Insertion and retrieval complexity comparison between cache data types, n denoting the number of cache entries, m denoting the number of cache entries following the closest entry.

5.3.3 Receive Events

The first change to the existing implementation is to modify the functionality of receiving events for relays. Homeservers process events as *Persistent Data Units* (PDU), which wrap events generated by the client in a data structure that holds all necessary information to establish the partial ordering in the DAG, and also information to determine the event's authenticity and integrity. Homeservers do not exchange PDUs directly, they wrap them in transactions, which are data structures holding multiple PDUs and exchanged only between homeservers, not between clients and homeservers.

The Dendrite components *Federation API* and *RoomServer API* are modified in order to implement the *receiveEvent* functionality (Listing 3, Section 4.2.4). While the *RoomServer API* is responsible for validating and authorizing the event, the *Federation API* forwards processed events to other homeservers/peers in the room.

Changes to the `processRoomEvent` function within the *RoomServer API* are described in Section 5.3.1. In short, from all successfully validated and authorized events, only state events are persistently stored in the AuthDAG, and then the events are cached via the *Store PDU* function with the current UTC timestamp in the respective room cache, as described in Section 5.3.2. Finally, events are added to the room servers output log, where they are processed further by the *Federation API*.

In the *Federation API*, the internal federation server receives a new event from the room server output log, the *processMessage* function configures the homeservers/peer the event should be forwarded to. In the original Dendrite implementation, the federation server sends the event to the peers that were joined before the event. In the relay extension, the federation server sends the event only to other room relays, since peers can receive from other peers directly when synchronously online, or resynchronize with a relay, when asynchronously online with other peers. At the end of the *processMessage* function, the events and their respective destinations are stored in an outgoing queue, where the events are eventually forwarded to their destinations.

5.3.4 Receive Resync Request

Processing resync requests is implemented in the new component *RoomRelay API*, where also making resync requests (Section 5.4) is implemented. This functionality corresponds to the resync function, as shown in Listing 3 from Section 4.2.4. A new http(s) GET endpoint is added, in the format

`http{s}://<relay>/_matrix/relay/resync/{roomID}/{userID}/{since}`.

After having received a request at this endpoint, the relay first checks whether room with the provided roomID exists. Then, the relay performs an authorization check, determines if the user with the provided userID is a member of the requested room and rejects the requests if the user is not a member. Additionally, to avoid unauthorized users providing the userID of a different user, an authentication mechanism is required. While in the Matrix specification's Client-Server API, clients are authenticated via a login/registration process, in the Server-Server API servers are authenticated via public key signatures. As the authentication mechanism of this request depends on the overall authentication mechanism of P2P Matrix, this work assumes that the P2P Matrix architecture provides peers authentication.

After having authenticated and authorized the requesting peer, the relay checks if the room exists and if a cache for the room exists. If so, the cache function *Get PDU by UTC timestamp* is called for the room cache and all events with a timestamp later than the *since* parameter of the resync request are returned.

5.4 Peer Functionality

The peer implementation is based on the Dendrite Pinecone Demo, described in Section 5.1.3. Because the demo already models peers in a P2P environment, the setup can be reused for peers interacting with relays. As Dendrite and Pinecone have already been successfully deployed as apps for mobile devices [28], [29], the deployment of peers with additional features for relay-related requests to mobile devices only requires updating Dendrite components. The implementation of peer functionality is described in the same order as in Section 4.3.2. First, the modifications for sending events are presented, then, the implementation of making resync requests is discussed.

5.4.1 Relay Selection

Relay selection is the process of peers selecting a relay for a specific room. While Section 4 abstracted from the relay selection problem, the implementation uses a pseudo-random selection mechanism. A selection seed can be passed to the peer, and is then used to instantiate a pseudo-random generator. From the set of relays currently in the room, each peer selects the relay provided by the pseudo-random generator. The selected relay is then stored in a global variable storing per-room relay selections, and thus is accessible from all other Dendrite components. Finally, peers use the selected relay for both sending events (Section 5.4.2) and making resync requests (Section 5.4.3). This relay selection mechanism allows reproducible evaluations, because with identical selection seeds, peers select the same relay in each evaluation run.

5.4.2 Send Event

When a client sends an event to the peers server, the room server component first determines whether the event is valid and authorized. After that, the event is added to the room servers output log, where it is processed further by the *Federation API*, as already seen in the relay's process of receiving events in Section 5.3.3. The *processMessage* function in the *Federation API* determines, which other peers/relays the event should be forwarded to. Intially, the peer has a list of room members, including peers and relays. The peer filters this list, and only forwards events to the selected relay and to other peers. The map of the selected relay for each room is configured in a global variable.

As mentioned in Section 5.3.3, the event is added to an outgoing queue, where it is eventually forwarded to its destinations. After an event is forwarded to the outgoing queue, the local federation server tries to forward the event to the destinations. If destinations are offline, the event remains in the queue and is forwarded to each destination, whenever it comes back online. Therefore, it is important to add all other room member peers (not only currently online member peers) to the event when adding it to the outgoing queue.

5.4.3 Make Resync Request

When returning online, a peer makes resync requests for each member room. If not already completed, a relay for the respective room is selected. The process of peers making resync requests to a given relay for a given room is implemented in the new component *RoomRelay API*. First, the peer queries the event ID of its own join event from the local room server, which provides all DAG-related operations. Then, the peer queries the last online time of the current device, as a peer can have multiple devices with different online times. At this point, the P2P Matrix architecture deviates from the Federated architecture, as there is a one-to-one relation between device and homeserver in P2P Matrix. If a peer has multiple devices, it also has multiple homeservers, because homeservers are assumed to be run on devices directly. Having determined the last online time of the current device, the peer makes the resync request to the selected relay, with the endpoint from Section 5.3.4. If the peer is authenticated, authorized, the room exists and the selected relay has events after the last online time, the relay returns a transaction, which holds multiple PDUs. The peer passes the transaction to the existing *processTransaction* function, where it is added to the local DAG after having verified and authorized each event.

5.5 Summary

In summary, this chapter discussed the implementation of both relays and peers based on the Dendrite Pinecone implementation. Dendrite is chosen due to its modular architecture and the publicly available P2P implementation. A new *RoomRelay* module is added to the architecture, including both functionality of processing resync requests for relays, and also sending resync requests to the relays. While the relay's AuthDAG functionality reuses most of Dendrite's existing DAG functionality, the event cache is implemented from scratch. For the peers, a relay selection mechanism is introduced, and the process of sending events is modified, in order to include the selected relay to the set of receivers.

6. Evaluation

Relays in P2P Matrix are evaluated based on the implementation described in Section 5. An overview of the evaluation setup is provided in Section 6.1. After that, the WhatsApp traffic dataset is analysed in Section 6.2. An overview of all evaluated aspects is given in Section 6.3. Section 6.4 describes the functional evaluation of the implementation and includes functional tests for both peers and relays. Then, Section 6.5 focuses on the performance enhancements of relay-enhanced P2P Matrix compared to pure P2P Matrix without relays.

6.1 Evaluation Setup

This section focuses on how the evaluations are performed and which components are used for the evaluation. After providing an overview on the evaluation setup, the traffic generator, the docker network and the logging mechanism are discussed.

6.1.1 Overview

The evaluation setup, consisting of used components and the information flow between those, is summarized in Figure 20. First, the traffic workloads between peers (and relays) are used to generate Matrix event traffic. Two types of workloads are used in the evaluation. First, manually written traffic files contain customized traffic for each functional evaluation experiment. Second, a public instant messaging dataset [56] containing WhatsApp group traffic is used for the performance evaluation experiments. The traffic generator uses either of the two workloads to trigger sending events as a Matrix client via the Matrix Client-Server API, or setting peers on-/offline via the Docker API.

All peers and relays are run in separate docker containers, connected by a network bridge of the host system. Peers and relays store their actions into a log files on the host system. Finally, all log files are merged into a single log file, that represents the result of the experiment and is used for further analysis. The following sections discuss the evaluation setup components in further detail.

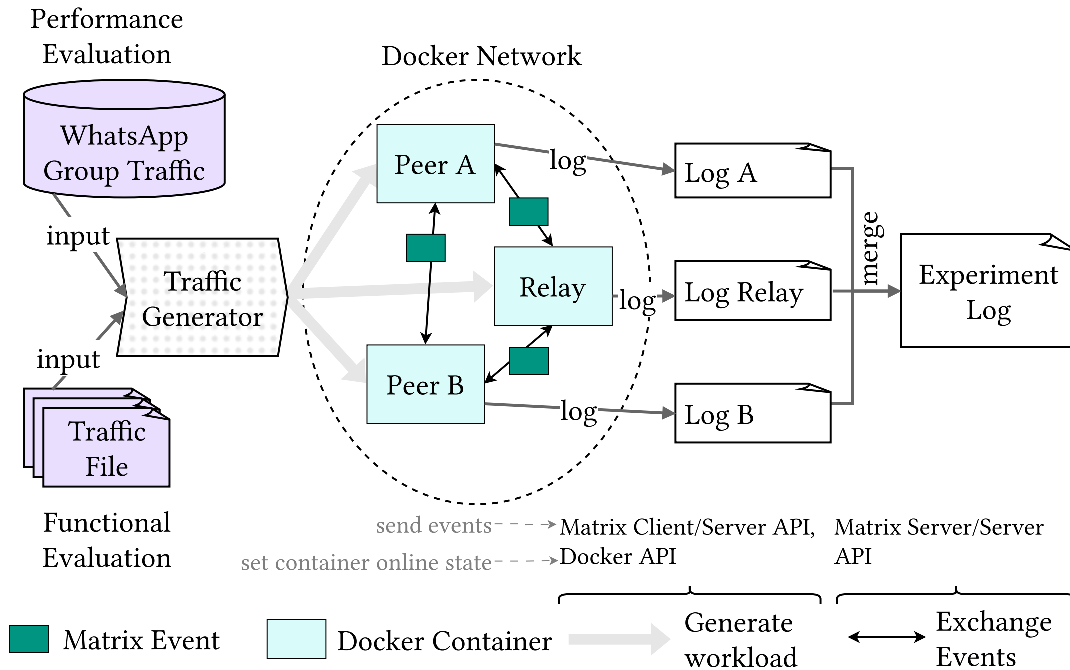

Figure 20: Evaluation Setup. The traffic generator generates workloads for the peers, sends traffic based on the provided input. Each peer and relay stores its actions (e.g, sending, receiving, online state) in a log. The merged log forms the basis for the evaluation.

6.1.2 Traffic Generator

The traffic generator triggers different actions for peers, based on either custom traffic files or instances of the WhatsApp group traffic set. Running from the host system, the traffic generator sends request as a Matrix client to the peers, that run in docker containers and act as Matrix servers. To generate client requests from the traffic generator to the peer, the Matrix SDK Mautrix-Go [57] is used. Because the evaluation requires peers to go offline in order to test asynchronous delivery, the traffic generator changes the online state of peers via the Go-Dockerclient [58] that implements the Docker Engine API [59].

Mautrix-Go [57] is a Golang Matrix SDK based on the gomatrix [60] client, a Matrix client written in go by the Matrix.org foundation. It implements the Client-Server API from the Matrix Specification [61], thus supports registering new users, creating rooms, inviting users, joining rooms, sending and reading events. The framework also supports authorization events, such as setting power levels or banning users. These functions are required to evaluate the relay-based solution, as relays have to be invited to rooms, that have to be created and store room events sent by other peers. Aside from the Client-Server API functions, Mautrix-Go also includes support for Application Services, End-to-end encryption and provides helpers for parsing event content. Therefore, the library provides sufficient functionality to run experiments for both functional and performance evaluations.

The traffic generator contains two separate scripts for functional and performance input files, as they have different input formats. The functional evaluation experiments require a more fine-grained control over the traffic compared to the WhatsApp dataset,

such as sending auth events (e.g., banning users, updating power levels) or changing a peer's on- or offline state via the traffic file. In contrast, in the performance evaluation experiments, only message events are exchanged, whereby no auth events have to be processed. Also, as the WhatsApp dataset does not contain on-/offline of users, the input format of the performance evaluation does not include actions changing a peer's online state. In the following, both input formats are described. The traffic characteristics of the WhatsApp dataset are analysed in Section 6.2.

For functional traffic, the traffic generator accepts a json traffic file containing a list of different actions. An example traffic file is provided in Appendix A, Listing 13. The traffic generator processes each action in sequence, and pauses for a configurable time until processing the next action. This pause time is provided for each action, because some actions require a longer period, until their results take effect. For example, when the action triggers a peer to come back online, it takes a certain time until the Pinecone peer has established connections to other peers in the network. Available actions for traffic files used as input for the traffic generator are listed in Table 4. Some actions require a set of action parameters. Every action requires the json parameter *user*, as every action corresponds to a request to the peer made by the Matrix-Go SDK. Actions from one user directed to another specific user, i.e., *ban* and *powerLevelEvent* require the parameter *targetUser*. The optional parameter *pauseAfter* allows configuring a pause time after performing an action. If not configured, the traffic generator uses a default pause time.

