Cloud Standby

Eine Methode zur Vorhaltung eines Notfallsystems in der Cloud

Zur Erlangung des akademischen Grades eines
Doktors der Ingenieurwissenschaften
(Dr.-Ing.)
von der Fakultät für
Wirtschaftswissenschaften
des Karlsruher Instituts für Technologie (KIT)
genehmigte
DISSERTATION

von

Dipl.-Inform.Wirt Alexander Lenk

Tag der mündlichen Prüfung: 18.12.2014
Referent: Prof. Dr.-Ing. Stefan Tai
Korreferent: Prof. Dr. Andreas Oberweis

Karlsruhe, 2014

Zusammenfassung

Kleine und Mittelständische Unternehmen (KMU) sehen sich in ihrem Alltag immer wieder Gefahren ausgesetzt. So hat sich in der Vergangenheit gezeigt, dass selbst zuverlässige Anbieter mit vielen eigenen Rechenzentren immer wieder mit Ausfällen durch Notfälle zu kämpfen haben und auch große Unternehmen plötzlich den Betrieb einstellen müssen. Hierdurch kann es zu einer Unterbrechung der Geschäftsprozesse und zum Datenverlust kommen. Dabei führen 94 % der KMUs in Deutschland zwar regelmäßig Datensicherungen durch, aber gerade einmal 50 % sichern ihre kritischen Prozesse und die daran beteiligten Systeme mit einem Notfallsystem bei einem anderen Anbieter ab. Rund 52 % der Unternehmen geben an, für die Absicherung ihrer Systeme eine zu geringe Ausstattung an Budget, IT-Ressourcen oder Wissen zu haben. Eine mögliche Lösung, um hohen Kosten bei der Bereithaltung von nur sporadisch genutzten Ressourcen zu begegnen, ist die Nutzung von öffentlichen Cloud Ressourcen.

Ziel dieser Arbeit ist es daher, eine neue Methode für die Vorhaltung eines Notfallsystems in der Cloud (sog. „Cloud Standby“) zu entwickeln, die aus einer Methode zur Notfallwiederherstellung und einer modellbasierten Deployment-Methode besteht. Die Methode zur Notfallwiederherstellung besteht aus einem Notfallwiederherstellungsprozess, eines Notfallwiederherstellungsprotokolls und einer Entscheidungsunterstützung zur Konfiguration des Prozesses besteht und dieses leistet. Dabei baut diese Methode auf bestehenden Datensicherungslösungen und einer Methode zum anbieterunabhängigen Deployment eines verteilten Systems in der Cloud auf. Eine solche Deployment-Methode wird auch in der Arbeit vorgestellt. Sie besteht aus einer Beschreibungssprache, einem Deploymentprozess, einem Deploymentprotokoll und einem Deploymentalgorithmus.

Diese beiden neuen Methoden heißen zusammen Cloud Standby und wurden im Rahmen der Arbeit prototypisch implementiert und veröffentlicht.

Da die Absicherung mit einem Standby-System immer das Ziel hat, bei einem Notfall das Notfallsystem rechtzeitig wieder in Betrieb zu haben, wurde hierauf in der Evaluation ein besonderes Augenmerk gelegt. Es wurde gezeigt, dass sich durch die Nutzung von Cloud Standby allgemein die Deploymentzeit und damit die kleinstmögliche Widerherstellungszeit (RTO) im konkreten Anwendungsfall bis um den Faktor 20 reduzieren lassen. Hierbei kommt ein Teil der Einsparung durch die Notfallwiederherstellungskomponente und ein Teil durch die Planung des Deployments mit dem Deploymentalgorithmus zustande. Im Anwendungsfall war es so möglich, ein RTO von ca. 30 Minuten mit ca. 3% zusätzlichen Kosten zu erreichen. Eine Langzeitbetrachtung der Kosten ergab außerdem, dass bei 400€ Ausfallkosten pro Stunde ein Update-Intervall von 9h ohne Mehrkosten gewählt werden kann und das System auf einem Stand gehalten werden könnte, sodass im Notfall nur die Startzeit der virtuellen Maschinen, nicht jedoch der Datentransfer zu Gewicht schlägt und damit diese minimal sind. Zusätzlich wurde im Rahmen der Evaluation gezeigt, dass es mit der Deployment-Methode möglich ist, ein verteiltes System auf 7 verschiedenen Clouds, bei 4 verschiedenen Anbietern, abzusichern.

Danksagung

Ich danke meinem Doktorvater Prof. Dr. Stefan Tai für die Übernahme der Betreuung dieser Arbeit und seine Unterstützung. Ohne seine hohen Ansprüche und kontinuierlichen Antrieb wäre die Arbeit in dieser Form nicht möglich gewesen. Ich danke auch meinem Korreferent Prof. Dr. Andreas Oberweis für seine Unterstützung, sowie Prof. Dr. Stefan Nickel und Prof. Dr. Bruno Neibecker, die das Prüfungskomitee vervollständigen.

Ich möchte mich ganz besonders bei meinen Kollegen vom FZI Forschungszentrum Informatik und von der Forschungsgruppe eOrganisation vom Karlsruher Institut für Technologie (KIT) bedanken. Ohne Euer kontinuierliches konstruktives Feedback, Diskussionen und vorbehaltlose Unterstützung wäre diese Arbeit nicht möglich gewesen. Speziell bei meinen Arbeitskollegen vom FZI-Forschungsbereich IPE und den Kollegen der FZI-Außenstelle Berlin möchte ich mich für die angenehme Arbeitsatmosphäre und den freundschaftlichen Umgang bedanken. Ein besonderer Dank geht auch an alle Wissenschaftler, die meinen Werdegang maßgeblich geprägt und mich kontinuierlich unterstützt haben. Hierzu zählen im Speziellen die Post-Docs Dr. Frank Pallas, Prof. Dr. Jens Nimis, Dr. Valentin Zacharias und Dr. Christian Janiesch, sowie meine Kollegen Robin Bühler und Dr. Erik Wittern, die das Wagnis Berlin mit mir zusammen eingegangen sind. Ich weiß, dass diese Arbeit ohne Eure Unterstützung nicht möglich gewesen wäre. Ein weiterer Dank geht an Tobias Bräuer, Gründer von barcoo, der es mir in mehreren Diskussionen ermöglicht hat, einen tiefen Einblick in die barcoo-Infrastruktur zu bekommen und diese als Anwendungsfall in dieser Arbeit zu nutzen.

Abschließend geht noch ein Dank an meine Eltern, Familie und Freunde, die mich auf dem Weg zu dieser Arbeit vorbehaltlos unterstützt, Geduld bewiesen und mir in unzähligen Fällen meinen Rücken freigehalten haben.

Alexander Lenk

Inhaltsverzeichnis

1. Einleitung

Unternehmen sehen sich in ihrem Alltag immer wieder unmittelbar auftretenden kritischen Ereignissen ausgesetzt, die schnell zu einer Gefahr für das gesamte Unternehmen werden können, wenn ihnen nicht richtig begegnet wird. Produktionsstätten, Vertriebskanäle sowie kritische Geschäftsprozesse gegen mögliche Gefahren abzusichern ist daher eine der Aufgaben, der sich jedes Unternehmen stellen muss. In diesem Zusammenhang sind Gefahren beispielsweise Stromausfälle, Epidemien oder auch Naturkatastrophen[1].

Seit Beginn der Industriellen Revolution wurden Pläne für den Notfall erstellt und Konzepte entwickelt, damit Unternehmen nach einem Ausfall möglichst schnell wieder in den geregelten Betrieb übergehen können. Diese Pläne, Konzepte und die dazugehörigen Prozesse werden in den Wirtschaftswissenschaften zu Business Continuity Management oder auch Notfallmanagement zusammengefasst [20], [51]. Seit den 1970er Jahren konzentriert sich das Notfallmanagement auch auf die IT-Ressourcen, da diese seitdem für viele Geschäftsprozesse von grundlegender Bedeutung sind [20], [51]. Hierzu existieren verschiedene Ansätze zur Erhöhung der Verfügbarkeit durch die Nutzung von Fehlertoleranz-Konzepten, wie beispielsweise das Vermeiden von Single-Points-of-Failure.

Die Überbrückung eines Notfalls durch die Bereitstellung von bereits dafür vorgehaltenen Standby-Ressourcen, wie einem Notfallsystem, an einem anderen Standort ist ein bewährter, jedoch auch kostspieliger Ansatz [69]. Speziell, wenn das Notfallsystem innerhalb weniger Minuten oder Stunden nach dem Notfall zur Verfügung stehen soll. Während große Un-

[1] Teile diese Kapitels wurden bereits veröffentlicht [86]

ternehmen durch ihre finanzielle Ausstattung die Möglichkeit haben, mehrere Rechenzentren selbst zu betreiben und so Single-Points-of-Failures zu vermeiden, haben Kleine und Mittelständische Unternehmen (KMUs) diese Möglichkeit nicht. Ihnen bietet sich nur die Möglichkeit, ihre Server selbst zu betreiben und/oder diese an einen Rechenzentrums-Anbieter auszulagern. Allerdings stellt die Bindung an einen einzelnen Anbieter wieder einen Single-Point-of-Failure dar [4]. Darüber hinaus es hat sich in der Vergangenheit gezeigt, dass selbst zuverlässige Anbieter mit vielen eigenen Rechenzentren immer wieder mit Ausfällen durch Notfälle zu kämpfen haben [88] und sogar große Unternehmen plötzlich den Betrieb einstellen müssen [33]. Ist der Anbieter durch den Ausfall eines kompletten Rechenzentrums oder gar mehrerer Rechenzentren durch einen Notfall betroffen, ist nicht davon auszugehen, dass ein darin laufendes System in naher Zukunft wiederherzustellen ist. In solchen Fällen ist es nur möglich auf Anbieterunabhängigkeit zu setzen, die Ausfallzeit also mit einem Notfallsystem bei einem Anbieter zu überbrücken [4].

Bei der Absicherung von IT-Systemen im Notfallmanagement wird üblicherweise der in Abbildung 1 dargestellte Prozess aus *Initiierung*, *Anforderung und Strategie*, *Implementierung* und *Operatives Management* angenommen [16].

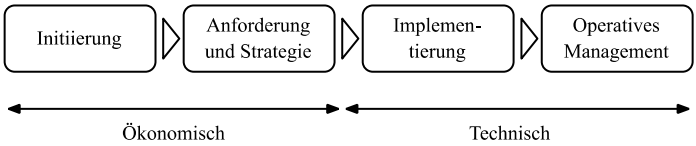

Abbildung 1: Notfallmanagement Prozess in der IT [16]

In den Schritten *Initiierung* und *Anforderung und Strategie* sind hauptsächlich unternehmensstrategische und ökonomische Fragen, wie der prinzipielle Einsatz von Notfallmanagement oder die Fragen nach den Ausfallkosten,

der Kritikalität von Geschäftsprozessen, den erlaubten Ausfallzeiten etc. zu beantworten. In den Schritten *Implementierung* und *Operatives Management* steht hingegen die technische Umsetzung des Notfallmanagements im Mittelpunkt: Wie wird ein Notfall festgestellt, welche Maßnahmen sind zu ergreifen und wie wird im laufenden Betrieb sichergestellt, dass das Notfallsystem bei einem Notfall aktuell ist und die Anforderungen, die in der Strategie definiert wurden, erfüllt. Gerade die technischen Fragen sind von KMUs nur schwer zu beantworten, da ihnen hierzu entweder das Wissen oder die passende finanzielle Ausstattung fehlt.

Dies wird durch Studien des Bundesministeriums für Wirtschaft und Energie[2] (BMWi) und Symantec bestätigt: Laut diesen Studien führen 94% der KMUs in Deutschland zwar regelmäßig Datensicherungen durch [19], jedoch sichern gerade einmal 50% ihre kritischen Prozesse und die daran beteiligten Systeme mit einem Notfallsystem bei einem anderen Anbieter ab [122]. Rund 52% der Unternehmen geben an, für die Absicherung ihrer Systeme eine zu geringe Ausstattung an Budget, IT-Ressourcen oder Wissen zu haben [122]. Die gleichen KMUs beziffern ihre Ausfallkosten im Durchschnitt mit $25.000 pro Tag [122].

Derzeit werden unterschiedliche Ansätze diskutiert, um dem Problem der kostspieligen und technisch aufwändigen Bereitstellung von Notfallsystemen im Rahmen des Notfallmanagements zu begegnen. Gerade Cloud Computing [94] wird immer wieder als eine mögliche Lösung vorgeschlagen, um ein Notfallsystem, das nicht laufend in Betrieb ist (Warm-Standby), zwar vorzuhalten, aber im Vergleich zu traditionellen Ansätzen Kosten zu sparen [69], [134]. Cloud Computing zeichnet sich dadurch aus, dass der Nutzer einen einfachen, direkten Zugang zu einem Pool von konfigurierbaren, elastischen Diensten (Netzwerke, Server, Speicher, Anwen-

[2] ehemals Bundesministerium für Wirtschaft und Technologie

dungen und sonstige Dienstleistungen) erhält und diese bedarfsgerecht abgerechnet werden [94]. Gerade durch diese bedarfsgerechte Abrechnung können die Kosten für die Vorhaltung von lediglich sporadisch genutzten Warm-Standby-Notfallsystemen stark reduziert werden [134]. Ein grundsätzliches Problem bei der Absicherung moderner Systeme ist aber, dass diese häufig verteilte Systeme, also aus mehreren, abhängigen Servern bestehende Systeme, sind [57], [123].

Bestehende Warm-Standby-Ansätze, die Cloud Computing zur Absicherung einsetzen, haben gemein, dass sie einzelne Server adressieren und die Abhängigkeiten zwischen ihnen bei der Absicherung nicht im Vordergrund stehen. Andere Ansätze, die auch für verteilte Systeme entworfen wurden, nutzen entweder kein Cloud Computing [119] oder konzentrieren sich auf kontinuierlich laufende Notfallsysteme (Hot-Standby) [32], [135], wodurch die Kostenspareffekte des Cloud Computing nicht ausreichend zum Tragen kommen. Für besonders wichtige Geschäftsprozesse, deren Ausfall dementsprechend kostspielig ist, kann ein Hot-Standby-Ansatz dennoch sinnvoll sein. Sollen jedoch nur weniger wichtige Prozesse für den Notfall abgesichert werden, so sind günstigere Warm-Standby-Ansätze, die auf Cloud Computing basieren, vorzuziehen.

Soll ein verteiltes System bei einem solchen Warm-Standby-Ansatz zur Aktualisierung oder im Notfall in der Cloud gestartet werden, wird eine Deployment-Methode benötigt, mit der verteilte Systeme anbieterunabhängig gestartet werden können. In der Literatur werden hierfür bereits Ansätze diskutiert. Diese Deployment-Methoden für die Cloud wurden jedoch nicht explizit dafür geschaffen, um ein Notfallsystem anbieterunabhängig bereitzustellen. Sie bewegen sich im klassischen Feld der Systemadministration und unterstützen Nutzer dabei, ein verteiltes System unkompliziert und automatisch bei einem einzelnen Cloud-Anbieter in Betrieb zu nehmen [72], [93] oder darauf eine einzelne Anwendung aufzuteilen und über Cloud-Anbieter-Grenzen hinweg zu betreiben [98]. Andere Ansätze bieten

ein Rahmenwerk, um verteilte Systeme interoperabel zu gestalten, sind aber bewusst sehr generisch gehalten, weshalb sie lediglich eine Basis für den Entwurf einer Deployment-Methode im Kontext der Bereitstellung von Notfallsystemen in der Cloud bieten können [65].

Für eine anbieterübergreifende Notfallwiederherstellung in der Cloud wird also eine neue Methode benötigt, um ein Notfallsystem für ein bestehendes verteiltes System in der Cloud vorzuhalten und im Notfall starten zu können. Hierfür bedarf es zum einen einer Methode zur kontinuierlichen Aktualisierung der Daten eines Notfallsystems in der Cloud, Erkennung von Ausfällen und Einleitung des Notbetriebs, zum anderen bedarf es einer Methode für das anbieterunabhängige Deployment in der Cloud, um dieses Notfallsystem automatisiert anstarten zu können. In der Literatur wurde weiterhin gezeigt, dass sich modell-getriebene Deployment-Ansätze bewähren, um verteilte Systeme automatisch bei einem Cloud-Anbieter zu deployen [34].

Ziel dieser Arbeit ist es daher, eine neue Methode für die Vorhaltung eines Notfallsystems in der Cloud (sog. „Cloud Standby“) zu entwickeln, die aus einer Methode zur Notfallwiederherstellung und einer modellbasierten Deployment-Methode besteht. Der Fokus dieser Arbeit liegt hierbei auf dem Bereich der Notfallwiederherstellung und nicht dem Störungsmanagement[3]. Dieses ist insbesondere dann relevant, wenn ein ganzes Rechenzentrum für einen längeren Zeitraum außer Betrieb geht oder ein Anbieter gänzlich vom Markt verschwindet. Die entwickelte Methode erhebt keinen Anspruch darauf alltägliche, punktuelle Ausfälle und Fehlfunktionen zu überbrücken oder eine Unternehmung gegen globale Katastrophen abzusichern. Auch konzentriert sich die Anwendung auf moderne verteilte Systeme. Ein Beispiel für ein solches modernes verteiltes System ist das

[3] siehe hierzu auch Kapitel 4.2.2

Backend einer der erfolgreichsten deutschen mobilen Apps, barcoo [8]. Es besteht aus mehreren, teilweise skalierbaren Schichten und ist über eine einheitliche Schnittstelle über das Internet zugreifbar. Mit Hilfe dieses verteilten Systems und der barcoo-App kann Verbrauchen vor dem Kauf eines Produkts Zusatzinformationen wie die Lebensmittelampel oder auch bedenkliche Zusatzstoffe angezeigt werden[4]. Es gibt jedoch auch verteilte Systeme, für die es keinen Sinn macht, diese mit Cloud Standby abzusichern: Existiert beispielsweise eine sehr komplexe Legacyanwendung bereits seit Jahren, so könnte es schwer sein, dieses verteilte System mit der in dieser Arbeit vorgestellten modellbasierten Deployment-Methode zu modellieren und zu deployen. In diesem Fall würde es sich eventuell anbieten, nur Teile der Anwendung, als eigene verteilte Systeme, mit Cloud Standby abzusichern. Gleiches gilt für eine Anwendung, die bereits als hochverfügbare Anwendung über anbietergrenzen konzipiert ist. Eine solche Anwendung benötigt keine weiteren Sicherheitsmechanismen wie die Absicherung mittels Cloud Standby und liegt damit auch nicht im Fokus dieser Arbeit [5].

1.1 Beiträge der Arbeit

Um das Ziel der Arbeit zu erreichen, werden zwei Teilbeiträge geleistet, die zusammen die *Methode zur Vorhaltung eines Notfallsystems in der Cloud (Cloud Standby)* bilden (siehe Abbildung 2):

[4] Eine detaillierte Darstellung des verteilten Systems hinter barcoo findet sich in der Evaluation in Kapitel 7.

[5] siehe zu diesen Punkten auch die kritische Betrachtung in Kapitel 8.2

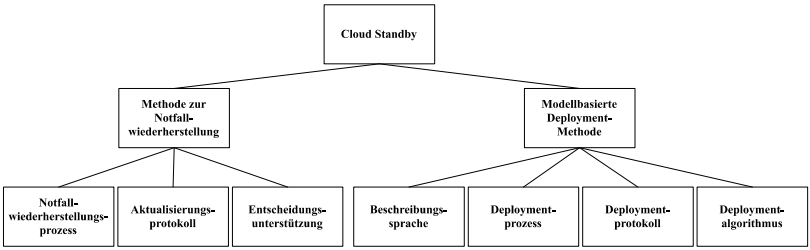

Abbildung 2: Übersicht über die in der Arbeit entwickelten Methoden

- *Eine Methode zur Notfallwiederherstellung von verteilten Systemen in der Cloud* – Die Methode setzt sich zusammen aus einem Notfallwiederherstellungsprozess, einem Protokoll für die Aktualisierung des Notfallsystems auf Basis bestehender, traditioneller Datensicherungslösungen und einer Entscheidungsunterstützung. Der Notfallwiederherstellungsprozess besteht aus den Teilprozessen für die Aktualisierung des Notfallsystems, für die Überwachung der Primär-Cloud, für die Initiierung des Notfallbetriebs und die Rückkehr in den Normalbetrieb. Die Entscheidungsunterstützung hilft bei der Konfiguration der Methode. Um die Prozesse durchführen zu können, nutzt die Methode zur Notfallwiederherstellung eine Deployment-Methode, die es ermöglicht, ein verteiltes System anbieterunabhängig automatisiert anzustarten.
- *Eine modellbasierte Deployment-Methode zum anbieterunabhängigen Deployment von verteilten Systemen in der Cloud* – Damit das abzusichernde verteilte System zur Aktualisierung und im Notbetrieb automatisch gestartet werden kann, wird eine Metamodellbasierte Beschreibungssprache entwickelt, die es ermöglicht, ein verteiltes System anbieterunabhängig zu beschreiben. Auf Basis der Beschreibungssprache werden weiterhin ein Deployment-Prozess, -Protokoll und -Algorithmus entwickelt, die es erlauben,

das beschriebene System automatisch bei verschiedenen Anbietern zu instanziieren.

Das Zusammenspiel der Komponenten und Methoden ist in Abbildung 3 grafisch dargestellt.

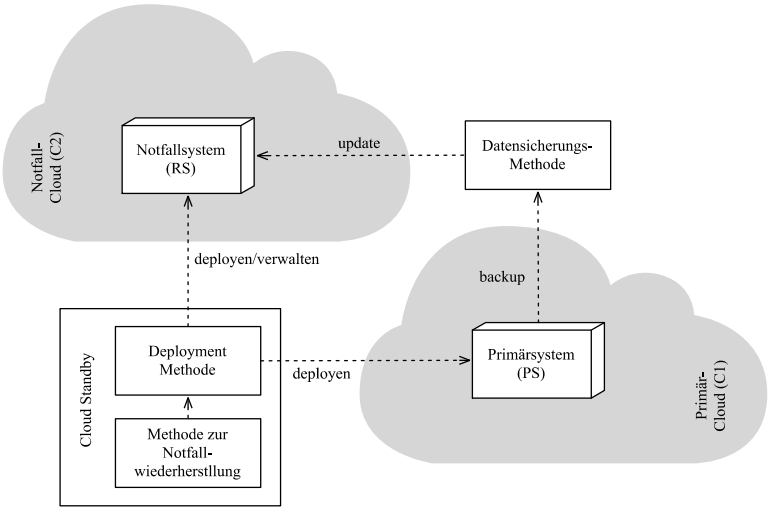

Abbildung 3: Zusammenspiel der Komponenten und Methoden (vgl. [86])

Die wesentlichen in dieser Arbeit zu betrachtenden Komponenten sind dabei das Primärsystem, das Notfallsystem und die verschiedenen Cloud-Anbieter.

Primärsystem (PS) – Das Primärsystem ist das verteilte System, das durch einen Warm-Standby-Ansatz abgesichert werden soll. Es wird in regelmäßigen Abständen mit einer Datensicherungsmethode gesichert, sodass die Daten im Notfall unabhängig von einer weiteren Absicherung mittels Notfallsystem auf einem neuen Primärsystem wiederhergestellt werden können.

Notfallsystem (RS) – Das Notfallsystem ist die Kopie des Primärsystems, das im Notfall den Betrieb übernimmt. Ziel des Cloud Standby Ansatzes ist es, dieses System bereitzustellen und aktuell zu halten. Damit das Replika System den Betrieb übernehmen kann, müssen die Anfragen der Nutzer des verteilten Systems auch das Notfallsystem erreichen. Hierfür gibt es verschiedene Verfahren, wie z.B. der Einsatz virtueller IP-Adressen oder dynamische DNS-Einträge. Diese Verfahren sind nicht Teil dieser Arbeit, sondern es wird hier auf etablierte Verfahren zurückgegriffen.

Cloud (C1 und C2) – Beim Cloud Computing wird häufig verschleiert, wo genau sich die Ressourcen wie virtuelle Maschinen oder Speicher befinden. In dieser Arbeit wird ein Cloud-Rechenzentrum-Standort vereinfacht als *Cloud* bezeichnet und ist die logische Einheit, in der vom Anbieter Rechen- und Speicherressourcen angeboten werden. Innerhalb einer Cloud sind die Transaktionskosten zwischen den Ressourcen, wie z.B. Kosten für Traffic zu vernachlässigen. Die Cloud dient also den *virtuellen Maschinen (VM)* als Laufzeitumgebung und bietet Speicherplatz an. Diese Cloud-Ressourcen können mit der *Cloud Management Schnittstelle* verwaltet werden. Typische Management-Aufgaben sind das Erstellen, Lesen, Aktualisieren und Löschen von Ressourcen (CRUD[6]).

Im Folgenden werden die Methoden dargestellt, die auf die zuvor vorgestellten Komponenten angewandt werden.

Datensicherungs-Methode – Die Datensicherungs-Methode ermöglicht es, mittels eines zentralen Datenspeichers alle Datensicherungen für das verteilte System zu speichern. Sie wird durch Datensicherungssoftware auf den einzelnen virtuellen Maschinen gefüllt. Das Intervall,

[6] create, read, update, delete

in dem die Datensicherungen durchgeführt werden, hängt von der Kritikalität der Daten ab. Diese Kritikalität leitet sich aus dem übergeordneten Geschäftsprozess und damit auch für die am Prozess beteiligten Systeme, virtuellen Maschinen und so weiter ab. Diese Bewertung wird im Rahmen des Notfallmanagements durchgeführt und wird im Cloud Standby System als gegeben angenommen. Es wird weiterhin angenommen, dass der aktuelle Stand der Datensicherung durch die bestehenden Datensicherungslösungen immer in einem Zustand ist, der den im Notfallmanagement definierten Anforderungen genügt (z.B. die letzte Datensicherung darf nicht älter als 24 Stunden sein). Die Datensicherung gewährleistet außerdem, dass die Daten, die auf das Notfallsystem repliziert werden, konsistent sind. Die Datensicherung ist erst dann abgeschlossen und zur Rücksicherung freigegeben, wenn alle virtuellen Server ihre Daten gesichert haben. Hierdurch wird auch die Replikation der Daten auf das Notfallsystem realisiert. Die Datensicherung dient als zentrale konsistente Datenquelle, auf die sowohl das Primärsystem als auch das Notfallsystem Zugriff haben. Durch die Nutzung der Datensicherung als gemeinsame Komponente werden bestehende Lösungen integriert. Zudem wird gewährleistet, dass sich das Notfallsystem immer auf einen, im Notfallmanagement definierten, hinreichend aktuellen Zustand bringen lässt.

Deployment-Methode – Die Deployment-Methode ist neben der Methode zur Notfallwiederherstellung der Kern von Cloud Standby. Die Deployment-Methode übernimmt die Aufgabe der koordinierten Verwaltung der am verteilten System beteiligten virtuellen Server. Die Deployment-Methode besitzt alle Informationen über das verteilte System, wie z.B. IP-Adressen oder Zugangsdaten und dient anderen Komponenten als Informationsquelle, wenn sie direkten Zugriff auf das verteilte System benötigen. In dieser Arbeit wird in Kapitel 5 eine modellbasierte Deployment-Methode vorgestellt, die den durch die Methode

zur Notfallwiederherstellung gestellten Anforderungen an eine Deployment-Methode, genügt.

Methode zur Notfallwiederherstellung – Die Aufgaben der Methode zur Notfallwiederherstellung sind die Aktualisierung des Notfallsystems, die Einleitung und Beendigung des Notbetriebs. Hierzu nutzt sie die Deployment-Methode für alle Aufgaben, die das Verwalten der einzelnen Instanzen betreffen. Die Methode zur Notfallwiederherstellung besteht aus dem Notfallwiederherstellungsprozess, dem Aktualisierungsprotokoll und einer Methode zur Wahl des Update-Intervalls.

Dabei können sowohl die Methode zur Notfallwiederherstellung als auch die Deployment-Methode für sich alleine stehen. Das heißt, es könnte im Rahmen der Notfallwiederherstellung ein anderer Ansatz als Deployment-Methode genutzt werden, falls er den im Rahmen der Arbeit identifizierten Anforderungen genügt. Weiterhin könnte die Deployment-Methode in einem anderen Kontext als in der Notfallwiederherstellung eingesetzt werden. Die beiden hier vorgestellten Methoden sind jedoch aufeinander abgestimmt, sodass deren gemeinsamer Einsatz als *Cloud Standby* empfohlen wird. Die Werkzeuge und Implementierungen, die diese Methoden umsetzen, können im produktiven Einsatz überall angesiedelt sein, es wird jedoch empfohlen, diese aus Verfügbarkeits- und Kostengründen in der Notfall-Cloud zu hosten: Falls die Primäre Cloud ausfällt, kann das Notfallsystem immer noch mit der letzten Version der Datensicherung gestartet werden. Außerdem berechnen viele Cloud-Anbieter nichts dafür, wenn Daten innerhalb einer Cloud transferiert werden. Für die Aktualisierung fallen somit keine Datentransferkosten an.

1.2 Aufbau der Arbeit

Die Entwicklung der Methode zur Vorhaltung eines Notfallsystems in der Cloud (Cloud Standby) gliedert sich in acht Kapitel. In Kapitel 2 werden

grundlegende Konzepte des Notfallmanagements, verteilter Systeme und Cloud Computing vorgestellt (siehe Kapitelübersicht in Abbildung 4). In Kapitel 3 werden die verwandten Arbeiten zu den Themengebieten „Methoden der Notfallwiederherstellung in der Cloud“ und „Modellbasierte Deployment-Methoden“ vorgestellt.

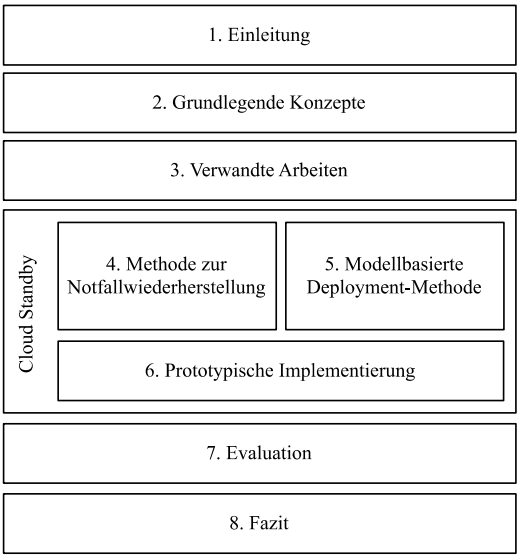

Abbildung 4: Kapitelübersicht

Der erste Teilbeitrag, eine neue Methode zur Notfallwiederherstellung von verteilten Systemen in der Cloud, wird in Kapitel 4 vorgestellt. Hierbei werden zunächst die angestrebten Eigenschaften, Annahmen und Terminologie beschrieben und darauf aufbauend die Methode entwickelt. Diese Methode besteht aus einem Prozess zur Notfallwiederherstellung, einem Protokoll zur Aktualisierung des Notfallsystems und einer Entscheidungsunterstützung zur Parametrierung des Prozesses. Zur Durchführung dieser

Methode ist es notwendig, dass ein verteiltes System bei mehreren Anbietern automatisch in Betrieb genommen werden kann. Diese Deployment-Methode wird in Kapitel 5 der Arbeit vorgestellt. Es wird auch hier zunächst auf die Anforderungen, Designentscheidungen und Annahmen eingegangen. Auf dieser Basis wird dann eine neue Beschreibungssprache entwickelt und ein Prozess zum Deployment eines so beschriebenen Systems bei mehreren Anbietern vorgestellt. Die Sprache, der Prozess und die Protokolle bilden zusammen den zweiten Teilbeitrag dieser Arbeit: Die modellbasierte Deployment-Methode zum anbieterunabhängigen Deployment von verteilten Systemen in der Cloud. Die Deployment-Methode bildet zusammen mit der Notfallwiederherstellungsmethode Cloud Standby. Im Anschluss wird zunächst mit einer prototypischen Implementierung in Kapitel 6 gezeigt, dass die entworfene Methode zur Vorhaltung eines Notfallsystems in der Cloud verwendet werden kann. Die hier beschriebenen Software-Komponenten und der Editor werden im Folgenden für die anwendungsfallgetriebene Evaluation in Kapitel 7 genutzt. In der Evaluation werden mittels Experimenten die Wiederherstellungsdauern und Kosten eines verteilten Systems mit und ohne den in der Arbeit vorgestellten Methoden verglichen. Da es bei der Absicherung mittels Standby-Ressourcen immer zu zusätzlichen Kosten kommt, wird aufbauend auf der experimentellen Evaluierung, unter Berücksichtigung der Ausfallkosten, eine simulative Langzeitbetrachtung der Kosten durchgeführt. Die Ergebnisse werden in Kapitel 8 präsentiert, kritisch diskutiert, mögliche weitergehende Arbeiten vorgestellt und die gesamte Arbeit nochmals zusammengefasst.

Um ein besseres Verständnis davon zu bekommen, wie die beiden Teilbeiträge bereitgestellt und genutzt werden können, wird im Folgenden erläutert, wie die Methode zur Notfallwiederherstellung und die modellbasierte Deployment-Methode formalisiert, implementiert und einem Anwender zur Nutzung bereitgestellt wird. Für die Notfallwiederherstellungsmethode ist dies in Abbildung 5 dargestellt.

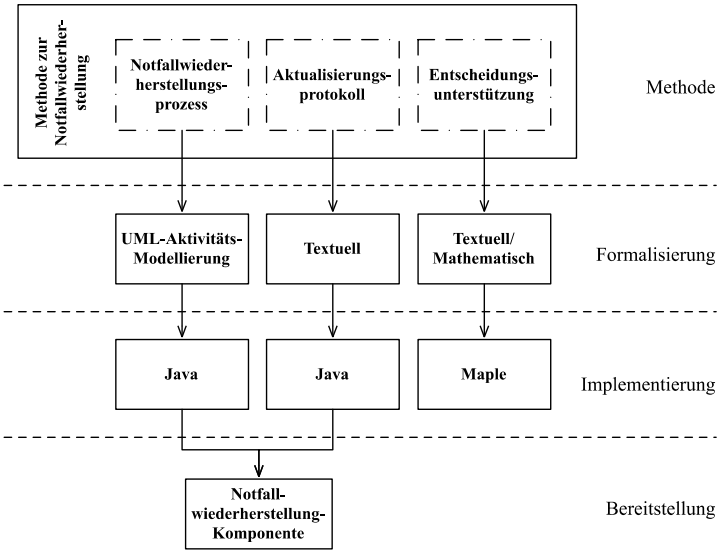

Abbildung 5: Übersicht über die Methode zur Notfallwiederherstellung und die Umsetzung in dieser Arbeit

Der Notfallwiederherstellungsprozess wird mittels UML-Aktivitäts-Modellierung [109], das Protokoll textuell und die Entscheidungsunterstützung textuell/mathematisch formalisiert. Sowohl der Prozess als auch das Protokoll werden in der prototypischen Implementierung in Kapitel 5 mittels Java implementiert und stehen als Tool in Form eines Java-Archives (JAR) [125] zur Verfügung. Die Entscheidungsunterstützung ist mit dem Computeralgebrasystem Maple [14] implementiert und wird auch in der Evaluation dieser Arbeit genutzt.

Auch die Deployment-Methode setzt sich aus mehreren Teilen zusammen, deren Formalisierung, Implementierung und Bereitstellung im Folgenden erklärt werden. Sie besteht aus einer Beschreibungssprache, einem

Deploymentprozess, einem Deploymentprotokoll und einem Deploymentalgorithmus (siehe Abbildung 6).

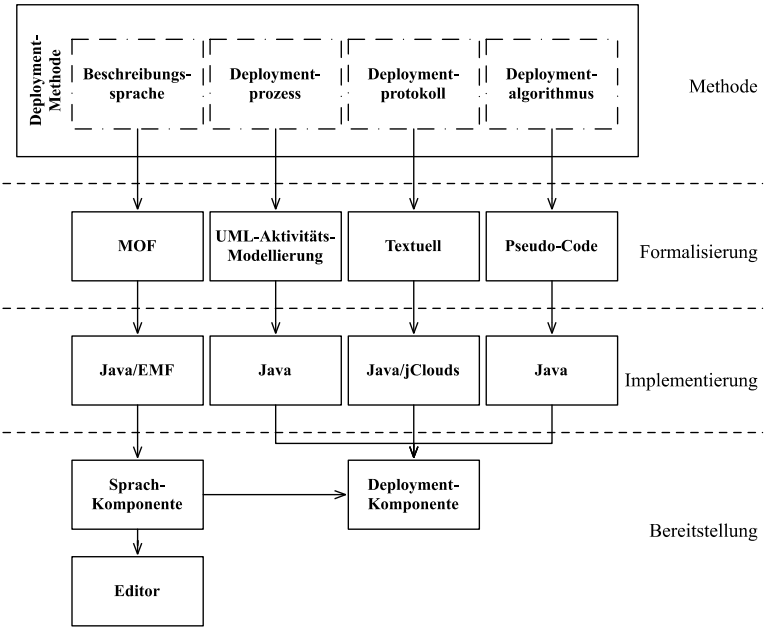

Abbildung 6: Übersicht über die Deployment-Methode und deren Umsetzung in dieser Arbeit

Die Beschreibungssprache ist mit Meta Object Facility (MOF) [106] formalisiert und wird mit dem Eclipse Modelling Framework (EMF) [121] in Java umgesetzt. Aus der Beschreibungssprache wird eine Sprachkomponente kompiliert, die als Bibliothek anderen Komponenten zur Verfügung stehen kann. Der Deploymentprozess wird ähnlich wie der Wiederherstellungsprozess mittels UML-Aktivitäts-Modellierung formalisiert und das Deploymentprotokoll mittels textueller Beschreibung. Um das Protokoll in Java zu implementieren, wird die externe Bibliothek jClouds [2] genutzt,

die die Kommunikation mit Cloud-Komponenten wie dem Anbieter oder einzelnen Servern erleichtert. Zusammen mit dem in Java implementierten Prozess wird das Protokoll als Deployment-Komponente bereitgestellt. Die Deployment-Komponente nutzt dabei die Sprachkomponente als externe Bibliothek. Ebenso nutzt der Editor, der als Eclipse-Plugin [121] bereitgestellt wird und mit dem sich formale Beschreibungen des Notfallsystems anfertigen lassen, die Sprach-Komponente.

Um die Notfallwiederherstellungs-Komponente nutzen zu können, müssen zunächst valide formale Beschreibungen erzeugt werden. Hierzu nutzt der *Modellierer* den Editor. Die formale Beschreibung wird dann vom *System Administrator* der Notfallwiederherstellungskomponente übergeben, welche die Deployment-Komponente zur Verwaltung des Notfallsystems nutzt (siehe Abbildung 7).

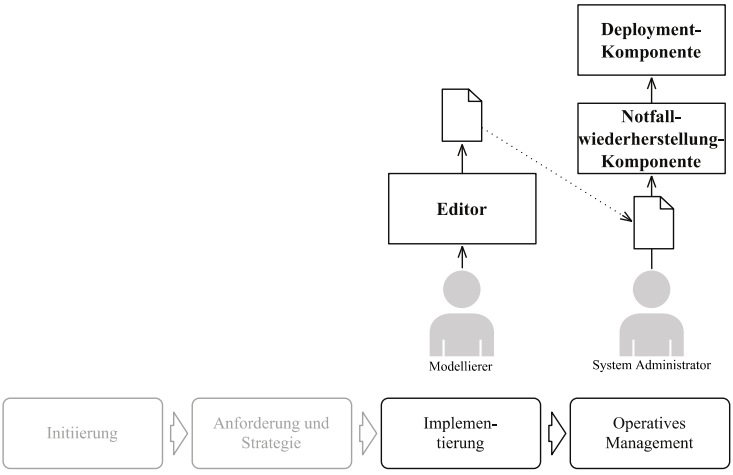

Abbildung 7: Erstellung und Nutzung der Deploymentbeschreibung

Die Beiträge der Arbeit und die Artefakte, die durch die prototypische Implementierung entstehen, unterstützen den Notfallmanagement-Prozess aus Abbildung 1. Im Implementierungsschritt wird der Editor als Werkzeug zur Spezifikation von Notfallsystemen zur Verfügung gestellt und im Operativen Management wird die Aktualisierung des Notfallsystems und die Überwachung des Primärsystems durch die Notfallwiederherstellungskomponente, die wiederrum die Deployment-Komponente nutzt, automatisiert.

2. Grundlegende Konzepte

In diesem Kapitel werden die grundlegenden Konzepte für diese Arbeit vorgestellt. Sie setzen sich zusammen aus den Grundlagen zum Notfallmanagement, zur Datensicherung, zu verteilten Systemen, zum Cloud Computing und zur Metamodellierung.

Der Zusammenhang zwischen den Grundlagen ergibt sich aus dem Ziel der Arbeit: Es soll eine neue Methode entwickelt werden, die sich in das Notfallmanagement eines Unternehmens eingliedert. Der Fokus wird dabei auf die Absicherung von verteilten Systemen gelegt. Um die Kosten gering zu halten, wird für die Vorhaltung des Notfallsystems Cloud Computing genutzt. Die Deployment-Methode nutzt Metamodellierung zur anbieterunabhängigen Beschreibung des verteilten Systems.

2.1 Notfallmanagement

Das Notfallmanagement ist nach BSI 100-4 *„ein Managementprozess mit dem Ziel, gravierende Risiken für eine Institution, die das Überleben gefährden, frühzeitig zu erkennen und Maßnahmen dagegen zu etablieren.“* [20]. Das Notfallmanagement ist somit nicht nur eine Maßnahme einer einzelnen Abteilung, wie der IT, sondern ein übergreifender Prozess des Managements und damit des gesamten Unternehmens, dem sich alle anderen Prozesse und Bereiche unterzuordnen haben [70]. Damit soll sichergestellt werden, dass das Unternehmen funktionsfähig bleibt. Hierzu sind auch geeignete Präventivmaßnahmen zu treffen, die zur Erhöhung der Robustheit und Ausfallsicherheit der Geschäftsprozesse beitragen [20], [51]. Damit umfasst das Notfallmanagement *„das geplante und organisierte Vorgehen, um die Widerstandsfähigkeit der (zeit-)kritischen Geschäftsprozesse einer Institution nachhaltig zu steigern, auf Schadensereignisse an-*

gemessen reagieren und die Geschäftstätigkeiten so schnell wie möglich wieder aufnehmen zu können" [20]. Maßnahmen, die beschreiben, wie bei einem Notfall die (eingeschränkte) Geschäftsfähigkeit wiederhergestellt werden kann, werden Notfallpläne oder auch Disaster Recovery Plan (DRP) genannt. Diese Dokumente beschreiben im IT-Umfeld zum Beispiel das Vorgehen, ein zerstörtes Rechenzentrum an einem anderen Ort wieder aufzubauen, wie die Daten zurückzuspielen sind und so weiter [50].

2.1.1 Ausfallklassen

Gerade die Schwere eines Ausfalls kann sich im Notfallmanagement stark unterscheiden: Kleine Ausfälle lassen sich im Tagesgeschäft beheben, große Ausfälle, die z.B. durch Katastrophen ausgelöst werden, müssen anders behandelt werden. Um die Schwere der Ausfälle zu kategorisieren, wird in dieser Arbeit auf das Modell nach BSI 100-4 zurückgegriffen [20]:

Störung – Eine Störung oder auch Problem ist eine alltägliche Fehlersituation, die die laufenden Prozesse und Ressourcen beeinträchtigt und deren entstandener Schaden gering ist [53]. Ein geringer Schaden ist ein Schaden, der im Verhältnis zum Gesamtjahresergebnis eines Unternehmens zu vernachlässigen ist oder die Aufgabenerfüllung nur unwesentlich beeinträchtigt [20].

Notfall – Ein Notfall ist ein Schadensereignis, bei dem die Prozesse oder Ressourcen der Unternehmung beeinträchtigt sind und die Verfügbarkeit nicht innerhalb einer geforderten Zeit wiederhergestellt werden kann [70]. Es entstehen hohe bis sehr hohe Schäden, die sich signifikant und in nicht akzeptablem Rahmen auf die Unternehmung auswirken [20]. Notfälle müssen über eine eigene Notfallbewältigung behandelt werden und können nicht im Tagesgeschäft gelöst werden [51], [132].

Krise – Unter einer Krise wird im Rahmen des Notfallmanagements ein Ereignis verstanden, das im Gegensatz zu einem Notfall trotz

vorbeugender Maßnahmen nicht durch die normale Aufbau- und Ablauforganisation bewältigt werden kann [20]. Bei einer Krise gibt es ein eigens dafür eingerichtetes Krisenmanagement (als Teil des Notfallmanagements), das jetzt aktiv wird. Notfälle können sich zu Krisen ausweiten, wenn nicht angemessen darauf reagiert wird und so Menschenleben oder die Existenz der Unternehmung in Gefahr gerät. Die Krise kann gewöhnlich innerhalb des Unternehmens behoben werden.

Katastrophe – Eine Katastrophe ist ein „Großschadensereignis“ [20], das weder zeitlich noch örtlich zu begrenzen ist. Neben der Existenz des Unternehmens sind Leben und die öffentliche Ordnung in Gefahr. Eine Katastrophe kann nicht vom Unternehmen allein, sondern muss in Zusammenarbeit mit externen Hilfsorganisationen bewältigt werden.

Dieser Zusammenhang ist nochmals in Abbildung 8 illustriert.

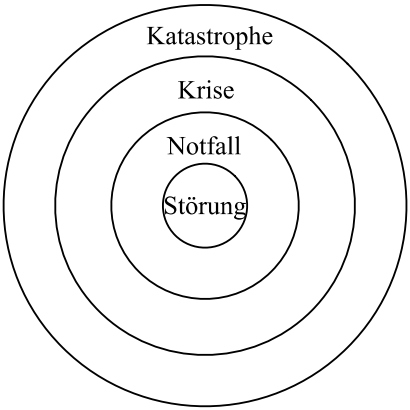

Abbildung 8: Ausfallklassen nach BSI 100-4

Andere Klassifikationen fassen unter Notfall auch Krisen und Katastrophen [70]. Um eine Störung des Systems zu vermeiden, gibt es verschiedene

Ansätze, die sich entweder allgemein auf die Vermeidung konzentrieren oder konkret zum Überbrücken eines der gerade beschriebenen Ereignisse geeignet sind.

Bei der Absicherung von IT-Systemen im Notfallmanagement wird häufig der folgende Prozess durchlaufen [16]: Initiierung, Anforderung und Strategie, Implementierung und Operatives Management (siehe Abbildung 9).

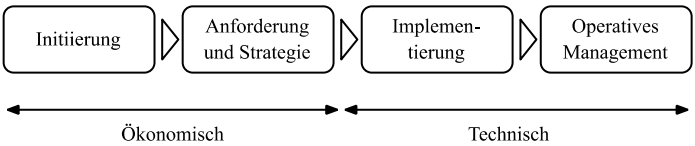

Abbildung 9: Notfallmanagement Prozess in der IT [16]

2.1.2 Notfallwiederherstellung

Soll die Verfügbarkeit eines Systems erhöht werden, so muss sichergestellt werden, dass keine Komponente des darunterliegenden Systems ausfällt. Im Bereich der fehlertoleranten Systeme ist das Vermeiden des „Single Point of Failure“ (SPOF) [4], [89], also von Systembestandteilen, die durch einen Ausfall das ganze System bedrohen, eine bewährte Methode [47], [119]. Zur Vermeidung dieser SPOFs, werden Komponenten oder Daten redundant, also an mehrfach (ggf. auch an verschiedenen Orten) vorgehalten [123]. Dabei kann es sinnvoll sein, auf homogene oder heterogene Redundanz zu setzen. Bei homogener Redundanz werden zusätzlich exakt gleiche Komponenten vorgehalten und bei heterogener Redundanz Komponenten gleicher Funktion, aber anderer Umsetzung (z.B. durch einen anderen Hersteller) vorgehalten. So kann beispielsweise sichergestellt werden, dass bei einem Fehler, der alle Komponenten des Herstellers betrifft,

der Schaden vom System abgewendet wird. Die Implementierung von heterogener Redundanz ist aber aufgrund des zusätzlichen Aufwands teurer.

Fällt eine redundante Komponente aus, so erfolgt dies transparent für den Nutzer. Für den Zeitraum des Ausfalls und bis zur vollständigen Wiederherstellung steigt zwar die Fehleranfälligkeit des Systems, aber es tritt kein Fehler beim Client auf; der Ausfall wird maskiert [123]. Versagt die Redundanz oder tritt ein nicht behandelter Fehler ein, der zu einer Störung führt, so wird diese im Rahmen des Störungsmanagements behoben [23].

Ist ein System komplett ausgefallen und kann mit seiner zeitnahen Verfügbarkeit nicht gerechnet werden, gewinnt die Begrenzung des Schadens an Priorität. Vom Notfallmanagement wird über die Einstufung des Notfalls entschieden [70]. Diese Entscheidung kann entweder ad hoc bei Auftreten eines Ausfalls oder nach bereits im Vorhinein klar festgelegten Kriterien erfolgen. Ist der Notfall festgestellt, so gilt es den Ausfall im Rahmen der Notfallwiederherstellung (Disaster Recovery) mit einem Notfallsystem zu überbrücken.

Gemäß der im Rahmen des Notfallmanagements durchzuführenden Auswirkungsanalyse oder auch Business Impact Analysis (BIA) [51], [132] werden den Prozessen zulässige Ausfallzeiten und Fenster, in denen ein Datenverlust toleriert wird, zugewiesen. Metriken für die Ausfallzeit und den Datenverlust sind das RTO und RPO:

RTO – Das Recovery Time Objective (RTO) bezeichnet die erlaubte Zeit, die der Geschäftsprozess maximal unterbrochen sein darf [67].