Name	**Description**
Message	Make send request to peer
Online	Connect container to the docker network
Offline	Disconnect container from the docker network
OnlineBatch	(Dis-)connect multiple containers to/from the docker network
Ban	Ban the provided peer from the room
PowerLevelEvent	Set power level for provided message type
PowerLevelUser	Set power level of a user
State	Change the room state of a state key to the provided value

Table 4: Configurable actions for functional traffic input files

For performance traffic, the traffic generator accepts a csv file containing sending actions in each row. The format of these actions is summarized in Table 5. The traffic generator processes each row from the performance traffic input. Random message contents of the provided char count length are generated. The generator pauses until processing the next row, depending on the relative timestamp. However, the traffic generator pauses for a maximum of 10 seconds, because waiting longer does not provide any benefit and would result in high simulation times. The upper bound of 10 seconds is chosen, so that peer re-discovery connection establishment in the Pinecone network works for larger topologies up to 30 peers.

Because waiting at most 10 seconds after the next action produces shorter interarrival times compared to the original data, after having run the experiment, the logged times are rescaled back to the original interarrival times. This rescaling mechanism allows a more realistic analysis of the experiment. If such a rescaling was not applied, evaluation results could be misleading, because all interarrival times longer than 10 seconds would be reduced to 10 seconds. Before running the performance experiments, the traffic gen-

erator adds online periods surrounding the sending timestamps. Therefore, the online periods of the peers are determined by the traffic generator based on the provided performance traffic. As the online periods themselves are more closely related to the traffic analysis than the input format for the traffic generator, they are discussed in Section 6.2.

Name	Description
Timestamp	Relative timestamp for sending message
User	User ID
Char Count	Number of characters in message

Table 5: Configurable parameters for performance traffic input files

6.1.3 Docker Network

Peers and relays run in docker containers, because the Docker framework provides several features that are utilized for the evaluation setup. First, the Docker environment provides a reproducible setup for running peers and relays. The host system therefore only requires a working docker installation and a working image for running peers and relays. Second, the Docker environment provides simple networking mechanisms to interconnect containers. Third, evaluation scenarios consisting of multiple containers for peers and relays can be created with a configuration file, i.e., a Docker-Compose file.

Docker-Compose is a tool for defining and running applications with multiple containers [62] in a single configuration file. In the context of relay-enhanced P2P Matrix, the technology can be used to deploy a P2P Matrix as the application, where the containers are either peers or relays. The entire application stack is managed in the configuration file and can be started with a single command. While docker itself already provides reproducibility with containers, docker-compose provides reproducibility for multi-container applications. Various inter-container dependencies can be configured, such as the startup order of containers. Docker-Compose also provides a customizable network configuration for the containers, where even multi-network definitions are possible and containers are assigned to the networks. For the evaluation scenarios, a custom docker network is configured, and each peer and relay is added to the subnet.

A Docker-Compose generation script is written, in order to automate the process of creating new Docker-Compose configuration files for various evaluation scenarios. The script requires two input parameters: A template file and a peer/relay configuration file. The template file serves as a skeleton for the generated Docker-Compose configuration file. The peer/relay configuration (Section 6.1.5) file includes information required for the Traffic Generator (Section 6.1.2) to send client requests to the peer/relay.

6.1.4 Logging

In order to be able to track peer/relay actions after an experiment, peers/relays save their actions in an output log with a timestamp. Even though peers/relays run in Docker containers, they store the actions in log files of the host system. This ensures that log files can be accessed after shutting the containers down or removing them. Logged actions are: sending, receiving, making resync requests, receiving resync requests, caching events, going online, going offline. These actions are logged with additional metadata, i.e., time of action, user, action type, event type and event hash.

The sending timestamp is logged in the *Federation* component, right before the event is added to the output queue, from where it is sent to all other online peers and the selected relay. The reception timestamp is logged after the transaction has been received, right before the events in the transaction are checked by the *RoomServer* component. The resync request is logged right before the request is made to the selected relay. The receive-resync action is logged right after a relay has received a request for the resync endpoint. The logging of on-/offline actions is performed, when the network interface has been activated or deactivated. After each experiment, the log files are merged together into a single experiment log, by sorting each entry by timestamp. The merged experiment log is used to interpret and evaluate the experiment, both for functional and performance evaluations.

6.1.5 Configuration Files

In order to setup and run experiments, the different experiment parameters have to be configured. The evaluation setup uses configuration files on different layers of abstraction, with the docker-compose file on the lowest level and running multiple experiments in succession on the highest level. The configuration files and their relation to the components in the Evaluation Setup are shown in Figure 21.

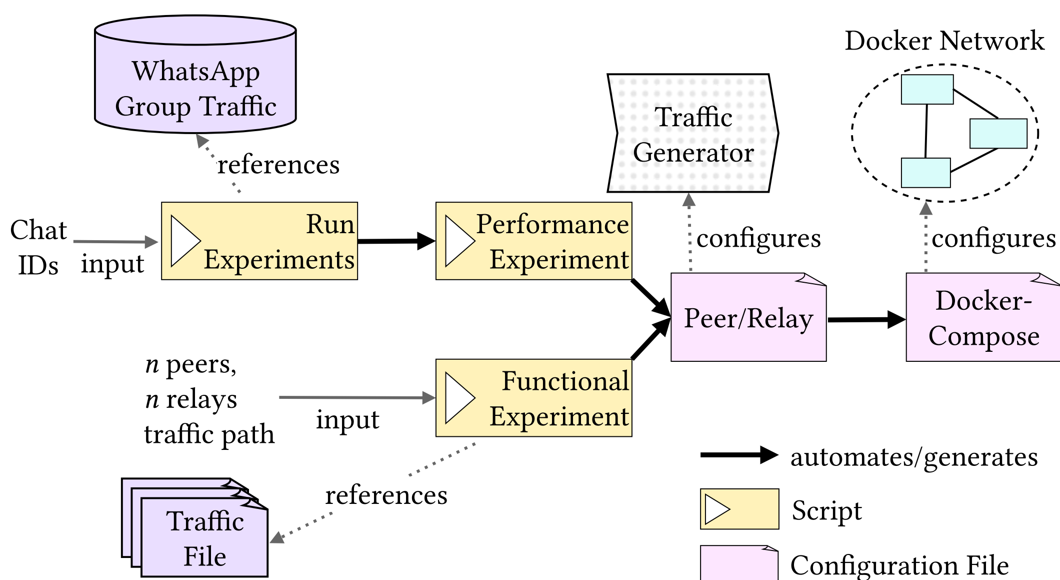

Figure 21: Hierarchy of configuration files and scripts. Files further to the left provide a higher degree of automation, e.g., *Run Experiments* accepts a list of chat IDs referencing datasets from the WhatsApp Group Traffic and executes the experiment.

Starting with the lowest level on the far right of Figure 21, an experiment requires a certain number of peers and relays to be configured in the Docker-Compose configuration file. As all peers and all relays have similar entries in the Docker-Compose configuration file, it can also be generated from a template file and the peer/relay configuration file, as described in Section 6.1.3. The peer/relay configuration file primarily stores information relevant for the traffic generator, i.e., user ID of peer, peer address, access token for client requests etc. Again, the construction of the peer/relay configuration file is automated, and can be generated by either the performance or functional experiment script.

The functional experiment script requires a number of peers, a number of relays and a path to the traffic file as input. It then generates a peer/relay configuration file, from which the docker-compose file is generated. The performance experiment script works in a similar way as the functional experiment script, only that the input file has a different format. Because the performance experiments are run with multiple WhatsApp Group Traffic datasets and multiple configuration options, the performance experiment script is further automated by the run experiments script. This script accepts a list of chat IDs, a certain online period, which is explained in Section 6.2, and a number of relays. The number of peers depends on the provided dataset.

On the whole, the configuration files and scripts provide an environment, where experiments can not only be configured on different levels of abstractions, but also can be run in a reproducible and automated way.

6.1.6 Experiment Process

This section elaborates on the process of executing experiments with one of the three experiment scripts, as depicted in Figure 21. As described in Section 6.1.5, the experiment files require a traffic file, the number of relays and – in the case of functional experiments – the number of relays as input parameters. Based on these parameters, a peer/relay configuration file is generated, from which the Docker-Compose file is generated. Next, the containers configured in the Docker-Compose file are started. After ensuring that all containers have started successfully, the Traffic Generator scripts are started.

The traffic generator uses the Mautrix-Go SDK to send client requests to the peer. A prerequisite for sending events to the peer via the Client-Server API is a user account. Therefore, the traffic generator registers new users for each peer running in a Docker container. The Traffic Generator then creates a new room, invites all peers and relays, and joins all peers and relays to the room. After this, the Traffic Generator starts the experiment and sends events to the peers based on the traffic file. Also, the online states of peers are controlled via the Docker API.

After the sending process is completed, the Docker containers are stopped. The log files are merged into a single experiment log. Finally, the experiment log is transformed to an evaluation-ready format. As the absolute timestamps are irrelevant to the evaluation, the timestamps are transformed into relative timestamps. Also, event hashes are ordered by the first event and transformed to numbers.

6.2 Performance Evaluation Traffic

To evaluate P2P Matrix with relays, two types of traffic are used: Synthetic traffic for functional evaluation and real-world traffic from the WhatsApp traffic dataset [56] for performance evaluation. Synthetic traffic files are written for each functional experiment, targeting a specific aspect of the relay's functionality. Because these synthetic traffic files for functional experiments are closely related to the experiment, they are described with each experiment in Section 6.4. This section focuses on the WhatsApp traffic dataset for performance evaluation.

6.2.1 Overview

The authors of [56] provide a dataset of 5.956 private WhatsApp chat histories, with a total of 76 million message from more than 117.000 users. The chat histories were collected by users voluntarily exporting histories of their chat groups with a web-based tool. Users were incentivized to share their chats by receiving statistical summaries and evaluations on the communication behavior within the chats after uploading their chat histories.

The chat histories include the time of sending, an anonymized user hash, the number of characters sent as contents and a message type, e.g., text or audio message. The dataset contains a high share of text messages, as messages in chats from 2021 have a share of 89%. The dataset contains group sizes up until 252 members. However, the distribution of group sizes in the dataset show, that 42% of the chats have two members, and 83% of the chats have thirty members or less. Due to different export formats for WhatsApp chats, only 27% of the chat traces have messages with timestamps of seconds granularity, the remaining chats only have a granularity of minutes. The interarrival times (IATs) between two consecutive messages, show a long tail characteristic and the authors fit the distribution of IATs to a beta prime distribution. In the provided distribution, 76% of the IATs are shorter than two minutes, indicating a high share of short IATs. The authors also show that IATs decrease with increasing room sizes. They claim that due to the long tail characteristic, both active phases and long breaks between messages occur in chats. The dataset provides only sending behavior of group members, it does not provide insights into message reading behavior.

6.2.2 Online Margins

As WhatsApp and Matrix both are group-based applications for instant messaging, the group chats of the WhatsApp dataset can be applied to Matrix rooms. The number of peers for an experiment can be derived by the number of different user hashes in the chat. One user hash maps to a distinct peer. A major limitation of the dataset towards the applicability to P2P Matrix with relays is that the dataset provides neither reception times nor on-/offline times of users. Moreover, asynchronous delivery with the help of relays can only be measured and evaluated when peers switch between on-/offline state over time. No distributions on real online times of instant messaging applications have been found in related work. Also, real distributions of online times cannot be directly derived from the sending times, because online times might deviate substantially from the sending times. In P2P Matrix, a peer being *online* denotes a peer being able to send and receive events to other peers in a room.

In order to evaluate P2P Matrix with relays with artificial online times, a spectrum of online periods is modeled for each dataset. Given the time of sending t_s and an online margin δ, an online period can be defined as tuple of start (going online) and end time (going offline).

$$(\text{start}, \text{end}) := (t_s - \delta, t_s + \delta)$$

To evaluate a range of online margins, different values can be used for δ. However, within an experiment all peers use the same δ-value, in order to have comparable online period durations for each peer. This range should model multiple on-/offline times for each peer, and result in different synchronous and asynchronous online periods, in order to show

a difference between P2P with relays and pure P2P. The different values in the range of online margins are shaped around the sending times, so that a peer is online for a certain time preceding a sending action, and goes offline after a certain time succeeding the sending action. Even though this model most likely does not represent real online times of peers, it allows measuring the effect of relays when the rate of synchronously online peers declines.

In order to apply online margins of different lengths to a dataset, the IATs of the chat are considered. Using a same set of online margins for each chat does not suffice, as traffic properties of chats differ, such as IATs. An online margin resulting in no overlapping online periods between peers in one chat higher IATs might result in many overlapping online periods in another chat with lower IATs. An example of different online times is shown in Figure 22. The two chats both have two group members and have very different distributions of IATs. While chat 5264 has shorter IATs, they are significantly longer for chat 305. Performing the evaluations with equal online margins for both chats would increase the probability of chat 305 having substantially less overlapping online periods than chat 5264, compared to performing the evaluations with relative online margins.

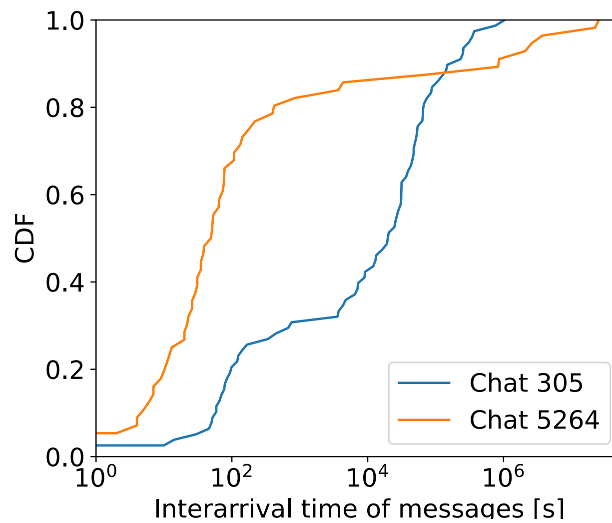

Figure 22: Interarrival times of two selected Whats App Chats with two participants. Messages are replied to in the first 100 seconds in 75% of the messages in chat 305, and 25% of the messages in chat 5264.