RPO – Das Recovery Point Objective (RPO) definiert in klassischen industriellen Verfügbarkeitsmodellen die maximal erlaubte Menge von produzierten Einheiten, die durch den Ausfall verloren gehen darf [51]. Für IT-Prozesse, bei denen Daten die produzierten Einheiten darstellen, ist also der RPO die maximale Zeit zwischen zwei Datensicherungen.

Fällt das System aus, so muss es innerhalb der Zeit, die als RTO spezifiziert ist, wiederhergestellt sein und sich aus Nutzersicht wieder im Normalzustand befinden. Die Festlegung des RTO und RPO sind Teil des Notfallmanagementprozesses (siehe Abbildung 10).

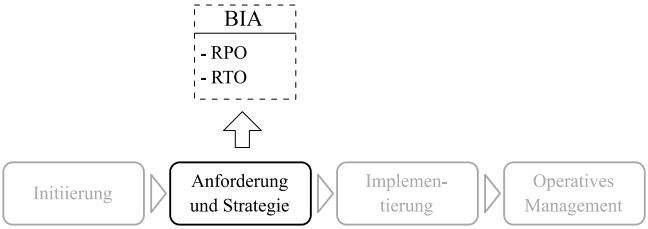

Abbildung 10: BIA als Teil der Anforderungsanalyse

Die Daten, die sich dann auf dem wiederhergestellten (Notfall-) System befinden, entsprechen den Daten seit dem letzten Backup oder dem Wiederherstellungspunkt. Dieser Zusammenhang ist in Abbildung 11 nochmals illustriert.

Ist das RTO bestimmt, dann bestehen die nächsten Schritte in der Identifikation und Realisierung derjenigen Maßnahmen, mittels derer das RTO erreicht werden kann. Ist das RTO groß genug, so bedarf es keiner weiteren Planung, da im Notfall das System unvorbereitet neu aufgesetzt werden kann. Soll jedoch ein kleineres RTO erreicht werden, so bedarf es spezieller Maßnahmen, zu denen die Vorbereitung einer Notfallumgebung, in der das System als Notfallsystem übergangsweise wieder in Betrieb genommen werden kann, gehört. Es werden also Teile des Systems oder der Laufzeitumgebung des Systems redundant als Notfallumgebung vorgehalten, für die zusätzliche Kosten anfallen.

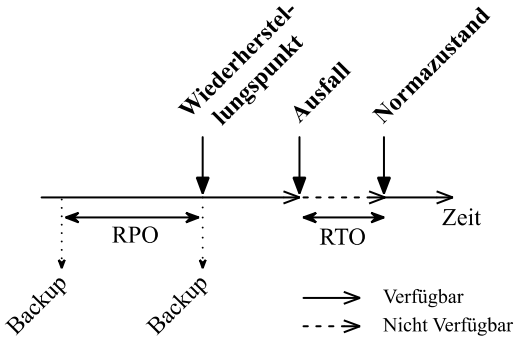

Abbildung 11: RPO und RTO im IT-Kontext (vgl. [86])

2.1.3 Standby Redundanz

Wird im Rahmen des Notfallmanagements ein Notfallsystem vorgehalten, so wird damit automatisch technische Redundanz hergestellt. Unter technischer Redundanz versteht man, dass eine Ressource mehrfach vorgehalten wird, sodass beim Ausfall eine andere den Betrieb übernehmen kann [62], [123]. Es kann also während des Betriebs zwischen den Ressourcen umgestaltet werden [41]. Grundsätzlich werden zumindest drei verschiedene Kategorien von Notfallumgebungen für redundante Ressourcen, wie ein Notfallsystem, unterschieden: Cold-, Warm- und Hot-Standby. Diese lassen sich vor allem anhand der zur Übernahme des Notfallbetriebs benötigten Zeit sowie anhand der entstehenden Kosten charakterisieren [21]:

Cold-Standby – Als Cold-Standby wird die einfachste Art der Notfallumgebung bezeichnet. Es werden klassischerweise lediglich die Umgebung (Räumlichkeiten, Rackplatz im Rechenzentrum, etc.), in denen später das Notfallsystem betrieben wird, vorgehalten [21], [132]. Im Falle einer Katastrophe müssen alle Ressourcen zur Überbrückung des

Notfalls neu angeschafft werden. Es werden lediglich Pläne für die Beschaffung und die Inbetriebnahme im Notfall vorgehalten, jedoch keine Hardware. Damit ist Cold-Standby zwar die günstigste Alternative, jedoch auch die mit der höchsten Wiederherstellungszeit. Die Overheadkosten für die Vorhaltung des Cold-Standby Systems sind: $c_{OH}^{Cold} = 0$

Warm-Standby – Notfallumgebungen dieser Klasse haben alle Eigenschaften von Cold-Standby Umgebungen, es werden jedoch zusätzlich noch die Ressourcen, auf denen später das Notfallsystem laufen soll, vorgehalten [21]. So müssen diese im Notfall lediglich in Betrieb genommen werden und stehen damit wesentlich schneller zur Verfügung [132]. Die laufenden Kosten für Warm-Standby sind die Abschreibungen auf die Anschaffungskosten (c_{Capex}) des Notfallsystems: $c_{OH}^{Warm} = c_{Capex}$

Da das Primärsystem und das Notfallsystem die gleichen Eigenschaften haben, werden die Abschreibungskosten für das Notfallsystem ungefähr denen des Primärsystems entsprechen.

Hot-Standby – Hot-Standby Umgebungen sind wie Warm-Standby Umgebungen bereits mit Hardware ausgestattet, jedoch müssen die Server nicht erst in Betrieb genommen werden [132]. Sie sind kontinuierlich in Betrieb erzeugen damit auch kontinuierlich Kosten. Ereignet sich ein Notfall, so kann direkt zwischen der Primär- und der Notfallumgebung umgeschaltet werden, sodass nahezu keine Unterbrechung der Verfügbarkeit entsteht [21]. Diese erhöhte Verfügbarkeit geht jedoch mit im Vergleich zum Warm-Standby nochmals erhöhten Kosten einher. Diese belaufen sich auf die Abschreibungen auf die Anschaffungskosten (c_{Capex}) und die Betriebskosten (c_{Opex}): $c_{OH}^{Hot} = c_{Capex} + c_{Opex}$. Ähnlich wie beim Warm-Standby sind die Kosten mit denen der Primärumgebung vergleichbar, wobei zu beachten ist, dass durch die fehlende Last des Systems die Opex-Kosten geringer ausfallen können. Grob kann man jedoch trotzdem von doppelten Kosten ausgehen.

In allen Fällen kommen zusätzlich zu den Kosten für die Bereithaltung des Notfallsystems die Kosten für den Transfer der Datensicherung in die Notfallumgebung und das Notfallsystem hinzu. Auch hier können verschiedene Strategien verfolgt werden. Während es bei einem Hot-Standby System sinnvoll ist, immer einen aktuellen Datenbestand verfügbar zu haben, so muss bei einem Warm-Standby System abgewogen werden, wie oft die Daten in der Notfallumgebung direkt zugreifbar gemacht werden.

Die Auswahl der passenden Notfallumgebung findet in der Strategie-Definition mit den Ergebnissen der BIA statt (siehe Abbildung 12).

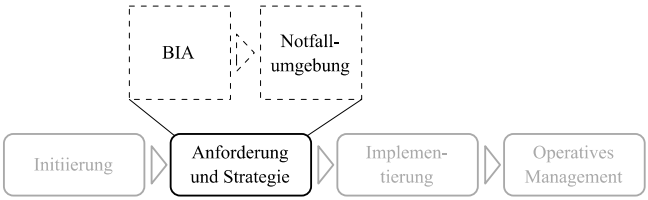

Abbildung 12: Auswahl der Notfallumgebung im Notfallmanagement-Prozess

2.1.4 Ausfallkosten

Eine Unternehmung muss also ihre kritischen Geschäftsprozesse sowie alle damit zusammenhängenden Systeme gegen Ausfälle absichern. Häufig ist jedoch gar nicht bekannt, inwieweit der Unternehmenserfolg von einzelnen Systemen und deren Verfügbarkeit abhängt. Ausfälle wirken sich dabei in verschiedenen Dimensionen auf den Erfolg eines Unternehmens aus [51]. Die Konsequenzen von Ausfällen sind in Abbildung 13 dargestellt.

	Bekannt	Geschätzt
Gewinn	Entgangener Gewinn	Verlorene Arbeitszeit
Kosten	Direkte Kosten	Zusätzliche Arbeitszeit

Abbildung 13: Konsequenzen von Ausfällen [119]

Direkte Kosten – Direkte Kosten entstehen, wenn bei einem Ausfall Anlagegüter zerstört werden und neu angeschafft werden müssen [1]. Zu dieser Neuanschaffung kommen noch die Kosten für externe Dienstleister, Lieferanten, etc. hinzu. Gerade bei kurzfristigen Anschaffungen können die Anschaffungskosten und Installationskosten höher liegen als bei einer geplanten Anschaffung. Pönale, die z.B. durch das verspätete Erbringen einer Leistung entstehen, zählen auch zu der Kategorie der direkten Kosten. [119]

Zusätzliche Arbeitszeit – Zusätzliche Arbeitszeit führt zu zusätzlichen indirekten Kosten, die dadurch entstehen, dass die IT-Mitarbeiter ihre Arbeitszeit nicht den üblichen Aufgaben, wie der Weiterentwicklung der IT Dienste, widmen können, sondern sich stattdessen um die Fehlerbehandlung kümmern müssen [87], [119]. Da die Kosten für die IT in der Regel von der gesamten Unternehmung getragen werden müssen, entstehen sie auch indirekt in jeder Abteilung. Zusätzlich kann jedoch auch in anderen Abteilungen zusätzliche Arbeit anfallen (so muss z.B. das Lager zusätzliche Arbeitsstunden einplanen, wenn das Lagerhaltungssystem ausgefallen ist) [119].

Verlorene Arbeitszeit – Die verlorene Arbeitszeit ist eine indirekte Kennzahl für den entgangenen Gewinn. Wenn z.B. ein ganzer Bereich mit mehreren hundert Mitarbeitern für viele Stunden oder Tage nicht

ihrer Arbeit nachgehen kann, dann weist dies auch auf den Gewinnverlust der Unternehmung hin [51], [119].

Entgangener Gewinn – Der entgangene Gewinn kann auch direkt einem Ausfall zugeordnet, aber nur schwer gemessen werden. Wenn z.B. durch einen Ausfall keine Verkäufe mehr getätigt werden, so ist es wahrscheinlich, dass die Kunden zu einem Konkurrenten abwandern [130]. Häufen sich die Ausfälle, so schlägt sich das zusätzlich auf den Ruf und das Vertrauen in die Unternehmung bei den Kunden nieder, sodass dies langfristige negative Auswirkungen haben kann [119].

Hierbei ist anzumerken, dass es einfacher ist, entstandene Kosten als entgangenen Gewinn zu bestimmen oder zu schätzen [119]. So kann insbesondere die verlorene Arbeitszeit nicht mit entgangenem Gewinn gleichgesetzt werden. Selbst wenn ein System gerade nicht verfügbar ist, heißt das nicht, dass die Unternehmung bzw. alle Angestellten still stehen. Oftmals wenden sie sich anderen Arbeiten zu und der Schaden durch die verlorene Arbeitszeit ist in der Regel geringer als die Summe der Ausfallstunden [119].

Die Berechnung der Ausfallkosten eines Geschäftsprozesses ist eine komplexe Aufgabe, die im Rahmen der Risiko-Abschätzung und der BIA durchgeführt wird. Im Rahmen dieser Arbeit ist die genaue Bestimmung der Ausfallkosten nicht von Relevanz. Diese Verfahren sind weitläufig bekannt und es ist lediglich die Tatsache von Interesse, dass diese durch ökonomische Verfahren bestimmt werden können [51], [132]. Die Ausfallkosten sind besonders in der Langzeit-Betrachtung in Kapitel 4.5.2 und 7.3 von Interesse, da hier davon ausgegangen wird, dass die Ausfallkosten für einen Geschäftsprozess und dem daran beteiligten Systemen bekannt sind.

2.1.5 Verfügbarkeit

Die Zuverlässigkeit oder auch Überlebenswahrscheinlichkeit $R(t)$ ist die Wahrscheinlichkeit, dass der zu beobachtende Prozess oder das System in einem Intervall $[0, t]$ alle zugesicherten Eigenschaften einhält [22]. Anders gesagt ist $R(t) = P(T \geq t)$ mit T als Zufallsgröße, die die Zeit bis zum ersten Fehler repräsentiert.

Die mittlere Lebensdauer oder Mean Time to Failure (MTTF) ist der Erwartungswert $E(T)$ der zuvor definierten Zufallsvariable T beziehungsweise die durchschnittliche Zeit bis zum Ausfall. Die mittlere Ausfalldauer oder auch Mean Time to Repair (MTTR) gibt die durchschnittliche Länge des Zeitintervalls zwischen Ausbleiben einer Eigenschaft und dem Wiedervorhandensein aller Eigenschaften [22] an. Sie beschreibt den Mittelwert der Zeit, die es dauert, die Reparatur durchzuführen oder den Notbetrieb zu starten. Während der Reparatur ist das System nicht verfügbar (downtime). Ist die Reparatur beendet, so ist das System wieder verfügbar (uptime). Die MTBF beschreibt das Zeitintervall das zwischen zwei Fehlern vergeht. Im Gegensatz zur MTTF wird also die Ausfalldauer mit dazu gerechnet (siehe Abbildung 14).

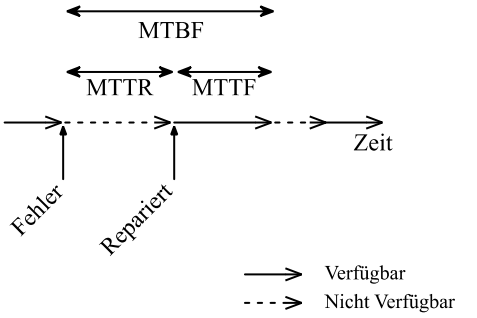

Abbildung 14: Übersicht über MTBF, MTTR und MTTF

Zur Berechnung der MTTR ist es unerheblich, ob die downtime durch eine geplante oder ungeplante Unterbrechung des Prozesses zustande kam. So könnte es sein, dass ein System zur Wartung ausgeschaltet werden muss und sich damit die MTTR erhöht. Gerade die geplanten downtimes können jedoch so gelegt werden, dass sie einen sehr geringen bis gar keinen Einfluss auf den Geschäftsbetrieb haben. Bei der Berechnung der Verfügbarkeit kann es daher angezeigt sein, die MTTR differenziert zu betrachten.
Wie bereits in Abbildung 14 erkennbar, beschreibt die Verfügbarkeit den Zustand des Systems durch zwei diskrete Werte [22]:

$$X(t) = \begin{cases} 1, & das\ System\ zeigt\ alle\ zugesicherten\ Eigenschaften \\ 0, & sonst \end{cases}$$

Nach [10], [11], [22] gibt es drei verschiedene Arten der Verfügbarkeit:

Die **Augenblickliche Verfügbarkeit** ist definiert als $A(t) = P(X(t) = 1)$. Es ist also die Wahrscheinlichkeit, dass zu einem Zeitpunkt t das System wie zugesichert funktioniert. In der Praxis ist diese Bestimmung der Verfügbarkeit jedoch nicht praktikabel, da es schwer ist, einen genauen Wert zu einem Zeitpunkt t zu ermitteln, und nach längerer Zeit strebt die Wahrscheinlichkeit immer auf einen festen, zeitunabhängigen Wert. [22]

Dieser Wert wird als **Stationäre Verfügbarkeit** bezeichnet. Sie ist definiert als der Grenzwert der augenblicklichen Verfügbarkeit nach t: $A_S = \lim_{t\to\infty} A(t)$. [22]

Die geläufigste Definition der Verfügbarkeit, die auch in dieser Arbeit Anwendung findet, ist die **stationäre mittlere Verfügbarkeit** A_∞. Sie ist definiert als: $A_\infty = \lim_{t\to\infty} \overline{A(t)}$ [22] und beschreibt die mittlere Verfügbarkeit über einem unendlichen Zeitintervall. In dieser Arbeit ist die Verfügbarkeit A also wie folgt definiert: $A := A_\infty$. Es wird also

immer von der stationären mittlernen Verfügbarkeit ausgegangen, wenn von Verfügbarkeit gesprochen wird.

Sind MTTF und MTTR bekannt, so kann die (stationäre mittlere) Verfügbarkeit auch über folgende Formel ausgedrückt werden [22], [119]:

$$A = \frac{MTTF}{MTTF + MTTR} = \frac{MTTF}{MTBF}$$

Die Zeiten, die in die MTTF- und MTTR-Berechnung einfließen, beinhalten in der Regel sowohl geplante als auch nicht geplante Nichtverfügbarkeiten. Wird ein System durchgehend benötigt und soll die Gesamtverfügbarkeit dieses Systems gemessen werden, so ist diese Definition ebenso sinnvoll. Je nach System kann es jedoch auch nützlich sein, keine dauerhafte (24/7) Verfügbarkeit, sondern lediglich eine Verfügbarkeit zu den Geschäftszeiten zu fordern [119]. In diesem Fall ist darauf zu achten, dass die MTTF und MTTR entsprechend bestimmt werden müssen.

Die Verfügbarkeit kann sowohl in Prozent als auch in Zeiteinheiten (Sekunden, Minuten, etc.) in einem bestimmten Zeitintervall angegeben werden. Insbesondere die prozentualen Angaben sind dabei trügerisch: Während eine Verfügbarkeit von 99% pro Jahr einen durchgehenden Ausfall von 3 Tagen erlaubt, so darf bei 99% Verfügbarkeit pro Woche eine maximale Ausfallzeit von 1,68 Stunden pro Ausfall nicht überschritten werden. Während im ersten Fall ein Ausfall von 5 Stunden am Stück nicht problematisch ist, stellt dieser im zweiten Fall bereits eine Verletzung der Verträge dar und könnte Strafzahlungen nach sich ziehen.

Im Bereich der hochverfügbaren Systeme ist es außerdem üblich, die Verfügbarkeit über die Anzahl der Neunen eine Klassifikation vorzunehmen. So wird beispielsweise bei einer Verfügbarkeit von 99,99% von der Verfügbarkeitsklasse 4, bei 99,999% von der Verfügbarkeitsklasse 5 gesprochen [48]. Der Aufwand bzw. die Kosten, um von einer Klasse in die

andere zu gelangen, sind dabei in der Regel auch nicht linear. So ist es viel aufwändiger und kostspieliger, von Klasse 3 in 4 als von Klasse 4 in 5 zu kommen [96].

2.2 Verteilte Systeme

Im Rahmen dieser Arbeit sollen verteilte Systeme im Rahmen des Notfallmanagements gegen Ausfälle abgesichert werden. Nach Tanenbaum et al. ist ein verteiltes System „*eine Ansammlung unabhängiger Computer, die den Benutzern wie ein einzelnes System erscheinen*“ [123] (siehe Abbildung 15).

Abbildung 15: Ein verteiltes System bestehend aus einem Clienten und einem Server, das von einem Benutzer genutzt wird

Dies bedeutet, dass ein verteiltes System aus verschiedenen Computern oder auch Komponenten besteht, die autonom sind. Über ihre Zusammensetzung und Art sind dem Nutzer des Systems keine Details bekannt. Es muss lediglich gewährleistet sein, dass sie gegenüber dem Nutzer als ein einziges System erscheinen. Dieses Prinzip heißt *Verteilungstransparenz* und ist neben *Zugriff auf Ressourcen*, *Offenheit* und *Skalierbarkeit* eines der Ziele verteilter Systeme [123]:

Verteilungstransparenz – In verteilten Systemen sind die Prozesse und Ressourcen über verschiedene Computer verteilt. Das System sollte jedoch in der Lage sein, dem Nutzer diese Tatsache zu verschleiern. Für den Nutzer oder die Anwendung wirkt das verteilte System wie eine Einheit [57], [123]. Diese Verteilungstransparenz wird gerade bei web-

basierten verteilten Systemen (siehe Kapitel 2.2.1) bewusst durchbrochen.

Zugriff auf Ressourcen – Nutzern und Anwendungen den Zugriff auf entfernte Ressourcen zu erlauben, ist das Hauptziel von verteilten Systemen. Ressourcen können dabei Daten, Peripheriegeräte wie Drucker, andere Computer, Netzwerke, etc. Durch die gemeinsame Nutzung von Ressourcen kann auch die Zusammenarbeit von Nutzern gefördert werden. [123]

Offenheit – Ein offenes verteiltes System bietet Dienste an, die standardisierte Schnittstellen und Protokolle nutzen [127]. Das heißt sowohl die Syntax als auch die Semantik der Dienste sind bekannt und hinreichend beschrieben. Dazu werden häufig Schnittstellendefinitionssprachen verwendet [123].

Skalierbarkeit – Ein verteiltes System sollte nach drei unterschiedlichen Dimensionen skalierbar sein [103]: Größe, Geographie und Administration. Es sollte demnach immer möglich sein, weitere Benutzer oder Ressourcen hinzuzufügen, es geographisch zu verteilen und von vielen unabhängigen Organisationen verwalten zu lassen. Fehlen eine oder mehrere dieser Dimensionen, so leidet bei der Vergrößerung des Systems häufig die Leistung [123].

Um ein verteiltes System zu betreiben, wird also eine Laufzeit benötigt, mit der sich die einzelnen Komponenten betreiben lassen, ein entfernter Zugriff auf die Ressourcen möglich ist, das über öffentliche Protokolle kommunizieren kann und Skalierung unterstützt wird.

Dem Netzwerk kommt bei verteilten Systemen eine besondere Rolle zu. Die Komponenten eines verteilten Systems können nur dann kommunizieren, wenn eine funktionierende Netzwerkverbindung besteht. Aus diesem Grund wird das Netzwerk in dieser Arbeit als eine Voraussetzung für verteilte Systeme angesehen. Es muss sichergestellt werden, dass zwischen kommunizierenden Komponenten immer eine Netzwerkverbindung besteht

und keine Firewall-Einstellungen oder ähnliches die Kommunikation unterbindet.

2.2.1 Webbasierte verteilte Systeme

Moderne Anwendungen, speziell im Cloud Computing, sind in der Regel webbasiert. Auch hier kommen verteilte Systeme zum Einsatz. Man unterscheidet zwischen herkömmlichen webbasierten Systemen und Webdiensten [123]:

Herkömmliche webbasierte Systeme – Im Web ist die Grundlage der Kommunikation, dass Dokumente ausgetauscht werden. Dies erfolgt in der Regel vom Client-Computer zum Server-Computer und wieder zurück. Da hier verschiedene Computer kommunizieren, kann man bereits von einem verteilten System ausgehen. Selten ist jedoch die Server-Seite so simpel aufgebaut. Hinter dem Server versteckt sich häufig ein weiteres verteiltes System mit einer Multi-Tier-Architektur. Das heißt, es gibt verschiedene Komponenten, die je nur eine Aufgabe erfüllen (z.B. Daten speichern oder dynamische Inhalte erzeugen). Zusammen stellen die Komponenten das vom Clienten angefragte Dokument zusammen und senden es an diesen zurück. Der Vorteil von solchen Multi-Tier-Architekturen ist, dass sie leicht erweiterbar und skalierbar sind (siehe Abbildung 16).

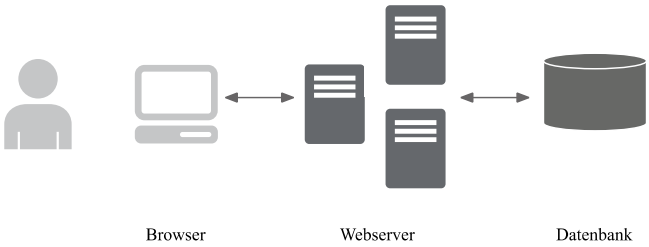

Abbildung 16: Webbasiertes verteiltes System bestehend aus Browser, Webserver-Komponente und Datenbank-Komponente

Webdienste – Eine Sonderform der webbasierten Systeme sind die Webdienste. Auch wenn die Bereitstellung wie bei den herkömmlichen webbasierten Systemen vonstattengeht, so ist die primäre Aufgabe dieser Systeme unterliegende APIs in Form von Webservices zur Verfügung zu stellen. Die Clienten binden diese Webservices direkt in die eigenen Prozesse ein und die webbasierten Systeme können so einfacher in bestehende, lokale Anwendungen integriert werden.

2.2.2 Fehlerklassen verteilter Systeme

Wenn ein verteiltes System in einen wichtigen Geschäftsprozess eingebunden ist, so ist es wichtig, dass es fehlerfrei funktioniert. Um die Zuverlässigkeit eines verteilten Systems zu bestimmen, wird wie in Kapitel 2.1.5 beschrieben unter anderem die Dauer zwischen zwei Fehlern herangezogen. Da Fehler auf verschiedene Weisen klassifiziert werden können, setzt diese Arbeit auf das bestehende Fehlermodell nach Tanenbaum et al. [123] auf:

Absturzausfall (Crash Failure) tritt dann auf, wenn die zugesicherte Eigenschaft oder die geforderte Funktion dauerhaft nicht erfüllt wird. Bei diesem Fehler ist immer eine Instandsetzung/Reparatur notwendig.

Dienstausfall (Omission Failure) ist ein Fehler, bei dem der Server nicht auf die eingehenden Anfragen reagiert. Die Ursachen für diese Art des Ausfalls können vielfältig sein. Es könnte beispielsweise sein, dass der Server die Anfrage nicht erhalten hat oder auch, dass er lediglich keine Antworten sendet.

Zeitbedingter Ausfall (Timing Failure) ist ein Ausfall, der passiert, wenn der Server außerhalb des festgelegten Zeitintervalls antwortet, also ein „Timeout" stattfindet.

Antwortfehler (Response Failure) bedeutet, dass zwar eine Antwort stattgefunden hat, diese jedoch falsch ist.

Byzantinischer oder zufälliger Ausfall (Byzantine oder Arbitrary Failure) beschreiben Fehler, die dadurch charakterisiert sind, dass zu zufälligen Zeitpunkten zufällige Antworten gesendet werden.

Im Rahmen dieser Arbeit sind hauptsächlich Absturzausfälle von Bedeutung. Möchte man nun die Zuverlässigkeit des gesamten Systems bestimmen, so sind die Zeiten, in denen keiner der genannten Fehler vorgelegen hat, mit denen zu vergleichen, in denen sich das System in einem Fehlerzustand befunden hat.

2.2.3 Datenredundanz

Wie bereits in Kapitel 2.1.3 erläutert, wird bei der Bereithaltung eines Notfallsystems eine technische Redundanz hergestellt, und es kann während des Betriebs von der Primär- auf die Standby-Ressource umgeschaltet werden. Soll die Standby-Ressource den Betrieb übernehmen können, so müssen jedoch nicht nur die Ressource selbst, sondern auch die Daten redundant gehalten werden. Nur so kann bei Auftreten eines Notfalls der Produktivbetrieb vom Primärsystem auf das Notfallsystem umgeschaltet werden, ohne dabei das RPO (siehe Kapitel 2.1.2) zu verletzen. Ein Problem dabei ist jedoch, die redundanten Ressourcen konsistent zu halten [123]. Man

unterscheidet allgemein, ob Daten kontinuierlich aktuell gehalten, also repliziert werden oder lediglich für Notfälle gesichert werden:

Replikation – Die Hauptgründe für Replikation von Daten sind Verlässlichkeit und Systemleistung [123]. Die Verlässlichkeit wird dadurch erhöht, dass Daten redundant an verschiedenen Speicherorten vorgehalten werden und so bei kleineren Ausfällen z.B. von Festplatten, die Daten weiterhin in Form eines Replika vorliegen. Eine Erhöhung der Systemleistung kann dadurch erzielt werden, dass replizierte Daten parallel gelesen werden. Hierdurch können mögliche Zugriffsraten erhöht und dadurch die Systemleistung insgesamt verbessert werden.

Datensicherung – Eine besondere Form der Redundanz ist die Datensicherung. Hier werden die Daten in einem zentralen Speicher vorgehalten und können bei Bedarf zurückgespielt werden. Es gibt verschiedene Datensicherungsmethoden, die genutzt werden können, um die Daten zentral zu speichern und im Notfall durch eine Rücksicherung wieder verfügbar zu machen. In Folgenden wird das Thema Datensicherung ausführlich erläutert.

Da für diese Arbeit besonders die Herstellung von Redundanz mittels Datensicherung von Relevanz ist, wird im Folgenden dieses Thema genauer erläutert.

2.3 Datensicherung

Datensicherung oder auch Backups sind Abbilder (Snapshots) von Daten, die zu einem gewissen Zeitpunkt erstellt, in einem einheitlichen Format gespeichert und unabhängig von den Originaldaten verwaltet werden [100]. Gewöhnlich werden von einem Datum nicht nur ein Snapshot, sondern gleich mehrere vorgehalten. So ist es auch möglich, vergangene Daten, die beispielsweise versehentlich gelöscht oder überschrieben wurden, wiederherzustellen. Außerdem lässt sich mit dem Backup bei einem Notfall der

letzte Stand wiederherstellen. Hierzu ist darauf zu achten, dass sich die Snapshots in einem konsistenten Zustand [123] befinden, also keine offenen Transaktionen oder ähnliches in dem Snapshot existieren. Neben dem Zeitpunkt, an dem ein Backup durchgeführt wurde, existieren bezüglich der Art und des Ortes verschiedene Möglichkeiten, die im Folgenden näher beschrieben werden. Je nach Auswahl der Kombination aus Art, Häufigkeit und Ort entstehen auch unterschiedliche Kosten. So muss bei der Erstellung der Backups aus Unternehmenssicht immer abgewogen werden, wie sicher ein Backup sein soll und welche Kosten dadurch entstehen.

2.3.1 Sicherungsarten

Das große und etablierte Gebiet der Datensicherung hat mit der Zeit viele Verfahren und Alternativen für die Sicherung von Daten hervorgebracht. So ist das Ausführen von Backups auf verschiedenen Ebenen möglich. Während traditionell Backups auf Blockebene durchgeführt wurden, kommen heute viele weitere Modelle wie dateibasierte Backups, Dateisysteme mit integrierten Backups, usw. auf den Markt. Wenn es, wie in dieser Arbeit, jedoch darum geht, ein Backup an einem anderen Ort vorzuhalten, so sind die Vollsicherung, die differenzielle und inkrementelle Sicherung die gängigsten Alternativen:

Vollsicherung – Die Vollsicherung beschreibt die einfachste Art der Sicherung. Alle Informationen werden kopiert. Alle weiteren Sicherungsarten setzen immer zumindest eine Vollsicherung voraus [100]. Eine besondere Art der Vollsicherung stellt die *Speicherabbildsicherung* oder auch das Image dar. Hier werden nicht nur die Nutzdaten, sondern z.B. eine gesamte Festplatte mit Partitionstabelle und so weiter gesichert. Ein Speicherabbild kann im Umkehrschluss auch wie eine Festplatte genutzt werden und bietet sich vor allem für die Vollsicherung von Betriebssystemen an. Images werden auf Blockebene erstellt

und benötigen oftmals mehr Speicherplatz als die Summe der eigentlich zu sichernden (Geschäfts-) Daten, die mit ihnen gespeichert wird.

Differenzielle Sicherung – Jeder differentiellen Sicherung liegt eine Vollsicherung zu Grunde. Es werden lediglich die Daten gesichert, die sich seit der letzten Vollsicherung geändert haben. Der Vorteil des differenziellen Backups ist, dass bei einer Wiederherstellung nur die letzte Voll- und letzte differentielle Sicherung benötigt werden. Der Nachteil ist: Ändern sich viele der im Vergleich zum letzten Vollbackup neu hinzugekommenen Dateien, so kann es vorkommen, dass das letzte differenzielle Backup größer ist als das letzte Vollbackup und trotzdem nicht alle Daten enthält [100].

Inkrementelle Sicherung – Die inkrementelle Sicherung verfolgt eine ähnliche Strategie wie die differentielle Sicherung, jedoch werden immer nur die seit der letzten Sicherung (vollständig, differentiell oder inkrementell) geänderten Daten gespeichert [100]. Auf diese Weise können große Mengen Speicherplatz gespart werden, allerdings muss auch beim Wiederherstellen nacheinander auf mehrere Backups zurückgegriffen werden, da nur so auch alle Änderungen erfasst werden können.

Sowohl bei der differentiellen als auch bei der inkrementellen Sicherung ist es sinnvoll von Zeit zu Zeit eine Vollsicherung zu machen, so dass bei der differentiellen Sicherung die Datenmenge und bei der inkrementellen Sicherung die Zeit und der Aufwand für die Wiederherstellung begrenzt werden.

2.3.2 Sicherungshäufigkeit und Sicherungsort

Die Auswahl des Sicherungsintervalls und des Sicherungsortes beeinflusst, ob gewisse Daten nach einer Katastrophe wiederhergestellt werden können oder nicht. Wird z.B. die Datensicherung auf derselben Festplatte gesichert, so sind bereits bei einem Festplattenausfall alle Daten und Sicherungen

verloren. Auch kann ein zu groß gewähltes Sicherungsintervall dafür sorgen, dass bei einem Totalausfall nur ein sehr alter Bestand überhaupt wiederhergestellt werden kann. Werden die Daten an einem anderen Ort oder auf einem anderen Kontinent gespeichert, so sind sie von einer Katastrophe höchstwahrscheinlich nicht betroffen, müssen aber zur Wiederherstellung auch wieder zurück übertragen werden. Zusätzlich sind die Kosten für die Speicherung an einem anderen Ort und die Speicherung in kurzen Zeitabständen üblicherweise höher, da die Daten über längere Strecken transportiert werden oder mehr Speicherplatz benötigt wird.

Speziell die Entscheidung bezüglich des Sicherungsintervalls lässt sich aus dem RPO ableiten. Wie oben dargestellt, gibt das RPO vor, wie viele Daten nach einem Ausfall maximal verloren gehen dürfen. Das Backupintervall (die Zeit zwischen zwei begonnenen Backups) muss daher kleiner sein als das RPO. Gleichzeitig führen kürzere Backupintervalle aber zu höheren Kosten, die bei einem zu kurz gewählten Intervall nicht mehr durch den Zugewinn an Ausfallsicherheit gerechtfertigt sind.

2.4 Cloud Computing

Ziel dieser Arbeit ist es, ein verteiltes System kostengünstig in der Cloud abzusichern. Unter Cloud Computing [3], [80], [94] versteht man den lau großen Paradigmenwechsel in der Informatik. IT-Ressourcen werden nicht mehr gekauft, sondern nach Bedarf gemietet. Ein wichtiger Teil von Cloud Computing ist also die Fokussierung auf Dienste statt auf physikalische Produkte.

Eine exakte Definition von Cloud Computing ist nicht einfach, da viele verschiedene Aspekte bei dem Paradigma Cloud Computing eine Rolle spielen. Neben einer Vielzahl von wissenschaftlichen Arbeiten, die sich mit dem Thema der Cloud Definition beschäftigen [3], [4], [12], [80], hat auch die NIST den Versuch unternommen, Cloud Computing zu fassen und zu definieren [94]. Auch wenn die NIST-Definition keine harte formale Defi-

nition für das Cloud Computing liefert, ist sie doch eine in der wissenschaftlichen Gemeinschaft anerkannte Grundlage zur Klassifikation von Cloud Diensten. Sie unterscheidet dabei die *Wichtigen Eigenschaften*, *Servicemodelle* und den *Betrieb* von Cloud-Diensten.

2.4.1 Wichtige Eigenschaften

Cloud Dienste haben nach der Definition von NIST [94] immer gewisse Eigenschaften, die sie von traditionellen Diensten unterscheiden. Wichtige Eigenschaften, die jeder Cloud-Dienst haben sollte, sind:

Selbstbedienung – Ein Kunde kann die gewünschten Cloud Ressourcen ohne Interaktion mit dem Anbieter provisionieren und verwalten.

Breitbandzugang – Die Dienste des Anbieters sind über das Netzwerk verfügbar. Dabei ist es unerheblich, ob der Zugang über ein Intranet, also ein abgeschottetes, privates Netz erfolgt oder öffentlich über das Internet.

Bündelung von Ressourcen – Die Ressourcen des Anbieters sind gebündelt und werden unterschiedlichen Kunden zur Verfügung gestellt. So kann z.B. durch Virtualisierung die Ausnutzung von physikalischen Ressourcen erhöht und bessere Skaleneffekte erzielt werden. Ressourcen werden dynamisch zugewiesen, wieder entfernt und der Kunde hat in der Regel keine Kontrolle darüber, wo genau sein Dienst bereitgestellt wird.

Elastizität – Die dynamisch zugewiesenen Ressourcen können in kurzer Zeit bereitgestellt und auch wieder freigegeben werden. Dies ermöglicht es, gemäß dem aktuellen Bedarf schnell hoch und runter zu skalieren.

Bedarfsgerechte Nutzung – Cloud Dienste überwachen und optimieren die Ressourcennutzung automatisch, indem sie auf verschiedenen Ebenen Daten erheben. Diese Daten werden dem Kunden aggregiert zur Verfügung gestellt, so dass dieser immer die Übersicht hat, was gerade

genutzt wird. Weiterhin können diese Daten für eine bedarfsgerechte Abrechnung genutzt werden.

Sollte ein Dienst eine dieser Eigenschaften nicht erfüllen, so kann nicht direkt davon gesprochen werden, dass dies nun kein Cloud-Dienst ist. Eine Bewertung muss immer in der Gesamtsicht erfolgen, die oben genannten Kriterien sind jedoch Indizien, dass es sich um einen Cloud-Dienst handelt.

2.4.2 Servicemodelle

Soll ein System in der Cloud bereitgestellt werden, so stellt sich die Frage, wie genau dies umgesetzt werden soll. Betrachtet man die Cloud-Landschaft, so lässt sich feststellen, dass es eine große Anzahl von verschiedenen Anbietern und Angeboten gibt. Eine Anwendung kann auf verschiedenste Arten betrieben werden. Um die Cloud Landschaft etwas besser durchschauen zu können, wird der „Cloud Computing Stack“ genutzt, der als Vorarbeit für diese Arbeit entwickelt und publiziert wurde [77], [80]. Ziel des Stacks (siehe Abbildung 17) ist es, die Beziehungen verschiedenster Cloud-Angebote darzustellen. Durch die Einführung von Abstraktionsschichten wird es möglich, die einzelnen Angebote leichter zu vergleichen und die Zusammenhänge besser zu verstehen. Im Folgenden werden dazu die einzelnen Teile des Stacks genauer beschrieben. Der Stack nutzt als Grobgliederung die gleichen Ebenen, wie die NIST [94], detailliert diese jedoch noch weiter.

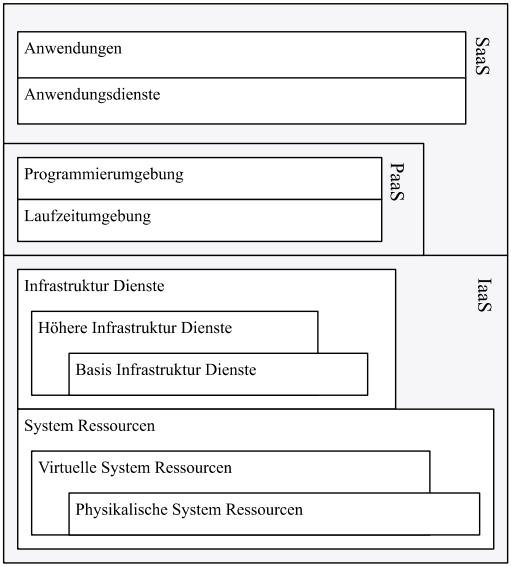

Abbildung 17: Cloud Computing Stack Architektur (vgl. [80])

Infrastructure as a Service (IaaS) – Ganz unten im Stack befindet sich die IaaS-Schicht. Hier wird wiederum auf unterster Ebene, innerhalb der *System Ressourcen*, zwischen zwei verschiedenen Arten von Angeboten unterschieden: *Virtuelle System Ressourcen* und *Physikalische System Ressourcen*. Beide Arten bieten eine grafische Bedienoberfläche und API, um die Ressourcen zu provisionieren und zu steuern. Mittels API können höherwertige Dienste genutzt werden, die ein automatisches Aufsetzen, ein Umschalten zwischen Ressourcen oder Herunterfahren erlauben. Auch wenn im Cloud Computing in der Regel die meisten Ressourcen mittels Hypervisoren, wie beispielsweise Xen [9], VMware [128] oder KVM [68] virtualisiert angeboten werden, gibt es einige Angebote, wie von Cisco [42], die Virtualisierung direkt in die

Hardware eingebaut haben und es erlauben, auch physikalische Hardware über eine API anzusteuern. Zu dieser Ebene zählen die Angebote von Amazons EC2 [6], Openstack [58], VMware vCloud [128] oder Rackspace [114]. Auf der nächsthöheren Ebene, den Infrastrukturdiensten, befinden sich höherwertige, auf System Ressourcen aufbauende Angebote. Einfache *Basis Infrastruktur Dienste* sind beispielsweise das verteilte Dateisystem HDFS [131], das für die verteilte Berechnung auf großen Datenbeständen genutzt werden kann, oder aber das Google Dateisystem [44]. Die *höheren Infrastrukturdienste* sind dann auf diesen Speicher und Rechendiensten aufbauende Services, wie z.B. das Hadoop [131], Redis [118] oder Google Bigtable [27].

Platform as a Service (PaaS) – Bewegt man sich im Stack weiter nach oben, so gelangt man zu PaaS. Der Begriff der Cloud Plattformen ist weit gefasst und wird von vielen Anbietern unterschiedlich interpretiert. In dieser Arbeit wird er jedoch so verstanden, dass ein echtes PaaS-Angebot auf IaaS aufsetzt. Nur so kann gewährleistet werden, dass die Plattform selbst der Anforderung der Elastizität genügt. Ein PaaS-Angebot setzt sich in der Regel aus einer Vielzahl von Diensten zusammen. Um auf diese Dienste zugreifen zu können, wird eine Programmierumgebung mit den passenden Bibliotheken und der dazu passenden Laufzeitumgebung bereitgestellt. Diese Laufzeitumgebung bildet den Kern von PaaS, alle anderen Dienste der Plattform werden über Programme, die in der Laufzeitumgebung laufen, genutzt. So kann der Anbieter auch sicherstellen, dass der Nutzer zwar Zugriff auf alle Dienste hat, sich aber auch an Einschränkungen hält. Einige Anbieter, wie die Google App-Engine [46] oder Salesforce [116], erlauben beispielsweise keine Nutzung von lokalem Speicher oder die Nutzung beliebiger Ports. Der PaaS-Anbieter kümmert sich in der Regel auch darum, dass den Programmen immer die passende Anzahl an Ressourcen

zur Verfügung steht. Anwendungen, die auf PaaS aufsetzen, sind damit meistens auch automatisch elastisch.

Software as a Service (SaaS) – *Anwendungen*, wie z.B. Microsoft Office 365 [52] oder Google Drive [45] können auf verschiedene Weisen realisiert werden. Sie können durch die Zusammenschaltung von unterschiedlichen *Anwendungsdiensten* erstellt werden, eigenständig sein oder Hybridformen annehmen. So ist beispielsweise die Amazon-Homepage ein Zusammenschluss von vielen einzelnen Services, die erst im Browser des Nutzers zusammenkommen. Auch ist die Laufzeitumgebung der Anwendung unterschiedlich. Eine Anwendung kann auf PaaS aufbauen und damit von Eigenschaften, wie der automatischen Elastizität profitieren, unterliegt dann jedoch auch den Einschränkungen des PaaS-Angebots. Sie kann aber auch direkt als verteilte Anwendung auf IaaS aufsetzen und damit alle Freiheiten haben, die beim Hosting im klassischen Rechenzentrum üblich sind. Die Skalierung, Absicherung, etc. der Anwendung liegen dann allerdings im Aufgabengebiet des Nutzers und stellen einen Mehraufwand dar. Häufig steht der Anwendungsentwickler jedoch nicht vor der Wahl, seinen Dienst mit PaaS zu realisieren, da die Einschränkungen vieler Plattformen sehr groß sind. Dann muss die Anwendung direkt auf IaaS aufsetzen und der Betreiber des Dienstes muss sich selbst um die Zusatzdienste wie Skalierung und Absicherung kümmern.

In dieser Arbeit stehen verteilte Systeme (siehe Kapitel 2.1.2) im Fokus, das heißt Systeme, die einen SaaS-Dienst bereitstellen und direkt auf IaaS aufsetzen. In Abbildung 18 ist dies noch einmal exemplarisch dargestellt.

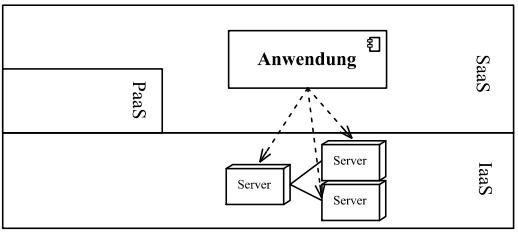

Abbildung 18: Exemplarische Darstellung eines verteilten Systems im Cloud Computing Stack

2.4.3 Betrieb

Nachdem die Eigenschaften und die Arten der Services beschrieben wurden, wird nun auf die Art der Bereitstellung eingegangen. Wie beim klassischen Hosting können Anwendungen entweder im eigenen Rechenzentrum betrieben oder ausgelagert werden. Bei dem Betrieb von Cloud Diensten unterscheidet die NIST vier verschiedene Modelle [94]:

Private Cloud – Die Cloud wird in dem Rechenzentrum einer Organisation betrieben und steht auch nur dieser Organisation, wohl aber z.B. verschiedenen Geschäftseinheiten, zur Verfügung.

Community Cloud – Diese gemeinschaftlich betriebene Cloud steht ähnlich wie die Private Cloud nur einer begrenzten Anzahl von Organisationen oder Individuen zur Verfügung. So könnten sich z.B. Organisationen wie Banken, die ähnliche Anforderungen haben, dafür entscheiden, gemeinsam eine Cloud zu betreiben.

Public Cloud – Die Public Cloud steht der Allgemeinheit offen. Sie wird von einer Organisation betrieben, aber diese beschränkt nicht den Zugang zu den Ressourcen.

Hybrid Cloud – Nicht immer kann ein Dienst einer der drei zuvor genannten Kategorien zugewiesen werden. Es gibt oft Mischformen, die

Hybrid Cloud genannt werden. So könnte ein Anbieter für kritische Berechnungen die Private Cloud nutzen, aber unkritische in die Public Cloud auslagen.

2.4.4 Zuverlässigkeit

Das Cloud Ökosystem wächst unaufhörlich, und waren es zu Anfang einige wenige Firmen, wie Google und Amazon, die das Cloud-Geschäft vorangetrieben haben, so gibt es heute mehr und mehr Anbieter.

Auch wenn sich die Anzahl der Anbieter auf dem Markt stetig wächst, sind es die Großen und diejenigen, die durch die Gunst der ersten Stunde einen großen Vorsprung erlangt haben, die ihre Marktmacht versuchen zu festigen. Anbieter wie Amazon, Google und Microsoft kämpfen darum, die gleiche marktbeherrschende Stellung einzunehmen, die Windows in den 90er Jahren innehatte [40]. Für die Kunden der Dienste stellen solche marktbeherrschenden Unternehmen ein Risiko dar. Durch das Defakto-Monopol können diese Anbieter Preise bestimmen und sich konsequent aus Standardisierungsbemühungen heraushalten. Hierdurch stärken diese Unternehmen ihre Marktmacht weiter, da ohne Standards Kunden zusätzlich über Lock-In-Effekte [3], [30], [73] gebunden werden können.

Neben den ökonomischen Risiken stellt die Fokussierung auf einen Anbieter auch das Risiko dar, dass hierdurch ein SPOF geschaffen wird [4]. So wurde im Jahr 2012 durch einen Konfigurationsfehler in der Rechenzentrumsinfrastruktur ein großer Teil des Amazon EC2 Rechenzentrums lahmgelegt (über verschiedene Verfügbarkeitsregionen hinweg) [88] und auch bei anderen Anbietern sind Ausfälle nicht ungewöhnlich. Es gibt jedoch nur wenige wissenschaftliche Untersuchungen, die differenziert betrachten, wann ein Cloud-Dienst nicht verfügbar ist. Seit 2007 sind in einer Studie von Li et al. jedoch auch 112 Ausfälle registriert worden (siehe Abbildung 19) [88]. Die Studie basiert dabei auf der nachträglichen

Analyse und zählt alle Ausfälle, unabhängig von ihrer Größe und dem Grad der Auswirkung auf den gesamten Cloud-Dienst.

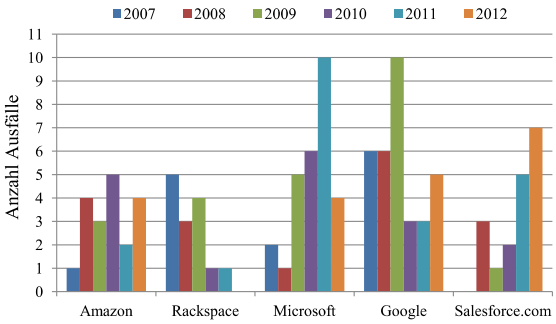

Abbildung 19: Ausfallverteilung über Anbieter und Jahre [88]

Unter diesen Einschränkungen gibt die Studie dennoch einen Hinweis darauf, dass auch als sicher geltende Anbieter wie Amazon, Google oder Microsoft mit Ausfällen zu kämpfen haben. Zusätzlich gibt es natürlich auch immer das Risiko, dass selbst ein erfolgreiches Unternehmen plötzlich seinen Betrieb einstellen muss, wie das z.B. bei Enron, einem der größten Unternehmen der USA, im Jahr 2001 geschehen ist [33].

Soll also ein System ausfallsicher betrieben werden, so sollte man sich möglichst nicht auf einen einzelnen Anbieter verlassen, sondern die Absicherung anbieterunabhängig, also bei einem anderen Anbieter als dem der das Primärsystem bereitet, vornehmen [4], [88]. Hierdurch verringert sich das Risiko, dass das System komplett ausfällt.