Therefore, online margins should be relative to the dataset's IATs. As the evaluations should be run in limited time, only five different online margins are chosen per chat. Considering the long-tail distribution of the IATs in the chats, i.e., including bursty message exchanges and long communication pauses, five evenly spaces online margins would overstate the share of long IATs. In order to evaluate the effect of relays in both asynchronous online periods and periods, where several peers are online at the same time, a logarithmic distribution is chosen for the online margin lengths. Finally, five values evenly spaced on a logarithmic scale are selected as online margins, with the maximum value being the maximum IAT . Therefore, chats are evaluated within at least five different scenarios representing the different online margin lengths.

6.2.3 Applying Online Margins to Dataset

After having established a configuration of online margins, these margins have to be applied to the dataset, so that for each message event, the sending peer comes online before and goes offline after the event occurs. However, for each offline time after a sending event, the succeeding sending event could occur earlier than the offline time. This can result in overlapping online periods per peer, which should be avoided. Otherwise, inconsistencies can occur, e.g., a peer going offline right before sending an event. Consequently, the overlapping online periods have to be merged together into one period.

Listing 12: Generate online periods from send events

1: **function** GENONLINEPERIODS(U: set<user>, M: set<message>, onlinemargin: int)
2: ▷ Set of online periods to be generated
3: P ← ∅
4: **for** u ∈ U **do**
5: $M_u \leftarrow \{ m \mid m.u = u, m \in M \}$
6: $P_u \leftarrow \emptyset$
7: ▷ Generate set of all possible online periods
8: $S \leftarrow \emptyset$
9: **for** $m \in M$ **do**
10: start ← $\min(0, m.\text{time} - \text{onlinemargin})$
11: end ← $m.\text{time} + \text{onlinemargin}$
12: $S \leftarrow S \cup (\text{start}, \text{end})$
13: ▷ Find conflicting online periods in state tuples
14: $i \leftarrow 0$
15: **while** $i < |S|$ **do**
16: $S_{\text{confl}} \leftarrow S_i$
17: ▷ Construct set of conflicting online periods
18: $j \leftarrow i$
19: **while** $j < |S| - 1 \wedge S_{j,1} > S_{j+1,0}$ **do**
20: $S_{\text{confl}} \leftarrow S_{\text{confl}} \cup S_j$
21: $j \leftarrow j + 1$
22: ▷ Save min and max from conflicting set
23: start ← $\min(\{s_0 \mid s \in S_{\text{confl}}\})$
24: end ← $\max(\{s_1 \mid s \in S_{\text{confl}}\})$
25: $P_u \leftarrow P_u \cup (\text{start}, \text{end})$
26: $i \leftarrow i + 1$
27: $P \leftarrow P \cup (u, P_u)$
28: **return** P

The generation of online periods and a merging mechanism for overlapping online periods is implemented in Listing 13. The function's inputs are the set of users and the set of messages, which represent a chat from the WhatsApp traffic set, and the online margin. The online periods are generated for each user (P_u), and merged together (P). In practice, the function takes a list of send messages and returns these messages extended with new online states. For each user, a set of all possible online periods is generated

first. Then, elements of this set are checked against conflicts. Two online periods stand in conflict, if the end time of the earlier period is larger than the start time of the later period. Starting at each of the user's events, a set of conflicting online periods is generated. These conflicting online periods are merged into a single online state, by having the merged online state start with the minimum start and maximum end time of all conflicting set elements. These merged online states are then added to the user's online state set. In the last step, the online periods of each user are added to the overall set of online periods. The overall set of online periods is then returned.

6.2.4 Data Selection

From the 5.956 chat histories, few chats are selected for comparing P2P Matrix with relays and pure P2P Matrix. Also, different online margins are applied to each of the selected chats, in order to model online periods of peers. Therefore, each chat is evaluated not only in the scenarios of pure P2P and P2P with relays, but also the five different online margins. This leads to a total of ten required runs for each chat, resulting in long evaluation times. To select only those chats suitable for showing differences between pure P2P and P2P with relays, these filters are applied to the dataset: Chats with seconds-granularity, group sizes of 30 or less, run times of three hours or less, 30 or more exchanged events, chats including periods of asynchronous message exchanges. These filters are described in the following.

As mentioned in Section 6.2.1, the WhatsApp traffic chat histories contain both datasets with second-granularity precision sending timestamps, and other datasets with only a minute-granularity precision. The IATs of all chats, as discussed in [56], show that 69% of all messages are replied within the same minute. Given this distribution and minute-granularity precision chat traces, the sending order is lost for a high number of messages. However, the temporal difference between sending times is important, in order to produce different online periods between peers, and to simulate asynchronous event exchanges. Consequently, minute-scale chat traces do not fully capture the temporal differences between user actions and are therefore filtered out. From the 5.956 chats, only 1.598 chats with second granularities of sending times remain.

Next, the chats are filtered by the number of group members. In larger groups, the probability of any two members being synchronously online is considered to be higher than in smaller groups. Since the advantage of relays most likely only takes significant effect, if peers are asynchronously online, the chats are filtered by group size. A maximum number of 30 group members is chosen, as 80% of the chats have a group size of 30 members or less, and these group sizes represent a high share in the overall dataset.

A factor constraining the range of possible chats used for performance evaluation is the time of running the chat traffic, as each chat is run in ten times. A script estimating the time of each run is written. This script takes a chat trace as an input, generates online periods and calculates the times the traffic generator requires between each event. It therefore uses the same mechanism from the traffic generator (Section 6.1.2) to wait for the next row to be processed, and adds these waiting times together. For each chat, the evaluation times are calculated with the online margin set to the median IAT. This produces rough estimates of how long running chats can take, although the times per chat may differ due to the five different online margins. Chats with an estimated evaluation time of three hours or less are chosen.

However, if a chat has a low estimated time running the chat, a low number of exchanged messages does not provide a reasonable amount of data for evaluation. Therefore, a sufficient amount of exchanged messages is required to make claims on the overall performance improvement of relays towards pure P2P. Because of that, chats with thirty or more messages are selected.

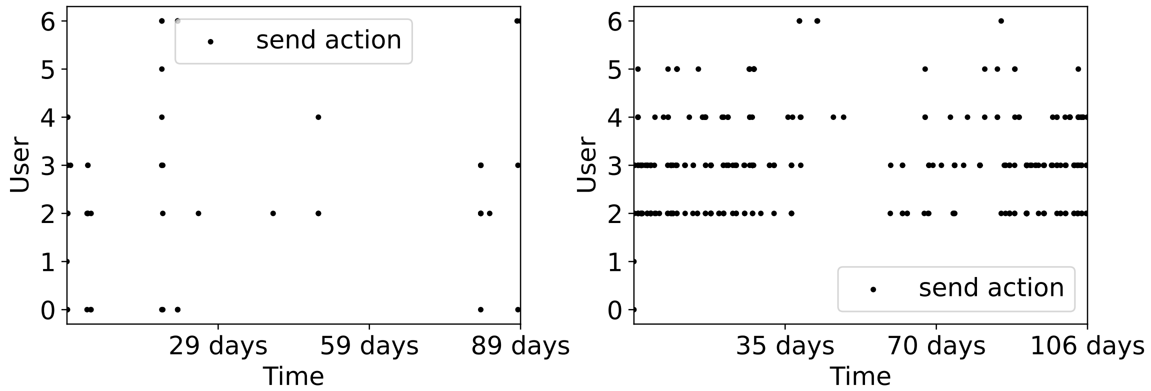

Figure 23: Comparison of two chats with six members. Chat 943 (left) has few bursty sending periods and fewer messages in total compared to chat 3543 (right).

After applying these filters to the whole WhatsApp dataset, 103 chats remain. For these remaining chats, a qualitative selection was made based on the traffic patterns in the dataset. The observed traffic patterns are the share of distribution of sending times and bursty sending periods. An example comparison between messages in two chats (943, 3543) of the same group size is shown in Figure 23. Chat 943 has more evenly distributed sending times compared to chat 3565. The more even distribution makes it more likely to have non-overlapping online periods, and thus asynchronous online times between peers. Also, chat 3565 has a higher number of periods of bursty message exchanges, leading to a high number of synchronous online times between peers. The filtered 103 chats were evaluated by observing their traffic patterns, resulting in 18 selected chats showing traffic patterns that are likely to show the advantages of relays in P2P Matrix. An overview of all selected chats is provided in Table 6.

Group Members	Number of Chats	Mean number of Messages
2	5	60
3 – 9	6	55
10 – 30	7	112

Table 6: Overview of selected chats from WhatsApp traffic set.

6.3 Evaluation Overview

The different experiments are summarized in Table 7. The evaluation metric describes an evaluated aspect of the system. An evaluation can be either functional, i.e., validating that P2P with relays works correctly, or performance-oriented, i.e., validating that P2P Matrix with relays is an improvement towards pure P2P. Each evaluation metric is

validated with one or more scenarios. A scenario consists of the room size, the number of relays and the traffic, which includes events, online periods and resync requests/responses. For the performance evaluations, the different online margins, based on which the online periods are generated, are also part of a scenario.

The evaluations are run in simplistic settings, where single rooms are evaluated, relays are constantly online, do not delete events and are used by every peer. This allows focusing on the core functionality of relays and peers. Each scenario evaluates the traffic in the context of only one room. Unless the experiment explicitly evaluates the aspect of authorization, the room state (i.e., memberships, permissions) is static and does not change within one scenario. Therefore, only message events are exchanged in most of the experiments. However, scenarios regarding relay access control with an up-to-date AuthDAG also include state events. Relays are assumed to be online within each scenario, since asynchronous delivery – the core functionality enhancement of P2P Matrix – is only possible when relays are online and the focus of the evaluation lies on the effect of relays in P2P Matrix. The cache retention time is set to infinity, i.e., relays do not remove cached events in the evaluation scenarios. This decision is made because the goal is to evaluate how asynchronous delivery with relays improves P2P Matrix, although in practice, peers can configure a retention time to prevent events and event metadata accumulating at relays. Also, all peers are assumed to use relays, although in practice, peers can choose to not forward any events to relays or request events from them.

Because the evaluation examines different aspects of the design, the scenarios in which the experiments take place vary. The different parameters are summarized as follows:

- *Room Size*: The room size represents the number of peers and relays in a room, and therefore also the proportion of relays to peers. The functional evaluations mostly use a minimal room size of two peers and two relays, while the room sizes for the performance evaluation depends on the traffic dataset.
- *Relay Selection of Peers*: The relation between peers and relays describing which relay a peer selects, forwards events to, and uses for resynchronization. This relation is static in most experiments, except for evaluating the functional metric "Resync with Different Relays".
- *Traffic*: Event traffic of peers within a Matrix room. The traffic used for the scenario has different characteristics, e.g., event sending times and interarrival times. The time of sending is the time when a peer sends a room event, while interarrival times are the times between receiving two subsequent events. Depending on the evaluation metric, either manually constructed or real-world traffic is used.
- *On-/Offline time distribution of Peers*: An online peer can send and receive events to and from other peers or relays while an offline peer cannot. A peer is considered online, if a connection to at least one other peer or relay is possible. When running the experiments, the on-/offline periods are part of the traffic input file.
- *Time between sending subsequent events*: The time in between sending events, where the traffic generator waits until sending the next event. This time can either be static between all events, or it can differ between events. This time is only configured for the functional experiments, as the traffic for performance experiments already includes inter-sending times.

Evaluation Metric	Type	Group Sizes	Traffic
Asynchronous Delivery: Minimal Setup	F	Dyadic	One asynchronous event
Asynchronous Delivery in larger Rooms	F	Small, Large	Round-Robin Online Traffic
ASAP Delivery	F	Dyadic	Multiple asynchronous online times
Resync with Since Parameter	F	Dyadic	Resync Traffic Scenario
Resync with Different Relays	F	Dyadic	Peer resyncs with multiple relays over time
AuthDAG: Access Control	F	Dyadic	State Event Traffic
Events not Delivered to any Peer	P	All	WhatsApp Group Traffic Instance
Event Delivery Time to any First Peer	P	All	WhatsApp Group Traffic Instance
Online Periods without Receiving Events	P	All	WhatsApp Group Traffic Instance
Reception Time after Returning Online	P	All	WhatsApp Group Traffic Instance

Table 7: Evaluation Overview. F denotes a functional evaluation experiment, and P denotes a performance experiment. There are three group sizes: Dyadic groups (two members), small groups (three to ten members), and large groups (ten to thirty members). One relay was used for dyadic and small groups, two relays were used for large groups.

6.4 Functional Evaluation

The functional evaluation determines whether the design is correctly implemented. Each experiment is first summarized by providing the evaluation goal, the scenario consisting of the room topology and traffic, the method to determine whether the evaluation goal is fulfilled, and the expected result. Then, after a brief evaluation description, the results of the experiments are discussed.