2.5 Metamodellierung

In der Informatik ist die exakte Definition eines Modells umstritten. So kann nach Mahr [90] jedes Objekt der realen Welt als Modell aufgefasst

werden und jedes Modell als reales Objekt. Je nach Kontext entscheidet sich, ob beispielsweise ein Stuhl ein Modell für die Produktion weiterer Stühle ist oder lediglich ein Stuhl ohne Modellcharakter [90]. Auch in der Informatik wird der Modellbegriff sehr umfassend benutzt, so existieren mathematische Modelle, beschreibende Modelle etc. Nach Mahr [90] haben diese Modelle jedoch alle gemein, dass sie etwas transportieren müssen und nur im Kontext des Einsatzzwecks als gut oder schlecht bewertet werden können. Soll ein Modell zur Ausführung von Anwendungen genutzt werden, dann ist zu beurteilen, ob es hierfür geeignet ist oder nicht. Es gibt daher auch eine ganze Reihe von Ansätzen, mit denen es möglich wird, reale Objekte formal zu beschreiben. Diese Ansätze und deren „Communities" stehen nicht selten in Konkurrenz zueinander. So wäre es beispielsweise möglich, eine Beschreibungssprache, wie sie in dieser Arbeit vorgestellt wird, mit *Meta Object Facility (MOF)* Metamodellierung [106] oder mit formalen Beschreibungssprachen, wie der *Backus-Nauer-Form (BNF)* [71] zu realisieren. Während BNFs die Grundlage vieler Programmiersprachen sind, versteht man unter Metamodellierung im Allgemeinen die Erstellung von Modellen, die die Struktur anderer Modelle beschreiben. Während die Metamodellierung an sich sehr allgemein genutzt werden kann, wird der Begriff im Bereich der Beschreibungssprachen meist synonym zur Modellierung von Modellierungssprachen anhand der Sprache MOF 2.0 verwendet [106]. MOF wurde von der Object Management Group (OMG)[7] entworfen und standardisiert, um damit über eine gemeinsame Basis für andere Modellierungssprachen, wie UML, WDSL, etc. zu verfügen. MOF ist somit eine Sprache zur Definition von Metamodellen, mit denen sich ihrerseits Modelle formulieren lassen. In Abbildung 20 sind die Hierarchien der MOF-Metamodellierung exemplarisch dargestellt. Im Beispiel

[7] http://www.omg.org

wird das Metamodell UML mit der Metamodellierungssprache MOF erstellt. Mit Hilfe des Metamodells kann wiederum ein UML Klassen- und Objektdiagramm für ein Element der realen Welt erstellt werden. Dabei sind sowohl das Klassen- als auch das Objektdiagramm ein Modell mit einer anderen Sicht auf das gleiche reale Objekt.

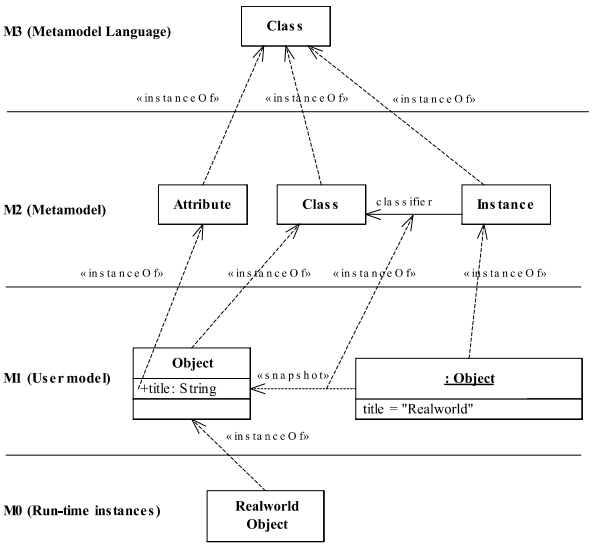

Abbildung 20: MOF Metamodellierungshierarchie [108]

Zur Definition von Metamodellen nach MOF stehen im Wesentlichen vier Elemente im Mittelpunkt: Typ, Assoziation, Komposition und Generalisierung. Auch wenn diese die grafische Repräsentation von UML nutzen, so ist deren Semantik verschieden.

Typ – Der Typ nutzt die Repräsentation der UML-Klasse. Er wird durch eine rechteckige Box dargestellt und kann Attribute mit eigener Typisierung haben (siehe Abbildung 21). Der Typ repräsentiert dabei abstrakt ein Gruppe von Objekten der realen Welt (z.B. Cloud Rechen-

zentrum oder virtuelle Maschine) und in einem darauf basierenden Modell können Elemente von einem Typ ein konkretes Element der realen Welt repräsentieren (z.B. AWS Cloud Rechenzentrum oder Kline AWS Virtuelle Maschine).

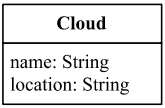

Abbildung 21: Beispiel für ein Metamodell Typ

Assoziation – Beziehungen zwischen zwei Typen wird durch die Assoziation beschrieben. Durch die in Abbildung 22 dargestellte Modellierung wird dem Cloud-Typ automatisch ein Attribut zugewiesen, dem bei Modellerstellung ein oder mehrere Elemente vom Typ Image zugewiesen werden müssen.

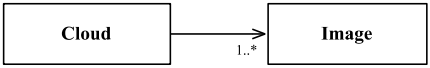

Abbildung 22: Beispiel für eine Metamodell Assoziation

Komposition – Die Semantik der Komposition unterscheidet sich leicht von der UML-Komposition. MOF-Kompositionen werden dazu genutzt verschiedene Bereiche darzustellen. Im Beispiel (Abbildung 23) würde es also bedeuten, dass die Cloud mehrere Images beinhaltet. Diese würden beispielsweise bei einer XML-Repräsentation als Kinder und bei einer Grafischen Darstellung durch das Zeichnen von mehreren Images in das Cloud-Element. Die Bindung zwischen den Typen ist also stärker als bei einer Assoziation, welche lediglich einen Verweis auf einen gleichwertigen, nicht einen untergeordneten Typ, darstellt.

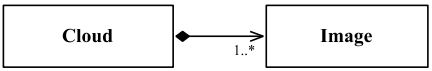

Abbildung 23: Beispiel für eine Metamodell Komposition

Generalisierung – Ähnlich wie bei UML kann ein Typ generalisiert werden. Im Beispiel in Abbildung 24 wird das Standard Image zu einem Image generalisiert. Der Image-Typ ist außerdem abstrakt. Von einem abstrakten Typ kann es im Modell kein Element geben, sondern nur von einem konkreten Typ (im Beispiel Standard Image). Bezieht sich jedoch eine Assoziation Komposition auf das Image so gilt dies auch für das Standard-Image, da dieses alle Beziehungen und Attribute von Image erbt.

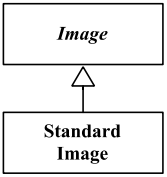

Abbildung 24: Beispiel für eine Metamodell Generalisierung

Prinzipiell kann die Metamodellierung zu sehr unterschiedlichen Zwecken eingesetzt werden. Die wichtigsten Funktionen eines Metamodells sind die Visualisierung von Modellen, die automatische Quellcodeerzeugung, die Definition eines Datenformats für den Datenaustausch, die Serialisierung und die Überprüfbarkeit.

2.5.1 Visualisierung

Elementen aus den Metamodellen kann eine grafische Repräsentation zugewiesen werden. Die grafische Darstellung der Modelle steht besonders bei sehr verbreiteten Metamodellfamilien, wie UML, klar im Mittelpunkt. Softwaresysteme werden in der Design-Phase als Klassendiagramme, Aktivitätsdiagramme etc. modelliert, um in der späteren Implementierung Fehler zu vermeiden und um sich über die Modelle austauschen zu können.

2.5.2 Überprüfbarkeit

Bei der Erfassung von Daten muss sichergestellt werden, dass diese Erfassung korrekt ist und keine Fehler oder Inkonsistenzen entstehen. Hierfür wird bei der Metamodellierung das Metamodell herangezogen. Das Metamodell ist die Grammatik für ein Modell und kann daher auch genutzt werden, um zu überprüfen, ob ein gegebenes Modell dieser Grammatik genügt. Diese Überprüfung bzw. Verifikation kann manuell oder automatisiert geschehen.

2.5.3 Quellcodeerzeugung

Sind bereits Modelle erstellt, so bietet es sich an, dass man diese auch in der weiteren Implementierungsphase bestmöglich nutzt. Das Gebiet des *Model Driven Development (MDD)* widmet sich der automatischen Erzeugung von Quellcodes aus Modellen, um so Fehler bei der manuellen Übersetzung von Modellen in Code zu vermeiden. Typische Erzeugungsschritte sind beispielsweise die Erstellung von Java-Klassen aus einem UML-Klassendiagramm.

2.5.4 Datenaustausch

Mit Hilfe von Metamodellen und den darauf basierenden Modellen lassen sich Informationen und Daten strukturiert erfassen. Es liegt daher auch in der Natur von Metamodellen, dass diese zur Definition von Datenaustauschformaten genutzt werden.

Zusammen mit der Metamodell-Beschreibungssprache MOF wurde zudem ein Standard verabschiedet, mit dem sich Modelle als XML-Dokumente serialisieren lassen: das *XML Metadata Interchange (XMI)* [107] Format. XMI ist ein Standard mit dessen Hilfe man Objekte eines Modells mittels XML codieren kann [18]. XMI ist eng mit MOF verknüpft und dadurch ein mächtiges Datenaustauschformat für alle möglichen Arten von Modellen. XMI bietet nach Brodsky [18] folgende Vorteile:

- XMI bietet einen Standard, um Modelle in XML darzustellen und auszutauschen
- XMI spezifiziert, wie man aus Metamodellen ein XML-Schema ableitet
- XMI erlaubt es, mit einfachen Dokumenten anzufangen und diese zu erweitern, sobald die Anwendung komplexer wird
- XMI bietet eine Abstraktionsebene auf XML, so dass man sich nicht mit den Implementierungsdetails von XML beschäftigen muss und trotzdem die Vorteile von XML nutzen kann

Im Kontext dieser Arbeit wird XMI hauptsächlich dafür genutzt, bestehende Modelle zu serialisieren, mit einem Interpreter wieder einzulesen und auf Korrektheit zu überprüfen.

Serialisierung – Unter Serialisierung versteht man verschiedene Verfahren, um Objekte in persistierbare und versendbare Datenströme zu verwandeln [49]. So können sie zu einem späteren Zeitpunkt als Kopien der Originalobjekte wieder deserialisiert werden. Die Serialisierung kann zur Speicherung oder zum Transfer auf ein entferntes System ge-

nutzt werden. Nach der Deserialisierung können die Objekte wie die Ursprungs-Objekte vor der Serialisierung genutzt werden. Klassisch werden Objekte bei der Serialisierung in Byteströme überführt. Es gibt jedoch weitere Verfahren, die, wie das bei XMI zum Einsatz kommende, die Objekte nicht in Byte-Ströme, sondern direkt in XML-Dokumente überführen.

Überprüfbarkeit – In XML werden XML-Schemata verwendet, um Dokumente zu verifizieren, also auf syntaktische Korrektheit zu prüfen. Ein XML-Schema ist selbst ein XML-Dokument, das Regeln für erlaubte Konstrukte in dem XML-Dokument, auf das es sich bezieht, definiert. Ein Beispiel für die Definition eines XML-Schemas in einem XML-Dokument ist in Abbildung 25 zu sehen. Mit Hilfe des XML-Schemas kann nun ein Parser das XML-Dokument auf syntaktische Korrektheit prüfen.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:xmi="http://www.omg.org/XMI"
            targetNamespace="http://cars"
            xmlns:cars="http://cars">
  <xsd:import namespace="http://www.omg.org/XMI"
              schemaLocation="xmi20.xsd"/>
</xsd:schema>
```

Abbildung 25: Beispiel XML-Schema Definition [18]

Nachdem die Grundlagen gelegt wurden, werden im Folgenden die verwandten Arbeiten beschrieben.

3. Verwandte Arbeiten

Im Rahmen dieser Arbeit wird eine Methode zur „Bereithaltung eines Notfallsystems in der Cloud“ (Cloud Standby) vorgestellt. Diese Methode besteht aus den zwei Teilen „Methode zur Notfallwiederherstellung“ und „Deployment-Methode“. Jede dieser beiden Methoden ist ein eigenständiger Teilbeitrag, zu dem sich verwandte Arbeiten in der Wissenschaft, aber auch in der Industrie, finden lassen. Diese werden im Folgenden vorgestellt.

3.1 Methoden zur Notfallwiederherstellung der Cloud

Die Methode zur Notfallwiederherstellung, wie sie in dieser Arbeit vorgestellt wird, verfolgt einen Warm-Standby-Ansatz, bei dem die Laufzeitumgebung des Notfallsystems eine Cloud ist. Es werden daher im Kapitel 3.1.1 zunächst verwandte Arbeiten zum Thema Warm-Standby in der Cloud vorgestellt. Neben Warm-Standby existieren in der Literatur auch weitere Ansätze zum Thema Hot-Standby, auf Basis derer sich ein Notfallsystem ebenfalls bereithalten lässt. Diese werden in Kapitel 3.1.2 vorgestellt. Zusätzlich werden komplementäre Ansätze aus dem Themengebiet Fehlertoleranz in Kapitel 3.1.3 vorgestellt.

3.1.1 Warm-Standby

Wood et al. [134] analysieren in ihrer Arbeit, ob Cloud Computing für die Bereitstellung von Notfallsystemen im Standby-Betrieb von Nutzen ist. Ihre Arbeit konzentriert sich dabei auf die ökonomische Bewertung von Cloud-gestützter Notfallwiederherstellung und zeigt, dass besonders beim Warm-Standby die Kostenvorteile von Cloud Computing zum Tragen

kommen. Wird das Standby-System wie beim Hot-Standby kontinuierlich repliziert, so sind die Effekte durch Cloud Computing eher gering. Werden die Abstände für die Replikation jedoch vergrößert, so kommen die Effekte der bedarfsgerechten Abrechnung zum Tragen.

Auch Pokharel et al. [113] setzen sich allgemein mit Standby-Techniken unter Nutzung von Cloud Computing auseinander. Die Autoren beschreiben, dass sich Cloud Computing für die Notfallwiederherstellung empfiehlt, da hierdurch die Kosten im Vergleich zu traditionellen Ansätzen, bei denen die Ressourcen tatsächlich vorgehalten werden müssen, reduziert werden können. Es wird außerdem vorgeschlagen, das Primärsystem und das Notfallsystem geographisch zu trennen. Mit einer Markov-basierten Simulation wird gezeigt, dass sich durch die Vorhaltung und geographische Trennung eine höhere Verfügbarkeit erreichen lässt. Sowohl der Ansatz von Wood et al. als auch der von Pokharel et al. beleuchten dabei sehr allgemein das Themengebiet der Notfallwiederherstellung unter Nutzung von Cloud Computing. Beide Ansätze beschreiben allerdings nur sehr abstrakt, wie sich ein solches Cloud-gestütztes Notfallmanagement technisch umsetzen ließe.

Anders verhält es sich bei der Arbeit von Klems et al. [69]. Hier wird ein konkreter Ansatz für den Warm-Standby in der Cloud vorgestellt. Die Autoren zeigen, dass durch Nutzung der Cloud im Kontext von Warm-Standby sowohl Ersparnisse auf der Kostenseite als auch eine Reduktion der Deploymentzeiten durch Automatisierung zu erreichen ist. Ähnlich wie die vorliegende Arbeit, verfolgt die Arbeit von Klems et al. den Ansatz, dass das Notfallsystem periodisch aktualisiert wird, um so die Geschäfts- und Konfigurationsdaten zu aktualisieren. Ihr Ansatz konzentriert sich dabei auf einzelne Server statt auf komplexe Server-Landschaften und schlägt vor, Deployment-Prozesse als BPEL-Workflows zu modellieren und in einer zentralen Datenbank abzuspeichern. Tritt ein Notfall ein, so kann ein verantwortlicher Operator auf diese Workflows zurückgreifen und

manuell den Notbetrieb einleiten. Alle weiteren Schritte der Workflows werden dann automatisch von einer Workflow-Engine ausgeführt. Der Ansatz basiert zudem darauf, dass nicht nur das Notfallsystem bei einem Cloud Anbieter vorgehalten wird, sondern auch ein sogenanntes Backup-System. Der Hintergrund hierfür ist, dass der Ansatz sich nicht alleine auf den RTO, sondern auch auf das RPO konzentriert und eine Möglichkeit bietet, Datensicherungen auf Basis virtueller Maschinen durchzuführen. Im ersten Schritt werden die Geschäfts- und Konfigurationsdaten zunächst vom Produktivsystem auf das Backup-System übertragen, um dann im zweiten Schritt vom Backup-System auf das Notfallsystem repliziert zu werden. Dies hat zur Folge, dass für jeden produktiven Server zwei virtuelle Server in der Cloud vorgehalten werden müssen und somit doppelte Kosten anfallen. Allerdings zeigt die Arbeit von Klems et al. auch, dass durch Automatisierung das RTO um den Faktor 5 reduziert werden kann.

Mit verteilten Systemen und deren Wiederherstellung beschäftigen sich Joshi et al. [63]. Sie beschreiben in ihrer Arbeit, wie im Rahmen der Notfallwiederherstellung von verteilten Systemen eine Notfallerkennung durchgeführt werden kann und stellen zwei Algorithmen vor, mittels derer eine Entscheidungsunterstützung zur Identifikation von Notfällen durchgeführt werden kann. Die Verfahren, die von Joshi et al. vorgestellt werden, sind für diese Arbeit relevant, da sie komplementär eingesetzt werden können, um Notfälle zu erkennen.

Auch der Ansatz *CloudSpider* von Bose et al. [15] ist im Kontext dieser Arbeit relevant. Es wird vorgeschlagen, mit Hilfe bestehender Migrations- und Replikationstechniken ein zusätzliches Replika zur Notfallwiederherstellung bereit zu halten. Der Ansatz konzentriert sich hierbei auf einen Algorithmus für die Auswahl und Verteilung der virtuellen Maschinen auf verschiedene Cloud-Rechenzentren sowie einen Scheduling-Algorithmus, der die Datenübertragung zur Laufzeit koordiniert. Die Algorithmen berücksichtigen hierbei sowohl die Kosten für Transport und Vorhaltung der

Daten bei einzelnen Cloud-Anbietern als auch die Ähnlichkeiten von vorhandenen virtuellen Festplatten (sodass Daten nur noch inkrementell übertragen werden müssen). Sind die virtuellen Maschinen in verschiedene Cloud-Rechenzentren repliziert, so können diese auch im Rahmen der Notfallwiederherstellung in Betrieb genommen werden. Der Ansatz beschreibt jedoch nur theoretische Algorithmen und belegt nicht deren Umsetzung in der realen Welt. Zur Evaluierung bedient er sich lediglich simulativer Ansätze, um die Überlegenheit des Ansatzes zu zeigen. Auf Grund der getroffenen Annahmen ist dieser Ansatz insbesondere in einem Public-Cloud-Umfeld jedoch nur schwer einsetzbar: So wird davon ausgegangen, dass das maßgebliche Problem die Übertragung der virtuellen Maschinen und der virtuellen Festplatten ist. Unterschiede in den Hypervisortechnologien und der Konfiguration der einzelnen Cloud-Rechenzentren können hier allerdings zu Problemen führen, sodass die übertragenen virtuellen Maschinen im neuen Rechenzentrum nicht lauffähig sind [79]. Innerhalb einer homogenen privaten Cloud-Umgebung könnte der Ansatz dennoch genutzt werden, um bestehende Warm-Standby- oder Migrations-Ansätze zu verbessern.

3.1.2 Hot-Standby

Bei dem Hot-Standby-Ansatz *PipeCloud* [135] werden Festplatten-Schreibzugriffe virtueller Maschinen in die Cloud repliziert. Hierzu ist es notwendig, in der Primärumgebung Zugang zum Hypervisor aller beteiligten Server zu haben. Der Hypervisor wird so manipuliert, dass Schreibzugriffe auf die virtuellen Festplatten der einzelnen virtuellen Maschinen kopiert und über das Netzwerk versendet werden können. Daraufhin werden diese asynchron in ein Cloud-Rechenzentrum übermittelt, um dort in einer „Pipe“, also einer Warteschlange, abgelegt und schließlich in eine dort gespeicherte virtuelle Festplatte übertragen zu werden. Hierdurch kann erreicht werden, dass ganze virtuelle Server mit einem geringen RPO repli-

ziert werden können. Es wird gezeigt, dass dieses Verfahren den etablierten Verfahren vor allem für die Replikation von Datenbanken überlegen ist. Die Anforderung, vollen Zugriff auf den Hypervisor zu erhalten, schränkt die Nutzbarkeit dieses Ansatzes jedoch auf die Private-Cloud ein, Public-Cloud-Anbieter ihren Kunden bewusst den direkten Zugang zum Hypervisor verwehren. Außerdem ist es notwendig, dass die beiden Hypervisor-Technologien von Primärumgebung und Notfallumgebung mit dem binär identischen Image umgehen können. Bereits kleine Änderungen auf Treiber-Ebene führen in manchen Fällen zu Problemen. Auch haben Studien gezeigt, dass selbst im Private-Cloud-Umfeld die Kompatibilität von Images zwischen verschiedenen Herstellern eingeschränkt ist [79]. Durch die Synchronisation auf Festplatten-Ebene entsteht außerdem ein nicht unerheblicher Overhead, da jeder schreibende Festplattenzugriff über das Internet verschickt wird. Es können so in sehr kurzer Zeit sehr hohe Datenmengen entstehen, was sowohl die verfügbare Bandbreite der virtuellen Maschine einschränkt, als auch Traffic und eine gesteigerte Grundlast entstehen lässt. Dies kann sich nachteilig auf die Performance und Verfügbarkeit aller Server auswirken. Für den Traffic können außerdem noch zusätzliche Kosten anfallen.

Weitere Ansätze für Hot-Standby in der Cloud sind *Remus* [32] und das darauf basierende *SecondSite* [115]. Der Ansatz von Remus ist sehr ähnlich zu *PipeCloud*. Auch bei diesem Ansatz werden durch direkten Zugriff auf den Hypervisor und hier speziell auf die Funktionalität von XEN virtuelle Maschinen während des Betriebs kopiert und durch Kopieren und Versenden von Schreibanfragen auf die Festplatte und den Arbeitsspeicher aktuell gehalten. Diese Daten werden wie bei *PipeCloud* über das Netzwerk versendet. Anders als *PipeCloud* konzentriert sich *Remus* jedoch nicht auf Cloud-Computing, sondern auf die Virtualisierung innerhalb eines einzelnen Rechenzentrums. Erst durch die Erweiterung *SecondSite* wird es möglich, Gruppen von virtuellen Maschinen über das Internet in ein entferntes

Rechenzentrum zu replizieren. Jedoch ist auch hier ein tiefer Eingriff sowohl in das Netzwerk als auch den Hypervisor notwendig, sodass sich diese Ansätze für den Einsatz in privaten, nicht jedoch in Public-Clouds anbieten.

Die Replikation von virtuellen Maschinen innerhalb einer Cloud wird auch im Ansatz von Caraman et al. [25] verfolgt. Dieser *Romulus* genannte Ansatz sieht sich als eine auf KVM basierende Konkurrenz zu *Remus*, die Schwächen bei der Nutzung von XEN behebt und explizit Algorithmen für die Replikation beschreibt, aber ebenfalls Schreibzugriffe auf die virtuelle Festplatte über das Netzwerk versendet. Anders als bei *Remus* erfolgt bei diesem Ansatz der Eingriff nicht in den Hypervisor, sondern in die Management Software. Die Konsequenz ist jedoch ähnlich: Wie bei den Ansätzen, die in den Hypervisor eingreifen, ist dieser Ansatz lediglich in privaten Clouds [94] nutzbar.

Neben den Ansätzen aus der Wissenschaft waren es in der Vergangenheit gerade große Speicher-Anbieter wie NetApp[8] oder EMC[9], die Standby-Techniken für die Absicherung von Daten genutzt haben. Ausgehend von klassischen RAID-Systemen, bei denen einzelne Festplatten redundant vorgehalten werden, haben diese Unternehmen ihre Systeme weiterentwickelt, um die RAID-Systeme über größere Strecken zu replizieren und mehrere Rechenzentren parallel zu betreiben. Diese proprietären Lösungen benötigen die Spezial-Hardware der Hersteller und sind teuer. Sie nutzen kein Cloud Computing für die Absicherung, sondern bilden lediglich die Basis für Cloud-Rechenzentren um die Daten auf Speicher-Ebene zu replizieren.

[8] http://www.netapp.com

[9] http://www.emc.com

Anbieter wie NetApp haben jedoch auch die Vorteile von Cloud Computing erkannt und bieten Lösungen an, mit denen die Daten nicht zu einem anderen NetApp-Speicher, sondern in die Cloud repliziert werden können. Dort können die Daten dann z.B. in Big Data Analyse-Techniken oder auch zu Notfallwiederherstellungs-Zwecken genutzt werden [102]. Diese Verfahren proprietärer Anbieter haben gemein, dass sie zwar performant sind, aber immer auch auf bestehende Lösungen dieser Anbieter aufbauen und damit auf teure und spezialisierte Hardware, wie Speicher-Controller, SANs oder intelligente Switches [99].

Diesem Problem der hohen Kosten traditioneller Speicher-Anbieter begegnen Nadgowda et al. [99] in ihrem Ansatz *I2Map*. Sie stellen ein Verfahren vor, mit dem virtuelle Maschinen Images durch die Protokollierung von Software-Installationsschritten und das Reproduzieren dieser Schritte, automatisiert in ein anderes Rechenzentrum repliziert werden können. Der Ansatz verfolgt damit ein ähnliches Vorgehen wie die in dieser Arbeit beschriebene Föderation auf Betriebssystem-Ebene (siehe Kapitel 5.2.2) und kann komplementär eingesetzt werden. So wäre es möglich, mit *I2Map* das Modell der Deployment-Methode (siehe Kapitel 5) automatisch zu füllen.

3.1.3 Fehlertoleranz

Weiterhin ist der Ansatz von Jhawar et al. [60], [61], der das Thema Fehlertoleranz bei IaaS Cloud Computing zum Inhalt hat, für diese Arbeit von Relevanz. In dem Ansatz wird ein Verfahren vorgestellt, mit dem Fehlertoleranz durch einen Drittanbieter realisiert werden kann. Der Ansatz konzentriert sich dabei auf die Anwendungsschicht und die Anwendungsentwicklung und nicht auf das gesamte verteilte System. Durch die Einführung eines Intermediärs kann der Eingriff in die Anwendung verkleinert werden, da das Management der Fehlertoleranz ausgelagert wird. Der Ansatz konzentriert sich dabei darauf, Fehlertoleranz in eine Cloud-

Anwendung zu integrieren und um Methoden wie Fehler in der Anwendung erkannt werden können.

Einen ähnlichen Fokus hat der Ansatz von Ayari et al. [7]. Auch er beschäftigt sich mit Fehlertoleranz und der Herstellung von Hochverfügbarkeit in verteilten Systemen. Der Fokus ihrer Arbeit liegt jedoch mehr auf dem Netzwerk als auf der Anwendung. Beide Fehlertoleranzansätze stehen nicht in Konkurrenz zu der hier vorgestellten Arbeit, sondern ergänzen sie.

Jayasinghe et al. [59] stellen in ihrem Ansatz *AESON* vor, wie auf Basis von Cloud Computing bestehende verteilte Systeme überwacht und fehlerhaft arbeitende Server wieder gestartet werden. Sie entwickeln dazu eine Deployment-Laufzeitumgebung, die nicht nur verteile Systeme deployen kann, sondern auch Fehler in einzelnen Komponenten erkennt und diese bei Bedarf neu startet.

3.1.4 Zusammenfassung

Zusammengefasst zeigt sich, dass ein Großteil der Ansätze aus Wissenschaft und Industrie sich auf das Replizieren von virtuellen Maschinen auf Basis kompletter Images konzentriert. Dies ermöglicht es, Hot-Standby-Notfallumgebungen aufzubauen und den Betrieb innerhalb von Sekunden umzuschalten. Diese Ansätze sind jedoch mit Einschränkungen verbunden: Häufig ist es notwendig, dass entweder in der Primärumgebung oder der Notfallumgebung Zugriff auf den Hypervisor besteht oder dass bei allen Anbietern die gleiche Hypervisor-Technologie im Einsatz ist. Dies ist jedoch in der Praxis in einem Public-Cloud-Umfeld nicht gegeben. Jeder Anbieter setzt eigene Virtualisierungslösungen ein und verbirgt einen Großteil der Funktionalitäten der Hypervisor-Anbieter vor dem Cloud-Nutzer. Soll also eine anbieterunabhängige Absicherung über unterschiedliche Public-Cloud-Anbieter hinweg stattfinden, so lassen sich diese Verfahren hierfür nicht einsetzen. Außerdem haben diese Ansätze gemein, dass das kontinuierliche Übertragen von Änderungen auf Dateisystem-Ebene

große Mengen an Traffic produziert, was neben Performanz-Problemen auch zu hohen Kosten führen kann. Andere Ansätze, die weniger hohe Anforderungen hinsichtlich RTO und RPO haben und als Warm-Standby zu klassifizieren sind, bieten lediglich rudimentäre Verfahren, um einzelne virtuelle Maschinen von einem Anbieter auf einen anderen abzusichern, ohne dabei jedoch die Abhängigkeiten in komplexeren Systemlandschaften zu betrachten. Für die Absicherung solcher komplexer Systeme und Integration bestehender Datensicherungslösungen lassen sich diese Ansätze daher nicht anwenden. Die in diesem Kapitel vorgestellten Ansätze sind nochmals in Tabelle 1 und Tabelle 2 zusammengefasst. Die Ansätze in Tabelle 1 stehen in Konkurrenz zu dieser Arbeit und sind nach Kriterien, die aus den Anforderungen aus Kapitel 1 abgeleitet wurden, bewertet. Die Ansätze in Tabelle 2 ergänzen diese Arbeit in einzelnen Aspekten, die sich nach dem Fokus des jeweiligen Beitrags richtet.

Tabelle 1: Bestehende Methoden zur Notfallwiederherstellung

Ansatz	**Art der Absicherung**	**Cloud**	**Kosten**	**Anbieter-unabhängig**	**Verteilte Systeme**
Klems et al. [69]	Warm	Public/Private	Niedrig	Ja	Nein
PipeCloud [135]	Hot	Private	Hoch	Ja	Ja
Remus [32]	Hot	Private	Hoch	Nein	Nein
SecondSite [115]	Hot	Private	Hoch	Nein	Ja
Romulus [25]	Hot	Private	Hoch	Ja	Ja
NetApp [102]	Hot	Public	Hoch	Ja	Ja

Tabelle 2: Weitere verwandte Arbeiten im Themengebiet Notfallwiederherstellung

Ansatz	Art der Absicherung	Fokus des Beitrags
Wood et al. [134]	Warm	Kostenbetrachtung
Pokharel et al. [113]	Warm	Kostenbetrachtung
Joshi et al. [63]	Warm	Notfallerkennung
Bose et al. [15]	Warm	Datenübertragung
Nadgowda et al. [99]	Hot	Softwareinstallation
Jhawar et al. [60], [61]	Fehlertoleranz	Anwendungsentwicklung
Ayari et al. [7]	Fehlertoleranz	Netzwerk
Jayasinghe et al. [59]	Fehlertoleranz	Monitoring

Nachdem die verwandten Arbeiten für den ersten Teilbeitrag dieser Arbeit beschrieben wurden, werden im weiteren Arbeiten zu den modellbasierten Deployment-Methoden vorgestellt.

3.2 Modellbasierte Deployment-Methoden

Für die Absicherung im Rahmen dieser Arbeit wird eine Deployment-Methode benötigt, mit der ein verteiltes System anbieterunabhängig gestartet werden kann. In der Literatur werden verschiedene Ansätze zur Automatisierung des Deployments komplexer Anwendungen, wie beispielsweise verteilte Systeme, diskutiert. Im Folgenden werden Ansätze vorgestellt, die sich mit dem Themenfeld der Deployment-Prozessautomatisierung, der Modellierung anbieterunabhängiger Deployments von verteilten Systemen und der Optimierung von Deployments beschäftigen.

3.2.1 Deploymentprozesse

Mietzner et al. [97], [98] stellen mit *Cafe* und darauf aufbauenden Arbeiten einen Ansatz vor, der mit Hilfe von Workflows das Deployment unterschiedlicher Anwendungs-Komponenten bei verschiedenen Anbietern er-

möglicht. Mit einer rudimentären formalen Sprache können Anwendungen in Komponenten zerlegt und für diese Komponenten dann automatisch entschieden werden, wo sie deployt werden. Mit Hilfe der Workflows wird dann der anbieterabhängige Deploymentprozess modelliert und mit einer Workflow-Engine ausgeführt. Der Ansatz wird stark aus der Workflow-Forschung getrieben und konzentriert sich auf die als Workflows definierten Deployment-Prozesse, sowie das Zusammenspiel verschiedener Cloud-Anbieter-spezifischer Prozesse.

Sampaio et al. [117] stellen mit *Uni4Cloud* ein Verfahren vor, mit dem ein verteiltes System anbieterunabhängig deployt werden kann. Dabei konzentriert das Verfahren sich nicht auf das verteilte System selbst oder dessen Beschreibung, sondern ähnlich wie bei Mietzner et al. auf die Deploymentprozesse. Speziell geht es bei *Uni4Cloud* darum, wie ein einheitliches Deployment auch bei verschiedenen Schnittstellen für das Management bei unterschiedlichen Anbietern durchgeführt werden kann. Hierzu stützt sich der Ansatz auf dem Standard Open Virtualization Format (OVF) [31]. Es wird angenommen, dass alle Images durch OVF interoperabel bei verschiedenen Anbietern ausgeführt werden können. In der Praxis zeigt sich jedoch, dass wenige Cloud-Anbieter auf OVF setzen und selbst auf Hypervisor-Ebene OVF nicht zwingend Interoperabilität zwischen Images verschiedener Anbieter schafft [82].

3.2.2 Formale Beschreibung verteilter Systeme

Arnold et al. [5] und Eilam et al. [34], [35] beschreiben in ihren Arbeiten, wie sich mittels Methoden aus dem Bereich der Modellgetriebenen Architektur oder auch Model-Driven Architecture das Deployment von verteilten Systemen in großen Rechenzentren vereinfachen lässt. In ihren Arbeiten wird ein besonderes Augenmerk auf die Modellierung durch verschiedene Experten und die Transformation der einzelnen Modelle gelegt. Es wird eine Architektur beschrieben, mit der die einzelnen Experten ihre eigenen

Modelle erzeugen und speichern können. Mittels automatischer Optimierung und Zuordnung kann die Anwendung so automatisiert in einem Rechenzentrum als verteiltes System deployt werden. Die genannten Arbeiten legen dabei Grundlagen für modellbasierte Deployment-Methoden in traditionellen Großrechenzentren ohne Virtualisierung.

Konstantinou et al. [72] stellen einen auf der zuvor beschriebenen Arbeit aufbauenden modellbasierten Ansatz vor, mit dem dieser auf Cloud Computing angewandt werden kann. Ihr Ansatz konzentriert sich auf die Images der Server und deren Kommunikation im laufenden Betrieb. Es wird eine Beschreibungssprache vorgestellt, mit der die Kommunikationsschnittstelle beschrieben wird und so beim Deployment die Infrastruktur, wie virtuelle Netzwerke, passend konfiguriert werden kann. Ähnlich wie der Ansatz von Eilam et al. konzentriert sich ihr Ansatz darauf, dass verschiedene Akteure unterschiedliche Modelle erzeugen, die dann im Deploymentprozess zusammen- und ausgeführt werden. Außerdem ist auch Konstantinous Ansatz darauf ausgelegt, ein verteiltes System bei einem einzelnen Anbieter auszuführen. Die Beschreibung des Anbieters ist nicht Teil ihres Ansatzes. Es wird außerdem angenommen, dass die Images der virtuellen Maschinen keinerlei Nachkonfiguration bedürfen. Aufbauend auf den Arbeiten von Arnold et al. und Eilam et al. wurde jüngst der Ansatz *Weaver* von Kalantar et al. vorgestellt. Er beschreibt erste Ideen wie man mittels einer auf Ruby aufbauenden domänenspezifischen Sprache ein verteiltes System in Code ausdrückt und anbieterunabhängig deployen kann.

Maximilien et al. [92], [93] stellen in ihrem Ansatz *Altocumulus* eine erste Idee für eine neuartige Middleware-Architektur vor, die Cloud-agnostisch ist. Hierbei wird auch ein Metamodell zur Beschreibung von Middlewares vorgestellt. Allerdings wird bei diesem Ansatz kein Unterschied gemacht, ob die Middleware auf einer IaaS oder PaaS Cloud deployt wird. Dies hat den Vorteil, dass von der bereitgestellten Hosting-

Infrastruktur abstrahiert wird und man nicht zwischen IaaS und PaaS Anwendungen unterscheiden muss. Um dies zu ermöglichen, wird an vielen Stellen für Funktionalitäten der kleinste gemeinsame Nenner gewählt oder Annahmen getroffen, so dass sich die Repräsentation von IaaS Images auch auf PaaS abbilden lässt.

Cosmo et al. [29] und Catan et al. [26] stellen in ihrem Ansatz *Aeolus* ein Modell vor, das nicht aus virtuellen Maschinen oder gar Komponenten besteht, sondern auf den Software-Paketen innerhalb der virtuellen Maschinen aufsetzt. Mit Zustandsautomaten wird der Zustand der Softwarepakete beschrieben und es ist möglich, Abhängigkeiten zwischen den Zuständen der Softwarepakete auf verschiedenen virtuellen Maschinen herzustellen. Das aufgestellte mathematische Modell erlaubt es, Software-Installationen auf verteilten Systemen sehr ausführlich und detailliert zu planen. Die Autoren stellen in ihrer Arbeit allerdings selbst fest, dass ein derart freies Modell schnell zu unentscheidbaren Situationen führt, so dass sie das Modell einschränken und als *Aeolus*$^{-}$ bezeichnen. In diesem Modell sind nicht mehr wahllos Verbindungen zwischen Softwarepaketen erlaubt, dafür ist es aber entscheidbar. Die Autoren konzentrieren sich in ihrem Ansatz rein auf die Softwareinstallation und lassen die Cloud-Infrastruktur außer Betracht. Es wäre jedoch möglich *Aeolus* zusammen mit dem in dieser Arbeit vorgestellten Modell zu benutzen um zu einem lauffähigen und anbieterunabhängigen Deployment zu gelangen.

Auch Brandtzæg et al. [17] stellen mit *Pim4Cloud* eine erste Idee für eine domänenspezifische Sprache zum Deployen von Anwendungen in der Cloud vor. Hierzu wird der für das Grid-Computing entworfene Ansatz *DeployWare* [39] für das Cloud Computing angepasst. Die *Pim4Cloud* Beschreibungssprache zeichnet sich dadurch aus, dass sie sehr einfach gehalten ist: Der Kern der Sprache besteht aus schachtelbaren Komponenten, die je nach Implementierung und Anwendungsfall typisiert werden. Es wird in ihrem Ansatz alles als Komponenten modelliert und es bleibt der

Ausführungsumgebung überlassen zum Beispiel die Unterscheidung zwischen der Komponente „Virtuelle Maschine“ und „WAR Artefakt“ vorzunehmen. Hierdurch ist die Sprache sehr flexibel einsetzbar, allerdings nicht sehr Cloud-spezifisch. Aus dem gleichen Forschungsprojekt stammt auch der Ansatz von Ferry et al. [37], [38]. In diesem Ansatz wird mittels Modellierung ein anbieterunabhängiges Modell erstellt, dann transformiert und anbieterunabhängig deployt. Der Ansatz ermöglicht damit zwar ein verteiltes System anbieterunabhängig zu deployen, die Beschreibungssprache konzentriert sich aber nur auf die Struktur des verteilten Systems. Es kann vom Modellierer nicht sichergestellt werden, dass die Softwarestacks aller virtuellen Maschinen des verteilten Systems bei allen beteiligten Anbietern gleich ist. Damit könnte es sein, dass das gestartete verteilte System nicht lauffähig ist. Auch gibt es in der Beschreibungssprache keine Möglichkeit, die anbieterspezifische Cloud-Infrastruktur mehrerer Anbieter zu modellieren.

Seit 2013 ist mit *TOSCA* auch ein Standard für das Beschreiben von verteilten Systemen in der Cloud verfügbar [105]. Er wird von der OASIS standardisiert und findet einige Unterstützung in der Industrie. Der Fokus des Standards liegt dabei stark auf Deploymentprozessen, die mit einer Business-Prozess-Beschreibungssprache formalisiert sind und ausgeführt werden können. Zudem ist es möglich, mit *TOSCA* die Struktur der Anwendung zu formalisieren. *TOSCA* bietet dem Anwender viele Freiheiten, was auch ein Nachteil des Standards ist. Obwohl er Cloud-spezifisch sein soll, fehlen derzeitig ähnlich wie bei dem Ansatz *Pim4Cloud* konkrete Cloud-Elemente für einen sinnvollen Einsatz. Im Zuge dieser Arbeit und eines Forschungsprojekts in Zusammenarbeit mit der Deutschen Telekom wurde eine Studie zu *TOSCA* für die OASIS erstellt, im Rahmen derer insbesondere die zu hohe Flexibilität und der geringe Cloud-Bezug als derzeitige Hemmnisse für den praktischen Einsatz von TOSCA identifiziert wurden [65], [66]. Die Standarisierungsorganisation OASIS wurde im

Rahmen der Durchführung der Studie auf diese Nachteile hingewiesen. Als möglicher Lösungsansatz wurden zudem Forschungsergebnisse dieser Arbeit präsentiert.

Ein weiterer de-facto Standard wurde vom Amazon mit Cloud *Formation* [6] entwickelt. Mit Cloud Formation können verteilte System auf der Amazon Cloud beschrieben und gestartet werden. Da dieser Standard von Amazon selbst entwickelt ist und auf die spezifischen Gegebenheiten der Amazon-Angebote abzielt, ist es nicht möglich, andere Anbieter zu integrieren und kann daher im Rahmen dieser Arbeit nicht genutzt werden. An der Portierung der Cloud Formation Beschreibungssprache für OpenStack wird in dem Projekt *Heat* [110] gearbeitet. Auch bei Heat ist es nicht möglich, ein verteiltes System anbieterunabhängig zu beschreiben, sondern lediglich die von Amazon entwickelte Beschreibungssprache auch in OpenStack einzusetzen.

Aus der Industrie gibt es mit IBM einen Anbieter von vorwiegend privaten Cloud-Lösungen, der mit *UrbanCode Deploy* [56] und *SmartCloud Provisioning* [55] Lösungen für Automatisierung des Deployments von verteilten Anwendungen bereit hält. *UrbanCode Deploy* ermöglicht es, hierbei verschiedene Deployment-Prozesse zu automatisieren, zu orchestrieren und Konfigurationen zu verwalten. Dadurch lassen sich Softwareinstallationen automatisiert auf einer großen Anzahl an Systemen deployen. *UrbanCode* lässt sich zusammen mit *SmartCloud* einsetzen, um so auch die IBM Cloud Infrastruktur zu verwalten. So lassen sich auch verteilte Systeme automatisiert in der IBM Private-Cloud deployen.

Für die Cloud-Management Lösung Openstack wurde von Canonical *Juju* [24] entwickelt. Ähnlich wie die properitären Lösungen von IBM ermöglicht es *Juju* für Openstack Deployments mittels einer formalen Beschreibung zu vereinfachen. Hierfür wird auch ein Editor angeboten, der die Erstellung von Deploymentbeschreibungen erleichtert. Ähnlich wie bei IBM liegt auch bei *Juju* der Fokus auf einer einzelnen Cloud-Management-

Lösung und es werden keine anbieterunabhängigen Deployments unterstützt.

3.2.3 Deploymentplanung

Hummer et al. [54] beschäftigen sich mit den Softwareinstallationsprozessen, die beim Deployment notwendig sind. Viele der existierenden Automatisierungstools wie Chef [111] verlangen von den Installations-Skripts, dass diese „idempotent“ sind. Das heißt mehrfach ausführbar, ohne dass dasErgebnis verändert wird. Ihr Ansatz beschäftigt sich mit der Frage, wie Installationsanweisungen im Rahmen von Deploymentprozessen auf Idempotenz getestet werden können.

Lascu et al. [75], [76] und Cosmo et al. [28] beschreiben in ihren auf *Aeolus* (siehe Kapitel 3.2.2) aufbauenden Arbeiten, wie man den Softwareinstallationsprozess besser planen und optimieren kann. Insbesondere wird dabei die Frage betrachtet, wie komplexe Software-Installationen so geplant werden können, dass diese ohne Dead-Locks durchführbar ist. Ähnlich wie das unterliegende *Aeolus* Modell fokussiert dieser Ansatz die Softwarepakete auf dem verteilten System und nicht die Cloud-Infrastruktur. Das *Aeolus* Modell sowie Optimierungen auf ihm lassen sich zusammen in der in dieser Arbeit vorgestellten Deployment-Methode einsetzen und können diese erweitern.

3.2.4 Zusammenfassung

Sowohl in der wissenschaftlichen Forschung wie auch im industriellen Umfeld existiert bereits eine Vielzahl von Ansätzen, die sich dem Themengebiet modellbasierte Methoden für das anbieterunabhängige Deployment von verteilten Systemen widmen. Einige beschäftigen sich mit den Deploymentprozessen, andere mit der Modellierung selbst und wieder andere mit der Planung des Deployments. Für die Notfallwiederherstellung

in der Cloud wird ein Ansatz benötigt, mit dem ein verteiltes System automatisiert bei verschiedenen Cloud-Anbietern deployt werden kann. In den verwandten Arbeiten gibt es keinen solchen Ansatz. Viele der Ansätze konzentrieren sich auf einzelne Anbieter oder sind so generisch, dass sie zwar für die Anwendung im Kontext der Absicherung verteilter Systeme adaptiert werden könnten, gleichzeitig aber durch den generischen Ansatz der Vorteil der Modellierung verloren geht. Dies trifft insbesondere auf den einzigen offiziellen Standard, *TOSCA*, zu. Durch den generischen Ansatz dieses Standards ist dieser nicht mehr Cloud-spezifisch und unterstützt den Nutzer des Standards ungenügend, wenn es darum geht, ein verteiltes System zu beschreiben und bei verschiedenen Anbietern auszuführen. Alle in diesem Kapitel vorgestellten Ansätze sind nochmals in Tabelle 3 dargestellt und nach ihrem Fokus, der Anwendbarkeit auf das Cloud Computing und ob sie nativ die Spezifikation des Deployments für mehr als einen Anbieter gleichzeitig erlauben, kategorisiert.

Im folgenden Kapitel 4 wird eine Methode zur Notfallwiederherstellung entwickelt, die eine Methode zum anbieterunabhängigen Deployment von verteilten Systemen voraussetzt. Da es, wie in diesem Kapitel dargelegt, keine solche Deployment-Methode gibt, wird in Kapitel 5 eine solche vorgestellt. Die Methoden-Kapitel 4 und 5 bilden zusammen mit deren Implementierung in Kapitel 6 die Methode „Cloud Standby“ und damit den Beitrag dieser Arbeit.

Tabelle 3: Übersicht der Verwandten Arbeiten zum Themengebiet modellbasierte Deployment-Methoden

Ansatz	**Fokus**	**Cloud**	**Anbieter-unabhängigkeit**
Mietzner et al. [97], [98]	Prozessmodellierung	Public/Private	Ja
Sampaio et al. [117]	Prozess	Public/Private	Ja
Arnold et al. [5]	Modelltransformation	Nein	Nein
Eilam et al. [34], [35]	Modelltransformation	Nein	Nein
Konstantinou et al. [72]	Beschreibungssprache	Public/Private	Nein
Kalantar et al. [64]	Beschreibungssprache	Public/Private	Ja
Maximilien et al. [92], [93]	Beschreibungssprache	Public/Private	Nein
Cosmo et al. [29]	Beschreibungssprache	Public/Private	Nein
Catan et al. [26]	Beschreibungssprache	Public/Private	Nein
Brandtzæg et al. [17]	Beschreibungssprache	Public/Private	Nein
Ferry et al. [37], [38]	Beschreibungssprache	Public/Private	Nein
TOSCA [105]	Beschreibungssprache	Public/Private	Ja
AWS Cloud Formation [6]	Beschreibungssprache	Public	Nein
Heat [110]	Beschreibungssprache	Public/Private	Nein
IBM UrbanCode Deploy [56]	Prozessautomatisierung	Private	Nein
IBM SmartCloud Provisioning [55]	Beschreibungssprache	Private	Nein
Canonical Juju [24]	Beschreibungssprache	Public/Private	Nein
Hummer et al. [54]	Softwaredeploymenttests	-	-
Lascu et al. [75], [76]	Deployment Planung	-	-
Cosmo et al. [28]	Deployment Planung	-	-

4. Methode zur Notfallwiederherstellung

Cloud Standby hat das Ziel, ein Notfallsystem mit einem kostengünstigen Warm-Standby-Ansatz in der Cloud vorzuhalten. In Kapitel 3.1 wurde gezeigt, dass es in den Verwandten Arbeiten keinen Ansatz gibt, der es ermöglicht, ein verteiltes System anbieterunabhängig in der Public-Cloud abzusichern und so von den Kostenvorteilen des Cloud Computing zu profitieren. Viele der Ansätze unterstützen keine verteilten Systeme [32], [69], konzentrieren sich auf das teurere Hot-Standby [25], [32], [102], [115], [135] oder betrachten Warm-Standby in der Cloud nur aus der Kostensicht [112], [134].

Daher wird im Zuge dieser Arbeit eine neue Methode zu Notfallwiederherstellung, bestehend aus einem Notfallwiederherstellungsprozess, dem dazugehörigen Aktualisierungsprotokoll und einer Entscheidungsunterstützung zur Konfiguration, entworfen. Die hier vorgestellte Methode basiert dabei auf den ersten Ideen für Warm-Standby in der Public Cloud, die in den verwandten Arbeiten präsentiert wurden [69], [134]. Die Zustände eines Standby-Systems, wie es in der Literatur beschrieben wird, sind vereinfachend in Abbildung 26 dargestellt. Während der Laufzeit des Produktiv-Systems wird das Standby-System regelmäßig aktualisiert und beim Eintritt eines Notfalls wird der Notbetrieb gestartet.

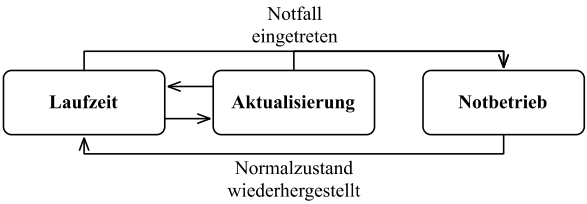

Abbildung 26: Zustände eines Standby Systems

In diesem Kapitel[10] wird eine neue Methode zur Notfallwiederherstellung präsentiert, die wie beststehende Ansätze auf periodischen Aktualisierungen basiert, jedoch auf verteilte Systeme anwendbar ist, bestehende Datensicherungs-Methoden integriert und ein Notfallsystem in der Cloud vorhält (siehe Abbildung 27).

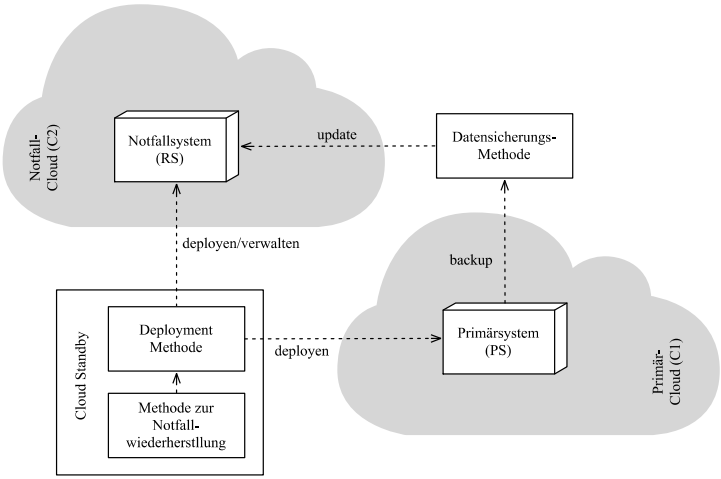

Abbildung 27: Methode zur Notfallwiederherstellung im Kontext der anderen Methoden bei Cloud Standby (vgl. [86])

Hierzu werden die zuvor vorgestellten allgemeinen Zustände eines Standby-Systems detailliert (siehe das UML-Zustandsdiagramm in Abbildung 28). Ist das Primärsystem[11] (PS) einmal gestartet, so wird das Notfallsys-

[10] Teile dieses Kapitel wurde bereits veröffentlicht [84]–[86]

[11] siehe hierzu auch die Annahmen über den Aufbau von verteilten Systemen im Rahmen dieser Arbeit in Kapitel 4.2.1 und die kritische Betrachtung in Kapitel 8.2

tem oder auch Recovery System (RS) periodisch gestartet und mit Hilfe der Datensicherung aktualisiert. Fällt die Primärcloud (C1) aus, so wird das Notfallsystem deployt und übernimmt den Produktivbetrieb.

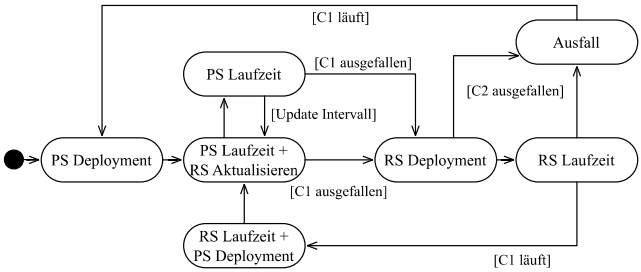

Abbildung 28: Cloud Standby Systemzustände (vgl. [86])

Die Zustände, die das System haben kann, sind wie folgt beschrieben:

PS Deployment – Weder das Primär- noch das Notfallsystem laufen, und zunächst muss das Primärsystem gestartet werden. Nachdem das Primärsystem gestartet ist, geht das System automatisch in den nächsten Zustand über.

PS Laufzeit und RS Aktualisieren – Das Primärsystem läuft nun produktiv und es werden regemäßig Datensicherungen durchgeführt. Parallel dazu wird das Notfallsystem hochgefahren und der aktuelle Zustand der Datensicherung eingespielt. Ist das Notfallsystem auf dem neusten Stand, wird es automatisch heruntergefahren, und das Cloud Standby System geht in den nächsten Zustand über.

PS Laufzeit – In diesem Zustand läuft lediglich das Primärsystem. Das Notfallsystem ist inaktiv, um so Kosten zu sparen. Nachdem das „Update Intervall“ abgelaufen ist, wird das Notfallsystem wieder automatisch gestartet und zurück in den Zustand „PS Laufzeit und RS Aktualisierung“ gekehrt. Das Cloud Standby System bewegt sich so lange zwi-

schen diesen Zuständen hin und her, bis ein Notfall eintritt und das Notfallsystem deployt werden muss.

RS Deployment – Das Notfallsystem wird gestartet. Die Dauer des Deployments ist davon abhängig, wie lange die letzte Aktualisierung des Notfallsystems her ist. Wurde gerade die aktuelle Datensicherung zurückgespielt, so kann das Notfallsystem schnell gestartet werden. Liegt diese Aktualisierung jedoch bereits länger zurück und haben sich in der Zwischenzeit viele Daten angesammelt, so dauert das Deployment entsprechend länger. Sobald das Notfallsystem läuft, wird automatisch in den nächsten Zustand übergegangen.

Notfallsystem Laufzeit– Das Notfallsystem läuft und hat den Produktivbetrieb übernommen. Anfragen von außen werden nun automatisch an das Notfallsystem gerichtet und nicht mehr an das Primärsystem. Das Notfallsystem nutzt jetzt dieselben Datensicherungslösungen wie das Primärsystem, um neu hinzugekommene Daten zu sichern. Sobald festgestellt wird, dass der primäre Cloud-Anbieter wieder verfügbar ist, erfolgt ein automatischer Übergang in den Zustand „Notfallsystem Laufzeit, Primärsystem Deployment“.

RS Laufzeit, PS Deployment – Der primäre Cloud-Anbieter ist wieder verfügbar, das Primärsystem wird angestartet und auf den aktuellen Stand gebracht. Läuft das Notfallsystem, so wird der Produktivbetrieb vom Notfallsystem auf das Primärsystem übertragen und automatisch in den nächsten Zustand übergegangen.

Ausfall – Sollten sowohl C1 als auch C2 gleichzeitig ausfallen, so ist der Geschäftsprozess unterbrochen und es ist nicht mehr möglich das RTO zu erfüllen. Das Primärsystem muss, wenn C1 wieder läuft, komplett neu deployt werden.

Dabei ist zu beachten, dass es zwischen den regelmäßigen Datensicherungen und der Aktualisierung des Notfallsystems keinen festen Zusammenhang gibt. Die Datensicherung richtet sich nach dem Datensicherungs-

Intervall, das durch das RPO bestimmt wird und die Aktualisierung nach dem Update-Intervall, welches wiederum durch das RTO festgelegt wird (siehe dazu die experimentell bestimmte Formel für das RTO in Abhängigkeit des Update-Intervalls aus Kapitel 7.2.2). Dieser Nicht-Zusammenhang ist exemplarisch in Abbildung 29 dargestellt.

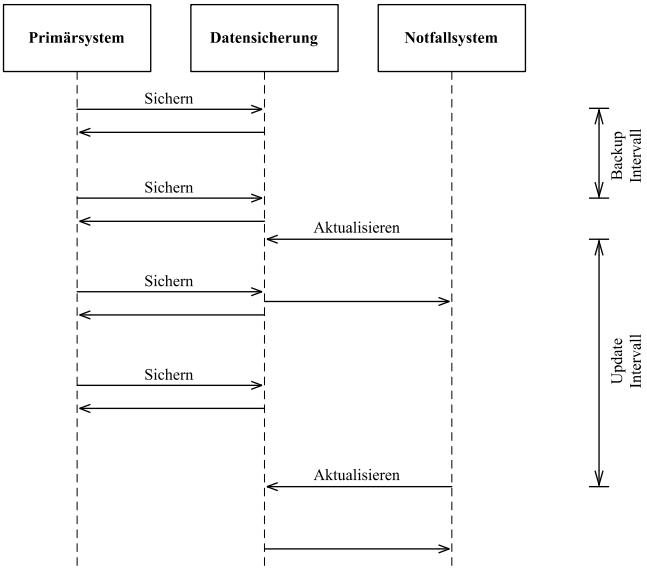

Abbildung 29: Exemplarische Darstellung des Nicht-Zusammenhangs des Update- und Backup-Intervalls als UML Sequenzdiagramm (vgl. [86])

Es wird deutlich, dass die Datensicherung in einem anderen Intervall als die Aktualisierung (Update) des Notfallsystems durchgeführt wird und dass es deshalb zu Problem mit der Konsistenz kommen könnte. Auf diese Problematik wird in Kapitel 4.2.3 näher eingegangen. Weiterhin wird in dieser Arbeit davon ausgegangen, dass der Zusammenhang von Datensicherung des Primärsystems und Aktualisierung des Notfallsystems unabhängig ist.

Es könnte jedoch sein, dass es in Ausnahmefällen Zusammenhänge gibt. So könnte es passieren, dass eine Datensicherung sehr lange dauert und damit bei zwei Aktualisierungen der gleiche Datensatz rückgesichert wird oder dass die Datensicherungsmethode nicht erlaubt, differentielle Datenrücksicherungen durchzuführen. Diese Fragestellungen sind Teil der zukünftigen Arbeiten, die aufbauend auf diese Arbeit ausgeführt werden sollten.

Im Folgenden werden zunächst die Anforderungen sowie die Annahmen an die neue Methode dargestellt und anschließend die Formalisierung des Notfallwiederherstellungsprozesses, des Aktualisierungsprotokoll und dar Entscheidungsunterstützung, wie in Abbildung 30 dargestellt, präsentiert. Als Unterstützung für die Nutzung der neuen Methode wird im Anschluss eine Entscheidungsunterstützung präsentiert, mit deren Hilfe die Methode zur Notfallwiederherstellung bewertet und konfiguriert werden kann.

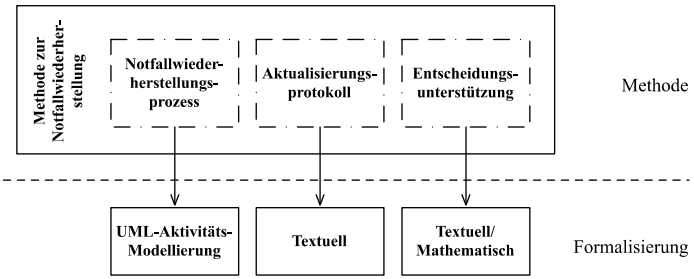

Abbildung 30: Formalisierung der Notfallwiederherstellungsmethode

4.1 Angestrebte Eigenschaften

Die in dieser Arbeit vorgestellte Methode zur Notfallwiederherstellung hat die Eigenschaften, dass sie es ermöglicht, ein verteiltes System anbieterunabhängig abzusichern und bestehende Datensicherungslösungen als Teil der Replikation zu integrieren.

4.1.1 Anbieterunabhängigkeit

Soll ein verteiltes System gegen den Ausfall eines ganzen Anbieters abgesichert werden, so ist die Anbieterunabhängigkeit eine wichtige Voraussetzung. Selbst wenn ein Anbieter vermeintlich als ausfallsicher gilt, kann es bei ihm zu Ausfällen kommen [88]. Verlässt man sich auf einen Anbieter, so führt das außerdem zu seinem Single Point of Failure und damit zu einem Risiko, das es im Notfallmanagement zu vermeiden gilt [4].

Bei der Schaffung einer neuen Methode zur Notfallwiederherstellung muss also darauf geachtet werden, dass ein verteiltes System nicht bei dem gleichen Anbieter, bei dem es auch betrieben wird, abgesichert wird. Das Notfallsystem muss anbieterunabhängig sein und da selbst etablierte Firmen unvorhergesehen vom Markt verschwinden können [33], sollte man auch das Notfallsystem nicht fest an einen zweiten Anbieter binden. Das ganze Notfallwiederherstellungs-System sollte so flexibel gestaltet werden, dass der zweite Anbieter bei Bedarf ausgetauscht werden kann.

4.1.2 Integration bestehender Datensicherungs-Methoden

Gemäß einer Studie des BMWi führen 94% der KMU in Deutschland bereits regelmäßig Datensicherungen durch [19]. Beim Entwurf einer neuen Notfallwiederherstellungs-Methode wäre es daher nicht sinnvoll, diesen Aspekt außen vor zu lassen und ebenfalls eine neue Methode zur Datensicherung zu entwerfen. Ziel dieser Arbeit ist daher nicht, die Kennzahl des RPO zu adressieren, sondern den RTO bei gegebenem RPO zu reduzieren. Es wird also davon ausgegangen, dass das verteilte System durch etablierte Verfahren im Rahmen des Notfallmanagements abgesichert wurde, und es ist Ziel dieser Arbeit, diese etablierten Verfahren zu integrieren Im Rahmen der BIA wurde auch für die einzelnen Systeme das RPO festgelegt, und die Daten sind an einer zentralen Stelle jederzeit und automatisiert zugreifbar.

4.1.3 Integration bestehender Deployment-Methoden

Im Rahmen der Methode zur Notfallwiederherstellung ist es notwendig, das Notfallsystem bei einem Ausfall des Primärsystems *vollautomatisch* zu deployen. Hierzu wird eine Deployment-Methode benötigt, die ein *verteiltes System anbieterunabhängig* deployen und dabei möglichst Rücksicherungen aus der Datensicherung initiieren kann. In Kapitel 3.2 wurden hierzu verschiedene Ansätze aus der Literatur vorgestellt. Wie jedoch auch in Kapitel 3.2 dargestellt, eignet sich keiner der Ansätze, um im Rahmen der Notfallwiederherstellung eingesetzt zu werden. Daher wird in Kapitel 5 eine modellbasierte Deployment-Methode vorgestellt, die speziell für die Absicherung mittels der hier vorgestellten Methode zur Notfallwiederherstellung entwickelt wurde.

4.2 Annahmen und Terminologie

Bei der Entwicklung der Methode zur Notfallwiederherstellung wurden gewisse Annahmen zu verteilten Systemen, dem zugrundeliegenden Fehlermodell und zu verschiedenen Arten von Konsistenz getroffen, die nun diskutiert werden.

4.2.1 Verteiltes System

Wird in dieser Arbeit von einem „System“ gesprochen, so ist immer ein verteiltes System gemeint. Wie in Kapitel 2.2 beschrieben, bestehen verteilte Systeme aus einem oder mehreren Servern, die miteinander kommunizieren. Verteilte Systeme können dabei sehr einfach sein (Client-Server Anwendung) oder sehr komplex mit redundanten Servern in verschiedenen, weltweit verteilten Cloud-Rechenzentren.