To validate whether relays fulfill the property of asynchronous delivery for a single event, comparable to Figure 14 in Section 4.1, it is validated in a minimal setup first. Then, the validation of asynchronous delivery for multiple events follows, utilizing scenarios with larger group sizes. After showing that the relay-based solution scales to large rooms, the remaining functional experiments are all performed in scenarios with dyadic rooms. To make things more intuitive, in dyadic rooms, the two peers are called *Alice* and *Bob*.

The functional evaluation not only verifies that this solution works as specified, but can also be used to evaluate the correctness of future solutions. As shown in Figure 21, the functional evaluations can be started by providing the number of peers and relays, as well as the traffic file. The remaining experiment components, i.e., Docker network setup, traffic generation etc., are managed by the functional experiment script, thereby providing a reproducible setup.

6.4.1 Asynchronous Delivery: Minimal Setup

Evaluation Goal	Show that solution provides asynchronous delivery for a single event.
Scenario	
↳ *Topologies*	(2,2) {(peers, relays)}
↳ *Traffic*	One asynchronous event exchange, as shown in Section 4.1, Figure 14.
Method	Check if event reception of asynchronously sent event appears in log.
Expected Result	Receive event exists in log for the receiving peer, and complies with asynchronous delivery definition, as described in Section 3.1.

This experiment determines, whether P2P Matrix with relays provides asynchronous delivery, a basic requirement to the solution. The scenario traffic consists of Alice coming online and Bob going offline, after which Alice sends an event, and goes online. After that, Bob returns online and makes a resync-request to the a relay. This traffic instance is comparable to the asynchronous delivery example, as shown in Section 4.1, Figure 14. The evaluation scenario evaluates whether asynchronous delivery works with one relay or two relays, where each peer uses a different relay. Evaluating asynchronous delivery with two relays also validates that forwarding events between relays works. The expected result is that Bob receives the sent event after resyncing with the relay.

	Time [s]	**User**	**Action**	**Type**	**Event ID**
0	0	Bob	offline		
1	2.8931	Alice	send	m.room.message	1
2	3.081	Relay 0	receive	m.room.message	1
3	3.3298	Relay 0	cache-event	m.room.message	1
4	3.3865	Relay 1	receive	m.room.message	1
5	4.0112	Relay 1	cache-event	m.room.message	1
6	7.03	Alice	offline		
7	10.5163	Bob	online		
8	11.8194	Bob	resync		
9	11.8245	Relay 1	receive-resync		
10	11.8247	Relay 1	reply-resync		
11	11.8255	Bob	resync-response		
12	11.8263	Bob	receive	m.room.message	1

Table 8: Event Log for functional Asynchronous Delivery evaluation in minimal setup, where only two peers exchange events asynchronously with the help of relays.

The results of the experiment with two peers and two relays are summarized in Table 8. The table represents a merged experiment log file, as described in Section 6.1.1. The original event hashes are translated to a relative numbering, so that event identifiers ordered by the time of sending are generated. In the minimal setup for asynchronous delivery, only a single event is sent. The table shows that after Alice sends the event (row 1), its chosen relay 0 receives and caches the event first (row 2,3), after which the other relay 1 does the same (row 4,5). This is an indication that the forwarding of events between relays works, since Alice sends the event to only her chosen relay. After returning online, Bob resynchronizes with his chosen relay 1 (row 8), and receives the event (row 12).

6.4.2 Asynchronous Delivery in Larger Rooms

Evaluation Goal	Show that solution provides asynchronous delivery in larger scenarios with multiple events, and that given the round-robin traffic, peers in P2PR receive events earlier than peers in P2P.
Scenario	
↳*Topologies*	(5,0),(5,1),(10,0),(10,1),(20,0),(20,2) {(peers, relays)}
↳*Traffic*	Round-Robin online traffic, where each peer comes online and sends an event one after the other, and all other peers are online. In the final state, all peers return online, and a final synchronization event is sent, forcing all peers (in pure P2P) to backfill missing events.
Method	Compare peer action (sending/receiving/online) timestamps.
Expected Result	Peers in P2PR receive events earlier than peers in P2P Section 3.1.

This experiment determines, whether P2P Matrix with relays provides asynchronous delivery not only in the minimal scenario, but also in larger rooms, i.e., ten and twenty members. Because this experiment compares reception times between pure P2P Matrix and P2P with relays, topologies with and without relays are used. The round-robin online traffic consists of each peer coming online, sending an event, and going offline, while all other peers are offline. In order to trigger a synchronization of peers at the end of the experiment, all peers come online at the end of the experiment.

The results are depicted in Figure 24. All events are sorted by time and their numerical order of sending is represented on the y-axis. The plot shows the send actions of both peers in P2P and P2PR. Sending times between P2P and P2PR are equal, as they are based on the same input traffic file. The reception period is the period between the first and last reception time for each event. Peers receive the event within this period.

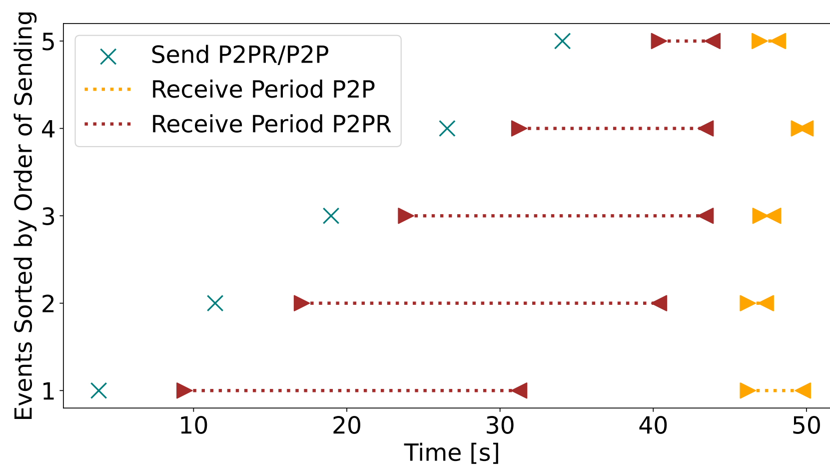

Figure 24: Asynchronous delivery evaluation with round-robin online traffic, comparing five-peer topologies in P2P and P2PR. The triangles represent the first/last reception times of each event. In the final state, all peers come back online.

The results show that after peer in P2PR returns online, which occurs shortly before sending the next event, the peer receives the last sent event, indicating that the resynchronization with the relay works. In contrast, peers in pure P2P receive events significantly later, i.e, when coming back online and synchronizing with other online peers.

6.4.3 ASAP Delivery

Evaluation Goal	Show that solution provides ASAP delivery, i.e., the received event should have an arrival time that is the earliest arrival time in the set of asynchronous online periods after sending.
Scenario	
↳*Topologies*	(2,1) {(peers, relays)}
↳*Traffic*	Traffic with multiple periods after the point in time of sending, where the receiver is online, while the sender is offline.
Method	Compare the event reception time with the set of post-send asynchronous online periods.
Expected Result	The event is received in the first post-send asynchronous period.

This experiment determines, whether the solution provides ASAP delivery. The scenario uses a minimal topology consisting of 2 peers and one relay. The output log should show that the received event fulfills the requirements for ASAP delivery, as formalized in Section 3.1. After having been offline, Bob should receive the event in the first asynchronous online period after the sending event, because he resyncs with the relay directly after coming back online.

	Time [s]	**User**	**Action**	**Type**	**Event ID**
0	0	Bob	offline		
1	3.3694	Alice	send	m.room.message	1
2	...				
3	7.5357	Alice	offline		
4	11.0178	Bob	online		
5	11.4486	Bob	resync		
6	...				
7	11.4556	Bob	receive	m.room.message	1
8	12.0232	Bob	offline		
9	...				
10	19.0319	Bob	online		
11	...				

Table 9: Event Log for ASAP Delivery Evaluation in minimal setup.

The result of the experiment is summarized in Table 9. The event log shows that there are two online periods for Bob (rows 4-8, rows 10-11). Also, during both online periods of Bob, Alice is offline. Therefore, there are two asynchronous periods following the sending event (row 1). Because the event delivery is logged within the first online period, the event is delivered to Bob ASAP.

6.4.4 Resync with Since Parameter

Evaluation Goal	Show that relay returns only a subset of events given the *since* parameter.
Scenario	
↳*Topologies*	(2,1) {(peers, relays)}

↳ *Traffic*	Asynchronously exchanged events from one peer, and two separate resync requests from a different peer, that goes offline between the requests.
Method	Check number of received events after the second resync request.
Expected Result	After the resync output log entry, the receiver should only receive the event sent asynchronously by the sender.

This experiment should verify that the relay only returns the subset of events, that have a later timestamp than the provided *since* parameter, in the minimal scenario with two peers and one relay. The traffic and the required functionality are depicted in Figure 25. First, Bob sends an event α to the relay, where it is stored in the event cache, and goes offline. Then, Alice comes online and resyncs with the timestamp t_0, after which the relay replies with the event α. After the resync, she goes back offline. Bob returns online, sends event β to the relay, and goes offline again. When subsequently Alice returns online and resyncs with the relay, it should only return the event β.

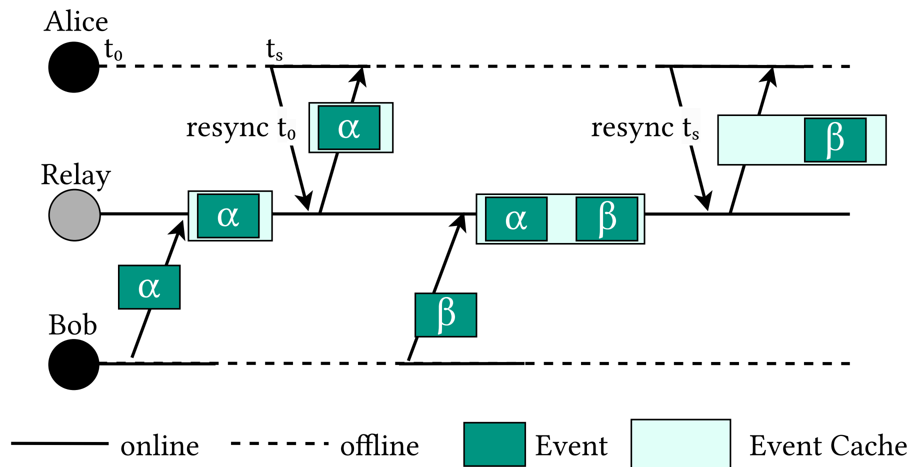

Figure 25: Resync evaluation scenario in a two-peer topology. In the second resync request, Alice should only receive the event β.

The results of the experiment can be found in Appendix A, Table 12. While for the first request, Alice receives event α, for the second resync request, Alice receives the event β only, indicating that the relay returned only the subset of events later than the provided *since* timestamp.

6.4.5 Resync with Different Relays

Evaluation Goal	Show that peer can resync with different relays over time.
Scenario	
↳ *Topologies*	(2,2) {(peers, relays)}
↳ *Traffic*	Asynchronously exchanged events from one peer, and two separate resync requests from a different peer, that goes offline between the resync requests.
Method	Compare two experiments. In the first run one peer resyncs with a relay, and later with the same relay again. In the second run one peer resyncs with a relay, and later with a different relay.

Expected Result The peer should receive the same sets of events in both runs.

This experiment verifies that resyncing with different relays over time works. The traffic is identical to the previous experiment, *resync with since parameter*. The difference to this evaluation is the number of relays, and the relays used for resynchronization. Since one of the two peer resyncs with two different relays, the scenario requires two relays. In order to have a baseline to which events are expected to be received, the received events from the peer performing the resync requests are compared with the previous scenario. In the experiment for resyncing with two relays, a peer resyncs with a relay first, goes offline, returns online, and resyncs with the other relay.

The results of the experiments can be found in Appendix A.2, Table 13. After resyncing with the two different relays, the Alice has received the same two messages, that were received in the previous experiment, *resync with since parameter*. Therefore, it is possible to resync with different relays without having to establish additional state with the new relay. A peer can simply send a resync request to a different relay, which responds after having authorized the peer.

6.4.6 AuthDAG: Access Control

Evaluation Goal Show that relay rejects unauthorized events.
Scenario
↳ *Topologies* (2,1) {(peers, relays)}
↳ *Traffic* Three state event traffic instances:
  a) Peer *a* bans Peer *b*, then Peer *b* resyncs with the relay.
  b) Peer *a* bans Peer *b*, then Peer *b* sends a room event to the relay.
  c) Peer *a* sets the power level of Peer *b* to zero, then Peer *b* sends a room event requiring a power level > 0 to the relay.
Method Verify that output log entry for relay cache does not exist for unauthorized events.
Expected Result Relay rejects unauthorized events.

This experiment determines, whether relays perform access control correctly, for both receiving events and receiving resync requests. The three scenarios use the minimal topology of two peers and one relay. The evaluation traffic contains state events, allowing to evaluate whether the relay performs access control according to the latest available room state.

In the first traffic instance, Alice bans Bob, after which Bob attempts to resync with the relay. This traffic instance verifies the working access control mechanism for the resync API endpoint. In the second traffic instance, Alice bans Bob from the room, after which Bob sends a room event to the relay. With this traffic instance, the working membership state check of the relay can be verified. In the third traffic instance, Alice sets the power level of Bob to zero, after which Bob sends a room event with a power level larger than his own power level. With this traffic instance, the relay's mechanism of checking peer power levels can be verified.