Im Rahmen dieser Arbeit, die sich auf KMUs und die Absicherung ihrer Systeme konzentriert, wird davon ausgegangen, dass diese ein

Mehrserver-System betreiben, das innerhalb eines einzelnen Cloud-Rechenzentrums läuft. Dieses kann spezielle Merkmale aufweisen, wie dass es in verschiedene Verfügbarkeitszonen aufgeteilt ist, nicht jedoch über mehrere Standorte. Die Gründe warum ein System als verteiltes System und nicht als Monolith deployt ist, sind vielfältig. So ist es möglich, durch die Verteilung auf verschiedene Server großer Nachfrage besser gerecht zu werden oder auch das System an sich robuster zu gestalten. Im Rahmen der Methode zur Notfallwiederherstellung werden diese „Interna“ an die Deployment-Methode delegiert, sodass aus der Notfallwiederherstellungssicht das verteilte System „als Ganzes“ betrachtet wird und auch als solches abgesichert werden kann. Es also wird lediglich gefordert, dass eine Deployment-Methode existiert, die das verteilte System (inkl. aller Konfigurationen z.B. der Server und des Netzwerks) vollautomatisch bei verschiedenen Anbietern deployt und dass das System mittels einer bestehenden Datensicherungs-Methode regelmäßig gesichert wird.

Da sowohl für das Notfallmanagement als auch auch für verteilte Systeme eigene Ausfall- bzw. Fehlermodelle existierten (siehe Kapitel 2.1.1 und 2.2.2), wird im Folgenden ein neues Fehlermodell, das in dieser Arbeit Anwendung findet, entwickelt.

4.2.2 Fehlermodell

Wie in Kapitel 2.1.1 dargestellt, gibt es im Bereich des Notfallmanagements verschiedene Ausfallklassen, die die Schwere eines Ausfalls beschreiben. Diese reichen von einer einfachen Störung innerhalb eines Rechenzentrums bis hin zu einer Katastrophe, also einem Großschadensereignis bei dem sogar das Leben und die öffentliche Ordnung in Gefahr sind. Dieser Makro-Sicht stehen die verschiedenen, feingranularen Fehlerzustände eines verteilten Systems gegenüber (siehe Kapitel 2.2.2), die Fehler von einem Absturz des gesamten System bis hin zu fehlerhaften Nachrichten abdecken. Das in dieser Arbeit angenommene Fehlermodell ist in Ta-

belle 4 dargestellt. Die Arbeit fokussiert dabei auf den Bereich des Notfalls, könnte aber auch bei Störungen oder Krisen zum Einsatz kommen.

Tabelle 4: Zugrunde liegendes Fehlermodell (der Fokus der Arbeit liegt auf dem grauen Bereich)

Ausfallklasse	Fehlerzustand	Betroffen	Vorsorge durch
Katastrophe	Absturzausfall	Eines oder mehrere Rechenzentren	Nicht möglich
Krise	Absturzausfall	Eines oder mehrere Rechenzentren	Krisenmanagement, Eindämmen von Notfällen
Notfall	Absturzausfall	Rechenzentrum	Notfallmanagement, Notfallwiederherstellung
Störung	Absturzausfall, Dienstausfall, Zeitbedingter Ausfall, Antwortfehler, Byzantischer Fehler	Verteiltes System oder Teile davon	Störungsmanagement

Eine Katastrophe ist ein Großschadensereignis, bei dem aus IT-Sicht mindestens ein Rechenzentrum, wahrscheinlich sogar mehrere betroffen sind. Durch den Ausfall des gesamten Rechenzentrums sind auch alle Systeme in dem Rechenzentrum durch einen Absturzausfall betroffen. Als Unternehmen ist es nicht möglich sich auf eine Katastrophe vorzubereiten.

Ähnlich wenige Vorbereitungsmöglichkeiten hat man gegen eine Krise. Auch hier ist zu erwarten, dass mindestens ein, vielleicht sogar auch mehrere Rechenzentren betroffen sind und alle Server und Systeme in diesen Rechenzentren ausfallen, d.h. den Zustand eines Absturzausfalls haben. Durch ein funktionierendes Notfallmanagement kann sichergestellt werden, dass sich keine Notfälle zu Krisen ausweiten.

Ein Notfall ist ein Schadensereignis, das nicht im normalen Tagesgeschäft bewältigt werden kann, sondern eines speziellen Notfallmanagements bedarf. Im Rahmen dieser Arbeit wird davon ausgegangen, dass dies aus IT-Sicht bedeutet, dass das Rechenzentrum oder der Anbieter bei dem sich ein verteiltes System befindet komplett ausgefallen ist und damit auch zu einem Absturzausfall des abzusichernden verteilten Systems führt. Der Notfall kann nur damit überwunden werden, dass das verteilte System in einem anderen Rechenzentrum als Notfallsystem bereitgestellt wird. Die Bereitstellung eines Notfallsystems ist hierbei auch der Fokus dieser Arbeit und andere Fehlerklassen werden explizit nicht behandelt.

In den Bereich der Störung fallen alle anderen Beeinträchtigungen des verteilten Systems und damit auch alle in Kapitel 2.2.2 beschriebenen Fehlerklassen. Selbst Aussturzausfälle, die das komplette verteilte System betreffen, aber innerhalb kurzer Zeit durch Maßnahmen des Störungsmanagements behoben werden können, befinden sich außerhalb des Fokus dieser Arbeit. Zwar könnte sich das Störungsmanagement entscheiden, zur Überbrückung der Störung das Notfallsystem zu nutzen, die in dieser Arbeit vorgestellten Methoden und automatisierten Verfahren beziehen sich jedoch ausschließlich auf einen Notfall nicht auf eine Störung.

4.2.3 Konsistenz

Das Thema Konsistenz wird speziell im Bereich der verteilten Systeme und Datenbanken sehr ausführlich diskutiert. Immer wenn Daten aus Performanz- oder Sicherheitsgründen redundant vorgehalten werden müssen, kommt die Frage auf, ob diese Daten auch konsistent gespeichert werden können. Während speziell im Feld der relationalen Datenbanken häufig eine strikte oder auch sequentielle Konsistenz [74] vorausgesetzt wird, so gibt es im Cloud Computing immer häufiger den Ansatz, dass die Daten innerhalb eines verteilten Systems nicht strikt konsistent sein müssen, son-

dern nur innerhalb eines gewissen Zeitfensters eine Konsistenz ausweisen müssen [13].

Im Rahmen der Methode zur Notfallwiederherstellung kann auch zwischen zwei verschiedenen Arten von Konsistenz unterschiedenen werden: Die Konsistenz, die ein Datensicherungsabbild aufweist, und die des Notfallsystems im Vergleich zum Primärsystem.

Datensicherungs-Konsistenz – Das Erstellen von Datensicherungen obliegt der Datensicherungs-Methode. Wie bereits in Kapitel 2.3 dargelegt, gibt es verschiedenste Arten der Datensicherung mit unterschiedlichen Vor- und Nachteilen. Auch gibt es für verschiedene Systeme (z.B. relationale Datenbanken) angepasste Datensicherungsmethoden, die sicherstellen, dass die Datensicherung in einer Art und Weise erstellt werden, in der sie auch problemlos rückgesichert werden können. Eines dieser Probleme könnte auftreten, wenn die Datensicherungs-Methode die Konsistenz der Datensicherung nicht beachtet. So könnte es passieren, dass offene Transaktionen in der Datensicherung gesichert werden und bei einer Rücksicherung ein inkonsistentes Abbild des ursprünglichen Systems geschaffen wird. In dieser Arbeit wird davon ausgegangen, dass die Datensicherungen, die über die bestehende Datensicherungsmethode angefertigt werden, auch koinzident sind, da sie sonst auch nicht im Rahmen anderer Rücksicherungsmaßnahmen zu gebrauchen wären. Es ist nicht Teil dieser Arbeit, dafür zu sorgen, dass diese Konsistenz sichergestellt wird, sondern es wird angenommen, dass die Datensicherungslösung es z.B. nicht zulässt, eine noch nicht abgeschlossene Datensicherung auf ein anderes System rückzusichern (siehe hierzu auch Abbildung 29). Unter Datensicherungs-Konsistenz versteht man also den konsistenten Zustand einer Datensicherung, die von einem verteilten System angefertigt wurde.

Notfallsystem-Konsistenz – Im Gegensatz zur Datensicherungs-Konsistenz, die rein durch die Methode zur Datensicherung beeinflusst

wird, hat die hier vorgestellte Methode zur Notfallwiederherstellung einen Einfluss auf die Konsistenz der Daten, die vom Primär- auf das Notfallsystem gespiegelt werden. Wie in Kapitel 2.1.3 beschrieben, wird bei der Notfallwiederherstellung immer auch eine Redundanz hergestellt und damit stellt sich unweigerlich die Frage nach Konsistenz. So ist es speziell bei einem Warm-Standby-Ansatz sehr wahrscheinlich, dass bei einem Ausfall des Primärsystems offene Transaktionen (z.B. Verkaufsprozesse) existieren, die mit dem Notfallsystem nicht zu Ende gebracht werden können, weil die Daten auf dem Notfallsystem zu alt sind. Die Frage nach der Notfallsystem-Konsistenz ist also auch eng mit dem RPO verbunden. Das RPO gibt nämlich vor, wie viele Daten bei einem Ausfall verloren gehen dürfen und damit auch, wie inkonsistent das System aus Sicht den Kunden sein darf. Möchte man, dass aus Sicht des Kunden gar keine Inkonsistenzen auftreten, so ist das Verfahren der Warm-Standby-Notfallwiederherstellung nicht geeignet. Es müsste auf ein Hot-Standby oder sonstige Verfahren zur Vermeidung von Inkonsistenzen ausgewichen werden. Unter Notfallsystem-Konsistenz versteht man also den möglichen inkonsistenten Zustand, den ein Standby-System einnehmen kann, nachdem es wiederhergestellt ist, aber auf Nutzerseite noch Transaktionen offen sind.

Nachdem nun sowohl die Anforderungen als auch die zugrundeliegenden Annahmen beschrieben wurden, werden im Folgenden der konkrete Notfallwiederherstellungsprozess und das Aktualisierungsprotokoll beschrieben.

4.3 Notfallwiederherstellungsprozess

Soll ein Notfallsystem nach einem großen Ausfall in Betrieb genommen werden, so sind die Möglichkeiten, die zur Verfügung stehen, davon abhängig, welche Vorkehrungen getroffen wurden. Es kann nichts wiederher-

gestellt werden, was nicht zuvor per Datensicherung an einen sicheren Ort gebracht wurde. Auch hängt die Zeit, die benötigt wird, um ein Notfallsystem in Betrieb zu nehmen, stark davon ab, inwieweit die Standby-Ressourcen vorbereitet wurden, die Prozesse automatisiert sind und wo die Daten in welchem Format vorgehalten werden. Der Prozess der Inbetriebnahme setzt sich zusammen aus:

1. Beschaffung und Deployment der Server
2. Deployment der verteilten Anwendung
3. Rückspielen der letzten Datensicherung

In der Literatur werden häufig die Datensicherung und die Bereithaltung eines Notfallsystems vermischt oder synonym genutzt. In der Praxis werden die zwei Verfahren auch Hand in Hand eingesetzt und müssen aufeinander abgestimmt sein. So macht es wenig Sinn, ein Notfallsystem in Sekunden zur Verfügung zu haben, wenn die Daten nicht verfügbar oder sehr alt sind und sehr lange brauchen, um auf den neusten Stand gebracht zu werden.

Der Notfallwiederherstellungsprozess, wie in Abbildung 31 dargestellt, umfasst die Aufgaben der Überwachung der Primär-Cloud, die Initialisierung und Beendigung des Notbetriebs, sowie die Aktualisierung des Notfallsystems mit Hilfe des Aktualisierungsprotokolls.

Hierbei gibt es zwei nebenläufige Aktivitäten, die während der Laufzeit des Primärsystems durchgeführt werden müssen: Überwachung der Primär-Cloud und Aktualisierung des Notfallsystems. Diese beiden Aufgaben werden kontinuierlich durchgeführt und periodisch erneut gestartet, wie in Abbildung 31 durch ein Zeit-Event modelliert.

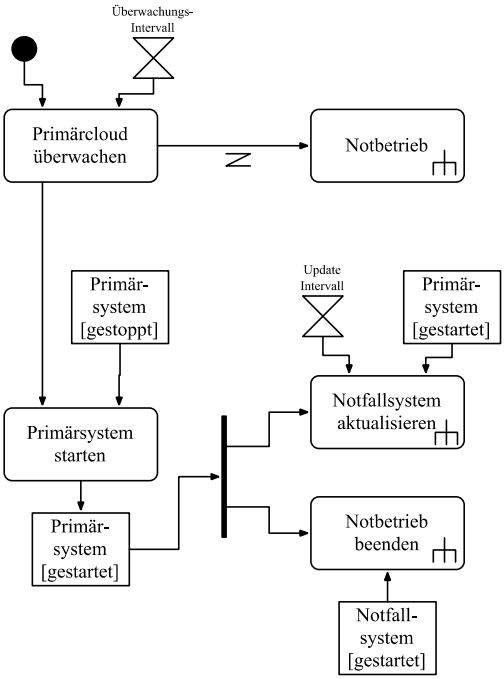

Abbildung 31: Notfallwiederherstellungsprozess (vgl. [86])

Gewisse Aktivitäten, wie z.B. die Aktualisierung des Notsystems, sind jedoch an weitere Bedingungen geknüpft. So darf sich das Notfallsystem nur aktualisieren, sofern sich das Primärsystem im Zustand „gestartet“ befindet. Dies wird im UML-Aktivitätsdiagramm dadurch verdeutlicht, dass „Notfallsystem aktualisieren“ nur dann ausgeführt werden kann, wenn zusätzlich der Objektknoten „Primärsystem“ den Zustand „gestartet“ hat. Ist das Primärsystem gestoppt so kann auch nach Ablauf des „Update Intervalls“ die Aktivität nicht ausgeführt werden.

Die Aktivität kann also nur dann ausgeführt werden, wenn das Primärsystem läuft. Sollte dies nicht der Fall sein, ist der gesamte Prozess im

Status „Notbetrieb“ und das Notfallsystem hat den Produktivbetrieb übernommen. Falls das Primärsystem gestoppt ist, so wird es, wenn die Primärcloud wieder verfügbar ist, automatisch gestartet. Das gesamte System befindet sich nun wieder im Normalbetrieb und die Aktualisierung kann ganz normal durchgeführt werden. Tritt bei der Überwachung der Primärcloud jedoch ein Fehler auf, d.h. ist diese nicht erreichbar, so wird in den Notbetrieb übergegangen. Im Folgenden werden die einzelnen Aktivitäten im Detail beschrieben.

4.3.1 Notbetrieb starten

Der Notbetrieb wird gestartet, wenn ein Notfall eingetreten ist. Die Primärcloud ist nicht mehr verfügbar und damit ist der zu schützende Geschäftsprozess unterbrochen. Nun muss das Notfallsystem so schnell wie möglich gestartet werden, damit es den Produktivbetrieb übernehmen kann. Das Anstarten des Systems wird von der Aktivität „Notbetrieb starten“ durchgeführt.

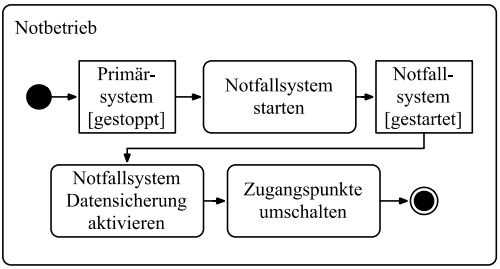

Abbildung 32: Aktivität „Notbetrieb starten“ (vgl. [86])

Der Notbetrieb ist dadurch gekennzeichnet, dass das Primärsystem gestoppt ist (siehe Abbildung 32). Zunächst wird das Notfallsystem gestartet und in den produktiven Zustand überführt. Das heißt, alle Zugriffe von

außen landen nun beim Notfallsystem, welches die Datensicherung in die zentrale Datenbank durchführt, sodass das RPO in der Zeit des Notbetriebs nicht verletzt wird.

4.3.2 Notbetrieb beenden

Ist der Notfall vorüber, das heißt die Primärcloud ist wieder verfügbar und das Primärsystem läuft wieder, so kann der Notbetrieb beendet werden. Die Aktivität „Notbetrieb beenden“ wird aus dem übergeordneten Notfallwiederherstellungsprozess (siehe Abbildung 31) direkt nachdem das Primärsystem gestartet wurde, ausgeführt.

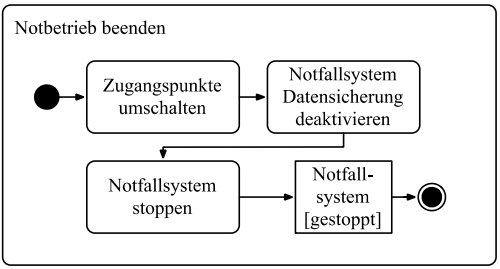

Abbildung 33: Aktivität „Notbetrieb beenden“

Ist die Primärcloud wieder verfügbar und das Primärsystem gestartet, so wird der Notbetrieb kontrolliert beendet (siehe Abbildung 33). Beim Start des Primärsystems werden die Zugangspunkte wieder auf dieses übertragen. Die Datensicherung läuft ab diesem Zeitpunkt auch wieder von dem Primärsystem in die gemeinsame Datenbank. Am Ende des Prozesses wird das Notfallsystem gestoppt und wartet von da an auf seine nächste Aktualisierung.

4.3.3 Notfallsystem aktualisieren

Die Aktualisierung des Notfallsystems ist eine zentrale Aktivität innerhalb des Notfallwiederherstellungsprozesses. Sie wird periodisch nach dem Ablauf des „Update Intervalls“ gestartet. Nach der Häufigkeit, in der das Notfallsystem aktualisiert wird, richtet sich auch die kleinstmögliche Deploymentzeit (siehe dazu auch Abbildung 34).

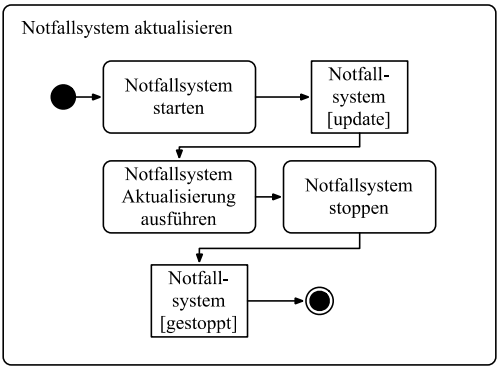

Abbildung 34: Aktivität „Notfallsystem aktualisieren“ (vgl. [86])

Die Aktualisierung des Systems kann nur gestartet werden, wenn das Primärsystem läuft, d.h. das Gesamtsystem sich im Normalbetrieb befindet. Hierbei wird zunächst das Notfallsystem auf Basis der formalen Beschreibung koordiniert und von der Deploymentsteuerung angestartet, um dann die letzte Datensicherung auf das Notfallsystem zurückzuspielen. Befindet sich das Notfallsystem auf dem aktuellen Stand und sind sowohl die neuste Version der Konfigurationsdaten als auch der Geschäftsdaten aufgespielt, so werden die Daten im Dateisystem persistiert und das System gestoppt. Beim nächsten Starten des Notfallsystems befindet es sich im gleichen Zustand wie beim letzten Stoppen.

4.4 Aktualisierungsprotokoll

Das Cloud Standby Aktualisierungsprotokoll setzt auf die bestehende Datensicherung auf und nutzt diese als Datenquelle, um das Notfallsystem im Rahmen der Daten-Replikation aktuell zu halten. Eine schematische Darstellung des Aktualisierungsprotokolls findet sich in Abbildung 35.

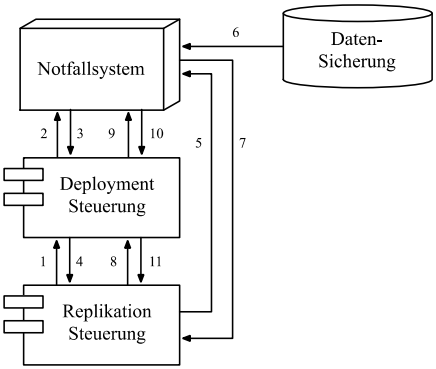

Abbildung 35: Aktualisierungsprotokoll

Im ersten Schritt ruft der Cloud Standby Controller die API des Cloud Managers auf und teilt diesem mit, dass alle am Notfallsystem beteiligten VMs gestartet werden sollen. Der Cloud Controller hat dabei auf die Abhängigkeiten zwischen den Komponenten des verteilten Systems zu achten und daraus eine Startreihenfolge abzuleiten. Anschließend setzt der Cloud Manager in Schritt 2 die Startbefehle Cloud-intern um und startet die einzelnen virtuellen Maschinen. Er überprüft weiterhin im 3. Schritt, wann alle Maschinen gestartet sind und gibt dies in Schritt 4 an den Cloud Standby Controller zurück. Ist das gesamte Notfallsystem gestartet, so beginnt der Cloud Standby Controller auf den einzelnen VMs im 5. und 6. Schritt die letzte Datensicherung einzuspielen. Ist das Notfallsystem nun

auf dem neusten Stand, wird dies der Replikationssteuerung im 7. Schritt gemeldet, die mit den Schritten 8 bis 11 das Notfallsystem wieder stoppen.

4.5 Entscheidungsunterstützung zur Wahl des Update-Intervalls

Nachdem gezeigt wurde, welche Prozesse und Protokolle an der Methode zur Notfallwiederherstellung beteiligt sind, stellt sich die Frage, wie genau Update-Intervall zu wählen ist. Die Wahl eines „guten“ Update-Intervalls ist hierbei nicht leicht zu treffen. Das Intervall hat einen großen Einfluss auf das erreichbare RTO und bestimmt die Kosten für die Absicherung. Wird das Intervall zu groß gewählt, so ist es im Notfall vielleicht nicht möglich, das Notfallsystem in der vorgegebenen Zeit zu starten (es müssen zu viele Daten transferiert und integriert werden). Wir das Intervall zu klein gewählt, so entstehen unnötige Kosten, da das Notfallsystem zu häufig aktualisiert wird, was nicht nötig wäre, da auch mit weniger Aktualisierungsvorgängen das RTO zu erfüllen wäre. Die Entscheidung wie ein Update-Intervall gewählt werden muss, hängt dabei von Faktoren wie dem RTO oder der Änderungsrate der Daten ab. Eine Entscheidungsunterstützung, wie ein Update-Intervall zu wählen ist, wird in dieser Arbeit im Rahmen der Evaluierung vorgestellt. Im Folgenden wird erläutert, wie ein bestehendes verteiltes System zu untersuchen ist, wie die Kosten berechnet werden und wie mittels einer Langzeitsimulation (unter Berücksichtigung von Ausfallkosten) ein kostenneutrales Update-Intervall zu wählen ist. Die hier vorgestellten Verfahren zur Bestimmung des Update-Intervalls werden in Kapitel 7 auf einen konkreten Anwendungsfall angewandt um diese Arbeit zu evaluieren.

4.5.1 Auf Basis des RTO

Die Wahl des Update-Intervalls hängt immer von dem konkreten Anwendungsfall ab und ist so zu wählen, dass hierdurch das vorgegebene RTO eingehalten wird. Im Folgenden wird eine auf Beobachtung oder Experimenten basierende Bestimmung des Verhältnisses von Update-Intervall und RTO beispielhaft gezeigt. Eine vollständige experimentelle Bestimmung anhand eines konkreten Anwendungsfalls wird in Kapitel 7.2 durchgeführt. In Abbildung 36 sind verschiedene Beobachtungspunkte abgetragen, aus denen sich erkennen lässt, wie sich die Deploymentdauer des verteilten Systems über die Zeit entwickelt.

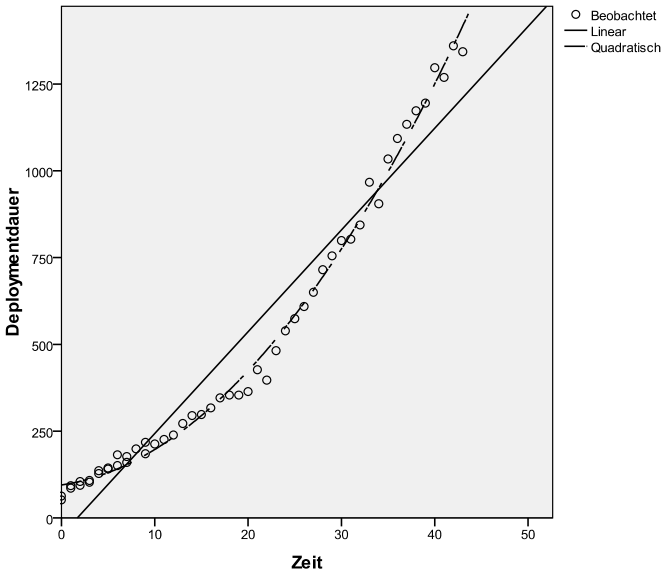

Abbildung 36: Beispielhafte Kurvenanpassung zur Bestimmung der Deploymentzeit-Funktion in Abhängigkeit von der verstrichenen Zeit

Diese Beobachtungen wurden mit Hilfe von Experimenten erhoben, indem verschieden alte Vollsicherungen (siehe Kapitel 2.3.1) rückgesichert wurden und die Zeit für die Rücksicherung gemessen wurde. Mit Hilfe einer Kurvenanpassung wurde ein lineares und quadratisches Modell getestet. Wie man leicht sehen kann, ist das Verhalten der Deploymentdauer in Abschnitt, im dem die Experimente durchgeführt wurden, quadratisch. Es ergibt sich also eine stetige Funktion, die eine Aussage darüber macht, wie sich die Deploymentzeit im Laufe der Zeit entwickelt. Das RTO sagt aus, was die erlaubte Dauer ist, bis das Notfallsystem bereitstehen muss. Hier ist also die Deploymentdauer das minimal mögliche RTO. Skaliert man nun noch mit dem Wissen, wie lange die einzelnen Vollsicherungen auseinander lagen, die Zeit-Achse auf Minuten und bildet die Umkehrfunktion, so erhält man eine Abschätzung für die Deploymentzeit:

$$\mathrm{t}_{Depl}\left(t_{UpdateInt}\right) \leq \ RTO$$

Für das in Abbildung 36 dargestellte Beispiel sieht die Funktion $\mathrm{t}_{UpdateInt}(RTO)$ wie in Abbildung 37 aus.

Ein derartiges Vorgehen ermöglicht es, eine Funktion der Deploymentzeit und des RTO in Abhängigkeit vom Update-Intervall für ein konkretes System zu bestimmen. Diese Funktionen dienen im Folgenden der Kostenberechnung und bei der Bestimmung eines sinnvollen Update-Intervalls als Grundlage. Eine detaillierte Beschreibung des konkreten Vorgehens an Hand eines Anwendungsfalls ist in Kapitel 7.2.2 beschrieben.

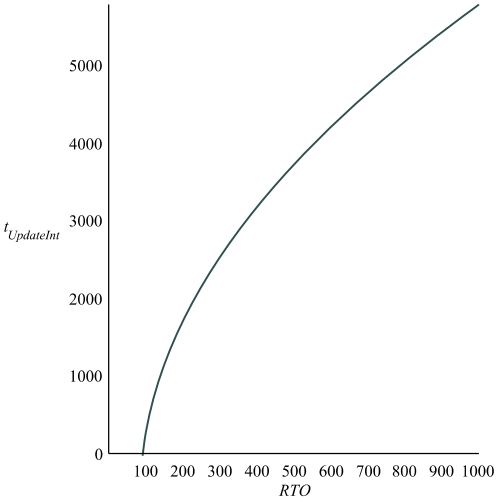

Abbildung 37: Beispielhafte Darstellung von $t_{UpdateInt}(RTO)$

4.5.2 Auf Basis von Kosten

Die Wahl des Update-Intervalls hat einen großen Einfluss darauf, wie schnell ein Notfallsystem zur Verfügung gestellt werden kann. Je geringer die Vorgabe des RTO ist, umso häufiger müssen Updates durchgeführt werden und damit auch das Notfallsystem für Aktualisierungen gestartet werden. Diese Startvorgänge und damit auch die Aktualisierungen verursachen Kosten. Die zusätzlichen Kosten für Cloud Standby setzen sich zusammen aus der Summe der Laufzeitkosten der virtuellen Maschinen $c_{C2,i}^{runtime}$, den Speicherkosten für das Vorhalten der Images $c_{C2}^{storage}$ und den Netzwerkkosten. Da sich im Cloud Standby System das Backup und das Notfallsystem im gleichen Rechenzentrum befinden und die Netzwerkkosten bei den großen Anbietern nicht berechnet werden, werden diese Kosten vereinfacht mit Null angenommen. Damit ergibt sich die folgende Formel, die im Weiteren für die Kostenbetrachtung genutzt wird:

$$cost = \sum_{i}^{n_{server}} c_{C2,i}^{runtime} \cdot \left\lceil \frac{t_{update}}{60} \right\rceil \cdot \frac{365{,}25 * 24}{\frac{t_{updateInt}}{60}} + \sum_{i}^{n_{server}} c_{C2}^{storage} \cdot size_i \cdot 12$$

Da die Abrechnung der virtuellen Maschinen bei den großen Anbietern stündlich erfolgt, muss bei jedem Update-Zyklus die Update-Dauer (t_{update}) mit der Aufrundungsfunktion $\lceil \cdot \rceil$ auf die volle Stunde aufgerundet werden. In diesem Kapitel wird die Methode zur Berechnung der Kosten detailliert vorgestellt und in Kapitel 7.2.3 die Kosten der Absicherung für einen konkreten Anwendungsfall analysiert. Beispielhaft sind die Kosten in Abbildung 38 dargestellt.

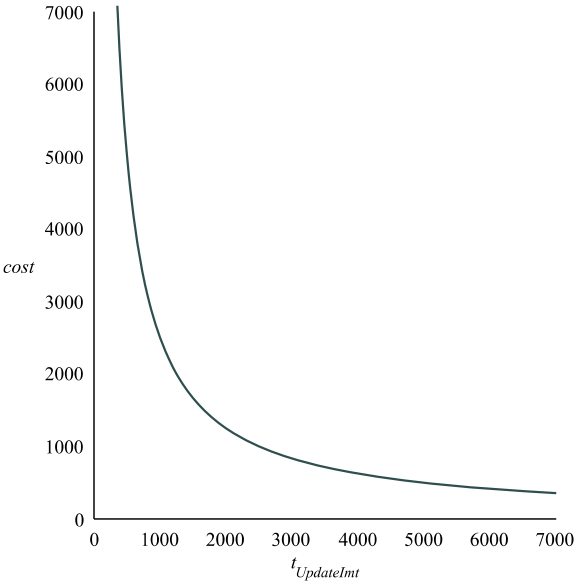

Abbildung 38: Beispielhafte Darstellung der Kosten für die Absicherung mit Cloud Standby

Über die Untersuchung des Startverhaltens des Primärsystems bei verschieden alten Datensicherungen und der Berechnung der Kosten auf Basis des Update-Intervalls kann nun bereits eine informierte Entscheidung getroffen werden, was für ein Update-Intervall gewählt werden soll.

Gerade bei Standby-Ansätzen ist es jedoch interessant, auch die Kosten auf lange Zeit hin zu untersuchen. Diese Arbeit konzentriert sich auf Notfälle und gemäß des Fehlermodells aus 4.2.2 ist das der schwerwiegende, längere Ausfall eines ganzen Rechenzentrums. Solche Ereignisse finden lediglich sehr selten statt und es stellt sich daher auch die Frage, ob man ein System überhaupt gegen diese Art von Ausfällen absichern oder einen solchen Ausfall nicht einfach hinnehmen sollte. Hierzu liefert diese Arbeit eine auf Simulation basierende Entscheidungsunterstützung, die unter Berücksichtigung der Ausfallkosten ($cost_e$) eines Geschäftsprozess die langfristigen Kosten ermittelt. Mit Hilfe dieser Entscheidungsunterstützung lassen sich nun Bereiche identifizieren, in denen das Update-Intervall beliebig gewählt werden kann und die Absicherung günstiger ist, wenn man davon ausgeht, dass der Ausfall des Geschäftsprozesses Kosten verursacht. Beispielhaft ist dies in Abbildung 39 dargestellt und wird detailliert in Kapitel 7.3 anhand eines realistischen Anwendungsfalls illustriert.

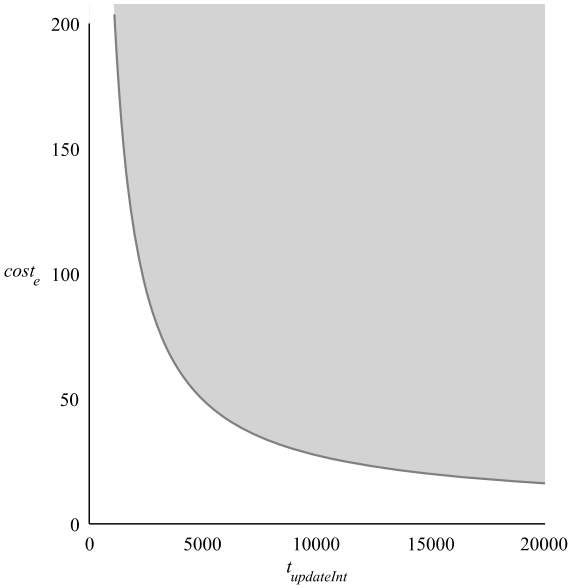

Abbildung 39: Beispielhafte Darstellung für Bereiche in denen Cloud Standby günstiger ist (grau)

Sind die Ausfallkosten auf Basis der als Teil von BCM durchgeführten BIA (siehe Kapitel 2.1.4) bekannt, so kann mittels der hier vorgestellten Simulation das Update-Intervall ermittelt werden, indem die langfristigen Kosten mit Absicherung gleich den langfristigen Kosten ohne Absicherung sind. Auch dieser Zusammenhang ist zunächst beispielhaft in Abbildung 40 dargestellt und wird im weiteren Verlauf dieses Kapitels detailliert erklärt.

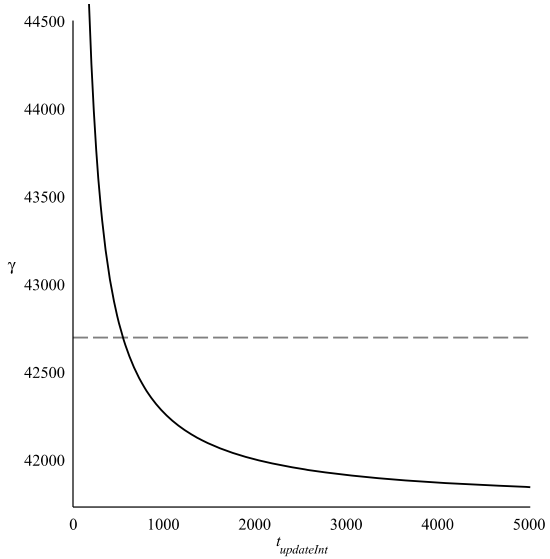

Abbildung 40: Beispielhafte Darstellung der Gesamtkosten mit (druchgezogene Linie) und ohne Absicherung (gestrichelte Linie) und dem kostenneutralen Update-Intervall im Schnittpunkt

Die Berechnung der Qualitätseigenschaften, wie Kosten oder Verfügbarkeit für ein Standby-System und der Vergleich mit einem System ohne Cloud Standby (Standardsystem), stellt eine wichtige Entscheidungsgrundlage sowohl bei der Einführung als auch der Wahl des Update-Intervalls[12] dar. Aufgrund der Struktur und der Natur von Standby-Systemen ist diese Berechnung allerdings nicht trivial. So haben Standby-Systeme verschiedene Zustände (siehe Abbildung 28) und in jedem Zustand fallen andere Kosten, wie die Laufzeitkosten, Ausfallkosten und so weiter an. Außerdem erweist

[12] Das folgende Kapitel ist bereits in den Arbeiten [84], [85] veröffentlicht.

sich die Bestimmung der Systemqualität aufgrund der langen Zeiträume und der möglichst geringen Ausfallwahrscheinlichkeiten (z.B. lediglich ein Notfall alle zehn Jahre) als schwierig. Eine rein experimentelle Ermittlung durch die jahrzehntelange Beobachtung eines Referenz-Systems ist nicht möglich. Es wird also eine Methode zur Berechnung der Qualitätseigenschaften über einen längeren Zeitraum benötigt.

Im Folgenden wird hierzu eine auf der Theorie der Markov-Ketten basierende Methode zur Berechnung vorgestellt[13]. Die Grundidee ist, dass über einen Zustandsgraphen ein „Random Walk", also ein zufälliges Abschreiten des Graphen gemäß der Übergangswahrscheinlichkeiten, durchgeführt wird. Lässt man die Schrittzahl des Random Walks gegen unendlich gehen, so kann berechnet werden, mit welcher Wahrscheinlichkeit jeder Zustand im Durchschnitt besucht wird. Aus dieser Wahrscheinlichkeitsverteilung für den Aufenthalt in jedem Zustand können dann die Kosten und die Verfügbarkeit berechnet werden. Um diese Berechnung der Qualitätseigenschaften überhaupt möglich zu machen, müssen für die Berechnung zunächst einige Variablen definiert und parametrisiert werden. Einige der Parameter sind dabei experimenteller Herkunft, andere können externen Quellen entnommen werden und wieder andere müssen geschätzt werden.

Speziell für die Berechnung der Zeiten und Kosten ist die Struktur des verteilten Systems eine wichtige Grundlage. Hierbei sind vor allem die Anzahl der Server (n_{server}) und die Kosten der Notfallcloud C2. Erst für einen konkreten Anwendungsfall (siehe Kapitel 7.1) lassen sich Startzeiten in Abhängigkeit von dem Update-Intervall berechnen. Tabelle 5 stellt die zu parametrisierenden Zeitvariablen sowie die hier zugrunde gelegte Quelle für deren Parametrisierung dar.

[13] Für die Umsetzung der Berechnung in Maple, siehe Anhang A.1

Tabelle 5: Temporale Parameter

Typ	Variable	Einheit	Quelle
Dauer des initialen Deployments	t_{depl}	Minuten	Experiment
Update-Intervall	$t_{updateInt}$	Minuten	Vorgabe
Updatedauer	t_{update}	Minuten	Experiment
Dauer des Notfallsystem-Deployments	$t_{replica}$	Minuten	Experiment
Dauer eines Notfalls	t_{error}	Minuten	Annahme/Historisch

Zur Berechnung der Gesamtkosten müssen auch die Kosten für die Laufzeit der einzelnen Server bekannt sein. Diese Daten lassen sich den Angeboten der Cloud-Anbieter entnehmen. Es ist dabei zu beachten, dass sich über einen so langen Betrachtungszeitraum die Kosten der Anbieter verändern werden, die Inflation eine Rolle spielt und so weiter. Da jedoch in dem hier vorgestellten Verfahren immer zwei Systeme verglichen werden und beide Varianten auf Cloud Computing basieren, so fallen diese Kosteneffekte nicht so sehr ins Gewicht, da sie in beiden Systemen anfallen. Dem Anwender des Verfahrens sollte diese Tatsache jedoch bewusst sein, wenn es darum geht absolute Zahlenwerte zu berechnen. Hierfür ist es notwendig nicht die realen Kosten der Cloud Anbieter zum aktuellen Zeitpunkt zu wählen, sondern die durchschnittlich zu erwarteten Kosten. Diese Berechnung sollte durch Experten aus dem Feld der Ökonomie durchgeführt werden und ist nicht Teil dieser Arbeit.

Zusätzlich ist für die Kostenbetrachtung wichtig zu wissen, wie viel Kosten/entgangene Gewinne dem Unternehmen entstehen, wenn das System nicht verfügbar ist. Alle in der folgenden Betrachtung berücksichtigten Kostenarten sind in Tabelle 6 zusammengestellt. Auch die Berechnung dieser Kosten ist Teil ökonomischer Berechnungen wie sie in der BIA durchgeführt werden und daher nicht Teil dieser Arbeit.

Tabelle 6: Monitäre Parameter

Typ	Variable	Einheit	Quelle
Primärer Cloud-Provider Kosten	$cost_1$	Euro/h/Server	Angebot
Sekundärer Cloud-Provider Kosten	$cost_2$	Euro/h/ Server	Angebot
Nichtverfügbarkeit Kosten	$cost_e$	Euro/h	Annahme/ Historisch

Die Verfügbarkeit der Cloud-Provider ist eine wichtige Grundlage für die Berechnung der Gesamtverfügbarkeit des Systems und damit auch der Kosten. Viele Cloud-Provider haben eine Verfügbarkeit in ihren SLA angegeben. Diese Verfügbarkeit ist jedoch im Kontext dieser Berechnung weniger interessant, da es im Rahmen dieser Arbeit nur um globale, längerfristige Ausfälle aufgrund von Katastrophen geht. Solch globale Ausfälle sind in der Regel sogar wesentlich seltener als die in den SLA vereinbarten Zahlen. Eine konservative Schätzung geht also von weitaus höheren Verfügbarkeiten aus als in den SLA angegeben. Es können natürlich aber auch die Daten aus den SLA direkt übernommen werden. Die in Tabelle 7 beschriebene Verfügbarkeit gibt den Zeitraum an, in dem durchschnittlich genau ein globaler Ausfall zu erwarten ist:

Tabelle 7: Verfügbarkeit der Cloud-Provider

Typ	Variable	Einheit	Quelle
Primärer Cloud-Provider Verfügbarkeit	$avail_1$	Jahre	Historisch/Annahme
Sekundärer Cloud-Provider	$avail_2$	Jahre	Historisch/Annahme

Aus den verschiedenen Zuständen des Zustandsdiagramms (siehe Kapitel 4, Abbildung 28) lassen sich Zustände für einen Zustandsgraphen, der die Grundlage für die weiteren Berechnungen darstellen soll, direkt ableiten. Dabei wird noch ein Fehlerzustand eingeführt, in den das System gelangt, wenn sowohl der Primäre als auch der Sekundäre Cloud-Anbieter ausgefallen sind.

Tabelle 8: Bezeichnung der Zustände aus den Prozessschritten

System-Zustand	Modell-Zustand
Primärsystem Deployment	S_1
Primärsystem Laufzeit	S_2
Primärsystem Laufzeit + Notfallsystem Update	S_3
Notfallsystem Deployment	S_4
Notfallsystem Laufzeit	S_5
Notfallsystem Laufzeit + Primärsystem Update	S_6
Fehler	S_7

Dabei entspricht S_i der Bezeichnung für den Zustand i aus dem Zustandsraum I. Um die Qualitätseigenschaften des Systems berechnen zu können, muss jedem der Zustände eine Aufenthaltsdauer zugeordnet werden. Dabei wird angenommen, dass der Takt der Markov-Kette eine Minute beträgt. Es sei $d_i \; \forall \; i \in I$ die Aufenthaltsdauer in einem Zustand S_i, mit:

$$d := \begin{bmatrix} t_{initDepl} \\ t_{updateInt} \\ t_{update} \\ t_{Depl}\left(t_{updateInt}\right) \\ t_{error} - t_{Depl}\left(t_{updateInt}\right) - \; t_{initDepl} \\ t_{initDepl} \\ t_{error} \end{bmatrix}$$

Es wird deutlich, dass sich alle Aufenthaltsdauern außer d_2, d_4 und d_5 mit den zuvor festgelegten Parametern bestimmen lassen. Das Update-Intervall $t_{updateInt}$ ist Teil der Konfiguration und hat einen großen Einfluss auf die Kosten und die Verfügbarkeit des Systems. Die Zeit, die für das Hochfahren des Notfallsystems benötigt wird (d_4), hängt stark davon ab, wann auf dmr Server das letzte Mal eine Aktualisierung durchgeführt wurde. Ist also das Update-Intervall sehr groß, vergrößert sich auch die Zeit für das Anstarten des Notfallsystems. Dieser Zusammenhang hat zur Folge, dass bei einer Erhöhung des Backup-Intervalls eine Reduktion der Deploymentzeit

zu erwarten ist. Daraus ergibt sich wiederrum, dass die Funktion $t_{Depl}(t_{updateInt})$ monoton steigend ist. Für d_5 und d_7 wird angenommen, dass die Zeit t_{error} unabhängig vom Einsatz eines Notfallsystems konstant ist. Damit setzt sich die Laufzeit des Notfallsystems aus der Ausfallzeit abzüglich der Notfallsystem-Deploymentzeit (d_4) und der Zeit für den Rückkehr in das Primärsystem (d_6) zusammen.

Die Qualitätseigenschaften des Replikationssystems lassen sich mit Hilfe der Modellierung der Zustände als Markov-Kette und einer Langzeitbetrachtung der Aufenthaltswahrscheinlichkeiten in den Zuständen *S* berechnen. Aufgrund der Gedächtnislosigkeit der Markov-Kette (Markov-Eigenschaft) ist es jedoch nicht möglich, die Aufenthaltsdauern direkt zu modellieren. Eine Möglichkeit dies indirekt zu tun, ist die Aufenthaltsdauern in Rückkehrwahrscheinlichkeiten zu überführen. Diese müssen so gestaltet sein, dass im Durchschnitt in d_i der Fälle der Zustand beibehalten wird und in einem Fall der Zustand verlassen wird. Daraus folgt, dass die Gesamtzahl der möglichen Fälle $d_i + 1$ beträgt. Damit ist die Rückkehrwahrscheinlichkeit $\lambda_i \; \forall \; i \in I$ zu berechnen:

$$\lambda_i = \frac{d_i}{d_i + 1} \forall \; i \in I$$

Neben den Rückkehrwahrscheinlichkeiten werden außerdem die Wahrscheinlichkeiten für einen Ausfall benötigt. Diese werden analog zu der Rückkehrwahrscheinlichkeiten berechnet. Im Schnitt soll, normiert auf den Iterationsschritt der Markov-Kette von einer Minute, einmal im Zeitraum von $avail_i, i \in \{1, 2\}$ ein Ausfall stattfinden:

$$\varepsilon_i = \frac{1}{avail_i * 365 * 24 * 60} \; , i \in \{1,2\}$$

Mit Hilfe der Wahrscheinlichkeiten lässt sich nun die Markov-Kette MK_1 für Cloud Standby, wie in Abbildung 41 dargestellt, aufstellen.

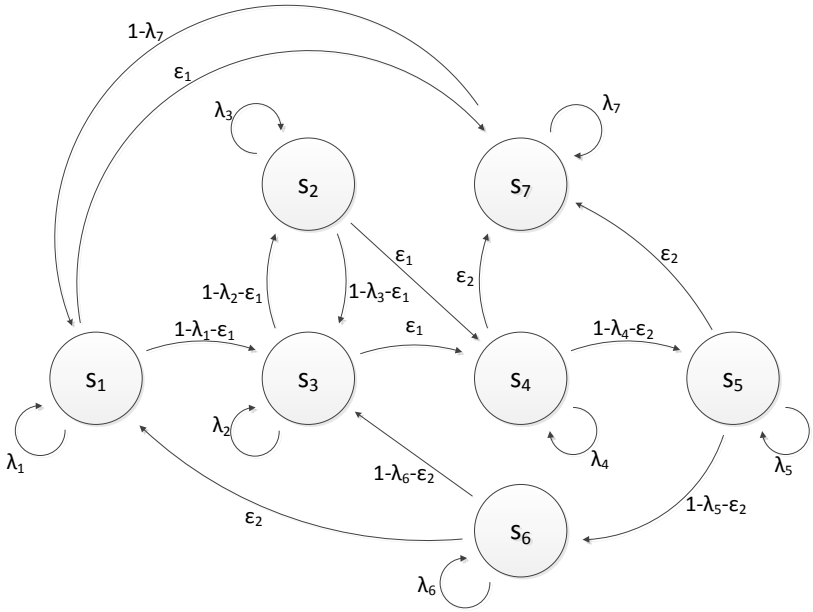

Abbildung 41: Markov-Kette für Cloud Standby (MK_1) [84], [85]

Aus der Markov-Kette MK_1 kann direkt die Übergangsmatrix P_1, wie in Abbildung 42 dargestellt, abgelesen werden.

$$\begin{pmatrix} \lambda_1 & 1-\lambda_1-\varepsilon_1 & 0 & 0 & 0 & \varepsilon_1 & 0 \\ 0 & \lambda_2 & 1-\lambda_2-\varepsilon_1 & \varepsilon_1 & 0 & 0 & 0 \\ 0 & 1-\lambda_3-\varepsilon_1 & \lambda_3 & \varepsilon_1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \lambda_4 & 1-\lambda_4-\varepsilon_2 & 0 & \varepsilon_2 \\ 0 & 0 & 0 & 0 & \lambda_5 & 1-\lambda_5-\varepsilon_2 & \varepsilon_2 \\ 0 & 1-\lambda_6-\varepsilon_2 & 0 & 0 & 0 & \lambda_6 & \varepsilon_2 \\ 1-\lambda_7 & 0 & 0 & 0 & 0 & 0 & \lambda_7 \end{pmatrix}$$

Abbildung 42: Übergangsmatrix P_1 zu MK_1[84], [85]

Da am Ende die Eigenschaften von Cloud Standby mit denen des ursprünglichen Systems verglichen werden sollen, müssen im nächsten Schritt die Markov-Kette MK_2 und die Übergangsmatrix P_2 als Referenz für das System ohne Cloud Standby erstellt werden. Die beiden Ketten unterscheiden sich darin, dass kein Update durchgeführt wird, also $t_{updateInt} \rightarrow \infty$ geht, die Aufenthaltsdauern in den Zuständen $S_3 - S_6$ gleich null sind und kein zweiter Anbieter vorhanden ist, die Ausfallwahrscheinlichkeit ε_2 also 1 entspricht. Wendet man diese Parameter auf MK_1 an, so sind die Zustände S_5 und S_6 nicht mehr erreichbar. Der Zustand S_4 geht mit der Wahrscheinlichkeit von 1 also unmittelbar in S_7 über und kann daher mit S_7 zusammengefasst werden. Da das Update-Intervall unendlich groß ist, geht die Rückkehrwahrscheinlichkeit von S_2 gegen 1[14]. Dies ergibt auch eine negative Übergangswahrscheinlichkeit S_2 zu S_3. Da die Rückkehrwahrscheinlichkeit von S_3 jedoch null ist, lässt diese negative Übergangswahrscheinlichkeit durch das Zusammenfassen der Knoten S_2 und S_3 zu S_2 auflösen. Für S_2 ergibt sich damit eine neue Rückkehrwahrscheinlichkeit von $1 - \varepsilon_1$.

[14] $\lim_{t_{updateInt} \rightarrow \infty} \lambda\left(t_{updateInt}\right) = \lim_{t_{updateInt} \rightarrow \infty} \frac{t_{updateInt}}{t_{updateInt}+1} = 1$

Die daraus resultierende Markov-Kette MK_2 ist in Abbildung 43 dargestellt.

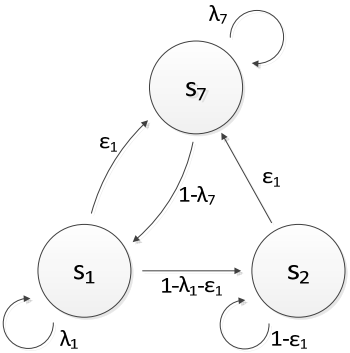

Abbildung 43: Markov-Kette ohne Cloud Standby (MK_2) [84], [85]

Die Übergangsmatrix P_2wird analog zu P_1 als $\mathbb{R}^{7x7}$ Matrix erstellt, sodass die gleichen Algorithmen auf beide Matrizen anwendbar sind. Die Übergänge von und nach den Zuständen $S_3 - S_6$ haben dabei eine Wahrscheinlichkeit von null. Die Matrix P_2 ist in Abbildung 44 dargestellt.

$$\begin{pmatrix} \lambda_1 & 1-\lambda_1-\varepsilon_1 & 0 & 0 & 0 & 0 & \varepsilon_1 \\ 0 & 1-\varepsilon_1 & 0 & 0 & 0 & 0 & \varepsilon_1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1-\lambda_7 & 0 & 0 & 0 & 0 & 0 & \lambda_7 \end{pmatrix}$$

Abbildung 44: Übergangsmatrix P_2 zu MK_2 [84], [85]

Um zu einer Langzeitbetrachtung des Systems zu gelangen, wird die stationäre Verteilung für die Markov-Kette $MK_l,\ l \in \{1,2\}$ berechnet. Diese

Verteilung $\pi_i, i \in I$ besagt, mit welcher Wahrscheinlichkeit das System sich zu jedem gegebenen Zeitpunkt $n \in \mathbb{N}$ in dem Zustand $S_i, i \in I$ befindet. Mit Hilfe der Wahrscheinlichkeitsverteilung können langfristige Qualitätseigenschaften, wie die Kosten γ oder die Gesamtverfügbarkeit α berechnet werden. Der Algorithmus zur Ermittlung der stationären Verteilung ist im Folgenden verkürzt dargestellt[15]. Dabei sei E_r die Einheitsmatrix und b_r der Einheitsvektor mit dem Rang r. Sei

$$Q = P - \mathrm{E}_7$$

und Q' die transponierte Matrix zu Q

$$Q' = \begin{pmatrix} q_{1,1} & \cdots & q_{1,7} & 1 \\ \vdots & \ddots & \vdots & \vdots \\ q_{7,1} & \cdots & q_{7,7} & 1 \end{pmatrix}^T$$

Mit der transponierten Matrix kann nun aus der Lösung des Gleichungssystems

$$Q' * \vec{\pi} = b_7$$

die stationäre Verteilung $\vec{\pi}$ berechnet werden. Diese Verteilung ist ein Vektor, in dem der Wert an der Stelle $\pi_i \in I$ die Wahrscheinlichkeit angibt sich in einem gegebenen Schritt n im Zustand S_i zu befinden.

[15] Eine ausführliche Beschreibung der Berechnung der stationären Verteilung kann dem Buch [129] entnommen werden.

Nachdem nun die stationäre Verteilung $\pi_i , i \in I$ bekannt ist, können hieraus die Kosten und Verfügbarkeit bestimmt und damit weitere Berechnungen durchgeführt werden, um das Update-Intervall zu bestimmen. Da die im Folgenden vorgestellten Berechnungsmethoden zur Bestimmung eines Update-Intervalls leichter zu verstehen sind, wenn sie auf einen konkreten Anwendungsfall angewandte werden, hilft es dem Verständnis beim Lesen der einzelnen Formeln zusätzlich, die konkreten Ergebnisse und Grafiken aus Kapitel 7.3 zu betrachten.

Langzeitkosten und Verfügbarkeit

Die Kosten ergeben sich aus der Summe der Kosten c_1 für die Primäre Cloud in den Zuständen S_1, S_2, S_3, S_6. Für die Sekundäre Cloud fallen die Kosten c_2 während des Updates und im Notbetrieb in S_3, S_4, S_5 und bei der Rückkehr über Zustand S_6 an. Die Kosten c_ε für die Nichtverfügbarkeit des Systems fallen in den Zuständen S_1, S_4, S_7 an.

$$\gamma \coloneqq c_1 n_{server} \sum_{i}^{i\in\{1,2,3,6\}} \pi_i + c_2 n_{server} \sum_{i}^{i\in\{3,4,5,6\}} \pi_i + c_\varepsilon \sum_{i}^{i\in\{1,4,7\}} \pi_i$$

Die Verfügbarkeit ergibt sich aus der Summe der Wahrscheinlichkeiten für die Zustände in denen das System verfügbar ist (S_2, S_3, S_4, S_6) oder aus der Gegenwahrscheinlichkeit für die Zustände, in denen das System nicht verfügbar ist (S_1, S_4, S_7):

$$\alpha \coloneqq \sum_{i}^{i\in\{2,3,5,6\}} \pi_i = 1 - \sum_{i}^{i\in\{1,4,7\}} \pi_i$$

Verhältnis von Ausfallkosten zum Update-Intervall

Um einen Vergleich der Gesamtkosten im Verhältnis zu den Ausfallkosten und Update-Intervall durchführen zu können, werden zunächst die Gesamtkosten mit variablen Ausfallkosten ($cost_e$) und Update-Intervall ($t_{updateInt}$) berechnet. Diese Gesamtkosten stellen sich dar als:

$$\gamma_i(cost_e, t_{updateInt}), i \in \{1,2\}$$

Mit Hilfe dieser variablen Kostenberechnungsfunktionen kann nun der Bereich gesucht werden, in dem die beiden Systeme die gleichen Kosten haben. Dies wird durch Schnitt der Funktionen erreicht:

$$cost_e(t_{updateInt}) := \gamma_1(cost_e, t_{updateInt}) \cap \gamma_2(cost_e, t_{updateInt})$$

Mit dieser Funktion können nun Grenzwertbetrachtungen durchgeführt werden. Grenzen für das Update-Intervall sind hierbei der Wert für kontinuierliche Updates und ein Update-Intervall gegen unendlich. Aufgrund der Kostenstruktur der Cloud-Provider (Abrechnungseinheit eine Stunde), ist die kontinuierliche Replikation gleichzusetzen mit einem Replikationsintervall von einer Stunde oder 60 Minuten:

$$cost_e^{min} := \lim_{t_{updateInt} \to \infty} cost_e\,(t_{updateInt})$$

$$cost_e^{max} := \lim_{t_{updateInt} \to 60} cost_e\,(t_{updateInt})$$

Außerhalb des Intervalls $[cost_e^{min}, cost_e^{max}]$ ist eine Cloud-Standby-Replikation, wie sie in dieser Arbeit beschrieben ist, nicht sinnvoll. Bei einer Verringerung der Kosten $cost_e^{min}$, ist der Einsatz von Cloud Standby

aus Kostengründen nicht sinnvoll. Beträgt das Update-Intervall 60 Minuten oder weniger, so fallen die gleichen Kosten an, wie beim Durchlaufen des Standby-Systems. In diesem Fall würde man jedoch direkt zu einem Hot-Standby-Ansatz übergehen, da dieser eine wesentlich höhere Verfügbarkeit verspricht.

Verhältnis der Verfügbarkeit zum Update-Intervall

Um ein Verhältnis zwischen Verfügbarkeit und Update-Intervall herzustellen, wird die Verfügbarkeit α als Funktion, die von $t_{updateInt}$ abhängig ist, dargestellt:

$$\alpha_i(t_{updateInt}), i \in \{1,2\}$$

Über diese Beziehung lässt sich das Intervall bestimmen, in dem das Replikationssystem eine Verfügbarkeit gewährleisten kann:

$$\alpha_1^{min} := \lim_{t_{updateInt} \to \infty} \alpha_1(t_{updateInt})$$

$$\alpha_1^{max} := \lim_{t_{updateInt} \to 60} \alpha_1(t_{updateInt})$$

Für den Fall ohne Cloud Standby ist die Verfügbarkeit unabhängig von $t_{updateInt}$:

$$\overline{\alpha}_2 = \alpha_2(t_{updateInt})$$

Da die Verfügbarkeitsfunktion $\alpha_1(t_{updateInt})$ konvex ist, gilt immer $\alpha_1^{min} < \alpha_1^{max}$. Zusätzlich gilt:

$$\overline{\alpha}_2 \leq \alpha_1^{min}$$

Dieser auf den ersten Blick überraschende Zusammenhang lässt sich dadurch erklären, dass im Fehlerfall ohne Cloud Standby direkt in den Zustand S_7 gewechselt wird, während bei Cloud Standby der Ausfall durch die Sekundäre Cloud überbrückt werden kann. Lediglich im Fall $t_{error} = 0$ gilt:

$$\overline{\alpha}_2 = \alpha_1^{min}$$

d.h. für $t_{error} > 0 \ \vee \ t_{updateInt} > 60$ gilt:

$$\overline{\alpha}_2 < \ \alpha_1(t_{updateInt})$$

Ist also anzunehmen, dass die Ausfallzeit $t_{error} > 0$ ist, so sollte im Sinne der Verfügbarkeit auf jeden Fall ein Replikationssystem eingesetzt werden, selbst wenn das Update-Intervall sehr groß gewählt wird.

Bestimmung des kostenneutralen Update-Intervalls

Für die Entscheidung, wie groß das Update-Intervall gewählt werden soll, kann es sinnvoll sein, einen Vergleich der Systeme auf Kostenbasis durchzuführen. Es wird angenommen, dass das Unternehmen die Ausfallkosten $cost_e$ beziffern kann. Um einen Kostenvergleich durchzuführen, werden die beiden Gesamtkostenfunktionen aufgestellt:

$$\gamma_{i,cost_e}(t_{updateInt}) = \gamma_i(t_{updateInt}, cost_e), i \in \{1,2\}$$

Mit Hilfe einer Grenzwertbetrachtung lassen sich die maximalen und minimalen Kosten für das Replikationssystem ermitteln:

$$\gamma_{1,cost_e}^{min} := \lim_{t_{updateInt} \to \infty} \gamma_{1,400}(t_{updateInt})$$

$$\gamma_{1,cost_e}^{max} := \lim_{t_{updateInt} \to 60} \gamma_{1,400}(t_{updateInt})$$

Durch den Schnitt der beiden Kostenfunktionen lässt sich das kostenneutrale Update-Intervall bestimmen:

$$\bar{t}_{updateInt} := \gamma_{1,cost_e}(t_{updateInt}) \cap \gamma_{2,cost_e}(t_{updateInt})$$

Durch die hier vorgestellte Entscheidungsunterstützung lässt sich also der Prozess aus Kapitel 4.3 parametrisieren und die Wahl des Update-Intervalls sowohl auf Basis von Kurzzeit- als auch Langzeit-Kosten entscheiden. Gerade bei der Langzeit-Kostenbetrachtung ist zu beachten, dass durch den hohen Planungshorizont von mehreren Jahren es schwer ist eine genaue Vorhersage der absoluten Kosten zu machen. Mit dieser Methode soll dem Entscheider ein Werkzeug an die Hand gegeben werden, das ihn bei der Parametrisierung des Prozesses unterstützt, jedoch nicht zur Kostenkalkulation genutzt werden kann. In der Praxis ist es sicher sinnvoll, mehrere Szenarien zu simulieren, um so eine noch fundiertere Entscheidung treffen zu können.

4.6 Zusammenfassung

In diesem Kapitel wurde die Cloud Standby Methode zur Notfallwiederherstellung vorgestellt. Diese besteht aus einem Notfallwiederherstellungsprozess und einem Aktualisierungsprotokoll. Der Notfallwiederherstellungsprozess überwacht die Primärcloud und löst den Notbetrieb aus oder beendet ihn. Außerdem wird mit Hilfe von periodischen Replikationenzyklen die aktuelle Datensicherungslösung auf das Notfallsystem aufgespielt. Ein zentraler Punkt ist die Wahl des richtigen Update-Intervalls. Wird es zu groß gewählt, so kann sein, dass im Notfall das System nicht schnell genug

gestartet und damit das RTO nicht eingehalten werden kann. Zur Wahl des Update-Intervalls wurde in diesem Kapitel eine Methode zur Entscheidungsunterstützung vorgestellt. Auf Basis des RTO und Experimenten oder auf Basis von weitergehenden Untersuchungen mittels Simulation wird der Systemadministrator, der das System zur Notfallwiederherstellung später verwaltet, bei der Wahl des Update-Intervalls unterstützt.

Die hier vorgestellte Methode schließt die in den verwandten Arbeiten [25], [32], [32], [69], [102], [115], [135] aufgezeigte Lücke indem eine konkrete Lösung bereitgestellt wird, mit der ein verteiltes System mittels eines Warm-Standby-Ansatzes kostengünstig abgesichert werden kann. Die Kostenersparnis gegenüber vor allem den Hot-Standby-Ansätzen [25], [32], [102], [115], [135] aus den verwandten Arbeiten gibt sich hierbei durch das periodische Starten, Rücksichern und wieder Stoppen des Notfallsystems. Hierdurch fallen, wie auch durch die Nutzung von Cloud Computing, auch lediglich sporadisch, nämlich zur Laufzeit, Kosten an [112], [134]. Damit eignet sich dieser Ansatz aber auch nicht zur Absicherung von hochkritischen Systemen, die mit einem RTO von Sekunden zurechtkommen müssen. Hier sollte weiter auf eine Hot-Standby-Absicherung zurückgegriffen werden. Siehe hierzu auch die Evaluierung in Kapitel 7.3.

Für den Notbetrieb, aber auch für die Aktualisierung des Notfallsystems setzt die hier vorgestellte Methode zur Notfallwiederherstellung eine Deployment-Methode voraus, die es ermöglicht, ein verteiltes System anbieterunabhängig abzusichern. Im Folgenden Kapitel wird eine solche Methode, die auf Metamodellierung basiert, vorgestellt.

5. Modellbasierte Deployment-Methode

Im letzten Kapitel wurde eine Methode zur Notfallwiederherstellung in der Cloud präsentiert. Mit dieser Methode wird das Notfallsystem periodisch bei einem anderen Cloud-Anbieter als dem ursprünglichen gestartet, mit den neuen Daten aus Datensicherung aktualisiert und wieder gespeichert. Wie in Kapitel 4.1.3 beschrieben wird hierzu eine Methode zum Deployment eines verteilten Systems bei verschiedenen Anbietern benötigt. In den Verwandten Arbeiten (siehe Kapitel 3.2) wurden bestehende Ansätze für das Deployment von verteilten Systemen in der Cloud diskutiert. Es gibt zwar bereits bestehende Ansätze zur Modellierung von verteilten Systemen in der Cloud [6], [17], [24], [26], [29], [37], [55], [64], [72], [93], [110], jedoch ermöglicht es keiner dieser Ansätze, ein verteiltes System auch anbieterunabhängig zu modellieren. Lediglich TOSCA [105] kann sowohl ein verteiltes System für die Cloud modellieren als auch dies für mehrere Anbieter tun. Wie in Kapitel 3.2.2 bereits dargelegt, liegt bei TOSCA in der Flexibilität auch der große Nachteil. Der Standard gibt keine Unterstützung bei der Modellierung und es obliegt dem Modellierer, alle Elemente, Prozesse etc. selbst zu entwerfen [65], was die Adaption dieses Standards im Rahmen der in Kapitel 4 vorgestellten Methode zur Notfallwiederherstellung sehr schwer macht.

Aus diesem Grund wird in diesem Kapitel eine eigene modellbasierte Deployment-Methode entwickelt, die im Rahmen der in Kapitel 2.5 vorgestellten Methode zur Notfallwiederherstellung genutzt werden kann, um das Notfallsystem zu deployen[16]. Hierzu werden zunächst die Anforderun-

[16] Es spricht nichts dagegen, alle in diesem Kapitel vorgestellten Modelle und Prozesse später in TOSCA zu modellieren und einzusetzen. Die in dieser Arbeit vorge-

gen, Designentscheidungen und Annahmen formuliert, um dann darauf aufbauend die Methoden-Teile *Beschreibungssprache*, *Deploymentprozess*, *Deploymentprotokoll* und *Deploymentalgorithmus* zu beschreiben. Aufbauend auf den in Kapitel 4 und 5 entwickelten Methoden, die zusammen „Cloud Standby“ bilden, wird dann in Kapitel 6 eine prototypische Implementierung vorgestellt.

In Abbildung 45 ist nochmals das Zusammenspiel aller an Cloud Standby beteiligten Methoden dargestellt.

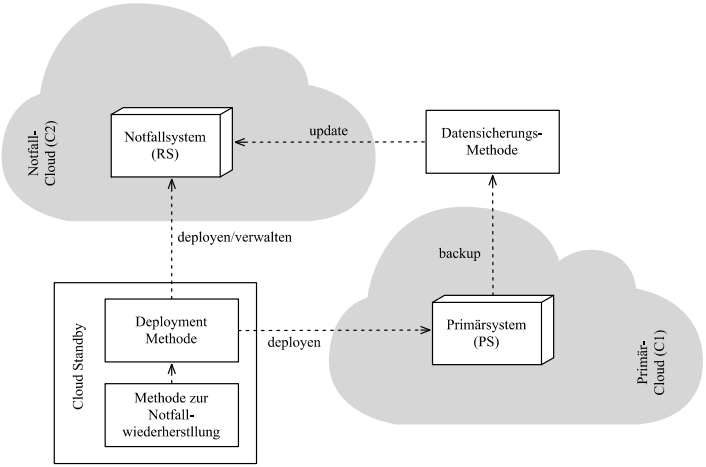

Abbildung 45: Deployment-Methode im Zusammenspiel der Methoden (vgl. [86])

Im Folgenden werden zunächst die Grundlagen, Anforderungen und Designentscheidungen/Annahmen dargelegt, um dann gemäß der Beschreibung in Abbildung 46 die einzelnen Bestandteile „Beschreibungssprache“,

stellten Konzepte können dazu dienen den Standard TOSCA cloud-spezifischer zu machen.

„Deploymentprozess“, „Deploymentprotokoll“ und „Deploymentalgorithmus“ zu beschreiben.

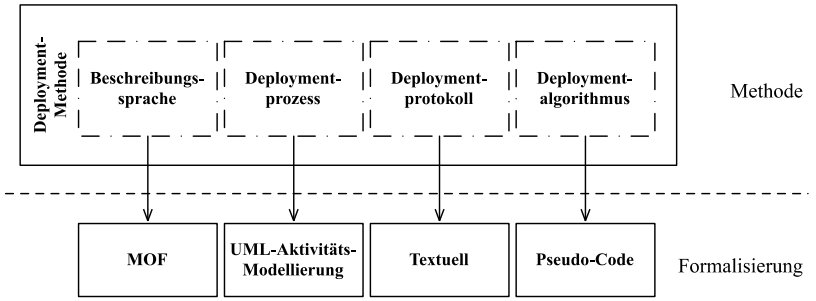

Abbildung 46: Formalisierung der Modellbasierten Deployment-Methode

5.1 Anforderungen

Das Absichern einzelner Server gegen Ausfälle mittels eines Standby-Systems stellt heute keine große Herausforderung dar und wurde bereits gezeigt [69]. Wenn jedoch nicht nur einzelne Server, sondern ganze Systemlandschaften bei mehreren Anbietern gegen Ausfälle abgesichert werden sollen, so gibt es eine Reihe von Anforderungen an die Deployment-Methode und die Beschreibungssprache, auf der die Deployment-Methode basiert. Diese Anforderungen wurden in Kapitel 4.1.3 formuliert und lauten „Verteiltes System“, „Anbieterunabhängigkeit“ und „Automatisierung“. Im Folgenden werden diese Anforderungen nochmals genauer betrachtet[17]:

- *Verteiltes System*: Gemäß der Definition in Kapitel 2.2 und Kapitel 4.2.1 besteht ein verteiltes System aus verschiedenen, abhängigen Komponenten. So können z.B. die Anwendungskomponenten eine

[17] Teile diese Kapitels wurden bereits veröffentlicht [78]

Datenbank für das Persistieren der Daten benötigen. Es muss möglich sein, diese Abhängigkeiten zu definieren, sodass diese beim Deployment berücksichtigt werden können.

- *Anbieterunabhängigkeit*: Damit das verteilte System ihm Rahmen der Notfallwiederherstellung deployed werden kann, muss die Deployment-Methode das verteilte System bei mehreren Anbietern deployen können. Diese Anbieterunabhängigkeit sollte bereits in der Sprache verankert sein, auf jeden Fall jedoch in den Deploymentprozessen. Es muss sichergestellt sein, dass das bestehende verteilte System bei verschiedenen Anbietern deployen lässt. Hierbei muss darauf geachtet werden, dass Verfahren bereitgehalten werden, die Datensicherung des Primärsystems auch auf dem Notfallsystem rücksichern zu können.
- *Automatisierung*: Damit ein verteiltes System im Rahmen der Notfallwiederherstellung automatisch gestartet werden kann, ist es notwendig, dass die Sprache in einem maschinenlesbarem Format vorliegt und automatisiert interpretiert werden kann. Auch diese Anforderung ergibt sich inhärent aus der in Kapitel 4.1.3 formulierten Annahme, welche Eigenschaften eine geeignete Deployment-Methode hat.

Im Folgenden wird eine Deployment-Methode entwickelt, die diesen Anforderungen entspricht und aus einer Beschreibungssprache, einem Deploymentprozess, Deploymentprotokoll und Deploymentalgorithmus besteht.

5.2 Designentscheidungen und Annahmen

Die zuvor formulierten Anforderungen können auf verschiedene Arten umgesetzt werden. Im Folgenden wird dargelegt, wie die Anforderungen aus Kapitel 5.1 gemäß der Dimensionen *Verteiltes System*, *Anbieterunab-*

hängigkeit und *Automatisierung* in der Deployment-Methode umgesetzt wurden.

5.2.1 Verteiltes System

Ein verteiltes System (siehe Kapitel 2.2 und Kapitel 4.2.1) kann aus einer fixen Anzahl virtueller Maschinen bestehen oder sich mittels Skalierung dem aktuellen Bedarf anpassen, also elastisch sein. Das Thema der Elastizität wurde vor allem durch das Aufkommen des Cloud Computing prominent. War vorher eine Skalierung nur in großen Zeitintervallen (neue Hardware musste angeschafft und installiert werden) realisierbar, so ist es mit Cloud Computing nicht nur möglich, Ressourcen in kurzer Zeit hinzuzufügen, sie können auch wieder freigegeben werden.

IaaS Lösungen wie die von Amazon EC2, Rackspace oder Openstack verwenden zur Skalierung sogenannte Skalierungsgruppen, auf die die angekommen Anfragen verteilt werden. Alle Instanzen einer Skalierungsgruppe sind homogen (gleiche Größe, gleiches Image etc.) und haben den gleichen Zugangspunkt. Sie können so in beliebiger Anzahl nach oben und nach unten skaliert werden.

In dieser Arbeit kommen auch Skalierungsgruppen zum Einsatz und werden gemäß der Terminologie der verteilten Systeme in der Literatur *Komponente* oder Tier genannt. Eine Komponente ist also die aus den verteilten Systemen bekannte Größe, auf der ein Teil der Gesamtanwendung bereitgestellt wird und die für sich funktional abgeschlossen ist.

5.2.2 Anbieterunabhängigkeit

Ein Problem bei der Bereitstellung von Notfallsystemen bei verschiedenen Anbietern sind Lock-In-Effekte [3], [30], [73], die durch proprietäre APIs,

Datenformate, etc. entstehen. Eine Lösung, um diese Lock-In-Effekte zu vermeiden, ist Cloud-Föderation[18]. Durch Cloud-Föderation werden verschiedene Cloud-Dienste so verbunden, dass dadurch ein Ressourcen-Pool entsteht. Ressourcen-Pools können über Anbietergrenzen hinweggehen, um damit Redundanz zu ermöglichen. Analog zur Skalierung gibt es auch bei der Föderation die zwei Dimensionen: *Horizontale Föderation* und *Vertikale Föderation*. Bei der horizontalen Föderation wird auf einer Ebene des Cloud Computing Stacks (siehe Kapitel 2.4) ein Ressourcenpool erstellt, wohingegen bei der vertikalen Föderation Dienste-Ressourcen über verschiedene Ebenen hinweg verbunden werden. Im Kontext dieser Arbeit, bei der eine virtuelle Maschine über Anbietergrenzen hinweg abgesichert werden soll, spricht man also von horizontaler Föderation, um Redundanz herzustellen. Hierzu müssen die Voraussetzungen geschaffen werden, dass eine virtuelle Maschine von ihren nicht-funktionalen (also z.B. Performance) aber auch ihren funktionalen Eigenschaften, der installierten Software und deren Konfiguration, gleich ist.

Ein klassischer und etablierter Ansatz, um Software-Installationen auf physikalischen Maschinen und deren Zusammensetzung zu beschreiben, ist die Deployment-Modellierung nach UML [109]. Gemäß dieser Spezifikation ist eine Laufzeitumgebung nichts anderes als ein Container, der maschinenlesbare Codes interpretieren und ausführen kann. In dieser Arbeit werden diese Laufzeitumgebungen als eine Abstraktionsebene genutzt, da sie eine einheitliche Schnittstelle bereitstellen. Beispiele für diese Abstraktionsebenen sind „virtuelle Maschinen“, „Java“, „Tomcat“ oder auch ein bestimmtes Betriebssystem. Hierdurch wird es möglich, eine virtuelle Maschine zu unterteilen und eine Föderation an jeder dieser Laufzeitumgebungsgrenzen durchzuführen (siehe auch Abbildung 47). Betrachtet man

[18] Teile dieses Kapitels wurden bereits publiziert [73]

die Cloud Computing Landschaft, so lässt sich feststellen, dass diese Interpretation von Laufzeitumgebungen als Abstraktionsebenen bei vielen Angeboten zu erkennen ist. So ist ein PaaS-Angebot nichts anderes als die Vereinheitlichung von verschiedenen virtuellen Maschinen auf Applikations-Server-Ebene. Auch für Datenbanken gibt es ähnliche Beispiele.

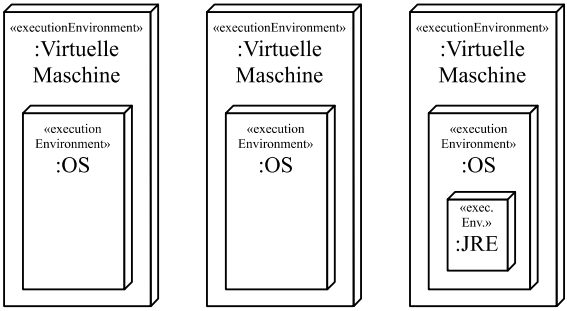

Abbildung 47: Beispiel für Laufzeitumgebungen nach UML im Cloud-Kontext

Diese Sicht der virtuellen Maschine in Form mehrere Abstraktionsebenen lässt sich auch für die Föderation, also Zusammenschluss von funktional gleichen Elementen über Anbietergrenzen hinweg nutzen. Eine Föderation auf Hypervisor-Ebene ermöglicht es beispielsweise, ganze virtuelle Maschinen von einem Anbieter zum anderen umzuziehen. Eine Föderation auf Betriebssystem-Ebene ermöglicht es, vom Betriebssystem unterstützte Anwendungen von einem Anbieter zum anderen zu übertragen usw. Es muss nur sichergestellt werden, dass die Laufzeitumgebungen, die föderiert werden, auf dem Quell- und auf dem Zielsystem funktional gleich sind. Das heißt auch, dass idealerweise alle Softwarekomponenten, die die Laufzeitumgebung bereitstellen, die gleiche Konfiguration aufweisen.

Da in dieser Arbeit ganze verteilte Systeme anbieterunabhängig deployt werden sollen, muss eine Abstraktionsebene bzw. Laufzeitumgebung ge-

funden werden, die flexibel genug ist, um dies zu ermöglichen. Zwar könnten ganze virtuelle Maschinen-Images von einem auf den anderen Anbieter übertragen werden, jedoch hat sich gezeigt, dass die Hypervisors nicht interoperabel sind [79]. Die nächsthöhere Ebene ist das Betriebssystem. Speziell die bei Linux-basierten Betriebssystemen standardisierte Softwarepakete und Paketmanager eignen sich um in dieser Ebene zur Umsetzung von Föderation genutzt zu werden. Es lässt sich über eine Beschreibungssprache die geforderte Version beschreiben und dann dezentral über das bestehende Netzwerk der Softwareauslieferung beziehen. Hierdurch kann ein Flaschenhals vermieden werden, der beim Eintreten eines Notfalls und dem Starten vieler virtueller Maschinen gleichzeitig entstehen könne.

In dieser Arbeit wird also die Föderation auf Betriebssystem-Ebene durchgeführt. Bei der Modellierung muss sichergestellt werden, dass sich auf allen beteiligten Clouds funktional gleiche Images, die ein Betriebssystem repräsentieren, und virtuelle Maschinen mit den gleichen nichtfunktionalen Eigenschaften befinden. Hierzu gibt es föderierte Elemente, die ermöglichen, anbieterübergreifend funktionale (Images) und nichtfunktionale (Virtuelle Maschinen) Elemente zu gruppieren. Diese föderierten Elemente werden im Folgenden im Detail vorgestellt.

Föderierte Elemente

Soll ein Server als Teil eines verteilten Systems auf mehreren Anbietern die gleiche Funktion erfüllen, so müssen die funktionalen und nichtfunktionalen Eigenschaften der beiden Server oder auch Instanzen vergleichbar sein. Unabhängig vom Anbieter muss also die gleiche Funktionalität mit ähnlicher Performance, Kosten, etc. bereitgestellt werden. Hierzu wird das Element *föderierte Instanz* eingeführt. Die föderierte Instanz hat ein *föderiertes Image* und eine *föderierte virtuelle Maschine*, in der das föderierte Image ausgeführt wird. Abbildung 48 verdeutlicht diesen Zusammenhang.

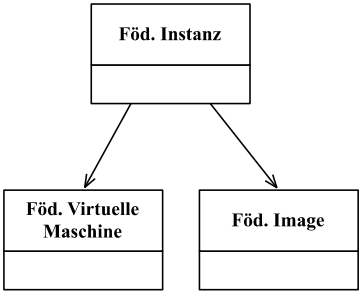

Abbildung 48: Modellierung horizontaler Föderation

Wie bereits dargelegt, erfordert eine Föderation auf Betriebssystem-Ebene die Möglichkeit, Elemente mit der gleichen Betriebssystem-Konfiguration zu gruppieren. Diese Elemente haben also das gleiche unterliegende Basis-Betriebssystem, die gleiche Software installiert, die identisch konfiguriert ist, und haben damit die gleichen funktionalen Eigenschaften. Diese funktional gleichen Betriebssystem-Pakete sind als Images persistiert und können so n-fach instanziiert werden.

Will man anbieterübergreifend funktional gleiche Images zu *föderierten Images* gruppieren, so kann dies wie bereits dargelegt auf Basis von standardisierten Software-Paketen passieren. Linux-Betriebssysteme bieten diese Standardisierung in Form von Distributionen an. Eine Distribution wird von einer zentralen Stelle verwaltet und bietet immer auch die Möglichkeit, die Software-Pakete über standardisierte Verteilungsmechanismen (z.B. yum[19] oder apt[20]) zu verwalten. Durch das Einführen föderierter Images und die Nutzung standardisierter Paketverwaltungen wird dem

[19] http://yum.baseurl.org/

[20] https://wiki.debian.org/Apt

Modellierer die Möglichkeit gegeben, ein verteiltes System unabhängig vom konkreten Anbieter zu beschreiben und so eine funktionale Anbieterunabhängigkeit zu gewährleisten. Es muss jedoch auch sichergestellt werden, dass die Qualität, mit denen diese funktionalen Eigenschaften zur Verfügung gestellt werden, anbieterunabhängig vergleichbar ist.

Für die Gruppierung virtueller Maschinen ist die Gruppierung auf Basis nicht-funktionaler Eigenschaften nicht trivial: Es gibt keine Standardisierung über die Anbieter hinweg, was Performance, Sicherheit, Kosten etc. betrifft. In vielen Fällen ist es jedoch auch nicht notwendig, dass exakt die gleichen Eigenschaften bei beiden Anbietern zur Verfügung stehen. Während es schon ein Problem darstellt, wenn eine Funktion vom Betriebssystem nicht zur Verfügung gestellt wird, so sind bei den virtuellen Maschinen die grundlegenden Funktionen standardisiert, die Güte der Eigenschaften, mit denen sie bereitgestellt werden, jedoch nicht. Es obliegt hier dem Modellierer, die passende Gruppierung der virtuellen Maschinen für den konkreten Anwendungsfall vorzunehmen. Hierzu können z.B. Benchmarks für die Performance durchgeführt und Cluster für die Kosten gebildet werden [81].

5.2.3 Automatisierung

Für den Einsatz der Deployment-Methode im Kontext der Notfallwiederherstellung ist es notwendig, dass ein mittels der hier zu erstellenden Methode beschriebenes verteiltes System vollautomatisch deployt werden kann. Nach Eilam et al. sind modellbasierte Ansätze zur Beschreibung von verteilten Systemen traditionellen Ansätzen überlegen, da sie einfache Visualisierung, Konzeptualisierung, Erweiterung, Standardisierung und Wiederverwendbarkeit zulassen [35]. Tatsächlich basieren mehrere etablierte moderne Deployment-Ansätze, wie TOSCA [105] oder Amazon Cloud Formation [6] auf einem modellbasierten Ansatz (siehe auch Kapitel 3.2).

Damit das beschriebene System am Ende auch vollautomatisch deployt werden kann, ist die Umsetzbarkeit der gewählten Beschreibungsmethodik in ein konkretes System ein wichtiges Entscheidungskriterium. Speziell für die auf Metamodellen basierenden Beschreibungssprachen gibt es eine große Anzahl an Werkzeugen, die direkt für die Implementierung der Deployment-Methode genutzt werden können.

Im Folgenden werden eine Beschreibungssprache und dazugehörige Prozesse, Protokolle und Algorithmen beschrieben, die diese Designentscheidungen umsetzen.

5.3 Beschreibungssprache

Die Beschreibungssprache stellt den Kern der hier vorgestellten Deployment-Methode dar. Diese ermöglicht es, ein verteiltes System anbieterunabhängig zu beschreiben. Zum einfacheren Verständnis wird die gesamte Beschreibungssprache in verschiedene Pakete unterteilt und diese einzeln beschrieben: *Software*, *Infrastruktur*, *Föderation*, *Verteiltes System* und *Datenrücksicherung* (siehe Abbildung 49).

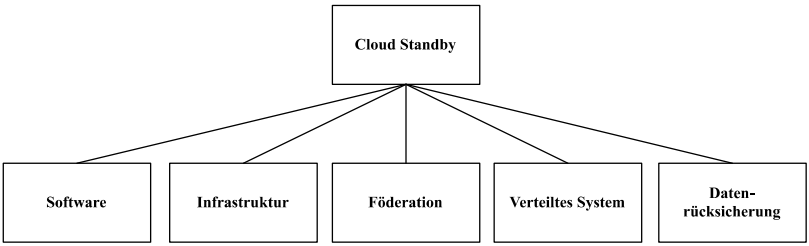

Abbildung 49: Sprachpakete der Cloud Standby Beschreibungssprache

Dabei ist unter der *Infrastruktur* die Menge der Elemente zu verstehen, die anbieterspezifisch sind (z.B. Cloud-Rechenzentrum, virtuelle Maschinen und Images). Da es gerade bei der Absicherung von Deployments auf ver-

schiedenen Anbietern wichtig ist, bestehende Lock-In-Effekte zu vermeiden, wird hier nicht nur eine simple, sondern eine föderierte Infrastruktur genutzt (siehe Kapitel 5.2.2). Im *Föderation*-Paket sind die anbieterspezifischen Elemente in anbieterübergreifende Elemente gruppiert. Das *Software*-Paket beinhaltet alle Elemente, die notwendig sind, um aus den Standard-Komponenten der Anbieter die individuelle, fertig konfigurierte, verteilte Anwendung zu formen. Das *Verteilte-System*-Paket bringt die Software mit der föderierten Infrastruktur zusammen und beschreibt die Beziehungen zwischen den verschiedenen Komponenten.

In Abbildung 50 sind diese Zusammenhänge als Paketdiagramm dargestellt. Die einzelnen Pakete bauen aufeinander auf, was in der Abbildung durch die „Merge“-Beziehung dargestellt wird. Beim Implementieren der Sprache sind diese Merge-Beziehungen zu beachten, sodass ihm Rahmen der Absicherung (bei der das Datenrücksicherungs-Paket der Ausgangspunkt ist) immer die gesamte Sprache genutzt wird. Es wäre jedoch in anderen Kontexten auch möglich, ein Subset der Sprache (z.B. ohne Datenrücksicherungs-Paket) zu nutzen.

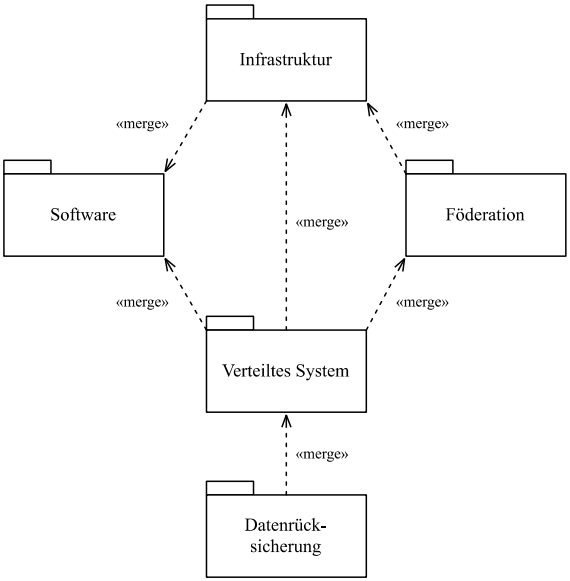

Abbildung 50: Die Beschreibungssprache als Paketdiagramm

Folgend werden die einzelnen Pakete näher beschrieben und im Anhang A.1 eine komplette Referenz bereitgestellt, bei der zu jedem Element zusätzlich die *Generalisierungen*, *Attribute* und *Assoziationen* beschrieben sind.

5.3.1 Software

Ein verteiltes System besteht nach Tanenbaum et al. [123] aus mehreren Recheneinheiten wie z.B. virtuellen Maschinen mit darauf installierter Software. Soll Software auf virtuellen Maschinen verteilt werden, gibt es hierfür verschiedene Methoden. Jedes Betriebssystem hat beispielsweise

eingebaute Paketmanager mit denen man Software warten kann. Neben diesen eingebauten Paketmanagern[21] existieren eine Reihe von Konfigurationsmanagern [101], [111], die betriebssystemübergreifend Softwarepakete verwalten können.

Ein Ziel beim Entwurf des hier vorgestellten Software-Pakets ist es, bestehende Paketmanager- und Konfigurationsmanager-Lösungen möglichst flexibel zu integrieren. Während dies bei Paketmanagern noch relativ einfach möglich ist, da diese aus dem Betriebssystem heraus mittels einfacher Stapelverarbeitungsdateien (Batch-Skripte) aufgerufen werden können, ist es bei Konfigurationsmanagern häufig schwieriger. Diese werden durch externe Komponenten bereitgestellt und verwalten die Installationsskripte, Variablen und alle sonstigen an der Konfiguration beteiligten Daten in einem eigenen Format. Die Übersetzung der in diesem Metamodell vorgestellten Elemente in die Datentypen des Konfigurationsmanagers erfolgt dann in der konkreten Implementierung. Das Metamodell gibt hierbei lediglich eine Datenstruktur vor, die es ermöglicht, alle notwendigen Informationen zu speichern. Eine Repräsentation des Metamodells ist in Abbildung 51 dargestellt.

[21] wie z.B. Apt (https://wiki.debian.org/Apt)

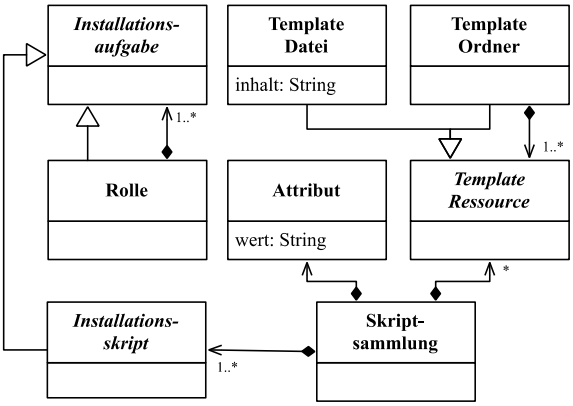

Abbildung 51: MOF Klassendiagramm des Software-Pakets

Das Element *Installationsaufgabe* beschreibt dabei eine abstrakte Aufgabe, die auf einer virtuellen Maschine ausgeführt wird, um eine Konfiguration oder Installation durchzuführen. Die Installationsaufgabe ist eine abstrakte Metaklasse und muss zur Modellierungszeit durch ein *Installationsskript* oder eine *Rolle* konkretisiert werden. Zusammen bilden diese drei Elemente das Kompositum-Entwurfsmuster [43], wodurch es möglich wird, Baumstrukturen mit Skripten als Blätter und Rollen als innere Knoten aufzubauen. Mit dem Element Rolle können also mehrere Skripte und Rollen zusammengefasst und in eine Ausführungsreihenfolge gebracht werden. Die Installationsaufgaben sind eine geordnete Liste von Skripten und Rollen, die so nacheinander zur Ausführung kommen. Mittels einer Tiefensuche kann der Installationsbaum zur Laufzeit aufgelöst und die Skripte, die die Blätter des Baums darstellen, ausgeführt werden. Es gibt in den verwandten Arbeiten auch Ansätze, bei denen Softwareinstallationen auf globaler Ebene für ein verteiltes System koordiniert werden [26], [28], im Rahmen dieser Arbeit werden die Installationsaufgaben jedoch lediglich lokal auf jeder Maschine geplant und die Abhängigkeiten zwischen den

virtuellen Maschinen auf der Ebene des verteilten Systems hergestellt (siehe dazu Kapitel 5.3.4)

Das abstrakte Element *Installationsskript* ist das Programm, das beim Deployment auf der virtuellen Maschine ausgeführt wird. Konkrete Ausprägungen werden in der Implementierung festgelegt und können im einfachsten Fall Bash Skripte sein oder Verweise auf die bestehenden Skripte von Konfigurationsmanagern, deren Ausführung dann von diesen übernommen wird. Dies erfolgt in der konkreten Implementierung (siehe Kapitel 6.2). Es sollte dabei darauf geachtet werden, dass die Installationsskripte idempotent sind [54]. Verschiedene Installationsskripte können in eine *Skriptsammlung* zusammengefasst werden. Dieses abstrakte Element muss auch in der Implementierung konkretisiert werden. Allen Skripten einer Skriptsammlung stehen zur Laufzeit außerdem die Daten aus *Template*- oder *Attribut*-Elementen zur Verfügung. Attribute beschreiben dabei die Variablen und Templates Datei-Inhalte. Daher kann auch bei den Templates das Kompositum-Entwurfsmuster Anwendung finden. *Template-Dateien* sind hierbei die Blätter und *Template-Ordner* die inneren Knoten. Das El*ement Template-Ressource* ist dabei lediglich ein Hilfsmittel, das für das Entwurfsmuster benötigt wird.

5.3.2 Infrastruktur

Die Infrastruktur ist die Basis des verteilten Systems und aller Software, die darauf läuft. Sie verbindet die Software mit der Hardware. Da es Ziel dieser Deployment-Methode ist, Deployments anbieterunabhängig durchzuführen, werden in dem Infrastruktur-Paket alle anbieterabhängigen Elemente zusammengefasst. Soll ein neuer Anbieter aufgenommen werden, so passiert dies im Infrastruktur-Paket. Das Infrastruktur-Paket beinhaltet somit alle Elemente, die von einem Anbieter direkt oder indirekt zur Verfügung gestellt werden. Das Metamodell ist in Abbildung 52 dargestellt.

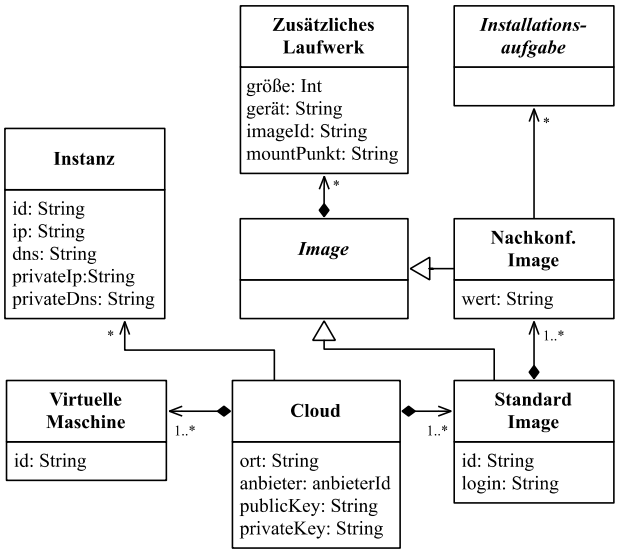

Abbildung 52: MOF Klassendiagramm des Infrastruktur-Pakets (vgl. [84])

Das Element *Cloud* steht für das Rechenzentrum, in dem die Ressourcen bereitgestellt werden. In einem Rechenzentrum gelten die gleichen Kosten und Dienste [93]. Das heißt, wenn ein Anbieter mehrere Rechenzentren in verschiedenen Regionen anbietet, so ist für jedes dieser Rechenzentren ein eigenes Cloud-Objekt zu erstellen. So kann ein verteiltes System auch bei einem einzelnen Anbieter in verschiedenen Regionen abgesichert werden. Um eine Software ausführen zu können, muss in dem Rechenzentrum (virtuelle) Hardware zur Verfügung stehen. Diese Virtuelle Hardware wird im Rahmen dieser Arbeit als *Virtuelle Maschine* bezeichnet. Neben der Hardware gibt es noch die auf der virtuellen Maschine befindliche Software, diese Software ist unter dem Begriff *Image* zusammengefasst. Ein Image ist also eine Sammlung von Software-Artefakten, die auf einer virtuellen Maschine ausgeführt werden können.

Dabei besteht das Image immer aus einem Betriebssystem inklusive Software- und Konfigurationseinstellungen. Das abstrakte Image-Element dient als gemeinsame Schnittstelle für *Standardisierte Images* und *Nachkonfigurierte Images*. Das Standardisierte Image wird von einem Cloud-Anbieter zur Verfügung gestellt und kann direkt gestartet werden. Genügen die Standard Images nicht den Anforderungen beim Deployment oder die Software auf einem Image ändert sich immer wieder, sodass es unpraktikabel wäre, immer wieder ein neues Standard Image zu erstellen und abzuspeichern, kann Software über das Nachkonfigurierte Image direkt beim Deployment nachinstalliert werden. Den virtuellen Maschinen wird dabei in der Regel ein fixer Speicherplatz für Software zur Verfügung gestellt. Reicht dies nicht aus, können je nach Anbieter zusätzliche Laufwerke hinzugefügt werden. Diese können auf Basis von bestehenden Images oder als leere Festplatte erstellt werden.

Das Element *Instanz* ist die Laufzeitrepräsentation der virtuellen Maschine und des Images. Auf einer Instanz laufen auf unterster Ebene immer ein Betriebssystem und darauf aufbauend verschiedene Softwarepakete, mit denen die Instanz mit der Außenwelt kommunizieren kann. Das Instanz-Element wird nicht modelliert, sondern zur Laufzeit automatisch erstellt und aktualisiert.

5.3.3 Föderation

Cloud-Anbieter versuchen im Allgemeinen, ihre Kunden durch Lock-In-Effekte an sich zu binden [73] und die Interoperabilität zwischen den Providern ist daher eher gering [79]. Sollen jedoch Anwendungen anbieterunabhängig deployt werden können, so ist es notwendig, diese Interoperabilität herzustellen. In dieser Arbeit wird hierzu Cloud Föderation genutzt (siehe Kapitel 5.2.2). Das verteilte System wird nicht direkt den anbieterspezifischen Elementen zugewiesen, sondern es wird eine weitere Abstraktionsebene, die Föderation, eingefügt. Erst beim Deployment wird ent-

schieden, welches der anbieterspezifischen Elemente aus den Föderations-Elementen ausgewählt und genutzt wird. Das Metamodell ist in Abbildung 53 dargestellt.

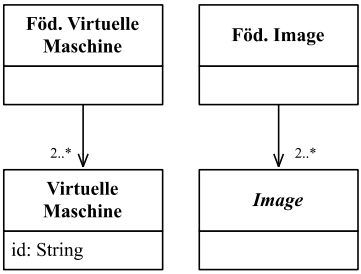

Abbildung 53: MOF Klassendiagramm des Föderation-Pakets (vgl. [84])

Die *Föderierte Virtuelle Maschine*, ist das Gruppierungselement für die virtuellen Maschinen aller an der föderierten Cloud beteiligten Anbieter. Die virtuellen Maschinen repräsentieren hierbei die nicht-funktionalen Eigenschaften der laufenden Instanz. Diese Eigenschaften werden nicht explizit modelliert, sondern sind dem Modellierer bekannt und er nimmt die Zuweisung zu der föderierten Virtuellen Maschine auf deren Basis vor. Eine föderierte Virtuelle Maschine repräsentiert also immer ein gewisses Spektrum an nicht-funktionalen Eigenschaften, die von ihr garantiert werden (z.B. ein Preisintervall oder eine minimale Performance).

Ähnlich wie das Image steht das *Föderierte Image* für die funktionalen Eigenschaften der Instanz. Der Modellierer stellt sicher, dass in dem Gruppierungselement ein Image von allen Anbietern zugewiesen wurde und

dass die funktionalen Eigenschaften aller Images identisch sind[22], d.h. das gleiche Betriebssystem und die gleiche Software installiert und konfiguriert ist.

5.3.4 Verteiltes System

Das Verteilte System Paket beinhaltet alle Elemente, die notwendig sind, um das verteilte System auf einer föderierten Cloud-Infrastruktur zu beschreiben. Dabei werden die in Kapitel 5.2.1 vorgestellten Anforderungen an die Skalierbarkeit des verteilten Systems umgesetzt. Das Metamodell ist in Abbildung 54 dargestellt.
Das *Verteilte System* repräsentiert das reale verteilte System, das anbieterunabhängig deployt werden soll. Es hat eine direkte Referenz zu dem Cloud-Element des Infrastruktur-Pakets. Hier wird eine geordnete Liste mit mindestens zwei Objekten hinterlegt, die besagt, in welcher Reihenfolge das verteilte System bei den Anbietern deployt werden soll. So ist es möglich, zu der Notfallcloud noch weiter Rückfallumgebungen zu spezifizieren. Dies muss jedoch auch von der konkreten Implementierung unterstützt werden. Die *Föderierte Instanz* ist eine logische Einheit und die Abstraktion der Instanz in der föderierten Cloud. Sie ist notwendig, um zustandsbehaftete Informationen der einzelnen Instanzen (Datensicherungen) zu definieren (siehe Kapitel 5.3.5). Zur Laufzeit aggregiert die föderierte Instanz die Instanzen, die parallel laufen und die gleiche Aufgabe bei verschiedenen Anbietern haben.

[22] In der Praxis wird es in der Regel nicht möglich sein, ein exakt identisches Image für jeden der Anbieter anzulegen. Es obliegt hier dem Modellierer zusammen mit einem Systemadministrator zu entscheiden, was für den konkreten Anwendungsfall als identisch angesehen werden kann.

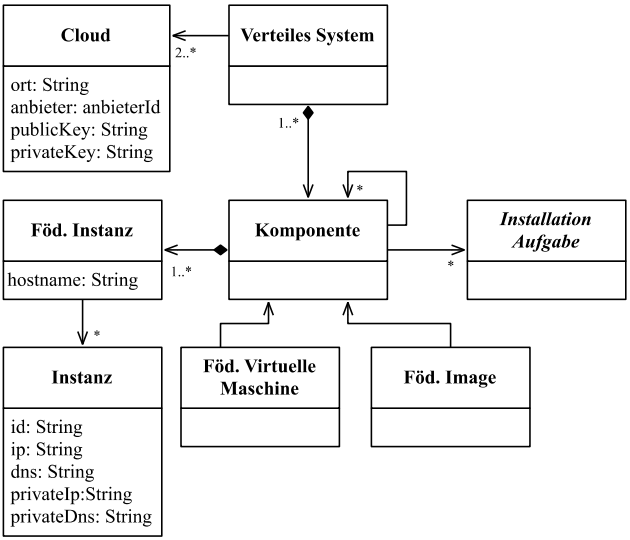

Abbildung 54: MOF Klassendiagramm des Verteiltes-System-Pakets (vgl. [84])

Die *Komponente* ist nach Tanenbaum [123] ein funktional abgeschlossener Teil des verteilten Systems. In der Praxis sind Komponenten häufig wieder verteilte Anwendungen oder Cloud-Cluster-Anwendungen [83]. Das Komponenten-Element stellt dabei eine logische Gruppierung von Föderierten Instanzen dar, die je nach Status des Gesamtsystems auf der Primär- oder Notfallcloud betrieben werden. Durch die Kapselung von mehreren Instanzen zu Komponenten und dadurch dass das Föderierte Image und die föderierte virtuelle Maschine über die Komponente spezifiziert ist, erfüllen die Komponenten die Funktion von Skalierungsgruppen. Alle Instanzen in einer Komponente sind homogen und es können zu Laufzeit beliebige Instanzen hinzu oder abgeschaltet werden. Es ist weiterhin möglich, Komponenten in Abhängigkeit zueinander zu setzen, sodass die Deployment-Reihenfolge der Komponenten beeinflusst werden kann (siehe Kapitel 5.6).

Es ist bei der Modellierung dieses Abhängigkeiten darauf zu achten, dass zyklenfrei sind, da sonst das Deployment nicht durchgeführt werden kann.

Um je nach Anwendungsfall die Modelle noch etwas einfacher gestalten zu können, wird dem Modellierer die Möglichkeit gegeben, weitere Installationsaufgaben direkt an die Komponente zu binden. So könne beispielweise für die Webserverkomponente die Webserverinstallation über die Komponente stattfinden und des müssten nicht mehrere Nachkonfigurierte Images für jede Komponente erstellt werden. Die Installationsaufgaben der Komponente werden dabei nach denen der Nachkonfigurierten Images ausgeführt.

5.3.5 Datenrücksicherung

Das Datenrücksicherungspaket bündelt alle Elemente, die notwendig sind, um eine bestehende Datensicherung (siehe Kapitel 2.3) wieder einzuspielen. Dieses Paket ist speziell für den Einsatzzweck innerhalb der Notfallwiederherstellung mittels Cloud Standby da. Sollte die Beschreibungssprache in einem anderen Kontext eingesetzt werden, so kann auf dieses Paket verzichtet werden.

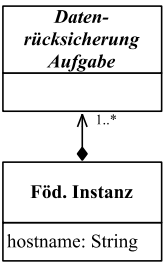

Abbildung 55: Metamodell des Datenrücksicherung-Pakets

Der *Datenrücksicherung Aufgabe* kommt eine zentrale Rolle in Cloud Standby zu. Hier wird definiert, wie genau der letzte Stand der Datenrücksicherung wieder eingespielt werden soll. Die Ausführung der Datenrücksicherungsaufgaben ist der letzte Schritt im Deployment-Prozess. Dieses Element ist nur als abstraktes Element definiert, da es eine große Anzahl an Datensicherungslösungen gibt. In der Implementierung wird dieses Element dann durch das konkrete Elemente erweitert.

5.3.6 Illustrierendes Beispiel

Um die zuvor entworfene Sprache zu veranschaulichen, werden einige Sprachkonstrukte an einem Beispiel verdeutlicht: Ein *Verteiltes System*, bestehend aus zwei *Komponenten*, soll auf einer Föderierten Cloud Infrastruktur, bestehend aus zwei Anbietern (AWS und Rackspace), ausführbar gemacht werden. Eine grafische Repräsentation des als Beispiel spezifizierten Modells ist in Abbildung 56 dargestellt.

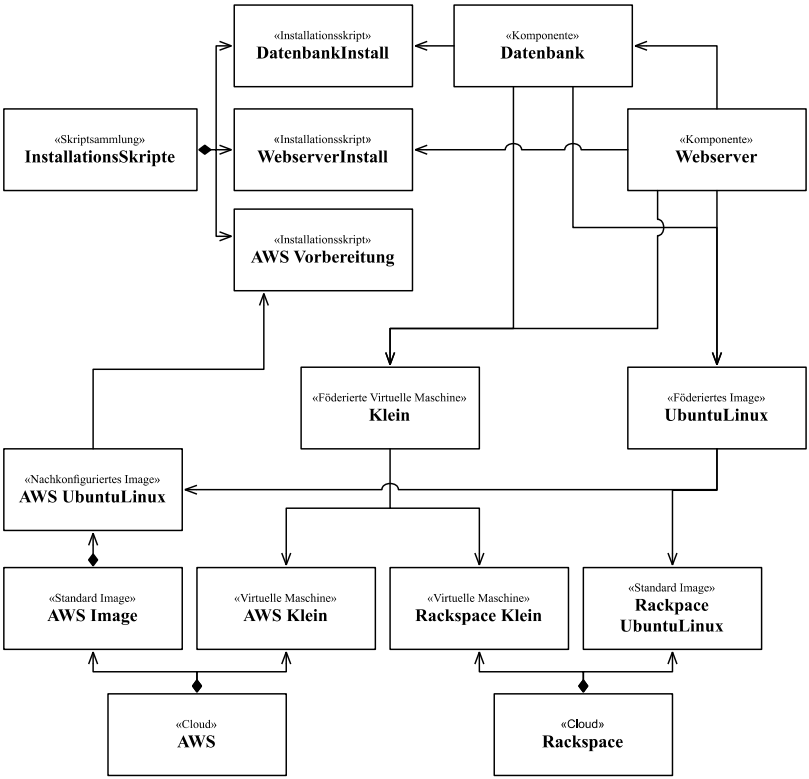

Abbildung 56: Beispielmodell für die föderierte Infrastruktur-Beschreibung basierend auf der Cloud Standby Beschreibungssprache

Die Komponenten sind ein einfacher Webserver sowie eine Datenbank, wobei der Webserver die Datenbank benötigt. Hierzu werden beide Komponenten zunächst mit Standard-Ubuntu Betriebssystemen versehen und die passenden Softwarepakete nachinstalliert. Bei einem Cloud-Anbieter (AWS) besteht außerdem die Besonderheit, dass das Standard Ubuntu-Image zunächst noch nachkonfiguriert werden muss, damit es die gleiche Funktionalität wie das Rackspace Ubuntu-Image hat. Als virtuelle Maschi-

ne wird für beide Clouds diejenige ausgesucht, die kostengünstig ist und wenig Performance bietet. Da die Anbieter ein unterschiedliches Preismodell haben, ist hier nur eine Annäherung zu erreichen.

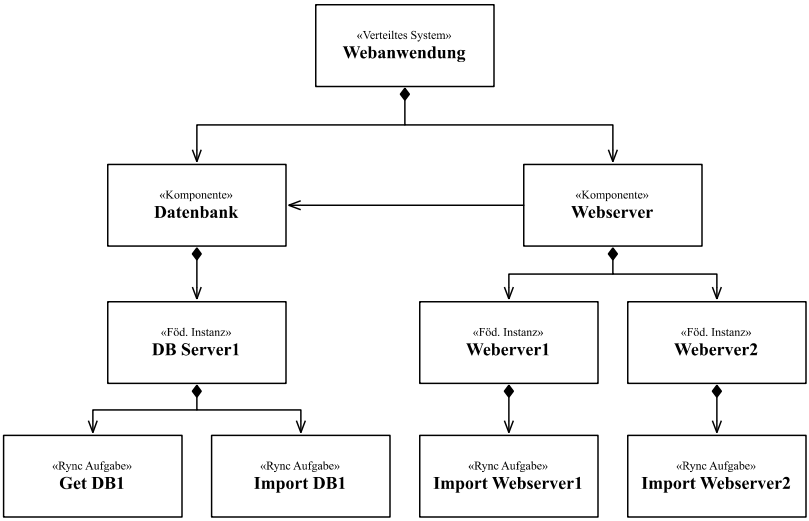

Abbildung 57: Beispielmodell für die Verteiltes System Beschreibung basierend auf der Cloud Standby Beschreibungssprache

Ausgehend von den Komponenten lässt sich auch der Rest des Verteilten Systems beschreiben. Dies ist in Abbildung 57 dargestellt. Die beiden sich bedingenden Komponenten sind Teil des Verteilten Systems „Webanwendung“ und setzen sich wiederum aus *Föderierten Instanzen* zusammen. Jeder der Instanzen ist eine *Datenrücksicherungsaufgabe* vom Typ Rsync zugewiesen, die sicherstellt, dass am Ende des Deployments auch die aktuellen Geschäftsdaten wieder auf der Maschine vorhanden sind. Während sich dieser Rücksicherungsprozess bei den zustandslosen Webserver-Instanzen noch einfach bewerkstelligen lässt, ist dieser bei der Datenbank

komplexer. Nach dem Rsync-Import wird noch ein Bash Skript ausgeführt, was einen Import der Daten in die Datenbank veranlasst. Die Reihenfolge der Aufgaben wird über die Anordnung der Objekte realisiert (in diesem Fall von links nach rechts).

5.4 Deploymentprozess

Die Cloud Standby Beschreibungssprache ist dafür ausgelegt, ein verteiltes System auf verschiedenen Cloud-Anbietern zu deployen. Zusätzlich du der Sprache ist es jedoch auch noch notwendig, dass darauf abgestimmte Prozesse definiert sind, die alle notwendigen Schritte um das System in Betrieb zu nehmen beinhalten. Im Folgenden sollen daher diese Prozesse entwickelt werden. Hierzu wird zunächst der Deploymentprozess und im Anschluss an diese Kapitel das Deploymentprotokoll beschrieben: Der Deploymentprozess verleiht der Beschreibungssprache Leben und nutzt das Deploymentprotokoll, um mit den Cloud-Anbietern zu kommunizieren. Der Deploymentprozess ist eng mit der Beschreibungssprache verbunden, da er die modellierten Elemente und deren Attribute nutzt, um daraus ein laufendes System zu instanziieren. Er ist in Abbildung 58 dargestellt.

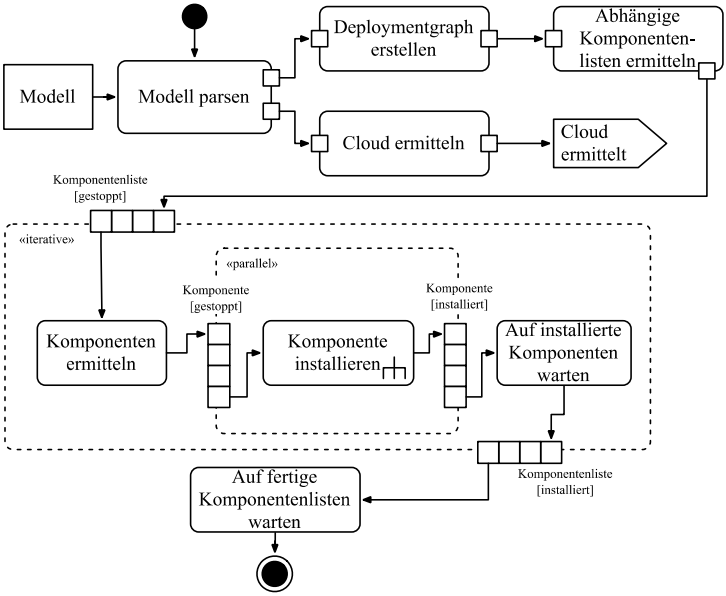

Abbildung 58: Gesamter Deploymentprozess

Zu Beginn des Prozesses wird die Deploymentbeschreibung, also das Modell, eingelesen und die Cloud ermittelt, auf der das Deployment ausgeführt werden soll. Zusätzlich zu der Cloud wird aus den „Benötigt-Beziehungen“ zwischen den Komponenten der Deploymentgraph erstellt. Mit Hilfe des Graphs kann nun eine Reihenfolge der Komponenten erstellt werden, in der sie gestartet werden müssen, ohne eine der „Benötigt-Beziehungen“ zu verletzen. Dabei ist zu beachten, dass dies nur bei einem zyklenfreien Deploymentgraphen möglich ist, was bei der Modellierung zu berücksichtigen ist. Je nach Algorithmus der Zerlegung der Komponenten (siehe hierzu auch Kapitel 5.6) ergeben sich eine oder mehrere Listen von Komponenten. Die Komponentenlisten werden nun sequentiell (iterativ) abgearbeitet und alle Komponenten einer Liste parallel gestartet und die Instanzen

der Komponente installiert. Sind alle Instanzen installiert, so hat auch die Komponente den Status „installiert“, und sind alle Komponenten einer Liste installiert, so bekommt die Liste diesen Status. Sind alle Listen abgearbeitet, ist das gesamte verteilte System deployt.

5.4.1 Komponente Installieren

Die Installation einer *Komponente* ist wiederum ein eigener Prozess, der in Abbildung 59 dargestellt ist.

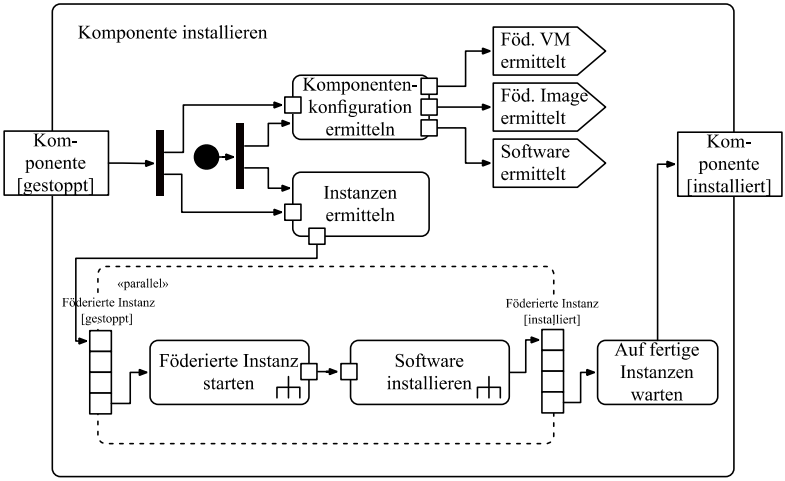

Abbildung 59: Subprozess Komponente installieren

Die Komponente hat zu Beginn des Prozesses den Status „gestoppt“. Im ersten Schritt wird die Konfiguration der Komponente ermittelt, d.h. die zugehörige *föderierte VM*, das *föderierte Image* und die passenden Software-Elemente des Modells. Diese Informationen werden beim Starten der *föderierten Instanz* benötigt.

Alle *föderierten Instanzen* der *Komponente* können nun parallel gestartet (siehe Kapitel 5.4.2) und installiert (siehe Kapitel 5.4.3) werden.

5.4.2 Föderierte Instanz starten

Der Prozess des Startens der anbieterunabhängigen föderierten Instanz ist in Abbildung 60 dargestellt. Erst wenn alle *föderierten Instanzen* installiert sind, hat auch die *Komponente* den Status „installiert“ und der Hauptprozess kann weiterlaufen.

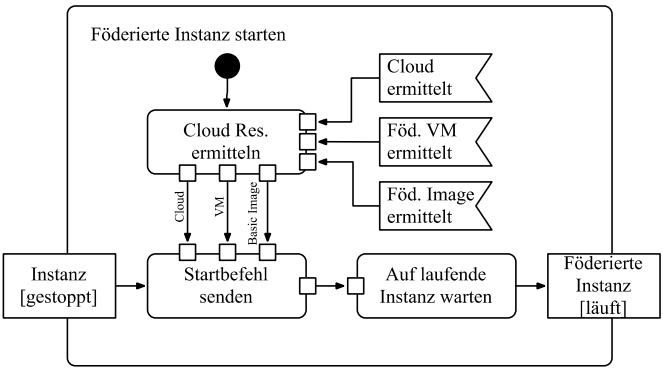

Abbildung 60: Subprozess Föderierte Instanz starten

Soll die *Föderierte Instanz* gestartet werden, so müssen alle zuvor aus dem Modell extrahierten und per Event bereitgestellten Informationen für diese Instanz ermittelt werden. Zunächst muss geklärt werden, auf welcher *Cloud* das gesamte *Verteilte System* und damit auch die *Föderierte Instanz* gestartet werden muss. Anhand dieser Information können dann aus den Elementen *Föderierte Virtuelle Maschine* und *Föderiertes Image* die passenden Elemente des Cloud-Anbieters ermittelt werden. Die Informationen werden genutzt, um den Startbefehl zu erzeugen und an den passenden Cloud-

Anbieter zu schicken. Im Anschluss muss dann lediglich auf die laufende *Instanz* gewartet werden, sodass damit auch die *Föderierte Instanz* den Status „läuft“ erhält.

5.4.3 Software installieren

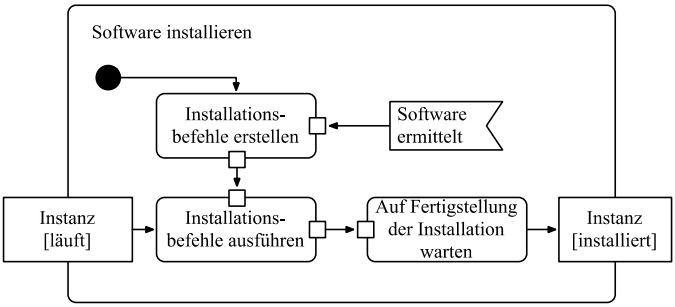

Abbildung 61: Subprozess Software installieren

Software, die auf einer *Instanz* installiert werden muss, kann an verschiedenen Stellen des Modells spezifiziert sein. So kann ein *Standard Image* mittels *Installationsaufgaben* zu einem *Nachkonfiguriertem Image* erweitert werden, eine *Komponente* kann für alle an ihr beteiligten *Instanzen* zusätzliche Software spezifiziert haben und die *Datenrücksicherung* kann außerdem *Installationsanweisungen* enthalten. Die Priorität, in der die einzelnen Installationsschritte auf der *Instanz* durchgeführt werden, entspricht der zuvor genannten Reihenfolge: *Image*, *Komponente*, *Datenrücksicherung*.

Sind die Installationsbefehle ermittelt, so können sie auf der *Instanz* ausgeführt werden (siehe Abbildung 61). Hierzu ist je nach Art der *Installationsanweisung* eine andere Methode notwendig. Ein Bash-Skript wird beispielsweise per Remote-Zugang direkt auf der *Instanz* ausgeführt, während es beim Einsatz eines Konfigurationsmanagers ausreichen kann, dem koordinierenden Server die passenden Befehle zu senden und diesem die

Kommunikation mit den einzelnen *Instanzen* zu überlassen. Wie auch immer die Installation durchgeführt wurde, die *Instanz* ist lediglich dann im Status „installiert“, wenn alle drei Gruppen von Installationsarten abgeschlossen sind.

Während des gesamten Deploymentprozesses muss sowohl mit der Cloud-Management-Schnittstelle als auch mit dem Notfallsystem kommuniziert werden. Diese Kommunikation ist im Deploymentprotokoll dargestellt.

5.5 Deploymentprotokoll

Das Deploymentprotokoll beschreibt die Kommunikation zwischen der Deployment-Methode, der Management-Schnittstelle des Cloud-Anbieters und dem Notfallsystem.

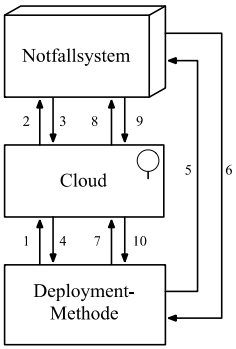

Abbildung 62: Deployment Protokoll

Der zentrale Punkt im Deploymentprotokoll ist die Management-Schnittstelle des Cloud-Anbieters. Cloud-Anbieter wie Amazon Webservices haben ihre eigene Schnittstellen-Spezifikation und Kommunikationsprotokolle. Diesbezügliche Standardisierungsbemühungen, wie OCCI,

waren in der Vergangenheit wenig erfolgreich. Aktuell haben die meisten Anbieter eine eigene Schnittstellenbeschreibung und bei der Umsetzung der Deployment-Methode (siehe Kapitel 6.3) muss darauf gedachtet werden, dass die verschiedenen Anbieter unterstützt werden. Unabhängig von der konkreten Implementierung der Management Schnittstelle sind die Befehle und die Schritte im Deploymentprotokoll grundsätzlich gleich.

Im ersten Schritt teilt die Deployment-Methode der Cloud Management Schnittstelle mit, dass ein neues verteiltes System zu starten ist. Die Schnittstelle prüft in den Schritten 2 und 3, ob die Befehle korrekt ausgeführt wurden und ob das System läuft. Dieser Status wird Deployment-Methode im 4. Schritt dann zusammen mit den Daten zum direkten Zugriff auf das verteilte System der mitgeteilt. Mit Hilfe dieser Zugangsdaten (IP, Benutzername, Passwort) kann die Deployment-Methode direkt über das interne oder öffentliche Netzwerk mit dem verteilten System kommunizieren. Diese Kommunikation findet in den Schritten 5 und 6 statt. Ist die Kommunikation abgeschlossen und wird das System nicht mehr benötigt, so kann es in den Schritten 7 bis 10 wieder gestoppt werden.

Insgesamt kann mit diesem Deploymentprotokoll sowohl die Kommunikation mit dem Cloud-Anbieter als auch mit dem Notfallsystem realisiert werden und im Deploymentprozess genutzt werden.

5.6 Deploymentalgorithmus

Im Deploymentprozess wird eine Menge von Komponenten-Listen benötigt, die parallel deployt werden können. Die Erstellung dieser Listen lässt sich jedoch nicht unmittelbar aus dem Modell herleiten. Die Beziehung „Komponente x benötigt Komponente y“ (siehe Kapitel 5.3.4) definiert einen Deployment-Graph, der genutzt werden kann, um die Deployment-Reihenfolge zu bestimmen. Die einfachste Möglichkeit ist, den Graph durch eine Tiefensuche zu serialisieren und alle Komponenten nacheinander zu deployen. Hiermit wird sichergestellt, dass keine benötigte Bezie-

hung verletzt wird, es wird jedoch auch auf eine mögliche parallele Ausführung des Deployment verzichtet[23]. Das Thema Scheduling und die Planung von Installationsaufgaben ist ein breit erforschtes Gebiet in den Disziplinen Betriebssysteme, Multicore etc. Algorithmen, die parallele Pfade in einem Graph erkennen können, jedoch liegt in der Regel die Annahme zu Grunde, dass die Anzahl der Ressourcen oder der möglichen parallelen Pfade beschränkt sind. Dies ist hier nicht der Fall. Durch das asynchrone Aufrufen der Cloud-Management-Schnittstelle können nahezu beliebig viele parallele Anfragen an den Cloud-Anbieter geschickt werden.

Die Annahme, dass es nur eine beschränkte Anzahl von parallelen Pfaden gibt, ist für die Informatik üblich, in anderen Disziplinen, wie zum Beispiel der Netzplantechnik im Operations Research, ist dies nicht der Fall. Im Rahmen dieser Arbeit wird daher ein auf dem MPM-Algorithmus [104] aufbauender Deploymentalgorithmus vorgestellt, der es erlaubt, in dem von den Komponenten aufgestellten Deployment-Graph parallele Pfade zu identifizieren und als Ausgabe eine geordnete Liste geordneter Listen ausgibt, die so direkt vom Deploymentprozess verarbeitet werden können.

[23] Dieses Kapitel wurde schon in der Veröffentlichung „Cloud Standby System and Quality Model“ [84] vorgestellt.