	Time [s]	User	Action	Type	Event ID
0	9.1567	Alice	ban Bob	m.room.member	1
1	9.3109	Bob	receive	m.room.member	1
2	9.3115	Relay	receive	m.room.member	1
3	9.7782	Relay	cache-event	m.room.member	1
4	14.6742	Bob	offline		
5	17.1787	Bob	online		
6	18.7587	Bob	resync		
7	18.7634	Relay	reject-resync		

Table 10: Event Log for evaluating that relays reject unauthorized resync-requests with the AuthDAG given the latest available room state.

The result of the experiment with the first traffic instance is summarized in Table 10. Alice bans Bob from the room and sends this ban event into the room (row 0). Both Bob and the relay receive this ban event. After going offline, Bob returns online (row 5) and attempts to resync with the relay (row 6). Because the relay has already appended the ban event to the AuthDAG, it can determine the member state of Bob, and rejects the resync request (row 7).

The result of the experiments with the second and third traffic instance can be found in Appendix A.2, Table 14 and Table 15. For the second experiment, since the relay only caches authorized room events, the *cache-event* action is missing for the event sent by the banned peer. For the third experiment, the relay also does not store the event in the event-cache, because the peer trying to modify the room state has an insufficient power level.

6.5 Performance Evaluation

While the functional evaluation determined the correctness of the implemented solution based on manually constructed traffic scenarios, the goal of the performance evaluation is to show the advantage of relay-enhanced P2P Matrix towards pure P2P Matrix based on real-world traffic. As the performance evaluation metrics were applied per chat, the results for one example chat are be presented. For the first two experiments, performance evaluation metrics will also be summarized for all selected chats.

To better understand the performance results, an example chat is selected, enabling the evaluation results to be related to the chat traffic. The example chat has ten users and 194 messages are exchanged within a duration of ten days. The sending times and the two longest online margins for the example chat are depicted in Figure 26. Shorter online margins are more clearly visible within a smaller time frame. For the example chat, the smaller time frames can be found in Appendix A.3, Figure 33. In the first day of the trace, there is a frequent sending activity of almost all users. After a break of four days, there is a lower sending frequency of sending. The chat provides a useful example, in which sending and online activity make a substantial difference in various delivery time metrics compared to pure P2P Matrix.

After applying the online margins, online periods emerge for each peer. An online period can be longer than an online margin, if the time in between two consecutive send

events is smaller than the online margin. In that case, the two emerging consecutive online periods are merged into one single online period. Thus, the online period can span multiple sending events, which can be seen for events within the first day in Figure 26. For example, the first events of user 7 all occur in close succession, resulting in the online periods of 2 hours, 41 minutes being merged together into a single long online period.

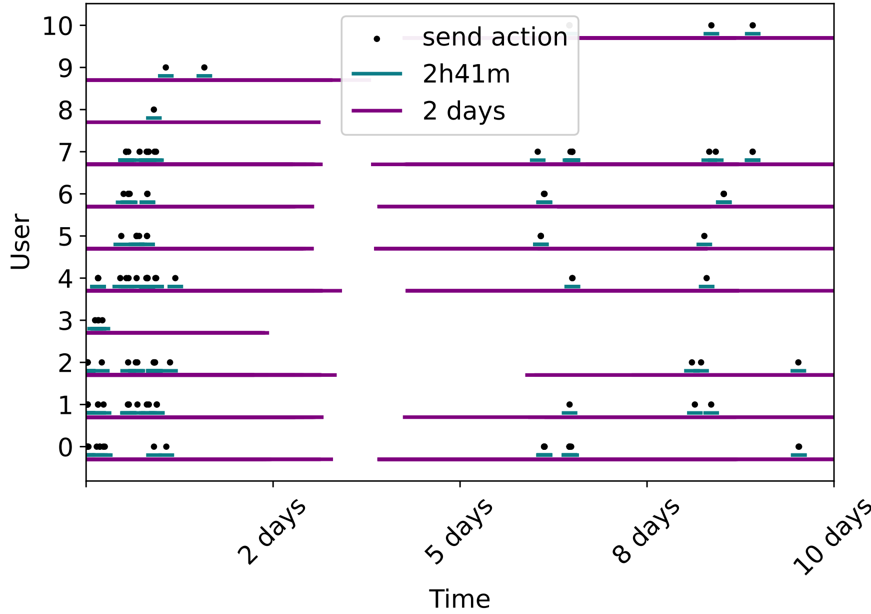

Figure 26: Send actions with the two longest online margins applied to chat 1796.

6.5.1 Events Not Delivered to Any Peer

Evaluation Goal	Show that with shorter online times, relays reduce the overall share of events not delivered to any peer.
Scenario 1	
↳*Topologies*	10 users, 194 messages exchanged in 10 days, 2 relays for P2PR
↳*Traffic*	Chat 1796 from [56]
Scenario 2	
↳*Topologies*	Users & messages from all chats; 1 relay for rooms with less than 10 users, otherwise 2 relays
↳*Traffic*	All selected chats, as summarized in Table 6
Method	Measure the number of events not delivered to any peer.
Expected Result	P2P with relays has a lower percentage of events not delivered to any peer.

This experiment should show that P2P Matrix with relays reduces the overall share of events not delivered to any peer compared to pure P2P Matrix. The first experiment scenario uses the example chat from the WhatsApp dataset, consisting of 10 room users and 194 messages over the period of 10 days. The second experiment scenario spans over all selected chats, thus making a broader assessment over various room sizes, number of messages and chat lengths.

P2P with relays is expected to have lower number of events not delivered to any peer compared to P2P Matrix. In pure P2P Matrix, some events are not delivered due to peers being asynchronously online, and because missing events are only re-requested when

after the missing event a new event is delivered to the receiving peer, who then backfills missing events from the other peers. If the receiving peer does not receive a successor event for the missing event, it will not obtain the missing event, because currently, Matrix peers do not proactively synchronize periodically. In contrast, peers in P2P with relays directly request events from the relay with the *resync* API endpoint, and thereby directly obtain missing events.

Scenario 1: Chat 1796

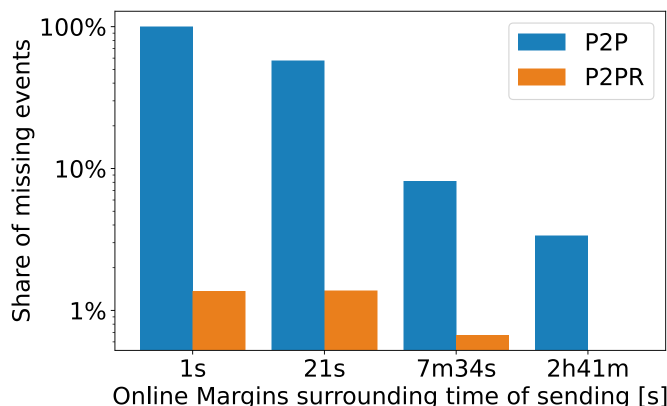

Figure 27: Number of events not delivered to any peer per online margin length. A longer online period results in more peers being synchronously online. In P2P Matrix, short online periods result in a substantial amount of events not being delivered.

The first scenario consists of the ten-user group chat 1796. As Figure 27 shows, the overall share of events not delivered to any peer is lower in P2P Matrix with relays compared to pure P2P Matrix. The fifth online period is not depicted, because for both P2P and P2PR, all events are delivered to at least one peer. Pure P2P Matrix has a high number of missing events for shorter online periods, because online periods of peers are less likely to overlap, i.e., peers are often not simultaneously online. Also, peers only re-request missing event when the missing event's successor is received. The probability of receiving the missing event's successor is lower for short online periods (less peers synchronously online), and higher for long online periods (more peers synchronously online). For the first three online margins, P2PR also has some events not delivered to any peer, because peer 0 sends the last two events before any other peer returns online, which cannot directly be seen in Figure 26 due to the scaling. Therefore, no peer resynchronizes with the relays, and as a result, the event is not delivered to any peer.

The experiment shows, that relays can ensure eventual delivery to at least one peer, provided that after sending another peer returns online. This also highlights the strong dependence between the relay's effectiveness in reducing the number of not delivered events and the online states of peers. Trivially, relays do not provide a substantial advantage towards reducing missing events, if all peers are online.

Scenario 2: All Chats

The second scenario comprises all chats and also measures the percentage of missing events per online margin. The results are shown in Figure 28 and are comparable to the results of the first scenario. The main difference to the first scenario is that also for the longest margin, a small number of events (less than 1 %) is missing. The longest margin equals the maximum IAT of events, which can deviate between chats, as seen in Figure 22. If the IAT is short, applying it as online margin to all send actions does not

result in all of the peer's online periods overlapping at all times. Therefore, event with the longest online margin, some events may still not be delivered to any peer.

The results show that less or equal 1% of the events are not delivered to any peer in P2PR, while the share is substantially higher in pure P2P. Again, the difference of missing events between P2PR and P2P decreases with increasing online margins.

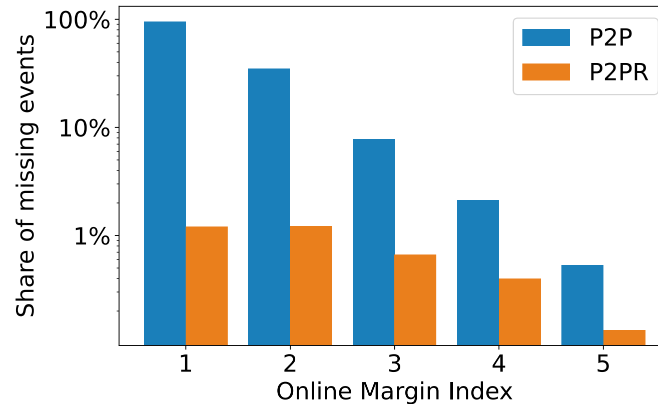

Figure 28: Number of events not delivered to any peer per online margin length, for all chats. For each chats, the individually calculated online margins were used. The online margins are indexed in ascending order based on their duration.

6.5.2 Event Delivery Time to any First Peer

Evaluation Goal	Show that relays provide lower event delivery times to any first peer compared to P2P Matrix, if peers are asynchronously online in a substantial number of periods.
Scenario 1	
↳*Topologies*	10 users, 194 messages exchanged in 10 days, 2 relays for P2PR
↳*Traffic*	Chat 1796 from [56]
Scenario 2	
↳*Topologies*	Users & messages from all chats; 1 relay for rooms with less than 10 users, otherwise 2 relays
↳*Traffic*	All selected chats, as summarized in Table 6
Method	Measure the delivery times to the first peer for all events by subtracting the time of sending from the time the event is first received by any peer.
Expected Result	P2P with relays has lower event delivery times for shorter online margins.

This experiment should show that P2P Matrix with relays provides lower event delivery times to any first peer compared to P2P Matrix, depending on the number of asynchronous online times. The motivation to measure the delivery times to any first peer, and not to all peers is that due to different online periods, a large majority of the events is not delivered to all peers for both P2P and P2PR – at least not, when the traffic trace ends. The delivery time to any first peer allows comparisons between P2P and P2PR using a substantial number of samples, because peers are more often online simultaneously with a single peer than with multiple peers.

For the first scenario, the same topology and traffic as in the previous experiment are used. The second scenario comprises all selected chats, inheriting all topologies and traffic from those chats. The delivery times to the first peer represents the interval difference from the time the event is sent until the earliest time at which the event is received by any peer. The delivery times for P2PR are expected to be lower than those of P2P for the lower online margins, because the first peer coming online after another peer sending an event asynchronously receives the event directly after resyncing with the relays. In contrast to the previous experiment, which assessed the success rate of events delivered to any peer, this experiment measures the difference in event delivery times.