```
calcActivityTimes := proc(G :: Graph)
  local topSort, startTime, duration, neighbors, max, n, v;

  # Lineare Sortierung berechnen
  topSort := TopologicalSort(Edges(G));
  for v in Reverse(topSort) do
     # Beziehungen prüfen
     if Departures(G, v) = [ ] then
          startTime := 0;
     else
          neighbors := Departures(G, v);
          for n in neighbors do
               max := GetVertexAttribute(G, n, "EFD");
               if max > startTime then
                    startTime := max;
               fi;
          od;
      fi;
      # Early Start Date berechnen
     SetVertexAttribute(G, v, "ESD" = startTime);
     duration := DurationSum(G, v);
     # Early Finish Date berechnen
     SetVertexAttribute(G, v, "EFD" = startTime + duration);
 od;
end proc:
```

Abbildung 63: Deploymentalgorithmus [84]

In dem Abbildung 63 vorgestellten Algorithmus kann die Dauer des Deployments der einzelnen Komponenten mit einbezogen werden, um so eine Abschätzung der Gesamten Deploymentzeit zu liefern. Sind diese Zeiten nicht bekannt, so werden diese einfach mit „1“ angenommen. Der Algorithmus liefert somit immer noch eine korrekte Deployment-Reihenfolge inklusive der parallel zu deployenden Komponenten, nicht jedoch eine Abschätzung für die Gesamtdauer des Deployments (EFD, early finish date). Neben der EFD bestimmt der Algorithmus auch noch die frühste Startzeit (ESD, early start date) der einzelnen Komponenten. Aus diesen ESD können nun die Listen der parallelen Deployments abgeleitet werden: Alle Komponenten mit der gleichen ESD können parallel ausgeführt werden, ohne dabei zuvor definierte Abhängigkeiten zu verletzen.

Beim Deployment aller Komponenten können nun sequentiell die Komponenten mit den EST in aufsteigender Reihenfolge ausgeführt wer-

den. Dabei wird immer gewartet, bis alle Komponenten einer ESD-Gruppe vollständig deployt sind. Die Komponenten gleicher ESD bilden im Deployment-Prozess (siehe Kapitel 5.4) eine Komponentenliste und somit ist der hier vorgestellte Algorithmus problemlos mit dem Deployment-Prozess verwendbar.

5.7 Zusammenfassung

In diesem Kapitel wurde eine neue Beschreibungssprache entwickelt, mit der sich verteilte Systeme anbieterunabhängig beschreiben lassen. Die Beschreibungssprache basiert auf dem Konzept der Metamodellierung und erlaubt eine Beschreibung elastischer verteilter Systeme. Dabei liegt ein besonderes Augenmerk auf der Absicherung und Aktualisierung im Rahmen des Notfallmanagements. Die Sprache setzt Konzepte der horizontalen Föderation um, wodurch es möglich wird, sowohl anbieterspezifische als auch anbieterunabhängige Teile in einem einzigen Modell zu vereinen. Im Vergleich zu bestehenden Ansätzen zur Modellierung und Deployment von verteilten Systemen in der Cloud [6], [17], [24], [26], [29], [37], [55], [64], [72], [93], [105], [110], bietet die hier vorgestellte Methode die Möglichkeit, ein verteiltes System durch ein integriertes Modell anbieterunabhängig zu beschreiben und zu deployen.

Die Beschreibungssprache genügt den Anforderungen die sich aus der Nutzung zusammen mit der Methode zur Notfallwiederherstellung ergeben und umfasst eine große Anzahl von Elementen. Je nach Anwendungsfall könnte es jedoch sinnvoll sein, die Sprache an der einen oder anderen Stelle zu erweitern. Dies ist vor allem dann sinnvoll, wenn die Sprache für außergewöhnlichere Cloud-Anbieter oder Konfigurationsmanager genutzt werden soll. Auch wäre es möglich, die Sprache in anderen Kontexten als der Notfallwiederherstellung einzusetzen und hierauf zu optimieren. Durch die modulare Struktur ist dies problemlos möglich, wird jedoch im Rahmen dieser Arbeit nicht behandelt. Der aktuelle Fokus der Beschreibungsspra-

che liegt weiterhin darin, das Deployment zu automatisieren und ist damit von der Ausrichtung sehr technisch. Dadurch dass jedoch diese technische Basis gelegt ist, wäre es auch leicht möglich, das die Sprache auch zur Unterstützung der Kommunikation von Experten zu nutzen. Hierfür müsste es, ähnlich wie das bei UML oder anderen Auszeichnugssprachen geschehen ist, den Metamodell-Elementen eine grafische Repräsentation zugewiesen werden. So könnte beispielsweise dem Element „Cloud“ eine Wolke zugewiesen werden und die Komposition-Beziehung zu einer „virtuellen Maschine“ darüber repräsentiert werden, dass die virtuelle Maschine in Form eines kleinen Servers in die Wolke gezeichnet wird. [24]

Die Sprache ist jedoch nur ein Baustein der Deployment-Methode. Die Beschreibungssprache ist eng mit dem Deploymentprozess, dem Deploymentprotokoll und Deploymentalgorithmus verzahnt. Sie basieren auf der Beschreibungssprache und beschreiben alle Aktivitäten und Kommunikationspfade, die durchlaufen werden müssen, um ein beschriebenes verteiltes System auf einem oder mehreren Cloud-Anbietern zu deployen. Der vorgestellte Deploymentalgorithmus macht dabei von den Beziehungen zwischen den Komponenten des verteilen Systems Gebrauch, um Komponenten parallel deployen zu können und damit die Deploymentzeit des verteilten Systems im Vergleich zu einem seriellen Deployment zu reduzieren.

Im Folgenden wird nunmehr eine solche Implementierung vorgestellt, die es ermöglicht, ein verteiltes System auf einer großen Anzahl verschiedener Anbieter auszuführen und somit gegen Ausfälle abzusichern.

[24] siehe hierzu auch den Ausblick in Kapitel 8.3

6. Prototypische Implementierung

In diesem Kapitel wird die prototypische Implementierung vorgestellt[25]. Durch sie werden die Methode zur Notfallwiederherstellung und die modellbasierte Deployment-Methode, die zusammen Cloud-Standby darstellen, implementiert. Durch die Implementierung wird auch die Umsetzbarkeit von Cloud Standby belegt. Die Implementierung dient weiterhin als Basis für die experimentelle und simulative Evaluierung in Kapitel 7.

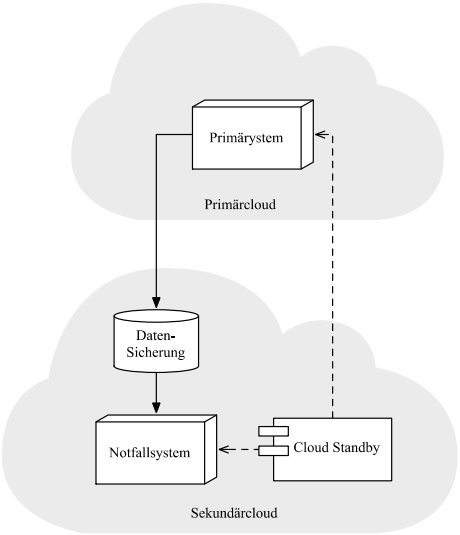

Abbildung 64: Deployment der Systemkomponenten

[25] Verfügbar unter:
https://github.com/alexlenk/CloudStandby/tree/master/org/cloudstandby/

Ziel von Cloud Standby ist es, durch die Bereitstellung eines Notfallsystems die Verfügbarkeit des verteilten Systems zu gewährleisten. Soll dieses Ziel erreicht werden, so ist es auch wichtig, dass die Systemkomponenten selbst gegen einen Ausfall geschützt sind. Dies wird erreicht, indem das Notfallsystem, die Datensicherung und das Cloud Standby System beim gleichen Anbieter betrieben werden (siehe Abbildung 64). Sollte der Primäranbieter ausfallen, so ist es immer noch möglich, dies zu erkennen und das Notfallsystem auf dem Sekundäranbieter zu starten. Fällt der Sekundäranbieter parallel zum Primäranbieter aus, so ist das Starten eines Notfallsystems nicht möglich und es ist irrelevant, ob auch Cloud Standby verfügbar ist oder nicht. Es wäre jedoch möglich, die Cloud Standby Systemkomponenten auf mehr als einen einzelnen Anbieter zu übertragen. Dieser erweiterte Cloud Standby Ansatz mit mehrfacher Redundanz ist jedoch nicht Teil dieser Arbeit.
Im Folgenden werden der Editor für die Erstellung von Deployment-Beschreibungen, die Deployment-Komponente und der Notfallwiederherstellungs-Komponente vorgestellt. Zusätzlich wird noch die Sprach-Komponente, die dem Editor und der Deployment-Komponente als Basis dient, beschrieben. Das Zusammenspiel der einzelnen Komponenten ist in Abbildung 65 dargestellt.

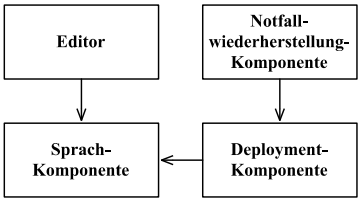

Abbildung 65: Zusammenspiel der Komponenten

Diese Komponenten gliedern sich wie in Abbildung 66 dargestellt in den Gesamtkontext dieser Arbeit (siehe dazu auch Kapitel 1). Der Nutzer nutzt in der Implementierungsphase den Editor und erstellt die Beschreibung des abzusichernden verteilten Systems. Diese Beschreibung wird dann von der Notfallwiederherstellung-komponente konsumiert und die Aktualisierung, der Notbetrieb etc. mittels der Deployment-Komponente initiiert.

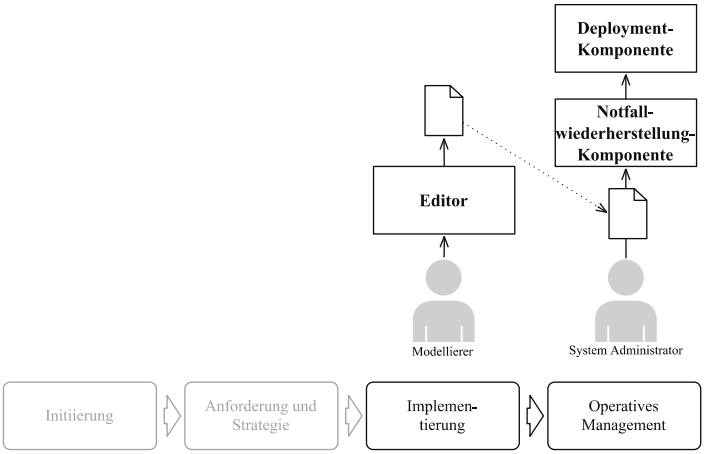

Abbildung 66: Komponenten im Kontext des Notfallwiederherstellungsprozesses

6.1 Editor

Für die Implementierung des Editors wird das Eclipse Modelling Framework (EMF) verwendet[26]. EMF ist ein von der Eclipse Foundation getriebenes quelloffenes Projekt. Die Eclipse Foundation hat zum Ziel, für die

[26] Implementierung verfügbar unter: https://github.com/alexlenk/CloudStandby/tree/master/org/cloudstandby/model.editor

Industrie eine robuste Technologie, eine erweiterbare Plattform sowie ein offenes Projektmanagement zu gewährleisten [121].

Da Eclipse auf dem modularen OSGi[27] Framework basiert, ist die Eclipse Plattform sehr leicht zu erweitern. Aus diesem Grund kann Eclipse mit zahllosen Plugins erweitert werden und bietet zugleich die Möglichkeit, als Laufzeitumgebung für eigene Erweiterungen zu fungieren. In EMF wird gerade von der Eigenschaft der automatischen Code-Generierung (siehe Kapitel 2.5) Gebrauch gemacht. Wird ein bestehendes Metamodell mit EMF umgesetzt, kann automatisch ein grafischer Editor bereitgestellt werden. Mit diesem lassen sich neue Modelle sehr leicht erstellen und im XMI Format speichern. Zusätzlich ist es möglich, aus dem Metamodell Java-Klassen zu erzeugen, die dann in anderen Projekten als externe Bibliothek genutzt werden können. Bestehende Modelle, die z.B. mit dem Editor erzeugt wurden, lassen sich über einen De-Serialisierer automatisch in Java-Objekte umwandeln und so weiterverwenden. Zusätzlich bietet EMF die Möglichkeit, bestehende Modelle auf syntaktische Korrektheit zu überprüfen (Validierung). Die Umsetzung eines Metamodells mit EMF bietet somit eine große Anzahl an Vorzügen. Der Kern von EMF bildet dabei „Ecore“, das den EMOF-Standard, eine Untermenge des Standards MOF [106], implementiert. Ecore selbst ist ein EMF-Modell und stellt die M3-Ebene der Metamodellierungshierarchien dar (siehe Kapitel 2.5). Hier sind die Grundelemente für Klassen (EClass), Attribute (EAttribute), Referenzen (EReference) und Datentypen (EDataType) definiert.

Der Cloud Standby Editor wird automatisch von EMF erstellt, wenn das Metamodell die passende Struktur hat. Abbildung 67 zeigt einen Screenshot einer laufenden Eclipse Instanz mit dem Cloud Standby Editor Plugin sowie einem exemplarischen Modell.

[27] http://www.osgi.org

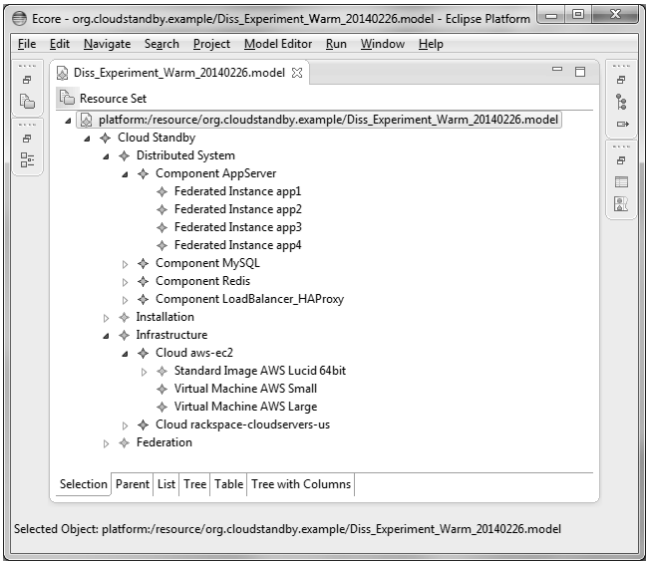

Abbildung 67: Beispielhafte Darstellung eines Modells im Cloud Standby Modell Editor

Will man die Funktionalität der automatischen Editor-Erstellung von EMF nutzen, ergeben sich jedoch einige Besonderheiten, die beachtet werden müssen und eine geringe Erweiterung des Modells bewirken. Der Editor basiert auf einer Baumstruktur, und so ist es notwendig, dass auch das Metamodell als Baum darstellbar ist. Bei der konkreten Umsetzung bedeutet dies, dass es einen Wurzelknoten geben muss, von dem aus alle anderen Knoten mittels „in containment“, also Kompositions-Beziehungen, erreichbar sind. Dies führt dazu, dass in der Umsetzung des Metamodells in die Sprache ein zusätzlicher Knoten „Cloud Standby“ sowie für jeden der unabhängigen Sprachbereiche ein eigener Knoten hinzukommt: „Infrastruktur“, „Software“ und „Föderation“ (siehe Abbildung 68). Das „Verteilte System“ hat bereits ein gruppierendes Element und die Datensicherung

wird per Komposition von der Föderierten Instanz abgeleitet und ist daher über diese zugänglich. Das einzige Element, das im Editor nicht erstellbar ist, ist das Element „Instanz“. Das Element wird automatisch zur Laufzeit erstellt und soll nicht zur Design-Zeit erstellbar oder modifizierbar sein.

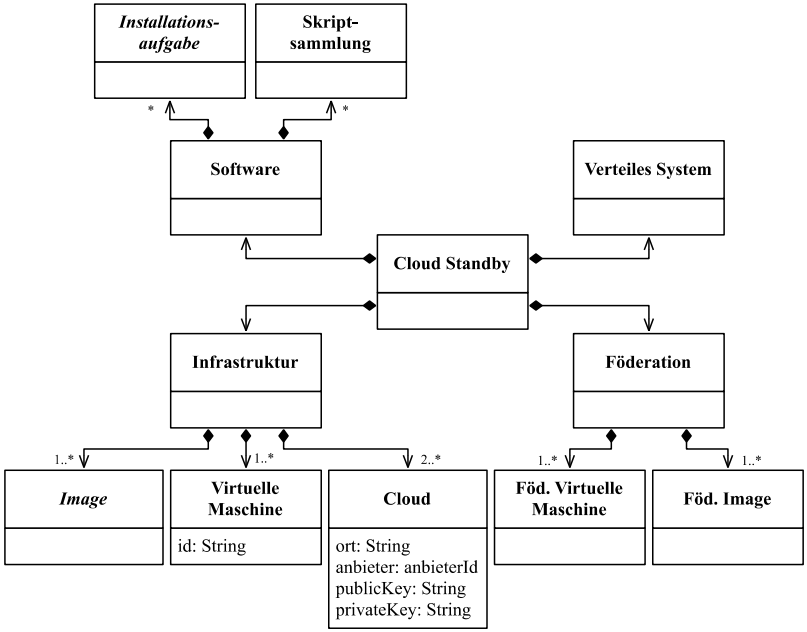

Abbildung 68: Sprach-Erweiterungen für den Editor

6.2 Sprach-Komponente

Die Beschreibungssprache aus Kapitel 5.3 kann zusammen mit den Editor-Erweiterungen aus Kapitel 6.1 direkt umgesetzt werden, da alle Metamodelle dieser Arbeit sich auf den Modellierungsstandrad EMOF [106] be-

schränken[28]. In EMOF beschriebene Modelle sind zu EMF Ecore kompatibel und müssen lediglich in EMF übertragen werden (für den Quelltext des Metamodells siehe Anhang A.1).

Aus dem fertigen Metamodell können mit EMF automatisch Java-Klassen generiert werden, auf deren Basis sich die Elemente des Metamodells direkt in Java-Programmen nutzen lassen. So wird sichergegangen, dass die vom Java-Programm gespeicherten Daten der vom Metamodell definierten Struktur folgen. Zudem kann direkt aus Java die Verifikation der Daten durchgeführt werden und diese als XMI serialisiert werden. Somit kann sichergestellt werden, dass alle benötigten Elemente angelegt wurden, keine syntaktischen Fehler im Modell existierten und es damit valide ist.

Die Klassenstruktur steht außerdem als externe Bibliothek anderen Java-Komponenten zur Verfügung. Alle zuvor als MOF definierten Konstrukte werden als Java-Klassen von dem zuvor vorgestellten Editor und auch von den folgenden Komponenten importiert. Sie werden daher auch in den folgenden UML Klassendiagrammen direkt als Java-Klassen verwendet. Um die Übersichtlichkeit zu wahren, werden jedoch immer nur die notwendigen Klassen dargestellt, auch wenn andere in dem jeweiligen Paket zur Verfügung stehen.

In der Beschreibungssprache in Kapitel 5.3 wurden an mehreren Stellen abstrakte Elemente ohne Konkretisierung definiert. Diese abstrakten Elemente hängen davon ab, welche konkreten Technologien in der Implementierung eingesetzt werden (z.B. ob ein Konfigurationsmanagement genutzt wird oder nicht). Im Folgenden werden die abstrakten Elemente „Installationsskript“ und „Datenrücksicherung Aufgabe“ für die prototypische Im-

[28] Implementierung verfügbar unter:
https://github.com/alexlenk/CloudStandby/tree/master/org/cloudstandby/model

plementierung konkretisiert. Ähnlich wie bei der Beschreibung der allgemeinen Metamodelle werden hier zu jedem Element die Generalisierungen, Attribute und Assoziationen beschrieben.

Installationsskript

Im Metamodell des Software-Pakets (siehe Kapitel 5.3.1) wurden das abstrakte Element Installationsskript um das konkrete *Bash Skript (BashScript)* erweitert, das direkt per SSH auf den Instanzen ausgeführt werden kann. Siehe dazu Abbildung 69.

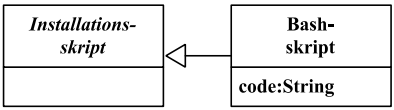

Abbildung 69: Konkretisierung des abstrakten Installationsskripts in der Implementierung

Datenrücksicherung Aufgabe

Neben dem Installationsskript wurde auch noch das abstrakte Element *Datenrücksicherung Aufgabe* konkretisiert. In der Implementierung wird Rsync[29] zum Rückspielen der Datensicherungen eingesetzt und Bash Skripte zur Erledigung zusätzlicher Aufgaben, wie dem Starten eines Datenbank-Import. Es wurde daher das neue Element *Rsync Aufgabe* und *Bash Aufgabe* als Konkretisierung der *Datenrücksicherung Aufgabe* erstellt (siehe Abbildung 70).

[29] https://rsync.samba.org/

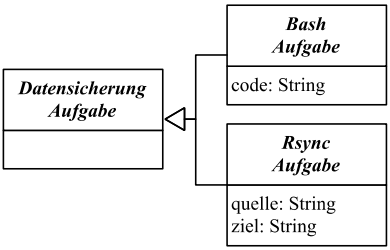

Abbildung 70: Konkretisierung der abstrakten Datensicherung Aufgabe in der Implementierung

Als Attribute hat die *Rsync Aufgabe (RsyncTask)* die Quelle und das Ziel der Datensicherung, wobei die Quelle in der Regel einen über das Netzwerk angebundenen Server darstellt, auf dem die Datensicherungen zur Verfügung stehen. Die *Bash Aufgabe (BashTask)* ist analog zum *Bash Skript* ein Stück Code, welches per SSH ausführbar ist.

6.3 Deployment-Komponente

Die Deploymentsteuerung fungiert als die Komponente, die verteilte Systeme mit Hilfe eines Modells, das in der Cloud Standby Beschreibungssprache modelliert ist, starten und stoppen kann[30]. Alle für die Deployment-Komponente relevanten Konzepte sind bereits in den Kapiteln 5.4 bis 5.6 beschrieben, sodass es im Folgenden lediglich um die technische Umsetzung als Java-Komponente geht. Für das Verständnis dieser Arbeit ist dieses Kapitel nicht essentiell, sodass Leser, die nicht an der technischen

[30] Implementierung verfügbar unter: https://github.com/alexlenk/CloudStandby/tree/master/org/cloudstandby/core/deployer

Umsetzung der Deployment- oder Notfallwiederherstellungs-Komponente interessiert sind, bei Kapitel 6.5 weiterlesen können.

Für die Implementierung der Deployment-Komponente und hier speziell für das Deploymentprotokoll, das auf mehrere Anbieter anzuwenden ist, wurde Apache jclouds genutzt. jclouds ist eine quelloffene Bibliothek, mit deren Hilfe man IaaS Cloud Dienste direkt aus Java heraus nutzen kann. Durch jclouds werden verschiedene[31] Cloud-Anbieter-APIs, wie z.B. die Amazon Web Services, Microsoft Azure, OpenStack, Rackspace oder VMware vCloud unterstützt [2]. Neben der reinen Abstraktion der API bietet jclouds auch umfangreiche Verwaltungsunterstützung, die in dieser Implementierung zum Einsatz kommt:

SSH Schlüssel – Die Verwaltung der öffentlichen und privaten Schlüsseln zum Zugreifen auf die virtuellen Maschinen wird komplett von jclouds übernommen.

Skripte ausführen – Durch die Verwaltung der SSH-Schlüssel wird es auch möglich, direkt auf den virtuellen Maschinen Skripte auszuführen. Die Skripte werden einfach in jclouds definiert und werden nach dem Deployment auf den virtuellen Maschinen ausgeführt.

In Abbildung 71 ist das Klassendiagramm der Deployment-Komponente dargestellt. Es orientiert sich eng an den in Kapitel 5 vorgestellten Prozessen, dem Deployment-Protokoll und dem Deployment-Algorithmus. Über die Sprach-Komponente (siehe Kapitel 6.2) stehen die Elemente aller zuvor beschriebenen Metamodelle als Klassen zur Verfügung und werden im Folgenden so genutzt.

[31] Eine Liste aller unterstützen Anbieter findet sich hier:
http://jclouds.apache.org/documentation/reference/supported-providers/

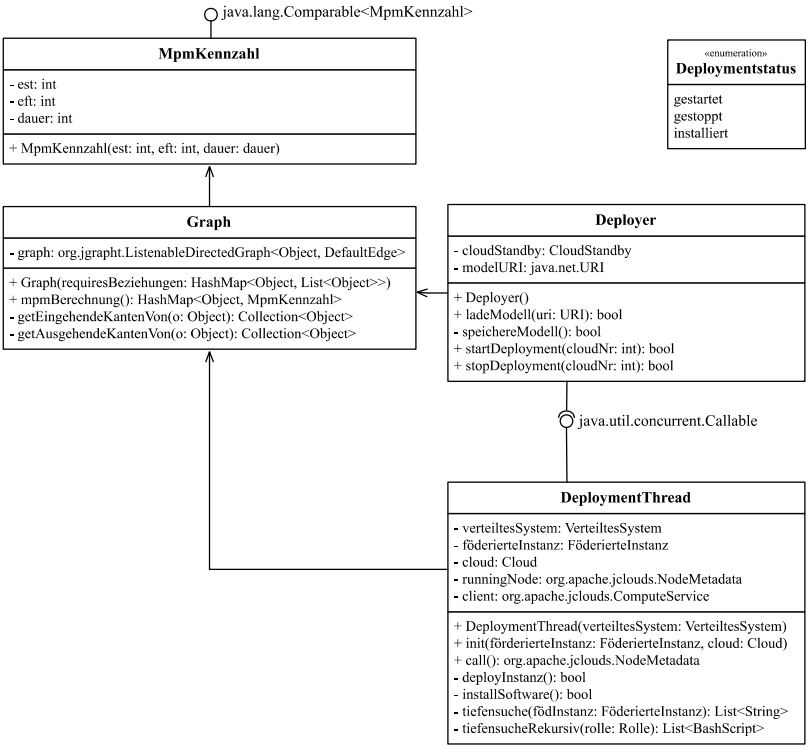

Abbildung 71: Klassendiagramm der Deploymentsteuerung

Deployer – Die Deployer-Klasse ist die Klasse, mit der das verteilte System zu verwalten ist. Mit der Methode „ladeModell“ wird eine neues Modell geladen und mit start/stop Deployment verwaltet. Die Methode „startDeployment“ implementiert dabei den Deploymentprozess aus Kapitel 5.4, Abbildung 58. Lediglich das parallele Starten aller Instanzen eines Cloud-Anbieters innerhalb einer Komponente ist an die Klasse DeploymentThread ausgelagert.

DeploymentThread – Die DeploymentThread-Klasse implementiert das Java-Interface „java.util.concurrent.Callable“. Es ermöglicht das Ausführen mehrerer Instanzen der Klasse in parallelen Threads. Beim Aufruf der Klasse wird automatisch die Methode „call“ ausgeführt, die den Einstiegspunkt in die Ausführung darstellt. Die Methode „deployInstanz“ implementiert dabei die Schritte „Startbefehl senden“ und „Auf Instanz warten“ aus dem Deploymentprozess (Kapitel 5.4, Abbildung 60) und die Installation der Softwarepakete aus der Aktivität „Software installieren“ (Kapitel 5.4, Abbildung 61).

Graph – Der Graph ist ein Datentyp, mit dem allgemein Graphen, hier im Speziellen der Deploymentgraph, gespeichert werden können, sodass später Berechnungen wie MPM darauf angewandt werden können. Die Speicherung der Knoten und Übergänge erfolgt in Form der Adjazenzmatrixdarstellung. Die Funktion mpmBerechnung führt den Algorithmus (siehe Kapitel 5.6) zur Berechnung der Deploymentreihenfolge der Komponenten im Deploymentgraph aus.

MpmKennzahl – MpmKennzahl ist ein Objekt, das als Datentyp für die MPM-Kennzahlen frühester Startzeitpunkt (EST), frühster Endzeitpunkt (EFT) und Dauer dient. Es implementiert das Java-Interface „java.lang.Comparable“, um so die Elemente in eine Reihenfolge bringen zu können.

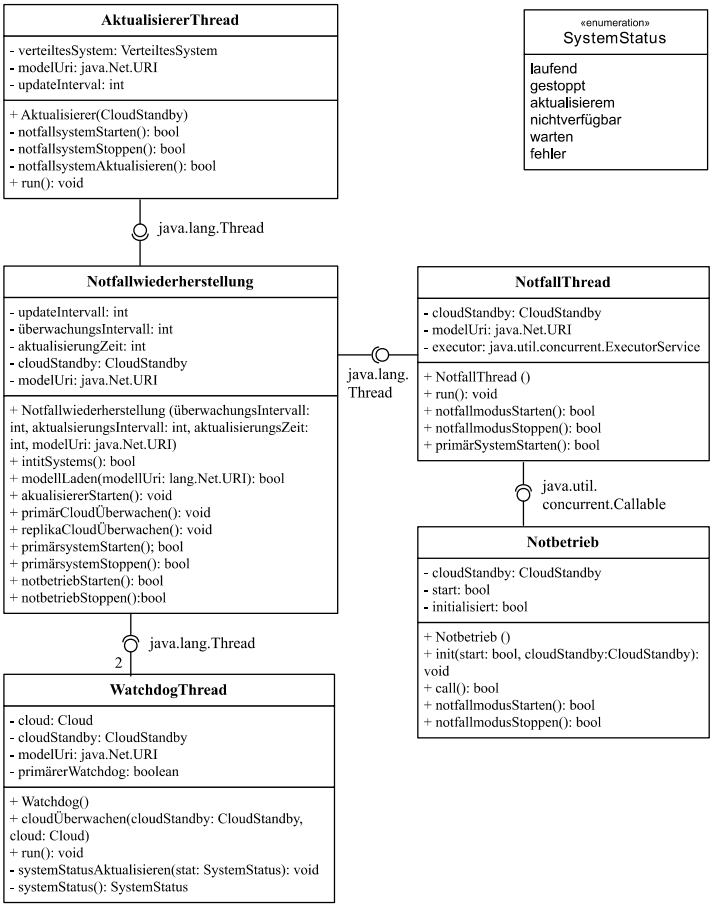

Abbildung 72: Klassendiagramm der Notfallwiederherstellungskomponente

6.4 Notfallwiederherstellungs-Komponente

Die Notfallwiederherstellungs-Komponente[32] ist die grundlegende Komponente vom Cloud Standby. Die gesamte Überwachung des Systems sowie die periodische Aktualisierung gehen von ihr aus. Alle in Kapitel 4 vorgestellten Prozesse und Protokolle werden in der Replikationssteuerung umgesetzt. Das UML-Klassendiagramm ist in Abbildung 72 dargestellt.

Notfallwiederherstellung – Die Klasse Notfallwiederherstellung hat die Aufgabe, das gesamte Cloud Standby System zu verwalten. Sie liest das Modell ein, startet die Überwachung der Cloud-Anbieter (WatchdogThreads), um einen Notfall zu erkennen und aktualisiert das Notfallsystem (AktualisiererThread). Ist ein Notfall erkannt, so wird automatisch der Notbetrieb gestartet (NotfallThread).

WatchdogThread – Die Klasse WatchdogThread implementiert das Java-Interface java.lang.Thread und lässt sich somit unabhängig von der aufrufenden Klasse ausführen. Dieses Design ist notwendig, da sonst die Überwachung der Clouds, die kontinuierlich durchgeführt werden muss, den Programmfluss unterbrechen würde und es nicht möglich wäre, parallel das Notfallsystem zu aktualisieren oder in den Notbetrieb überzugehen. Die WatchdogThread Klasse implementiert den Subprozess des Notfallwiederherstellungsprozesses, der durch das Zeitevent „Überwachungsintervall“ angestoßen wird (siehe Abbildung 31).

AktualisiererThread – Die Aktualisierung des Notfallsystems wird periodisch durchgeführt und läuft parallel zum sonstigen Programmfluss ab. Aus diesem Grund implementiert auch diese Klasse das Java-

[32] Implementierung verfügbar unter: https://github.com/alexlenk/CloudStandby/tree/master/org/cloudstandby/core/replication

Interface java.lang.Thread, um so als eigener Thread ausgeführt zu werden. Die Aktualisierung des Notfallsystems (siehe Kapitel 4.3.3) repräsentiert dabei den Subprozess, der durch das Zeitevent „Update-Intervall“ ausgelöst wird (siehe Abbildung 31).

NotfallThread – Wird ein Notfall durch den Watchdog erkannt, so muss der Notbetrieb gestartet werden. Dies übernimmt die Klasse NotfallThread.

Notbetrieb – Der Notbetrieb an sich wird von der Klasse „Notbetrieb“ verwaltet. Mit dieser Klasse wird der Notbetrieb gestartet (siehe Kapitel 4.3.1) und beendet (siehe Kapitel 4.3.2). Dabei werden die Zugangspunkte umgeschaltet und die Datensicherung für das jeweils produktive System aktiviert.

Systemstatus – Der Datentyp Systemstatus repräsentiert alle in dem Notfallwiederherstellungsprozess erlaubten Zustände der Systemkomponenten. Der Systemstatus wird von einer statischen Klasse „System-Modell“ verwaltet, die aus Übersichtlichkeitsgründen nicht in Abbildung 72 dargestellt ist.

6.5 Zusammenfassung

In diesem Kapitel wurde eine prototypische Implementierung des Cloud Standby Systems vorgestellt. Hierzu wurden die Methoden aus Kapitel 5 und 6 in Java umgesetzt. Diese Implementierung besteht aus einem Editor, mit dem die Deployment-Beschreibung eines verteilten Systems bei mehreren Cloud-Anbietern erstellt werden kann, der Deployment-Komponente, die das verteilte System mit Hilfe der Deploymentbeschreibung instanziiert und der Notfallwiederherstellungs-Komponente, die den gesamten Notfallwiederherstellungsprozess überwacht. Als Grundlage für alle Komponenten wird mit der Sprach-Komponente das Deployment-Metamodell als Java-Klassen bereitgestellt.

Mit Hilfe des Cloud Standby Systems lassen sich nun verteilte Anwendungen auf einem zweiten Cloud-Anbieter absichern. Das hier vorgestellte Cloud Standby System ist Grundlage für die Bewertung im nächsten Abschnitt.

7. Evaluation

Ziel dieser Arbeit ist es eine neue Methode zur Bereithaltung eines Notfallsystems in der Cloud zu entwickeln. Da es bereits bestehende Ansätze geben könnte, die dieses Ziel erfüllen, wurden in Kapitel 3 eine Untersuchung der verwandten Arbeiten durchgeführt. Diese hat jedoch gezeigt, dass es im Themenfeld der Notfallwiederherstellung in der Cloud zwar bestehende Ansätze gibt [25], [32], [69], [102], [115], [135], diese es jedoch nicht ermöglichen, ein verteiltes System in der öffentlichen Cloud abzusichern. Aus diesem Grund wurde in Kapitel 4 eine neue Methode entwickelt, die aus einem Notfallwiederherstellungsprozess, eines Notfallwiederherstellungsprotokolls und einer Entscheidungsunterstützung zur Konfiguration des Prozesses besteht und damit das Ziel der Arbeit erfüllt. Dabei baut diese Methode auf bestehenden Datensicherungslösungen und einer Methode zum anbieterunabhängigen Deployment eines verteilten Systems in der Cloud auf. Auch hier hat sich bei der Untersuchung in Kapitel 3 gezeigt, dass es zwar diverse modellbasierte Deployment-Ansätze für die Cloud gibt [6], [17], [24], [26], [29], [37], [55], [64], [72], [93], [105], [110], keiner dieser Ansätze es jedoch ermöglicht, sowohl ein verteiltes System zu modellieren als auch dies so zu tun, dass in dem Modell mehrere Anbieter beschrieben werden können. Aus diesem Grund wurde in Kapitel 5 eine solche Deployment-Methode entwickelt, die aus einem Deploymentprozess, einem Deploymentprotokoll und einem Deploymentalgorithmus besteht. Die beiden neuen Methoden heißen zusammen Cloud Standby und wurden, wie in Kapitel 6 beschrieben, prototypisch implementiert.

Um einen tieferen Einblick zu bekommen, wie gut sich Cloud Standby zur Absicherung eines verteilten Systems in der Cloud eignet, wird im Folgenden ein Cloud Standby System mit einer Installation ohne Cloud Standby anhand eines Anwendungsfalls verglichen. Gewisse Metriken, wie

die Deploymentzeit und damit das minimal mögliche RTO, können so für den gegebenen Anwendungsfall experimentell bestimmt werden. Nach der Beschreibung des Anwendungsfalls erfolgt eine Vorstellung dieser experimentellen Evaluierung.

Würde man sich jedoch rein auf den RTO konzentrieren und die Kosten außer Acht lassen, so wäre immer ein Hot-Standby-Ansatz das Mittel der Wahl. Bei der Entscheidung, ob ein Warm-Standby-Ansatz wie Cloud Standby genutzt werden soll, sind also auch immer die Kosten von Bedeutung. Für die Evaluierung dieser Arbeit wird daher neben der experimentellen Evaluierung des RTO eine weitere, simulative Methode vorgestellt, mit der die Kosten und Verfügbarkeit des Systems auf lange Sicht und unter Einbeziehung der Ausfallkosten im Notfall berücksichtigt werden. Diese Methode durch Langzeitsimulation wurde speziell für diese Arbeit entwickelt[33].

7.1 Anwendungsfall

Das Unternehmen checkitmobile betreibt mit seiner Handy-Scan-App „barcoo“ eine der erfolgreichsten mobilen Anwendungen Deutschlands. Mit Hilfe eines Smartphones kann der Nutzer der App Barcodes von Produkten scannen und bekommt hierdurch vielfältige Zusatzinformationen, wie zum Beispiel den Preis, die Lebensmittelampel [126], bedenkliche Zusatzstoffe usw. Im Zuge des Jubiläums zum 50. Milliardsten Download aus dem App-Store veröffentlichte Apple eine Liste mit den „Top 10 Gratis-Apps aller Zeiten“ und barcoo belegte Platz 9. Außerdem hat barcoo im April 2013 die 10 Millionen-Download-Marke durchbrochen [91].

[33] Die Methode zur Langzeitsimulation wurde als Vorarbeit für diese Arbeit bereits veröffentlicht [84], [85].

Um diese große Anzahl von Nutzern überhaupt bedienen zu können, nutzt barcoo die Amazon EC2 IaaS Cloud. In Zeiten, in denen die Last nicht hoch ist, werden Server automatisch abgeschaltet und zu Hochzeiten Server hinzugeschaltet. Es handelt sich bei barcoo also um ein elastisches IaaS Deployment im Sinne dieser Arbeit. Der serverseitige Teil der Anwendung unterliegt außerdem ständiger Weiterentwicklung und auch die Daten, die gespeichert werden, unterliegen einer permanenten Änderung. Die barcoo-Infrastruktur stellt somit eine ideale Möglichkeit dar, die Konzepte dieser Arbeit in einem realen Szenario zu evaluieren.

7.1.1 Anwendungsarchitektur

Die barcoo-Anwendung besteht aus vier Komponenten: Loadbalancer, Applikationsserver, Datenbank und NoSQL-Datenbank. Die Komponenten sind in Abbildung 47 dargestellt.

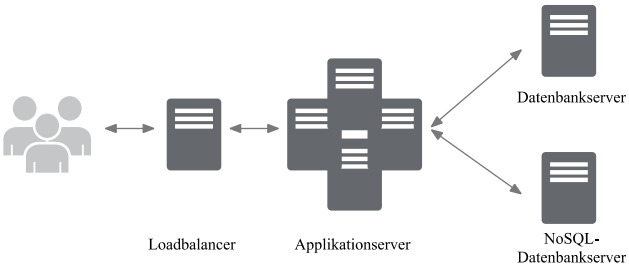

Abbildung 73: Systemkomponenten des Anwendungsfalls (vgl. [86], [133])

Der Loadbalancer dient als Zugangspunkt für alle Clients (Browser, Smartphone-App etc.). Hier werden die Anfragen auf die einzelnen Anwendungsserver weitergeleitet. Die Anwendungsserver nutzen die Daten aus der Datenbank oder dem NoSQL-Cache, um dem Nutzer die passenden Informationen zur Verfügung zu stellen. Zusätzlich sind über die Anwen-

dungsserver noch weitere Dienste angebunden (Preis-Dienstleister, Warnungen zu Lebensmittelskandalen, etc.). Diese externen Anfragen werden im Cache zwischengespeichert.

7.1.2 Formale Beschreibung des verteilten Systems

Die Anwendungsarchitektur und die Daten wurden mit Hilfe des Cloud Standby EMF-Editors in ein Modell überführt (siehe Abbildung 74).

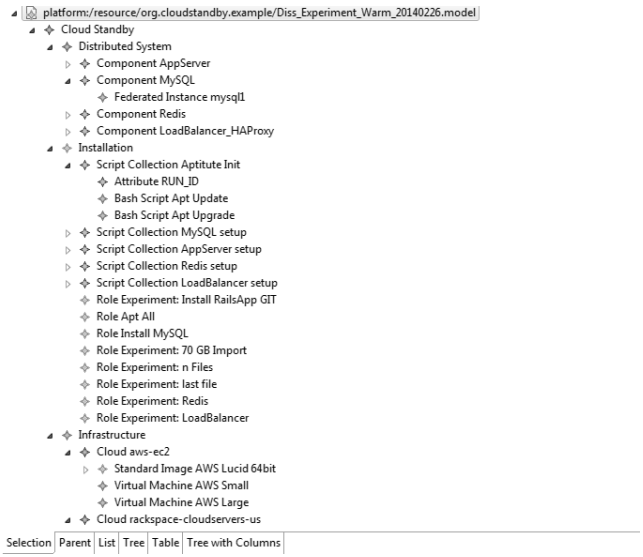

Abbildung 74: Modell des Anwendungsfalls im Cloud Standby Editor

Als föderierte Cloud Infrastruktur wurden die Clouds „Rackspace“ und „Amazon Webservices EU West“ ausgewählt. Als föderiertes Image wurde ein auf Ubuntu basierendes Image gewählt und alle Software-Pakete werden beim Deployment nachinstalliert. Als Virtuelle Maschinen stehen sowohl eine „Kleine VM“ als auch eine „Große VM“ zur Verfügung. Die

kleine föderierte Virtuelle Maschine hat 1-2 Kerne und 1700-2048 MB Arbeitsspeicher. Die große föderierte Virtuelle Maschine hat vier Kerne und ca. 15 GB Arbeitsspeicher.

Alle Komponenten außer der MySQL Datenbank nutzen die Virtuelle Maschine „KleineVM“, diese nutzt die Virtuelle Maschine „GroßeVM“. Auch hat jede der Komponenten, mit Ausnahme des Applikationsservers, eine einzelne Föderierte Instanz, wohingegen der Applikationsserver vier besitzt.

7.1.3 Daten