Scenario 1: Chat 1796

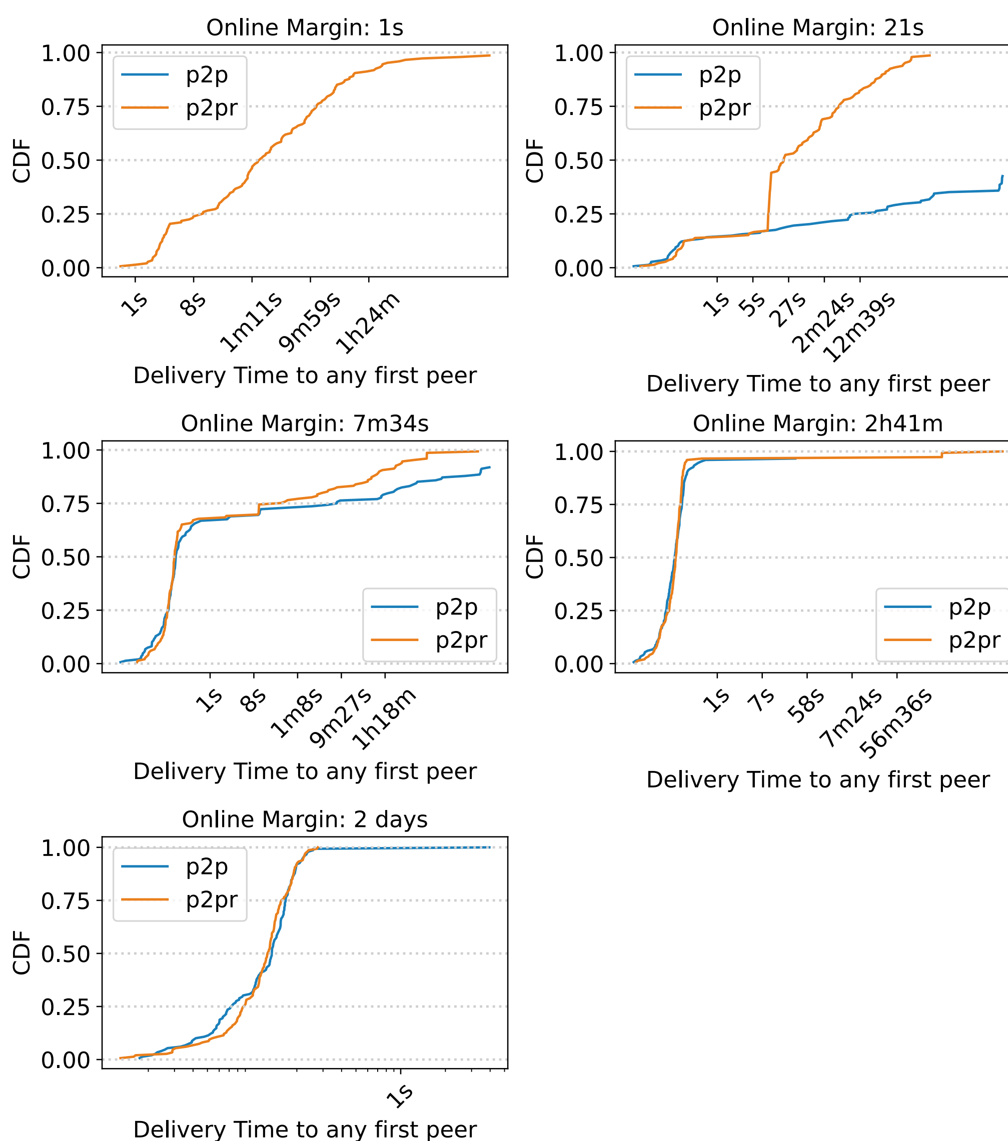

Figure 29: Delivery times to any first peer for each online margin.

For the first scenario of the ten-user chat, the CDFs for each applied online margin are shown in Figure 29. The CDF for the first online margin includes no delivery times for P2P, because for this margin no events are delivered to any peer, i.e., all are missing, as shown in Figure 27, and therefore no delivery times can be measured. The CDF for the second online margin shows a steep increase for the delivery times at 12 seconds, while

before the increase the delivery time distributions are similar. This increase correlates with the online times, because for delivery times lower equals 10 seconds, which is half of the online time, the peers exchange the event synchronously. This indicates a frequent overlap of online periods for half of the online margin. Consequently, the relay advantage is apparent for delivery times larger than the online margin. The share of delivery times is shorter than 12 seconds in 40% of the cases in P2PR, and only in 17% of the cases in P2P, for the second online margin of 21 seconds. While for the third online margin, the delivery time difference decreases substantially, for the fourth and fifth margins, the difference becomes marginal.

This experiment demonstrates that, in addition to improving the success rate of event delivery to any peer, relays can also reduce the delivery times of events to the first peer, particularly when peers exhibit asynchronous online activity. The extent of the relay advantage, in terms of how many delivery times are reduced, depends on the asynchronous online activity of peers.

Scenario 2: All Chats

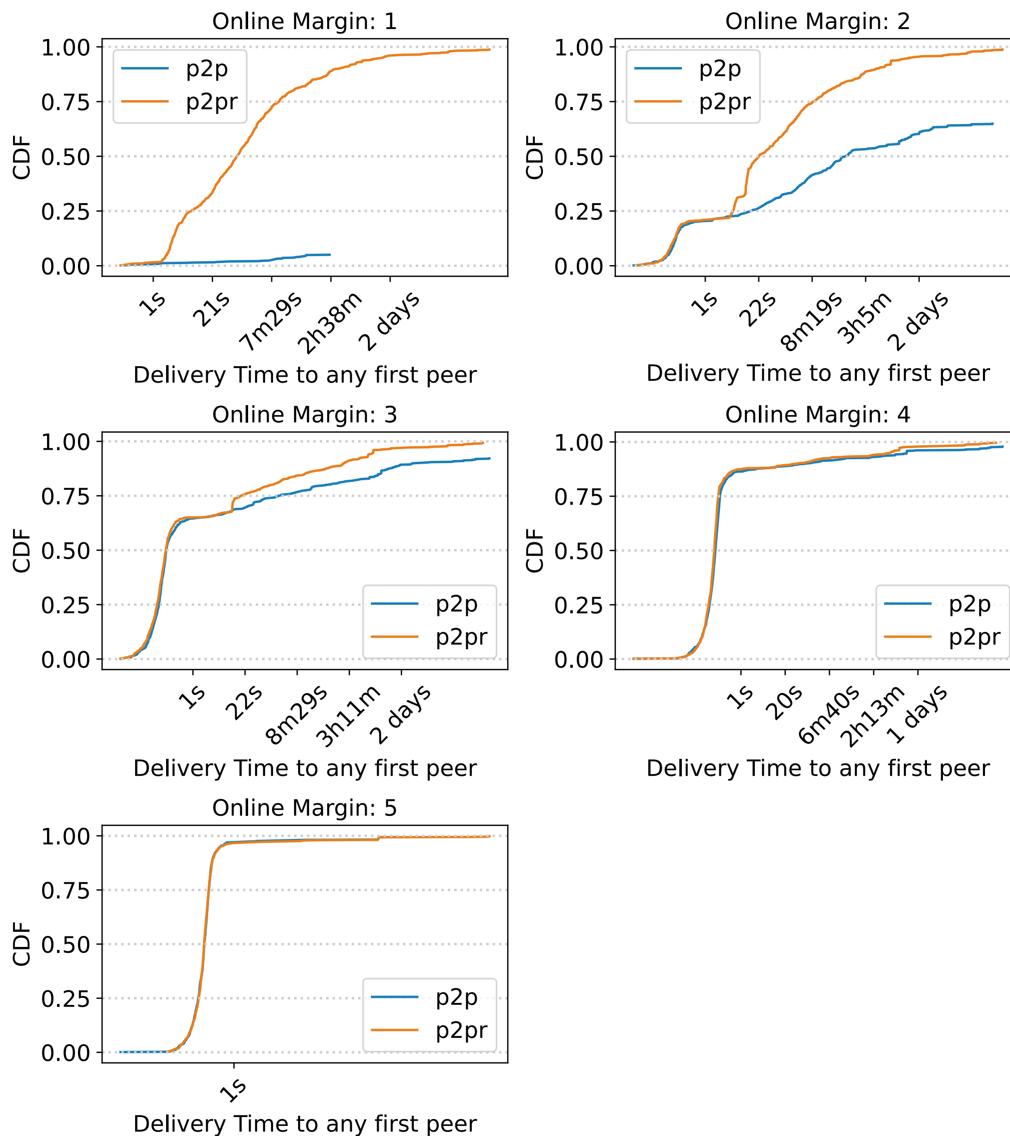

Figure 30: Delivery times to any first peer for each online margin.

For the second scenario comprising all chats, the CDFs for the applied online margins of each individual chat are shown in Figure 30. Therefore, the margin length is always relative to the chats traffic characteristics, especially the IATs. The results are comparable to the previous ten-user chat. Regarding the first margins, some events were successfully delivered to at least one peer in P2P. The increase of the curve for the second margin is less apparent than the increase of the previous ten-user scenario. However, a slight increase can be observed, indicating that several second online margins fall into a close range.

As mentioned in the previous scenario, the distributions of delivery times depend on the lengths on the online margins, and therefore the synchronous online times. Even though the delivery time difference between P2P and P2PR appears insignificant for the fourth and fifth online margin, relays can still improve the rate of events delivered to at least one peer, as shown in Figure 27 and Figure 28.

6.5.3 Online Periods without Receiving Events

Evaluation Goal	Show that P2PR has fewer online periods without receiving any events than P2P.
Scenario	
↳*Topologies*	10 users, 194 messages exchanged in 10 days, 2 relays for P2PR
↳*Traffic*	Chat 1796 from [56]
Method	Measure share of online periods of peers, where the peer does not receive any events.
Expected Result	P2P with relays has lower share of online periods without event reception for shorter online margins.

This experiment should show that P2P Matrix with relays has a lower number of "empty" online periods, where the peer does not receive any events. An empty online period could occur either because no events are exchanged during this period, or despite events having been sent, they are not delivered to the receiver. The scenario from the previous experiments is used again. The empty online periods are expected to be lower for P2PR, because if events exchanged before the online period, relays cache the event, and forward the it to the receiver after it returns online and resyncs.

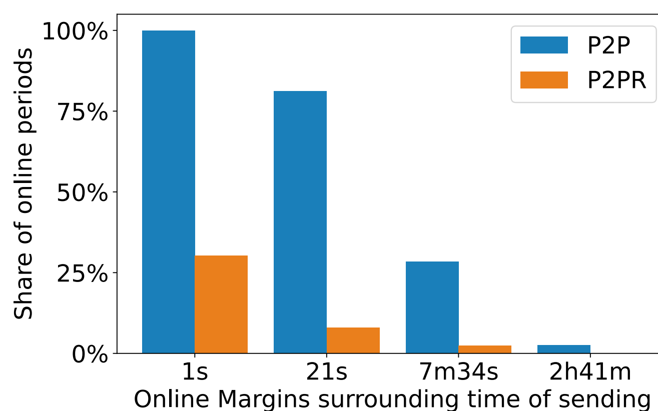

Figure 31: Number of peer online periods, where peers do not receive any events for each online margin. In P2P, short online periods result in a higher percentage of online periods without an event receptions.

The percentages of online periods without peers receiving events are depicted in Figure 31. The largest online margin is not depicted, as every online period contains at least one event reception. As expected, for the shortest online margin, peers in P2P do not receive any events, because there are no overlapping online periods. In P2PR, some online periods are empty, because a peer coming online twice in close succession without any other peer being online in between, does not receive any events. In the second and third online margin the differences in the number of empty online periods are substantial between P2P and P2PR, while becoming insignificant for the fourth online margin.

This experiment provides insight into how many online periods occur without a peer receiving any event. The difference between the empty online periods between P2P and P2PR represents the amount of online periods, where peers in a P2P system with the asynchronous delivery property would have received any event. In summary, for a certain distribution of online periods, a peer is more likely to receive any event after returning online in P2PR compared to P2P.

6.5.4 Reception Time after Returning Online

Evaluation Goal	Show that peers receive events faster after returning online in P2PR compared to P2P.
Scenario	
↳ *Topologies*	10 users, 194 messages exchanged in 10 days, 2 relays for P2PR
↳ *Traffic*	Chat 1796 from [56]
Method	Measure time between returning online and receiving first event.
Expected Result	Reception times are lower for P2PR.

This experiment should show the reception times after returning online are shorter for P2PR compared to P2P. The scenario from the previous experiments is used. For each online period and for all peers, the time between a peer returning online and receiving an event is measured. The reception times are expected to be lower for P2PR, because peers receive previously exchanged events after returning online and resyncing, regardless of which other peers are online. In contrast, the reception times after returning online depend on which other peers are online, for peers in P2P. They also depend on which other peers are online, because only those peers that have sent events in the previous offline period of the receiving peer, and are synchronously online, proactively re-try sending the event to the peer returning online. As missing events are only sent when a new event after the missing event is propagated to the peer, and the receiving peers then re-requests the missing event, the existing Matrix protocol also impacts the reception time.

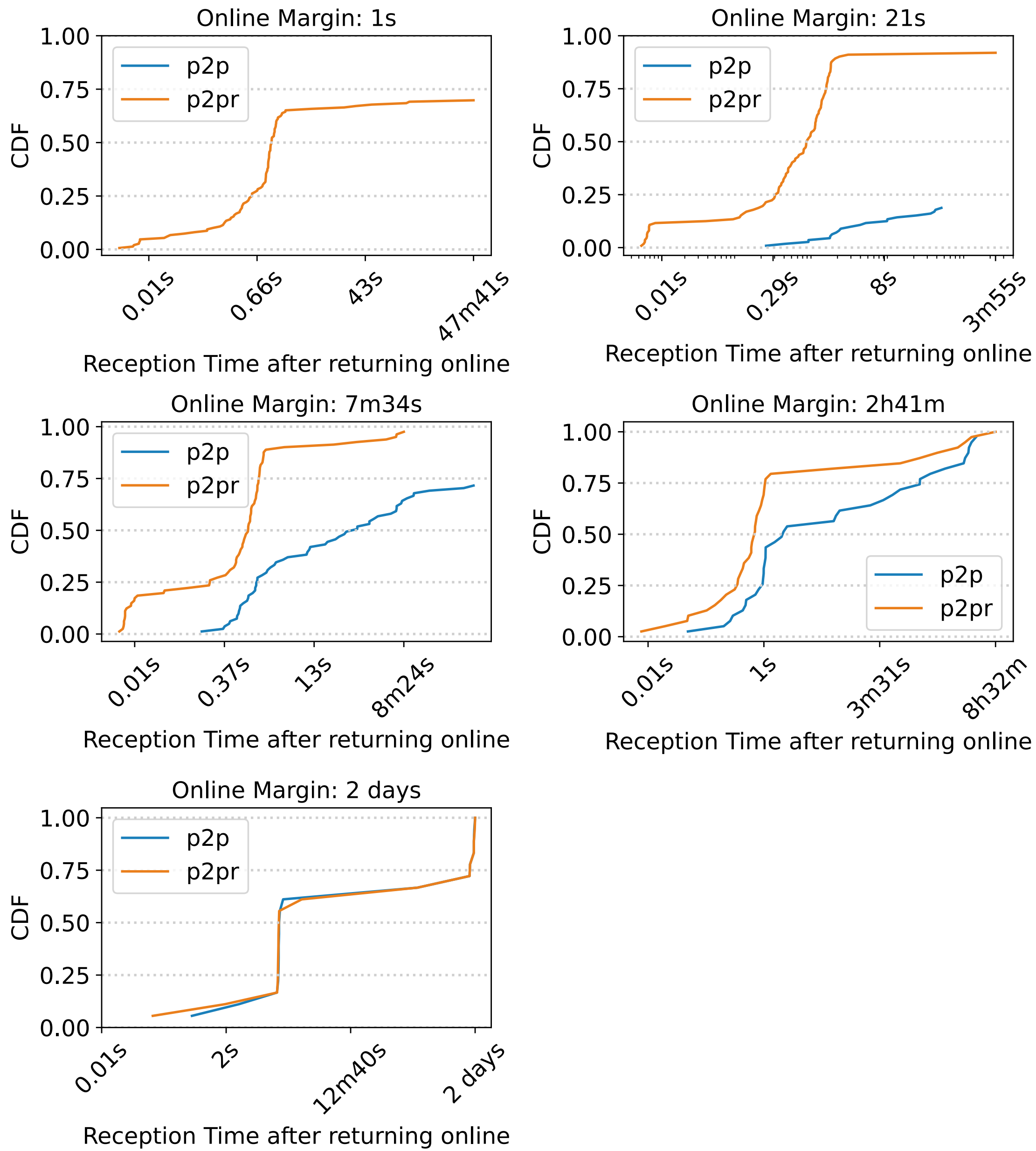

Figure 32: Reception times after returning online per online margin, i.e., time between returning online and receiving first event.

The CDFs of reception times after returning online for each online margin are depicted in Figure 32. There are no reception times for any peers in the first online margin for P2P, because no events are delivered within the online periods, as shown in Figure 31. For P2PR, the CDF curve of reception times is similar between the online margins, although the scaling of reception times is slightly different. While in the third online margin, the exponential increase ends at 1.6 seconds (77%), in the fifth online margin, it ends after 30 seconds (52%). Except for the fifth online margin, P2PR has a higher share of shorter reception times compared to P2P. The reception time difference stems not only from the fact that peers have to wait until other peers return online and provide missing events, but also because the round-trip times for a peer's resync with the relay are shorter than the discovery of other online peers of the Pinecone network. The latter aspect depends solely on the efficiency both the round-trip times of resyncing and the underlying P2P

network's peer discovery process, and could prove negligible for more efficient discovery mechanisms.

This experiment provides an insight into the time it takes for peers to receive the first events after returning online. Reducing reception times after returning online can contribute to the user experience, as peer users receive events faster, without having to wait for the peer reconnecting to the P2P network.

6.6 Summary

The goal of the evaluation was to validate both the functionality of the implementation, and also the performance improvements relays make in P2P Matrix compared to pure P2P Matrix. In order to validate both aspects, an evaluation setup was introduced. The setup included input formats for functional- and performance-oriented traffic files, a network configuration for peers and relays running in Docker containers, and a logging mechanism to track the activity of peers and relays. The evaluation setup allows running multiple experiments automatically, by generating lower level configurations from higher-level configuration files and scripts.

The functional evaluation consists of manually constructed traffic files, each designed to validate one certain aspect of P2PR. The results of the functional evaluation show that the implementation works correctly for the constructed traffic and the tested topologies. For evaluating performance, 18 chats were selected from a publicly available WhatsApp group traffic dataset. Since asynchronous online state of peers is a prerequisite for evaluating the improvement of P2PR towards P2P, a mechanism that models online periods around sending times is introduced. The results of the performance evaluation show that relays do not only improve the rate of successful delivery, but also can improve the overall delivery times, provided that peers are asynchronously online for a substantial amount of time.

7. Conclusion and Future Work

This work addresses the challenge of providing distributed asynchronous delivery for events in P2P Matrix, with the goal of eliminating the need of peers to be online simultaneously in order to exchange events. The notions of asynchronous, as-soon-as-possible delivery were formalized in such a way that this formalization can be applied to various classes of messaging systems. Dedicated relays were introduced to provide this functionality to rooms in P2P Matrix. The relays were designed in a way that avoids the drawbacks of homeservers in the current, federated Matrix architecture, such as accumulating metadata, load centralization towards few homeservers, single point of failure for users. Consequently, the relays store less state than homeservers in federated Matrix, are associated with rooms rather than users, and can be switched arbitrarily. Also, relays are designed to authorize peers by only storing a subset of the room state, i.e., an Authorization DAG, thereby reducing the stored metadata.

A proof-of-concept implementation was developed, incorporating relays and the necessary modifications to peers in P2P Matrix. Also, a reproducible evaluation setup was implemented, allowing to run peers both with or without relays in a controlled environment. Using this setup, the relay solution was systematically evaluated. Traffic data from a publicly accessible WhatsApp group traffic dataset was applied to the proof-of-concept implementation within the automated evaluation setup, in order to evaluate the system's performance. To evaluate asynchronous online times between peers, a model was defined that generates online periods of varying lengths around sending times. By applying both the WhatsApp group traffic and the model of online periods, relay-enhanced P2P Matrix was compared to pure P2P Matrix consisting solely of peers. The results demonstrate that relays can improve both the number and the duration of successfully delivered events to any first peer, as well as reduce the time until events are received after peers return online.

This work leads to several further questions that are worth pursuing in future work. New event dissemination mechanisms between relays could be introduced, to either maximize the performance of delivery times, or for the system to become more resilient against sophisticated adversaries. Also, to reduce the state stored at the relays required for authorization, new authorization techniques could be evaluated, e.g., Zero-Trust-Mechanisms, where peers could prove their membership or privileges without disclos-

ing too much information. Moreover, relays could be utilized not only to provide asynchronous delivery for the peers, but also to reduce the peer's energy consumption, by handling resource-intensive computations or reducing the number of targets to which a peer must forward its events. The aspect of energy-efficiency with relays could be evaluated in scenarios, where efficient energy consumption is critical, e.g., IoT networks.

Bibliography

[1] Eurostat, “Internet use: Instant messaging.” Jun. 16, 2024. Accessed: Jul. 12, 2024. [Online]. Available: https://ec.europa.eu/eurostat/databrowser/bookmark/fe0f0561-feaa-40ef-ba43-1ace06a2f9c7?lang=en

[2] G. M. Volpicelli, “How governments and spies text each other.” Accessed: Dec. 07, 2023. [Online]. Available: https://www.wired.co.uk/article/matrix-encrypted-messaging-app-governments

[3] N. V. Ltd, “Secure collaboration for a wide range of industries and sectors.” Accessed: Dec. 07, 2023. [Online]. Available: https://element.io/sectors

[4] F. Jacob, J. Grashöfer, and H. Hartenstein, “A Glimpse of the Matrix (Extended Version): Scalability Issues of a New Message-Oriented Data Synchronization Middleware,” *CoRR*, 2019, [Online]. Available: http://arxiv.org/abs/1910.06295

[5] M. Hodgson, “Designing Matrix: A Global Decentralised End-to-End Encrypted Communication Network,” Dublin: USENIX Association, Oct. 2023. Accessed: Jul. 14, 2024. [Online]. Available: https://www.usenix.org/conference/srecon23emea/presentation/hodgson

[6] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, “The many faces of publish/subscribe,” *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114–131, Jun. 2003, doi: 10.1145/857076.857078.

[7] S. Gilbert and N. Lynch, “Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services,” *SIGACT News*, vol. 33, no. 2, pp. 51–59, Jun. 2002, doi: 10.1145/564585.564601.

[8] E. Brewer, “CAP twelve years later: How the 'rules' have changed,” *Computer*, vol. 45, no. 2, pp. 23–29, Feb. 2012, doi: 10.1109/mc.2012.37.

[9] F. Jacob, S. Bayreuther, and H. Hartenstein, “On CRDTs in Byzantine Environments,” in *Sicherheit, Schutz und Zuverlässigkeit: Konferenzband der 11. Jahrestagung des Fachbereichs Sicherheit der Gesellschaft für Informatik e.V. (GI), Sicherheit 2022, Karlsruhe, Germany, April 5-8, 2022*, 2022, pp. 113–126. doi: https://doi.org/10.18420/sicherheit2022_07.

[10] W. Vogels, “Eventually consistent,” *Commun. ACM*, vol. 52, no. 1, pp. 40–44, Jan. 2009, doi: 10.1145/1435417.1435432.

[11] The Matrix.org Foundation CIC, “Matrix Specification: 3. Architecture.” Accessed: Apr. 10, 2024. [Online]. Available: https://spec.matrix.org/v1.10/#room-structure

[12] The Matrix.org Foundation CIC, “Room Version 11: 3.3 State Resolution.” Accessed: Apr. 10, 2024. [Online]. Available: https://spec.matrix.org/v1.10/rooms/v11/#state-resolution

[13] Neil Alexander, “State Resolution v2.” Accessed: Apr. 10, 2024. [Online]. Available: https://matrix.org/docs/older/stateres-v2/

[14] Hubert Chathi, “State Resolution: Reloaded.” Accessed: Jun. 06, 2024. [Online]. Available: https://matrix.uhoreg.ca/stateres/reloaded.html

[15] The Matrix.org Foundation CIC, “Matrix Specification: 7.5 Room Events.” Accessed: Apr. 10, 2024. [Online]. Available: https://spec.matrix.org/v1.10/client-server-api/#mroompower_levels

[16] F. Jacob, L. Becker, J. Grashöfer, and H. Hartenstein, “Matrix Decomposition: Analysis of an Access Control Approach on Transaction-based DAGs without Finality,” in *Proceedings of the 25th ACM Symposium on Access Control Models and Technologies*, in SACMAT '20. Barcelona, Spain: Association for Computing Machinery, 2020, pp. 81–92. doi: 10.1145/3381991.3395399.

[17] The Matrix.org Foundation CIC, “Matrix Specification: Administration & privileges.” Accessed: Apr. 10, 2024. [Online]. Available: https://matrix.org/docs/matrix-concepts/rooms_and_events/#administration-privileges

[18] The Matrix.org Foundation CIC, “Matrix Specification: 8.4 Permissions.” Accessed: Apr. 10, 2024. [Online]. Available: https://spec.matrix.org/v1.10/client-server-api/#permissions

[19] A. Tanenbaum and M. van Steen, *Distributed Systems*, Third edition, Version 3.01. Pearson Education Inc., 2017.

[20] R. Schollmeier, “A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications,” in *Proceedings First International Conference on Peer-to-Peer Computing*, in PTP-01. IEEE Comput. Soc. doi: 10.1109/p2p.2001.990434.

[21] The Annotated Gnutella Protocol Specification v0.4, Accessed: Apr. 29, 2024. [Online]. Available: https://rfc-gnutella.sourceforge.net/developer/stable/

[22] S. Saroiu, K. P. Gummadi, and S. D. Gribble, “Measuring and analyzing the characteristics of Napster and Gnutella hosts,” *Multimedia Systems*, vol. 9, no. 2, pp. 170–184, Aug. 2003, doi: 10.1007/s00530-003-0088-1.

[23] M. Hodgson, “Decentralised user accounts.” Accessed: Feb. 26, 2024. [Online]. Available: https://github.com/matrix-org/matrix-spec/issues/246

[24] N. Alexander, “Dendrite.” Accessed: Mar. 11, 2024. [Online]. Available: https://github.com/matrix-org/dendrite/blob/928c8c8c4af0e933e52dbf42d9237234d1a3bc9d/README.md

[25] Protocol Labs, “libp2p Connectivity.” Accessed: Mar. 06, 2024. [Online]. Available: https://connectivity.libp2p.io/

[26] N. Alexander, “Yggdrasil Network.” Accessed: Apr. 25, 2024. [Online]. Available: https://yggdrasil-network.github.io/

[27] N. Alexander, “Pinecone.” Accessed: Dec. 07, 2023. [Online]. Available: https://matrix-org.github.io/pinecone/

[28] New Vector Ltd, Accessed: Apr. 22, 2024. [Online]. Available: https://github.com/element-hq/element-android-p2p/tree/7883fa21824e124e71a452a4636e4a3afef3785c

[29] New Vector Ltd, Accessed: Apr. 22, 2024. [Online]. Available: https://github.com/element-hq/element-ios-p2p/commit/43b52b75750dfaea67b2c638ad5715d980db51bb

[30] M. Hodgson, “Introducing P2P Matrix.” Accessed: Jun. 02, 2020. [Online]. Available: https://matrix.org/blog/2020/06/02/introducing-p2p-matrix/

[31] devonh, “MSC4080: Cryptographic Identities (Client-Owned Identities).” Accessed: Nov. 15, 2023. [Online]. Available: https://github.com/devonh/matrix-spec-proposals/blob/cryptoIDs/proposals/4080-cryptographic-identities.md

[32] W. W. W. Consortium, “Decentralized Identifiers.” Accessed: Jul. 19, 2022. [Online]. Available: https://www.w3.org/TR/did-core/

[33] C. Scheideler, “Relays: Towards a Link Layer for Robust and Secure Fog Computing,” in *Proceedings of the 2018 Workshop on Theory and Practice for Integrated Cloud, Fog and Edge Computing Paradigms*, in TOPIC '18. Egham, United Kingdom: Association for Computing Machinery, 2018, pp. 1–2. doi: 10.1145/3229774.3229781.

[34] V. Setty, M. van Steen, R. Vitenberg, and S. Voulgaris, “PolderCast: Fast, Robust, and Scalable Architecture for P2P Topic-Based Pub/Sub,” in *Middleware 2012*, P. Narasimhan and P. Triantafillou, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 271–291.

[35] S. A. Baset and H. Schulzrinne, “Reliability and relay selection in peer-to-peer communication systems,” in *Principles, Systems and Applications of IP Telecommunications*, in IPTComm '10. Munich, Germany: Association for Computing Machinery, 2010, pp. 111–121. doi: 10.1145/1941530.1941547.

[36] Y. D. Max Skibinsky, “Asynchronous Mobile Peer-to-peer Relay,” 2015, Accessed: Mar. 06, 2024. [Online]. Available: https://s3-us-west-1.amazonaws.com/vault12/crypto_relay.pdf

[37] Edgegap Technologies Inc., “Distributed Relay Manager.” Accessed: Mar. 06, 2024. [Online]. Available: https://docs.edgegap.com/docs/distributed-relay-manager/

[38] R. Banno, J. Sun, M. Fujita, S. Takeuchi, and K. Shudo, “Dissemination of edge-heavy data on heterogeneous MQTT brokers,” in *2017 IEEE 6th International Conference on Cloud Networking (CloudNet)*, IEEE, Sep. 2017. doi: 10.1109/cloudnet.2017.8071523.

[39] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The Second-Generation Onion Router," in *13th USENIX Security Symposium (USENIX Security 04)*, San Diego, CA: USENIX Association, Aug. 2004.

[40] Nostr, "A decentralized social network with a chance of working." Accessed: Mar. 09, 2024. [Online]. Available: https://nostr.com/

[41] fiatjaf, "Nostr: Basic protocol flow description." Accessed: Mar. 09, 2024. [Online]. Available: https://github.com/nostr-protocol/nips/blob/a46338bd6a183f17a292ea578ee69ff15ea6723f/01.md

[42] D. Tarr, E. Lavoie, A. Meyer, and C. Tschudin, "Scuttlebutt Protocol Guide." Accessed: Mar. 11, 2024. [Online]. Available: https://ssbc.github.io/scuttlebutt-protocol-guide/#introduction

[43] D. Tarr, E. Lavoie, A. Meyer, and C. Tschudin, "Secure Scuttlebutt: An Identity-Centric Protocol for Subjective and Decentralized Applications," in *Proceedings of the 6th ACM Conference on Information-Centric Networking*, in ICN '19. Macao, China: Association for Computing Machinery, 2019, pp. 1–11. doi: 10.1145/3357150.3357396.

[44] Berty Technologies, "Wesh protocol." Accessed: Mar. 11, 2024. [Online]. Available: https://berty.tech/docs/protocol/#high-availability

[45] N. Alexander, "Pinecones and Dendrites - P2P Matrix Progress." Accessed: Feb. 06, 2021. [Online]. Available: https://archive.fosdem.org/2021/schedule/event/matrix_pinecones/

[46] D. Hudson, "Relay Server Architecture." Accessed: Mar. 11, 2024. [Online]. Available: https://github.com/matrix-org/dendrite/blob/24a865aeb76dbff1a8a95d549e19070ff102e441/relayapi/ARCHITECTURE.md

[47] Briar, "How It Works." Accessed: Mar. 27, 2024. [Online]. Available: https://briarproject.org/how-it-works/

[48] Nico Alt, "Briar Wiki." Accessed: Nov. 27, 2023. [Online]. Available: https://code.briarproject.org/briar/briar/-/wikis/home

[49] Torsten Grote and Michael Rogers, "Briar Mailbox released to improve connectivity." Accessed: Jun. 15, 2023. [Online]. Available: https://briarproject.org/news/2023-briar-mailbox-released/

[50] The Matrix.org Foundation CIC, "Matrix Specification: 5. PDUs." Accessed: May 06, 2021. [Online]. Available: https://spec.matrix.org/v1.9/server-server-api/#pdus

[51] Element, "Synapse." Accessed: May 28, 2024. [Online]. Available: https://github.com/element-hq/synapse/tree/9edb725ebcd41c0ca1ee8cbb833dcb28df47a402

[52] Famedly, "Conduit." Accessed: Jul. 12, 2024. [Online]. Available: https://gitlab.com/famedly/conduit/-/tree/8abab8c8a0e3ef406f8a107936f07031d964ac5f

[53] J. Volk, J. Tjoelker, and A. Conill, "Construct." Accessed: Jul. 12, 2024. [Online]. Available: https://github.com/matrix-construct/construct/tree/0624b69246878da592d3f5c2c3737ad0b5ff6277

[54] M. Hodgson, "Dendrite is entering Beta!." Accessed: May 28, 2024. [Online]. Available: https://matrix.org/blog/2020/10/08/dendrite-is-entering-beta/

[55] The Matrix.org Foundation CIC, "gomatrixserverlib." Accessed: May 28, 2024. [Online]. Available: https://godoc.org/github.com/matrix-org/gomatrixserverlib

[56] A. Seufert, F. Poignée, M. Seufert, and T. Hoßfeld, "Share and Multiply: Modeling Communication and Generated Traffic in Private WhatsApp Groups," *IEEE Access*, vol. 11, no. , pp. 25401–25414, 2023, doi: 10.1109/ACCESS.2023.3254913.

[57] T. Asokan, "mautrix-go." Accessed: Jun. 06, 2024. [Online]. Available: https://pkg.go.dev/maunium.net/go/mautrix

[58] F. Souza, "go-dockerclient." Accessed: Jul. 12, 2024. [Online]. Available: https://pkg.go.dev/github.com/fsouza/go-dockerclient@v1.11.0

[59] D. Inc, "Docker Engine API (1.46)." Accessed: Jul. 12, 2024. [Online]. Available: https://docs.docker.com/engine/api/v1.46/

[60] The Matrix.org Foundation CIC, "gomatrix." Accessed: Sep. 26, 2022. [Online]. Available: https://github.com/matrix-org/gomatrix/tree/ceba4d9f75305223c2598cda1b1090f438b1e2fa

[61] The Matrix.org Foundation CIC, "Client-Server API." Accessed: 2024. [Online]. Available: https://spec.matrix.org/v1.10/client-server-api/

[62] Docker Inc., "Docker Compose overview." Accessed: Jun. 06, 2024. [Online]. Available: https://docs.docker.com/compose/

A. Appendix

A.1. Evaluation Setup

```
{
  "messages": [
    {
      "action" : "online",
      "user": "peer0",
      "pauseAfter": "1s"
    },
    {
      "action" : "offline",
      "user": "peer1",
      "pauseAfter": "3s"
    },
    {
      "action" : "message",
      "user": "peer0",
      "content": "This is an asynchronous message.",
      "pauseAfter": "4s"
    },
    {
      "action" : "offline",
      "user": "peer0",
      "pauseAfter": "3s"
    },
    {
      "action" : "online",
      "user": "peer1",
      "pauseAfter": "1s"
    }
  ]
}
```

Listing 13: Example input file for functional evaluation experiment, as described in Section 6.1.2.

	Date	User	Message Type	Char Count
0	00:00:00	0	1	22
1	00:00:19	0	1	14
2	00:00:29	0	1	16
3	00:01:24	1	1	50
4	00:01:54	0	1	18
5	00:13:40	1	1	21
…				

Table 11: Example input file for performance evaluation experiment, as described in Section 6.1.2.

A.2. Functional Evaluation Log Tables

	Time [s]	User	Action	Type	Event ID
0	…				
1	3.6956	Bob	send	m.room.message	1
2	7.9685	Bob	offline		
3	11.0332	Alice	online		
4	…				
5	12.5473	Alice	resync		
6	…				
7	12.5544	Alice	receive	m.room.message	1
8	16.5459	Alice	offline		
9	23.9915	Bob	online		
10	…				
11	28.8353	Bob	send	m.room.message	2
12	…				
13	33.0557	Bob	offline		
14	35.5785	Alice	online		
15	…				
16	36.546	Alice	resync		
17	…				
18	36.5518	Alice	receive	m.room.message	2

Table 12: Event Log for evaluating resync with a since-parameter in minimal setup. After the second resync, the peer receives the second event only.

	Time [s]	User	Action	Type	Event ID
0	...				
1	11.0174	Alice	online		
2	11.659	Alice	resync		
3	11.6604	Relay 0	receive-resync		
4	11.6607	Relay 0	reply-resync		
5	11.6616	Alice	resync-response		
6	11.6624	Alice	receive	m.room.message	1
7	...				
8	49.5103	Alice	online		
9	50.343	Alice	resync		
10	50.348	Relay 1	receive-resync		
11	50.3483	Relay 1	reply-resync		
12	50.3492	Alice	resync-response		
13	50.3501	Alice	receive	m.room.message	2

Table 13: Event Log for evaluating resync with different relays. Peer 0 receives the event in the first asynchronous online period, after peer 1 has sent the event. Since the traffic is otherwise equal to Table 12, the actions of the other peer are omitted.

	Time [s]	User	Action	Type	Event ID
0	8.2266	Alice	ban Bob	m.room.member	5
1	8.2676	Bob	receive	m.room.member	5
2	8.2681	Relay	receive	m.room.member	5
3	8.9696	Relay	cache-event	m.room.member	5
4	10.4409	Alice	send	m.room.message	6
5	10.7431	Relay	receive	m.room.message	6
6	10.9828	Relay	cache-event	m.room.message	6
7	12.6924	Bob	send	m.room.message	7
8	12.7957	Alice	receive	m.room.message	7
9	12.7992	Relay	receive	m.room.message	7

Table 14: Event Log for evaluating relays rejecting events of non-members with the AuthDAG given the latest available room state. Peer 0 bans Peer 1 from the room (row 0). When the banned Peer 1 attempts to send an event to the relay (row 7), the relay receives it (row 9), but does not store it in the event cache.

	Time [s]	User	Action	Type	Event ID
0	...				
1	13.3725	Alice	name 100	m.room.power_levels	5
2	13.5555	Bob	receive	m.room.power_levels	5
3	13.556	Relay	receive	m.room.power_levels	5
4	14.1243	Relay	cache-event	m.room.power_levels	5
5	15.5638	Alice	Bob: level 0	m.room.power_levels	6
6	15.5886	Bob	receive	m.room.power_levels	6
7	15.5887	Relay	receive	m.room.power_levels	6
8	16.2004	Relay	cache-event	m.room.power_levels	6
9	21.0904	Alice	offline		7
10	24.01	Bob	send	m.room.name	8
11	24.2674	Relay	receive	m.room.name	8

Table 15: Event Log for evaluating relays rejecting events of members with insufficient power level with the AuthDAG given the latest available room state. Peer 0 sets the required power level to change the room name to 100 (row 1). Peer 0 then sets the power level of Peer 1 to 0 (row 5). Peer 1 attempts to change the room name (row 10), and the relay receives it (row 11), but does not add it to the event cache, as Peer 0 only has the power level 0, but 100 is required.

A.3. Performance Evaluation Plots

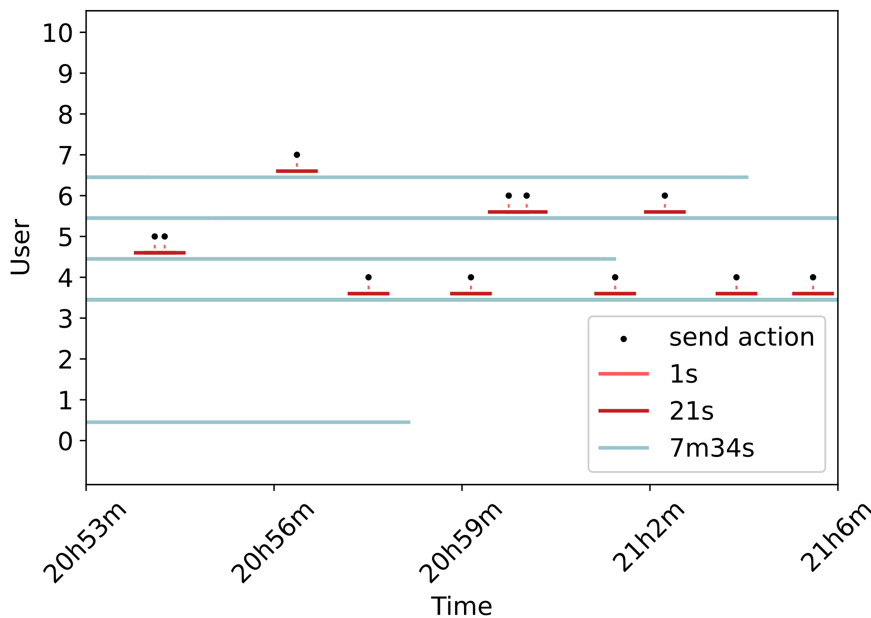

Figure 33: Send actions with the three shortest online margins applied to chat 1796. Only a small timeframe of the chat is depicted.