Das Produktivsystem von barcoo enthält neben unkritischen Cachingdaten weitere unternehmenskritische und datenschutzrelevante Datensätze. Für die unbeschränkte Nutzung dieser Daten bedarf es besonderer Vorkehrungen und Einwilligungen dritter Personen, die im Rahmen der Evaluierung nicht eingeholt werden können. Aus diesem Grund sind für die Evaluierung nicht alle Daten von barcoo verfügbar und werden im Rahmen dieser Arbeit durch unkritische Daten ersetzt. So wird es außerdem möglich, die Erkenntnisse auch ohne Zugang zu den internen barcoo-Daten nachzuvollziehen.

Änderungsraten der Daten

Die Geschäftsdaten von barcoo unterliegen ständigen Änderungen: Neue Produkte werden in die Datenbank aufgenommen, Preise werden aktualisiert, Benutzer kommen hinzu und so weiter. Die Anfragen, die auf die Datenbank treffen, sind eine Mischung aus Einfüge-, Aktualisierungs- und Lösch-Operationen. Bei einer Betrachtung der Daten in der MySQL-Datenbank über mehrere Jahre lässt sich jedoch der Trend erkennen, dass die Datenmenge kontinuierlich steigt. Im Rahmen dieser Evaluation wer-

den daher lediglich die Einfüge-Operationen betrachtet. Es wird davon ausgegangen, dass die Datenbank in einer Woche um 400 MB anwächst.

Auch die barcoo-Anwendung unterliegt kontinuierlicher Weiterentwicklung. Als Teil des agilen Scrum-Entwicklungsprozesses gibt es viele kleinere und größere Veröffentlichungen von neuen Versionen. Im Rahmen der Evaluierung wird davon ausgegangen, dass wie bei Scrum üblich alle zwei Wochen eine neue und größere Version herausgegeben wird, die es eventuell auch notwendig macht, die unterliegenden Softwarepakete des Betriebssystems zu aktualisieren. Abbildung 75 illustriert nochmals den Zusammenhang der Änderung der MySQL-Daten und den Applikationsserver-Aktualisierungen.

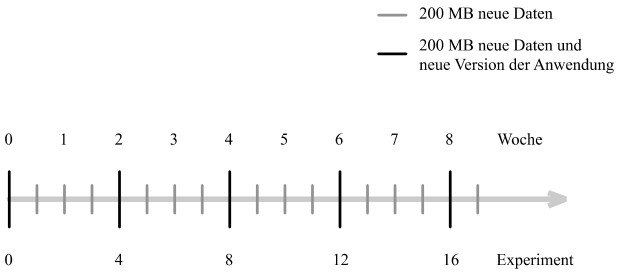

Abbildung 75: Abfolge der Experimente (vgl. [86])

Simulation von kritischen Geschäftsdaten

Im Rahmen dieser Evaluation werden die sensiblen Geschäfts- und Kundendaten durch öffentlich verfügbare Datensätze und Anwendungen ersetzt. Um dennoch möglichst realistische Ergebnisse zu erhalten, wurden die Daten so angepasst, dass sie hinsichtlich der Menge und der Struktur der Daten den realen Daten nahe kommen. Konkret wurde die MySQL Datenbank durch Daten der Ensembl-Gendatenbank [36] befüllt. Diese Daten enthalten sowohl größere Mengen an Zahlen, die mit den bei barcoo

gespeicherten Preisen vergleichbar sind als auch längere Texte, die den sonstigen in der Datenbank gespeicherten Daten entsprechen.

In der Ensembl-Datenbank liegen die Daten in Form von Datenbanken vor, bei der jede Datenbank den genetischen Code einer Lebensform enthält. Erste Untersuchungen der Daten haben gezeigt, dass sich Teile der Daten schneller als andere importieren lassen. Gründe hierfür sind, dass einige Tabellen nur aus Zahlen, andere aus größeren Textmengen bestehen. Damit diese unterschiedlichen Import-Zeiten nicht die Messungen verfälschen, wurden zunächst alle Daten einer Ensemble-Datenbank in MySQL-Import-Befehle umgewandelt und in eine große MySQL-Datei geschrieben. In der Datei entsprach jede Zeile einem Import-Befehl. Im Anschluss erfolgte eine zufällige Mischung der Zeilen der großen, mehreren Gigabyte umfassenden Datei. Das Ergebnis wurde in kleine Dateien zerteilt, die sich alle einzeln in ein bestehendes Schema importieren lassen und die finale Datenbank um 100 MB vergrößern. Zum besseren Transport wurden diese 100-MB-Teile mit GZip[34] gepackt.

Die Daten der NoSQL-Datenbank werden im Arbeitsspeicher gehalten und sind für den Betrieb nicht von zentraler Bedeutung. Über die Zeit reduzieren sie lediglich die Anzahl der Anfragen auf externe Anbieter. Im Notfall wird die NoSQL-Datenbank leer aufgesetzt und füllt sich mit der Zeit allein.

Simulation der Anwendung

Die barcoo-Anwendung ist der zentrale Kern des Unternehmens. Sollte diese öffentlich werden, so wäre dies geschäftsschädigend. Im Rahmen dieser Evaluierung waren alle Installationsschritte für das Aufsetzen eines

[34] https://www.gnu.org/software/gzip/

Servers, der die barcoo-Anwendung auf Ruby-on-Rails[35] beschreibt, verfügbar, jedoch nicht die Anwendung selbst.

Um diese Anwendung zu simulieren, wurde ein zum Zeitpunkt der Evaluierung sehr aktives Github-Projekt „dispora“[36] genutzt. Dispora[37] hat die Entwicklung eines verteilten sozialen Netzwerks zum Ziel. Alle Komponenten sind open Source und es sind bereits mehrere Versionen der Software auf Github verfügbar, wodurch sich die Entwicklung der Software über die Zeit simulieren lässt.

7.2 Experimentelle Evaluierung

In der experimentellen Evaluierung wird das im Anwendungsfall beschriebene Modell ausgeführt und die Deploymentzeiten für einzelne Deployment-Schritte sowie das gesamte Deployment. Es wird dabei die Methode aus Kapitel 7.2.1 zunächst auf das System ohne Cloud Standby angewandt und mit einer Cloud Standby Installation verglichen.

7.2.1 Methode

Im Folgenden wird der Experimentaufbau zur Messung der Deploymentzeiten vorgestellt. Im Rahmen der Evaluation müssen eine ganze Reihe von Experimenten durchgeführt werden, die teilweise eine sehr lange Ausführungszeit haben. Damit diese Experimente nicht manuell durchgeführt werden müssen, übernimmt die Komponente „Experimente Steuerung“ die Ausführung. Abweichend von dem Referenzdeployment der Systemkom-

[35] http://rubyonrails.org/

[36] https://github.com/diaspora/diaspora

[37] https://joindiaspora.com/

ponenten in Kapitel 6 ist die Cloud Standby Komponente nicht in der Notfall-Cloud gehostet, sonderern auf dem lokalen Rechner, der auch die Experimente ausführt und auswertet. Dies ermöglicht einen einfacheren Zugriff auf die Ergebnisse. Da das System nicht produktiv betrieben wird, ist es zudem unerheblich, wo genau die Cloud Standby Komponente betrieben wird. Eine Übersicht über den Experimentaufbau findet sich in Abbildung 76.

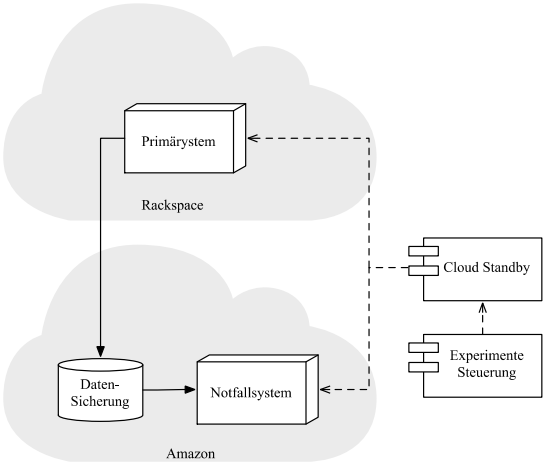

Abbildung 76: Experimentaufbau

Für die Ausführung der Experimente unterscheidet die Experimentsteuerung, ob das Deployment mit oder ohne Cloud Standby durchgeführt werden soll. Ohne Cloud Standby wird die Instanz nach dem erfolgreichen Deployment und der Entnahme der Messergebnisse wieder terminiert, also gestoppt und gelöscht, sodass beim nächsten Start ein komplett leeres Image wieder mit Software und Daten befüllt werden muss. Aus diesem

Grund erhöht sich ohne Cloud Standby in jedem Experimentschritt die Datenmenge.

Ist Cloud Standby aktiviert und wird auf diese Weise der Cloud Standby Ansatz simuliert, so wird die Instanz lediglich gestoppt, wodurch die Daten auf dem Image erhalten bleiben. Im nächsten Experimentschritt müssen nur noch die neuen, nicht aber die alten Daten auf die Instanz aus der Datensicherung zurückgespielt werden. Der Ablauf der Experimente ist in Abbildung 77 dargestellt.

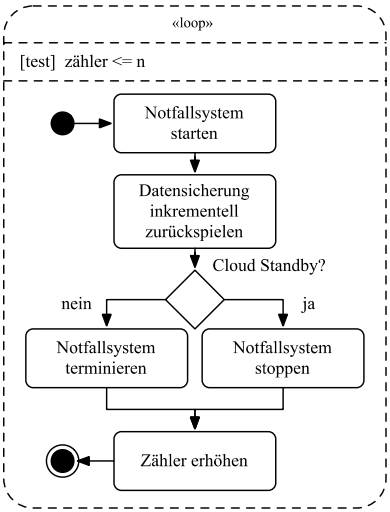

Abbildung 77: Experimentablauf

Im Folgenden werden die Ergebnisse der Experimente präsentiert, die durch die vorgestellte Experiment-Komponente durchgeführt wurden.

Die hier vorgestellte Methode zur experimentellen Evaluierung der Absicherung von verteilten Systemen wird in den folgenden beiden Kapiteln angewandt ($n = 48$), um das Deploymentzeitverhalten des im Anwen-

dungsfalls vorgestellten verteilten Systems ohne und mit Cloud Standby zu analysieren.

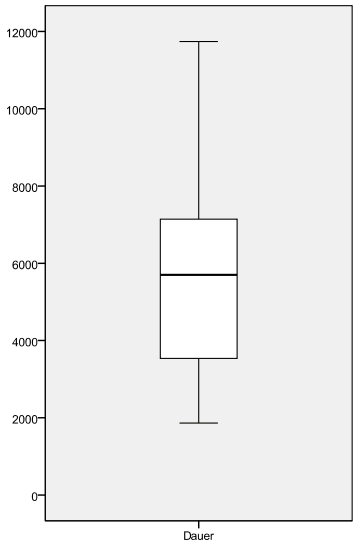

Abbildung 78: Verteilung der Deploymentzeiten ohne Cloud Standby

7.2.2 Analyse des Anwendungsfall-Primärsystems

Zunächst wird das verteilte System ohne Cloud Standby gemäß der in Kapitel 4.5.1 vorgestellten Methode untersucht. Es werden die zuvor vorgestellten Experimente durchgeführt. Die Verteilung der Gesamtdeploymentzeiten in Sekunden sind in Abbildung 78 dargestellt. Die weite Streuung der Daten ist in diesem Fall wenig verwunderlich, da die Größe der Daten über die Zeit zunimmt und sowohl der Transfer als auch der Import der Datenansteigen. Durch die Zunahme der Datenmenge in jedem Experimentschritt ist eine weitere Zunahme der Deploymentzeiten zu erwarten.

Die Entwicklung der Gesamtdeploymentzeiten ist in Abbildung 79 dargestellt. Wie erwartet, ist mit der Zunahme der Datenmenge auch ein Anstieg der Zeit für das Deployment zu erkennen. Eine Gesetzmäßigkeit lässt sich aus der Grafik nicht direkt ablesen, jedoch können verschiedene Kurven-Anpassungs-Modelle getestet werden. Mit Hilfe einer solchen Kurve kann dann die weitere Entwicklung der Zahlen approximiert werden.

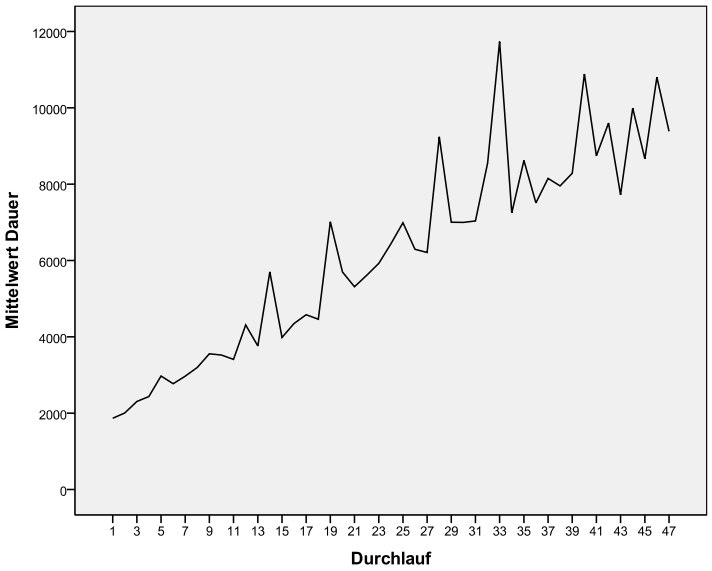

Abbildung 79: Entwicklung der Deploymentzeit ohne Cloud Standby bei voranschreitender Alterung der Daten

In Tabelle 9 sind verschiedene Arten der Kurvenanpassungsmodelle im Vergleich aufgelistet. Man kann sehen, dass insbesondere sowohl die kubische als auch die Potenzfunktion als Modell für die Kurvenanpassungsmethode einen hohen R^2-Wert erreichen und damit eine gute Annäherung an die tatsächlichen Messwerte darstellt.

Tabelle 9: Kurvenanpassungsmodelle im Vergleich

	R^2	F	Sig.
Linear	0,867	293,517	0,000
Logarithmisch	0,754	138,149	0,000
Invers	0,323	21,433	0,000
Quadratisch	0,874	152,599	0,000
Kubisch	0,877	101,856	0,000
Zusammengesetzt	0,874	312,281	0,000
Potenzfunktion	0,898	397,464	0,000
S-förmig	0,483	42,081	0,000
Aufbaufunktion	0,874	312,281	0,000
Exponentiell	0,874	312,281	0,000

Die Kubische- und Potenzfunktion-Anpassung ist nochmals in Abbildung 80 dargestellt. Da die Potenzfunktion-Anpassung ein höheres Bestimmtheitsmaß (R^2) besitzt, wird diese künftig für die Beschreibung der Deploymentzeit des im Anwendungsfall modellierten, verteilten Systems benutzt. Das Bestimmtheitsmaß ist so zu interpretieren, dass in 89,8% der Fälle die Potenzfunktion eine Aussage über die realen Messungen macht. Die im Folgenden genutzte Potenzfunktion lautet:

$$\mathrm{f}(x) = 1234{,}742\, x^{0{,}519} \qquad [sek]$$

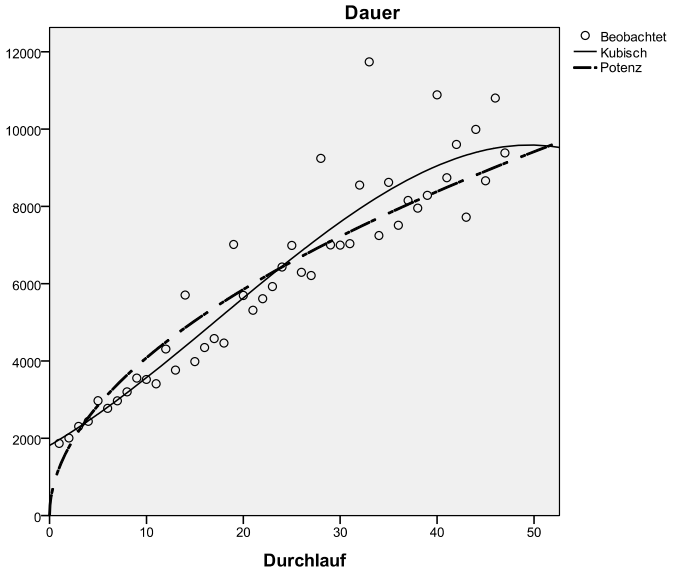

Abbildung 80: Kubische- und Potenzfunktion-Kurvenanpassung im Vergleich

Da zwei Durchläufe einer Woche entsprechen, entspricht ein Experimentdurchlauf 84 Stunden. Die folgende Formel bestimmt die Deploymentzeit in Minuten in Abhängigkeit von der Dauer t in der keine Aktualisierung stattgefunden hat (ebenfalls in Minuten):

$$\mathrm{t}_{Depl}(t) = 0{,}2459\, x^{0{,}519} \quad [min]$$

Für den Wertebereich 0 – 201.600 Minuten (140 Tage) repräsentiert die Funktion, die in den Experimenten bestimmten Werte. Werte über 201.600 ergeben sich aus dem weiteren Verlauf der Funktion und stellen lediglich eine Schätzung dar.

Die Deploymentzeit des verteilten Systems setzt sich aus den Deploymentzeiten der einzelnen Komponenten und der Struktur der Abhängigkeiten zwischen den Komponenten zusammen. Sollten z.B. keine Abhängigkeiten bestehen, so würden sich alle Komponenten parallel deployen lassen. Um den Einfluss der Daten oder der einzelnen Applikationsserver-Aktualisierungen etwas genauer zu durchleuchten, ist eine genauere Betrachtung der Deploymentzeiten sinnvoll. In Abbildung 81 sind diese als Schaubild dargestellt.

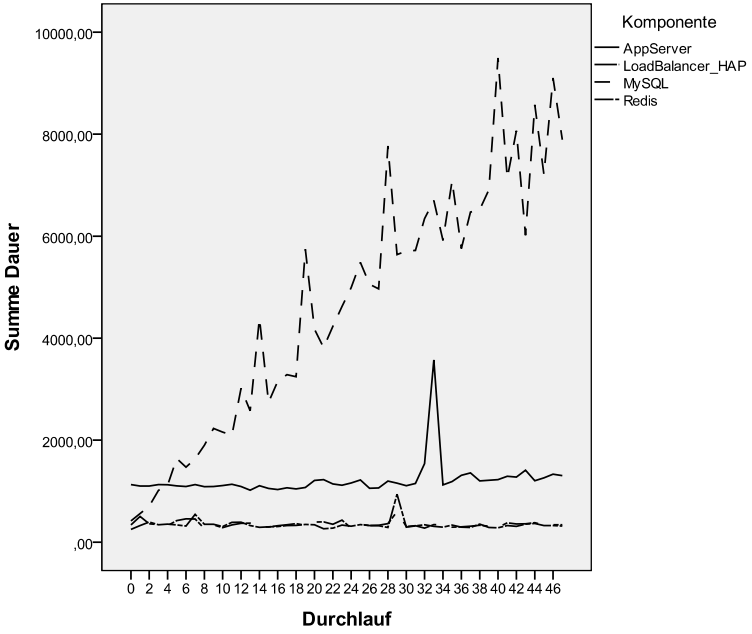

Abbildung 81: Vergleich der Deploymentzeiten nach Komponente (ohne Cloud Standby)

Das Schaubild verdeutlicht, dass vor allem der MySQL-Datenbankserver zu der charakteristischen Kurve des Gesamtdeployments beiträgt. Bereits ab einer Datenmenge von 800 MB dauert das datenintensive Deployment der Datenbank länger als das installationsaufwendige Deployment der Applikationsserver. Die Applikationsserver-Komponente hat lediglich im 33. Experimentschritt eine mehr als doppelt so hohe Deploymentzeit. Im Folgenden wird die Deploymentdauer der einzelnen Komponenten etwas näher betrachtet.

Datenbankserver – In Abbildung 82 sieht man die Dauer der einzelnen Installationsschritte und den Start der Instanz in Schritt 000. Die Aktualisierung des Betriebssystems in den Installationsschritten 001 und 002 ist ebenso vernachlässigbar klein wie die Installation der MySQL-Software in Schritt 003 als auch die Einbindung eines externen Datenträgers in Schritt 004. Es lässt sich erkennen, dass die steigende Datenmenge einen Einfluss auf den 005. Installationsschritt und das Herunterladen der inkrementellen Datenbankbackups hat. Der Großteil der Zeit wird jedoch für den Import der einzelnen Datenpakete in die Datenbank benötigt.

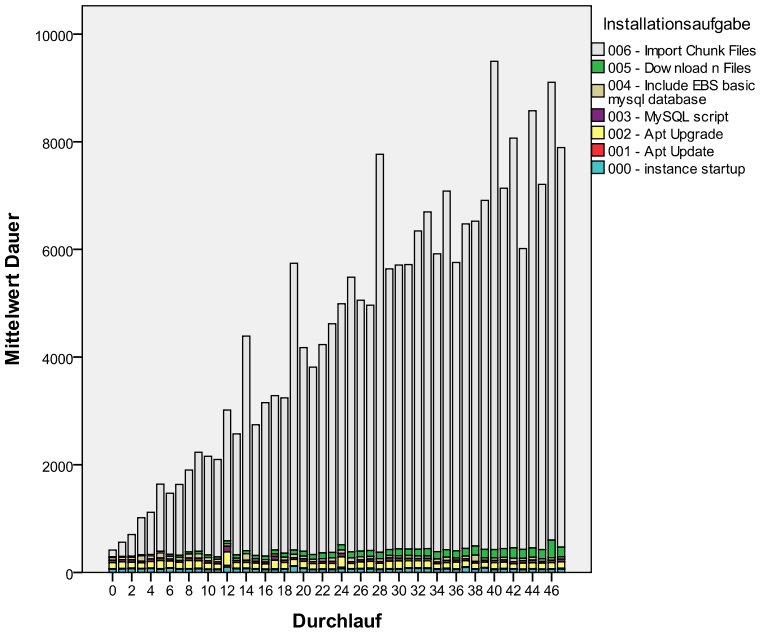

Abbildung 82: Datenbankserver Deploymentzeiten (ohne Cloud Standby)

Applikationsserver – Das installationsintensive Deployment der vier parallel zu installierenden Applikationsserver ist in Abbildung 83 dargestellt. Von Ausreißern abgesehen stellt sich der Bereitstellungsprozess sehr homogen dar. Zwar ist der Trend zu erkennen, dass eine Installation der dispora-Anwendung im 006. Schritt mit fortschreitender Weiterentwicklung der Anwendung länger dauert, jedoch stellt dies nur eine Momentaufnahme dar und könnte bei einer neuen Veröffentlichung wieder umkehren.

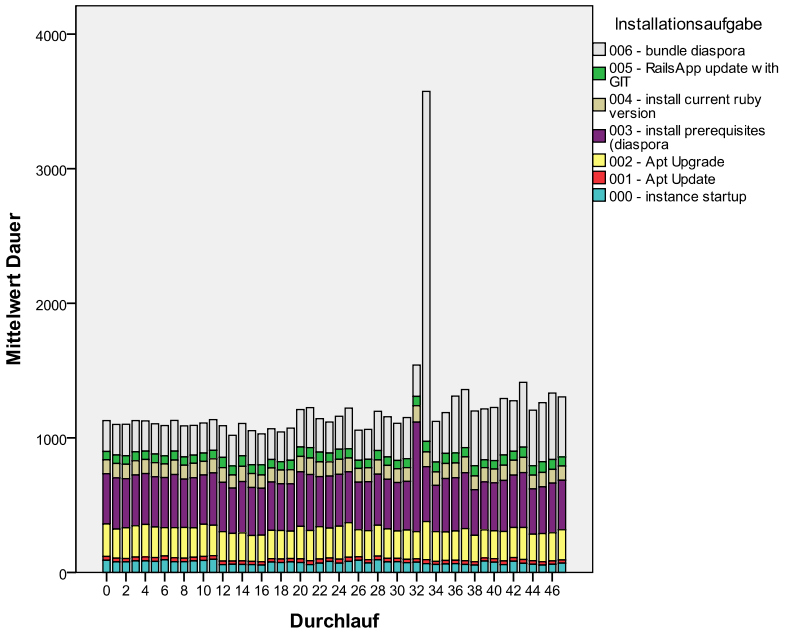

Abbildung 83: Applikationsserver Deploymentzeiten (ohne Cloud Standby)

Loadbalancer – Der Loadbalancer ist im Vergleich zur Datenbank und zum Applikationsserver schnell deployt. Das Deployment des Loadbalancers dauert ungefähr halb so lang wie das des Applikationsservers. Abbildung 84 zeigt deutlich, dass die zeitintensivsten Abläufe die Standardaufgaben, wie das Starten der Instanz und das Aktualisieren des Betriebssystems, sind. Die Installation und die Konfiguration der HA-Proxy Loadbalancer Komponente sind hingegen innerhalb von Sekunden ausgeführt.

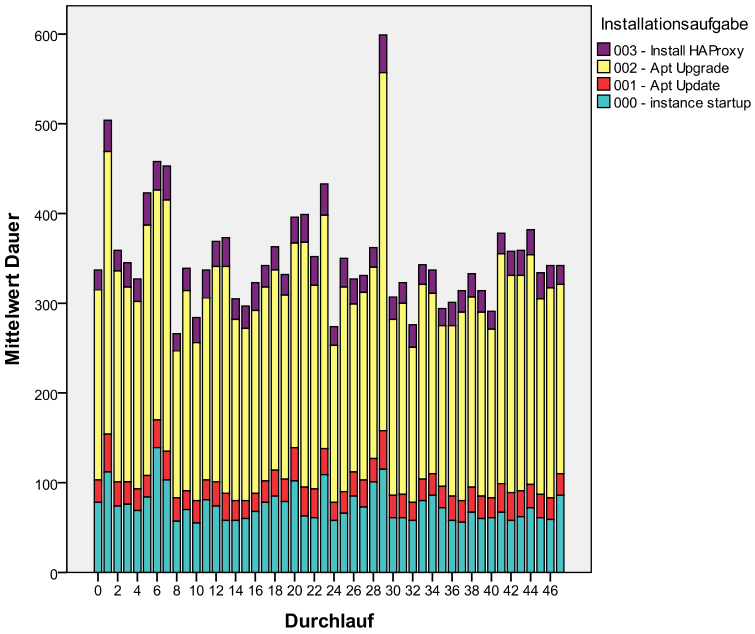

Abbildung 84: Loadbalancer Deploymentzeiten (ohne Cloud Standby)

NoSQL-Datenbank – Ähnlich wie beim Loadbalancer ist die NoSQL-Datanbank schnell deployt. Die Installation der Datenbank Redis nimmt im Vergleich zum Starten der Instanz wenig Zeit in Anspruch. Normalerweise wäre aufgrund der Datenbankgröße eine längere Dauer des NoSQL-Datenbank-Deployments zu erwarten. Würde man die komplette 30 GB Cache-Datenbank von barcoo einspielen, wäre dies auch der Fall. Da es sich bei dieser Datenbank jedoch nur um einen Cache handelt, der auch immer die gleiche, maximale Größe hat und nicht weiter anwächst, wird beim Notfallsystem aus Zeitgründen darauf verzichtet.

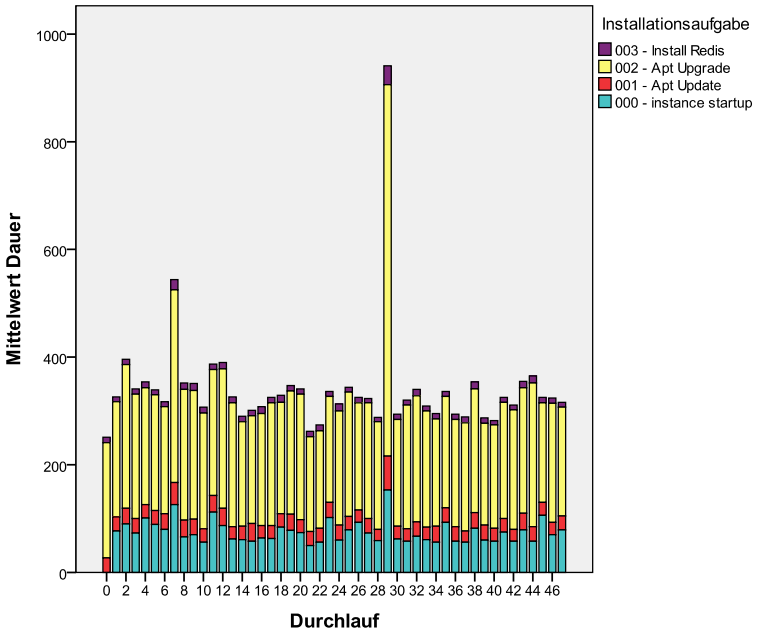

Abbildung 85: NoSQL Datenbankserver Deploymentzeiten (ohne Cloud Standby)

7.2.3 Analyse von Cloud Standby im Anwendungsfall

Nachdem das verteilte System des Anwendungsfalls ohne Cloud Standby untersucht wurde, werden im Folgenden die gleichen Experimente mit aktiviertem Cloud Standby durchgeführt, um vergleichbare Ergebnisse zu erhalten (siehe auch Kapitel 4.5.1). Außerdem erfolgt im Anschluss eine Analyse, die den Einfluss der Modellierung auf das RTO und dem aus der Absicherung zwangsläufig entstehenden Kosten-Overhead, näher beleuchtet.

Reduktion des RTO durch die Methode zur Notfallwiederherstellung

Betrachtet man das System zur Notfallwiederherstellung mit regelmäßiger Replikation, so lässt sich feststellen, dass die Deploymentzeiten nahe zusammenliegen. Neben einigen Ausreißern nach oben und unten im Verlauf der Experimente lässt sich vor allem feststellen, dass die erste Messung (Durchlauf 0) wesentlich länger dauert als die anderen (siehe Abbildung 86). Dies ist nicht überraschend, da beim erstmaligen Ausführen der Experimente für jeden Server initial alle Software installiert werden muss und nicht nur reine Aktualisierungen ausgeführt werden. Daher ist in den folgenden Analysen dieser Spezialfall ausgeklammert.

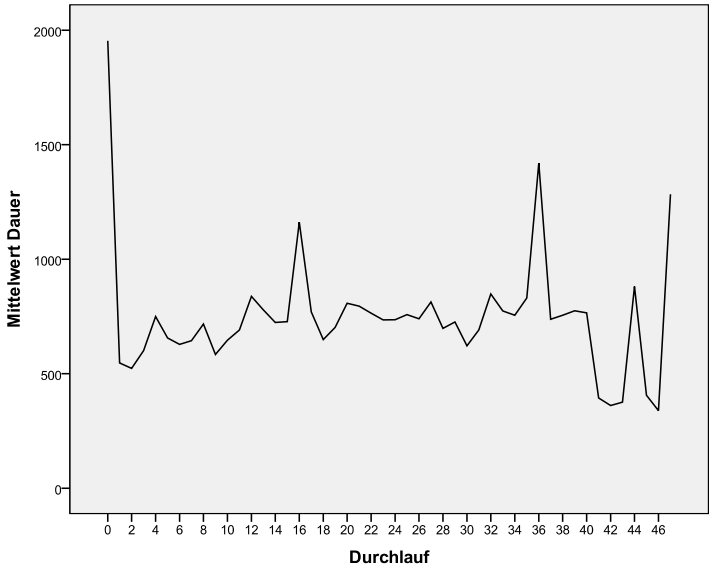

Abbildung 86: Entwicklung der Deploymentzeit mit Cloud Standby bei voranschreitender Alterung der Daten

Die Verteilung der restlichen Messpunkte lässt sich in der folgenden Abbildung 87 genauer betrachten. Es lässt sich erkennen, dass es zwar Ausreißer gibt, aber der Großteil (95%-Konfidenzintervall des Mittelwerts) der Datenpunkte befindet sich im Intervall von 662,88 bis 780,82 Sekunden. Der Mittelwert der Daten liegt bei 721,85 Sekunden, der Median bei 735 Sekunden und die Standardabweichung beträgt 200,84.

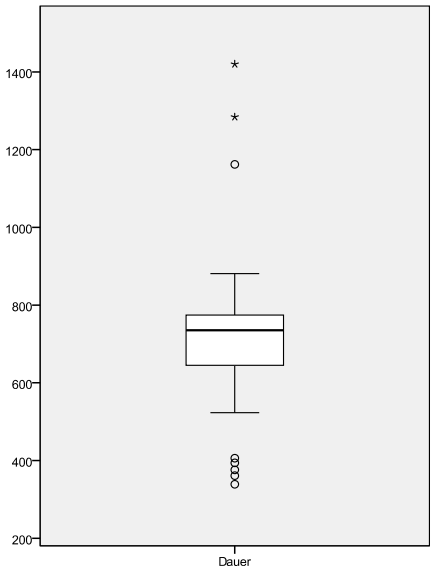

Abbildung 87: Verteilung der Deploymentzeiten mit Cloud Standby

Betrachtet man das Streudiagramm in Abbildung 88 analog zu der Analyse ohne Cloud Standby auf dem gleichen Intervall wie Abbildung 80, so sieht man, dass sich die Messergebnisse sehr nahe um den Mittelwert bewegen. Somit eignet sich der Mittelwert gut für eine Abschätzung der Deploymentzeit.

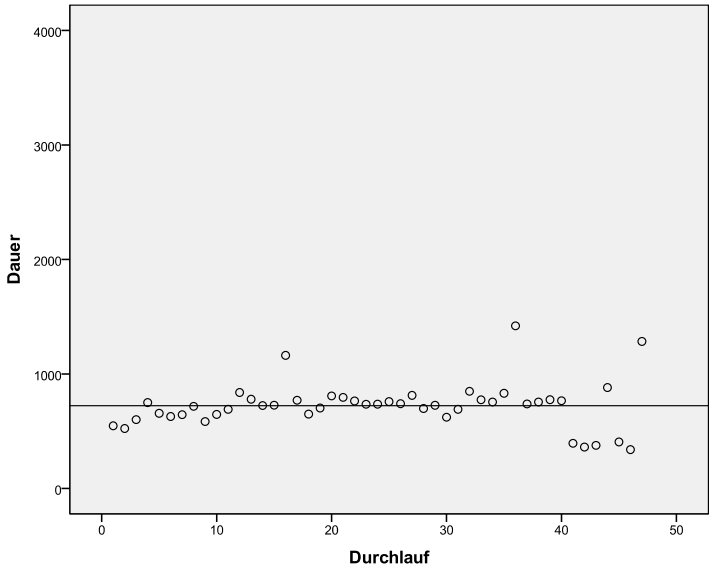

Abbildung 88: Streudiagramm mit Gerade (mit Cloud Standby)

Ein Vergleich der Untersuchungsergebnisse mit und ohne Cloud Standby verdeutlicht, dass Cloud Standby eine Reduzierung der Wiederherstellungszeit und damit auch des kleinstmöglichen RTO eines verteilten Systems bewirkt. Die anfangs aufgestellte Hypothese *ein durch Cloud Technologien unterstütztes Notfallmanagement ermöglicht die Absicherung eines verteilten Systems und verkürzt damit die Wiederherstellungszeit* wird somit bestätigt. Die beschriebene Reduktion der Deploymentzeit ist in Abbildung 89 als Kurve und in Abbildung 90 als Boxplot dargestellt.

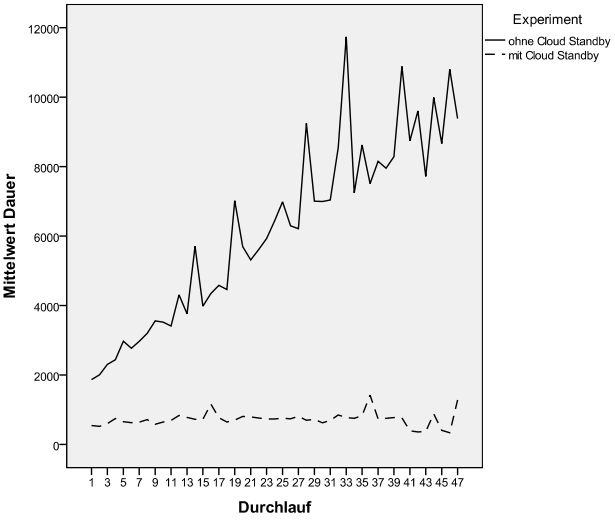

Abbildung 89: Vergleich der Deploymentzeiten über die Zeit (vgl. [86])

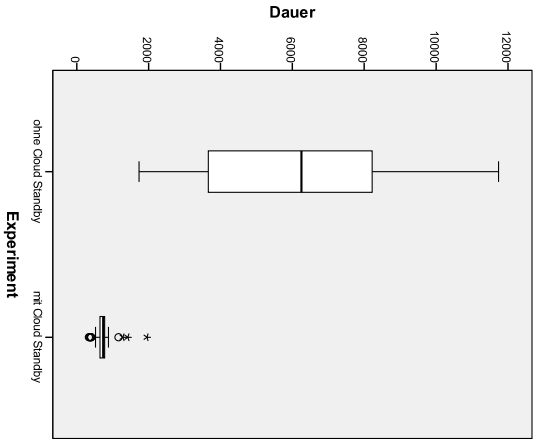

Abbildung 90: Vergleich der Deploymentzeit-Verteilung mit und ohne Cloud Standby

Untersucht man in der weitergehenden Analyse die Deploymentzeiten der einzelnen Komponenten (siehe Abbildung 91), so lässt sich feststellen, dass die einzelnen Komponenten durchaus unterschiedlichen Mustern folgen. Insbesondere fällt auf, dass der Applikationsserver mehrere Spitzen in der Deploymentzeit hat. Der Vollständigkeit halber werden nachfolgend jedoch auch die anderen Komponenten detailliert analysiert.

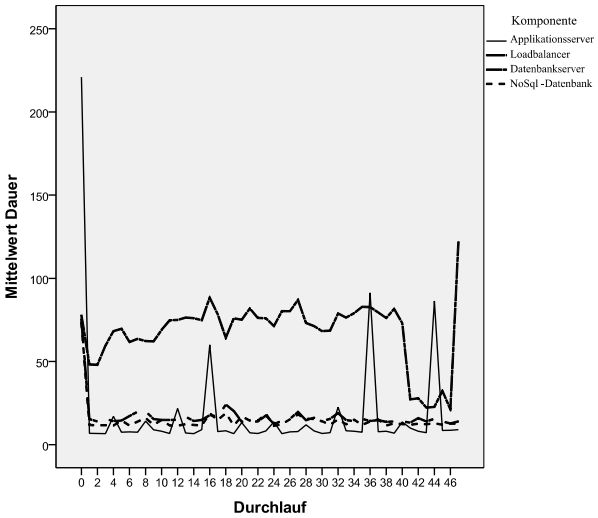

Abbildung 91: Vergleich der Deploymentzeiten nach Komponente

Datenbankserver – Der Datenbankserver zeigt das charakteristische Muster der Cloud Standby Replikation. Aufgrund der längeren Dauer des Deployments sind beim ersten Start die Installations-Betriebssystemaufgaben teurer, ab der zweiten Ausführung ist die Deploymentzeit jedoch hauptsächlich durch den Import des Backups im Schritt 006 bestimmt und die Deploymentzeit bleibt somit annähernd konstant.

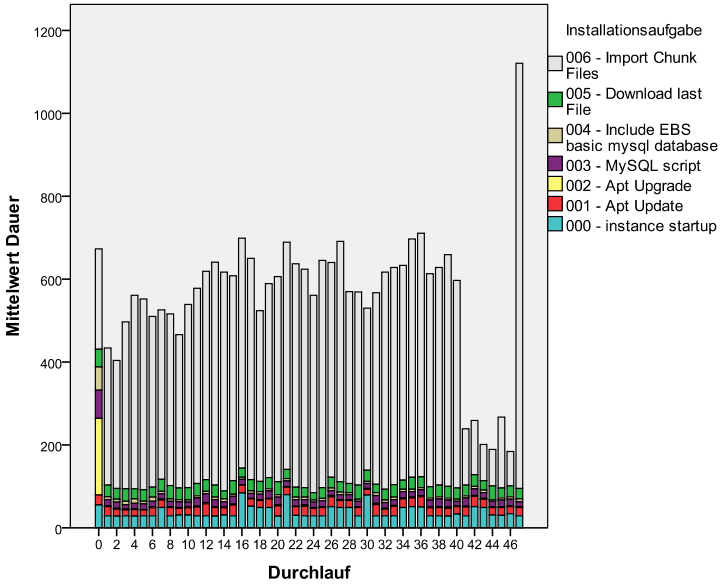

Abbildung 92: Datenbankserver Deploymentzeiten (mit Cloud Standby)

Applikationsserver – Die Deploymentzeiten des Applikationsservers fallen durch mehrere Spitzen auf. Während die kleinen Spitzen darauf zurückzuführen sind, dass eine neue dispora-Version nur alle vier Durchläufe vorliegt, so lassen sich die großen Spitzen nicht direkt aus der Versionierung von dispora und der Ausgestaltung der Simulation erklären. Bei der Betrachtung des gesamten Installations-Prozesses lässt sich jedoch feststellen, dass die hohen Spitzen immer dann auftreten, wenn eine neue Ruby Version installiert werden muss. Nach der Neuinstallation der Ruby Version muss offensichtlich auch beim „bundling“ eine zeitaufwendige Aufgabe durchgeführt werden.

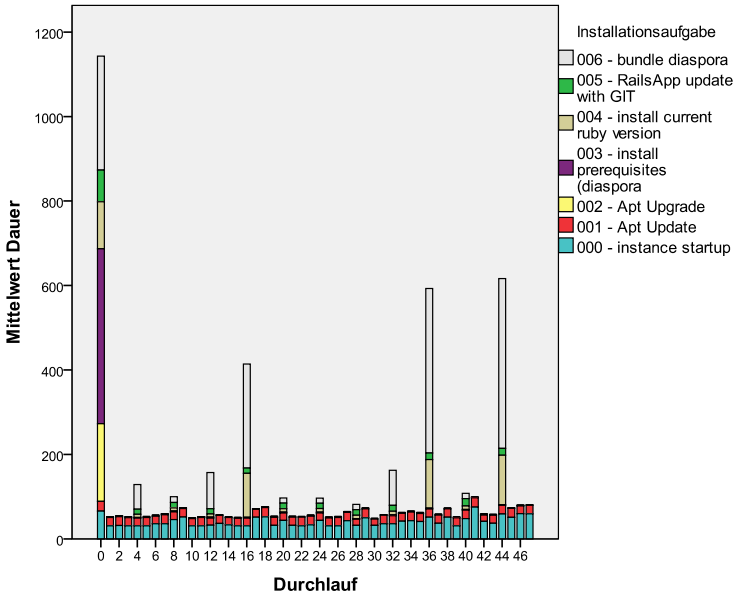

Abbildung 93: Applikationsserver Deploymentzeiten (mit Cloud Standby)

Loadbalancer – Das Deployment des Loadbalancers birgt keine Überraschungen, da der erste Durchlauf ähnlich wie beim Datenbankserver länger als die anderen Durchläufe dauert.

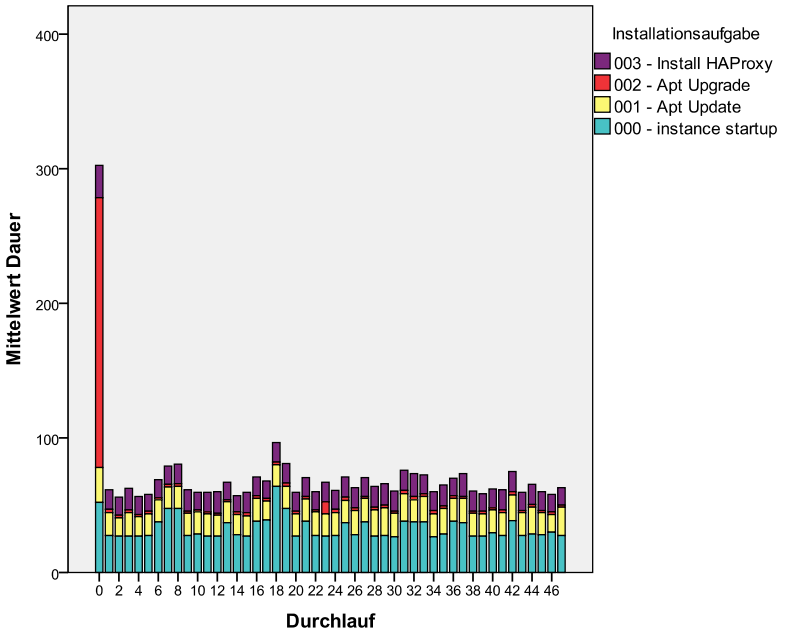

Abbildung 94: Loadbalancer Deploymentzeiten (mit Cloud Standby)

NoSQL-Datenbank – Auch bei der NoSQL-Datenbank verläuft die Installation wie erwartet. Nachdem das System installiert ist und nur noch aktualisiert werden muss, fällt die Deploymentzeit und bleibt daraufhin annähernd konstant.

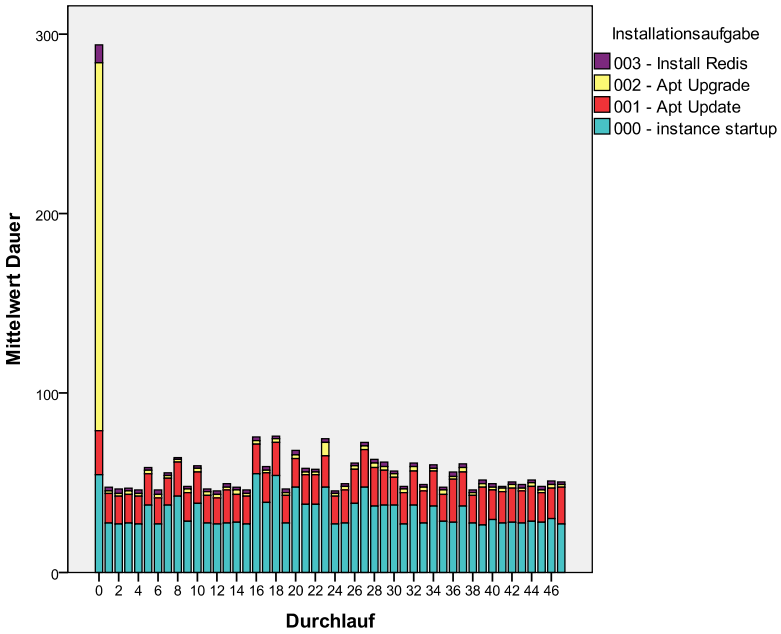

Abbildung 95: NoSQL Datenbankserver Deploymentzeiten (mit Cloud Standby)

Reduktion des RTO durch die modellbasierte Deployment-Methode

Soll ein verteiltes System deployt werden, sind wie bereits in Kapitel 1 beschrieben verschiedene Abhängigkeiten zu beachten. So könnte beispielsweise der Deploymentprozess des Applikationsservers voraussetzen, dass bereits eine Datenbank verfügbar ist. In der Beschreibungssprache

werden diese Abhängigkeiten mittels der Assoziation „Komponente benötigt Komponente“ explizit gemacht. Dieses Vorgehen ermöglicht es Fehler beim Deployment zu vermeiden. Aus der oben genannten Assoziation kann außerdem ein Deployment-Graph erstellt werden, um das Deployment automatisch durchzuführen. Hierzu muss aus dem Deployment-Graph die Reihenfolge abgeleitet werden, in der die verschiedenen Komponenten deployt werden müssen. Im einfachsten Fall wird der Deployment-Graph mittels einer Tiefensuche serialisiert und somit sichergestellt, dass keine Abhängigkeiten verletzt werden. Hierbei werden jedoch parallelisierbare Deployment-Schritte vernachlässigt. Aus diesem Grund wurde in Kapitel 5.6 ein Deploymentalgorithmus vorgestellt, der parallele Pfade berücksichtigt, wodurch es zu einer Reduktion der Deploymentzeit kommt. In Abbildung 96 ist ein Vergleich der Deploymentzeit bei seriellem Deployment und dem Deployment gemäß des hier vorgeschlagenen Algorithmus dargestellt. Die Experimente wurde gemäß der Methode in Kapitel 7.2.1 mit $n = 9$ durchgeführt.

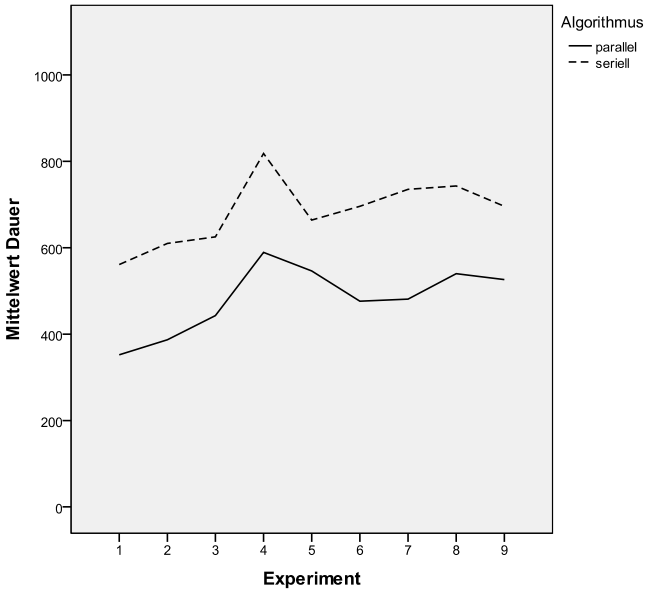

Abbildung 96: Einfluss des parallelen Deployments auf das RTO

Es lässt sich erkennen, dass sich bei allen durchgeführten Experimenten Deploymentzeit um ca. 200 Sekunden reduziert. In Abbildung 97 ist der absolute Wert der Reduktion bei jedem Experiment dargestellt.

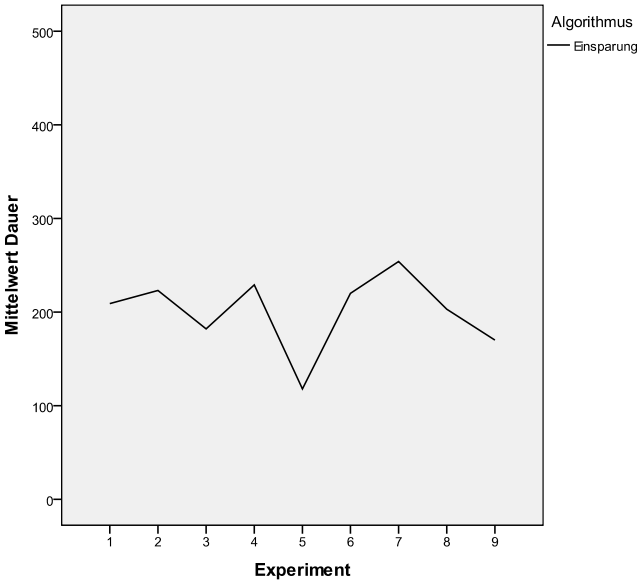

Abbildung 97: Einsparungen in der Deploymentzeit durch die Modellierung

Anbieterunabhängigkeit durch die modellbasierte Deployment-Methode

Neben der Reduktion der Deploymentzeit ist das Ziel der modellbasierten Deployment-Methode, das anbieterunabhängige Deployment zu ermöglichen. Um dies zu belegen, wurde das im Anwendungsfall beschriebene verteilte System bei mehreren Anbietern deployt.

Wie man in Abbildung 98 erkennen kann, war das Deployment bei den Anbietern AWS EC2[38], Google[39], HP[40] und Rackspace[41] in zu Teil ver-

[38] http://aws.amazon.com

schiedenen Cloud-Standorten erfolgreich. Es hat sich außerdem gezeigt, dass sich die Deploymentzeiten bei den verschiedenen Anbietern oftmals stark unterscheiden. Warum dies so ist, wurde im Rahmen dieser Arbeit nicht weiter untersucht, sollte jedoch in künftigen Arbeiten näher untersucht werden.

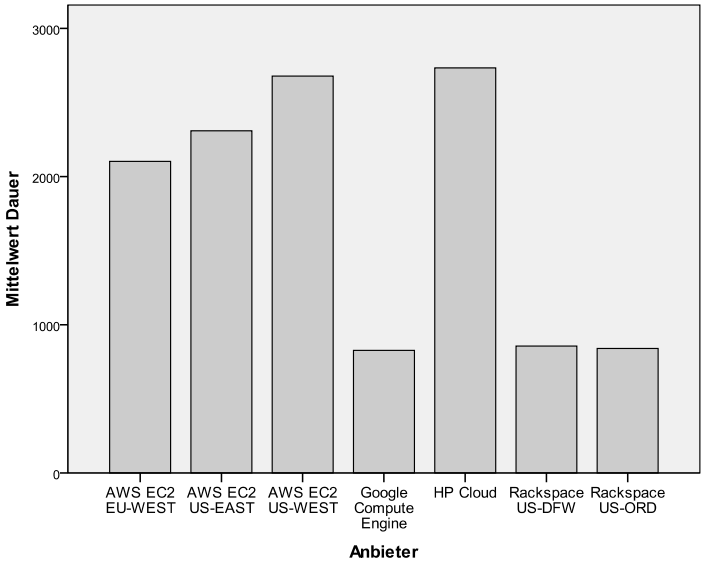

Abbildung 98: Vergleich der Deploymentzeiten des initialen Deployments bei verschiedenen Anbietern

[39] http://cloud.google.com/products/compute-engine

[40] http://www.hpcloud.com

[41] http://www.rackspace.com

Kosten für die Absicherung

Wie in Kapitel 7.2.1 dargelegt, hängen die Kosten von Cloud Standby hauptsächlich von dem gewählten Update-Intervall ab. Für den Anwendungsfall sind die Kosten in Abhängigkeit des Update-Intervalls in Abbildung 99 dargestellt[42].

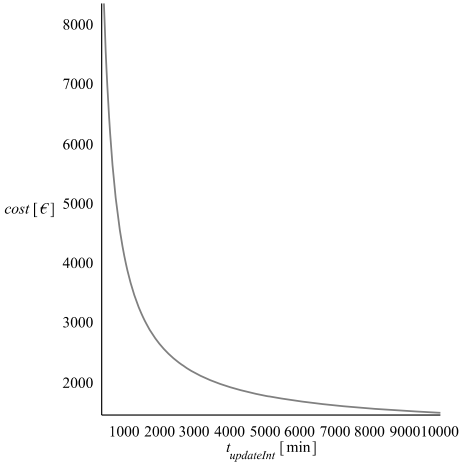

Abbildung 99: Kosten für Cloud Standby im Anwendungsfall

Um eine Aussage über die Kosten für ein bestimmtes RTO treffen zu können, muss ein Zusammenhang zwischen dem Update-Intervall und dem RTO hergestellt werden. Die Untersuchungen des Systems ohne Cloud

[42] Da die Kosten für die Speicherung der Images marginal sind, werden diese Kosten hier nicht mit betrachtet. Es würden pro 30GB Server nochmals zusätzlich 36€ pro Jahr, also 252€ für das gesamte verteilte System anfallen.

Standby in Kapitel 7.2.2 haben gezeigt, dass das Alter der Daten die Deploymentzeit beeinflusst und damit das kleinstmögliche RTO. Über die Funktion der Deploymentzeit lässt sich eine Approximation des RTO für verschiedene Update-Intervalle herstellen, indem man die Funktion für die Deploymentzeit des Primärsystems als obere Schranke für die Deploymentzeit des Systems mit Cloud Standby nutzt. Damit ergibt sich für das RTO:

$$t_{Depl}(t_{UpdateInt}) \leq RTO$$

Mit Hilfe dieser Approximation lassen sich dann weitere Berechnungen, wie z.B. der Kosten in Abhängigkeit des RTO, durchführen.

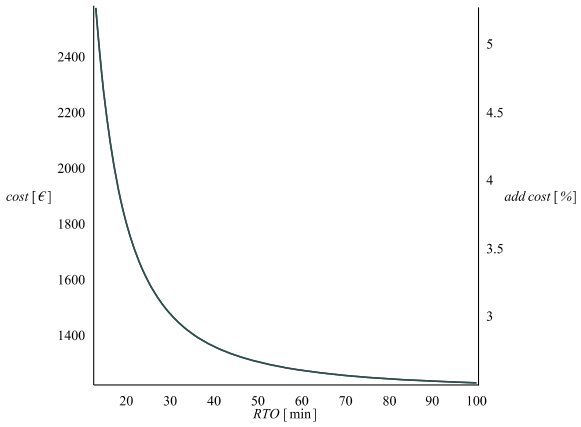

Abbildung 100: Absolute Kosten und prozentuale zusätzliche Kosten für Cloud Standby in Bezug auf das RTO

In Abbildung 100 sind auf der linken vertikalen Achse die Gesamtkosten für die Absicherung zu sehen, während auf der horizontalen Achse das

kleinstmögliche RTO abgetragen ist. Betrachtet man die absoluten Werte, so erscheinen diese hoch, setzt man sie jedoch in Relation zu den ursprünglichen Betriebskosten des Primärsystems, so wird deutlich, dass auch mit geringen prozentualen Mehrkosten (siehe rechte vertikale Ache) ein geringes RTO zu erreichen ist. Beispielsweise ist bereits mit 4% Mehrkosten ein RTO von ca. 20 Minuten möglich. Als Grenze für den kleinstmöglichen RTO wurde hier der Mittelwert der Experimente (720 Sekunden = 12 Minuten) gewählt.

Zusammenfassend wurde gezeigt, dass Cloud Standby eine Reduktion des RTO ermöglicht und dies mit vertretbaren Kosten zu erreichen ist. Für die Bewertung des Nutzens eines solchen Standby-Ansatzes könnte es jedoch auch sinnvoll sein, neben der reinen RTO und der Kosten außerdem die Langzeitbetrachtung mit einzubeziehen. Sollten beispielsweise die Kosten, die durch die Nichtverfügbarkeit eines Systems entstehen, sehr gering oder sehr hoch sein, könnte die Nutzung von Cloud Standby auf lange Sicht eher nachteilig sein, sodass der Einsatz eines Hot-Standbys oder der Verzicht auf eine Absicherung sinnvoller wären.

Im Folgenden wird nun in einer weiteren simulativen Evaluierung der Nutzen von Cloud Standby über einen längeren Zeitraum untersucht. Hierzu wird zunächst eine neue Berechnungsmethode für die Kosten und Verfügbarkeit über einen längeren Zeitraum unter Berücksichtigung der Ausfallkosten vorgestellt, die dann zur Evaluierung auf den Anwendungsfall angewandt wird.

7.3 Simulative Evaluierung

Im Folgenden werden die Berechnungen anhand des Anwendungsfalls aus Kapitel 7.1 mit Hilfe der Berechnungsmethodik aus Kapitel 4.5.2 durchgeführt. Aus dem Anwendungsfall und der experimentellen Evaluierung in Kapitel 7.2 ergeben sich die in Tabelle 10 aufgeführten Parameter. $t_{initDepl}$ leitet sich aus der Deploymentzeit des ersten Experiments ohne Standby ab

(siehe Kapitel 7.2.2), t_{update} entspricht dem Mittelwert der Experimente mit Cloud Standby (siehe Kapitel 7.2.3) und t_{error} ist eine Annahme, dass es einen Tag dauert, das System aus einem Totalausfall manuell wiederherzustellen. Die Kosten ergeben sich aus dem Anwendungsfall und werden als fix angenommen[43].

Tabelle 10: Parameter aus dem Anwendungsfall und der experimentellen Evaluierung

Variable	Wert
$t_{initDepl}$	35 min
t_{update}	12 min
t_{error}	1440 min
n_{server}	7
$cost_1$	0,68€/h/Server[44]
$cost_2$	0,68€/h/Server
$avail_1$	10 Jahre
$avail_2$	10 Jahre

Für die Berechnung der Kosten und Verfügbarkeit ist die Bestimmung der Deploymentzeit des Notfallsystems (t_{Depl}) für verschiedene Update-Intervalle erforderlich. Auch hierzu werden die in Kapitel 7.2.2 durchgeführten Experimente genutzt. Dank der Untersuchung des Systems ohne

[43] Es ist dabei zu beachten, dass sich über einen so langen Betrachtungszeitraum die Kosten der Anbieter verändern werden, die Inflation eine Rolle spielt und so weiter. Da jedoch in dem hier vorgestellten Verfahren immer zwei Systeme verglichen werden und beide Varianten auf Cloud Computing basieren, fallen diese Kosteneffekte nicht so sehr ins Gewicht, da sie in beiden Systemen anfallen. Siehe hierzu auch Kapitel 4.5.2.

[44] Preis für eine „Extra Large“ Amazon EC2-Instanz in der Verfügbarkeitszone EU-West im Juni 2014

Cloud Standby ist deutlich geworden, wie sich die Deploymentzeit über die Zeit verhält, wenn keine Aktualisierung des Notfallsystems durchgeführt wird. Die in diesem Kapitel interpolierte Funktion kann somit genutzt werden, um die Deploymentzeit für das Notfallsystem vorherzusagen, wenn das Update-Intervall eine gewisse Größe hat:

$$\mathrm{t}_{Depl}(t) = 0{,}2459\, x^{0{,}519}, t_{updateInt} \in [60, \infty]$$

Grafisch lässt sich die Funktion, wie in Abbildung 101 gezeigt, darstellen.

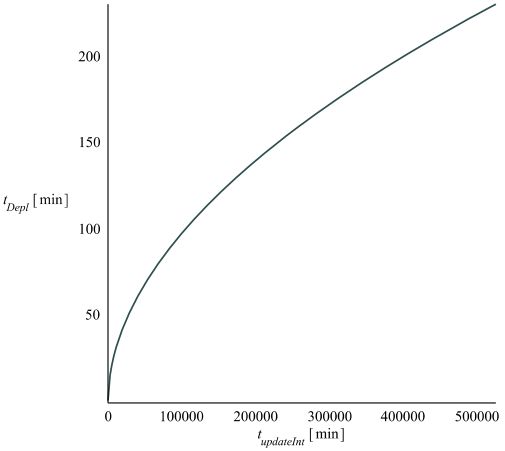

Abbildung 101: Verhältnis der Deploymentzeit zum Update-Intervall (vgl. [84], [85])

Verhältnis der Ausfallkosten zu dem Update-Intervall

Mit Hilfe der stationären Verteilungen und der Kosten aus Tabelle 10 können nun die Kostenfunktionen $\gamma_1$und $\gamma_2$in Abhängigkeit von $t_{\mathrm{updateInt}}$ und

$cost_e$ bestimmt und in einem Schaubild dargestellt werden (Abbildung 102).

So erkennt man, dass es Kombinationen gibt, in denen die γ_1 geringere Funktionswerte (Gesamtkosten) besitzt, und andere, in denen γ_2 geringer ist.

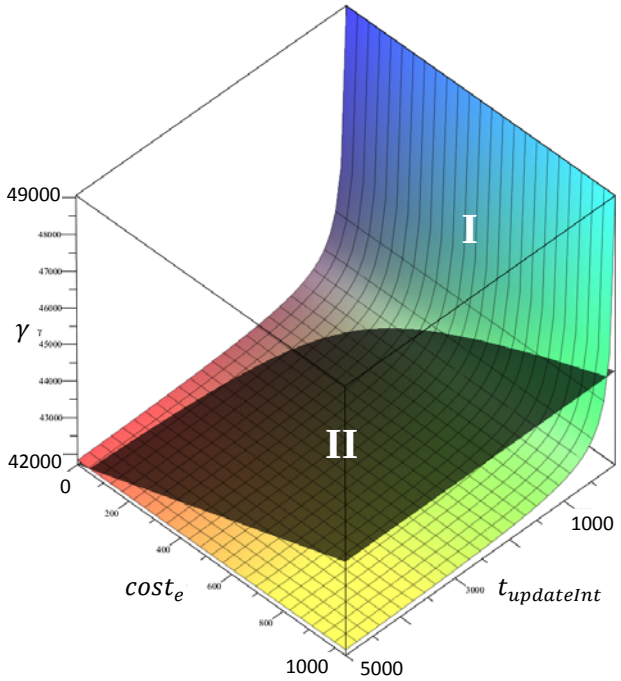

Abbildung 102: Vergleich der Gesamtkosten γ_1(helle Fläche I) und γ_2(dunkle Ebene II) bei variablem $t_{updateInt}$ und $cost_e$ (vgl. [84], [85])

Durch den Schnitt der Funktionen erhält man eine Kurve, auf der beide Systeme die gleichen Kosten haben. Diese Funktion ist in Abbildung 103 dargestellt. Neben den gleichen Kosten (graue Linie) lassen sich auch die Kombinationen ablesen, in denen Cloud Standby dem System ohne Cloud

Standby monetär unterlegen ist (weißer Bereich), und in denen Cloud Standby günstiger ist (grauer Bereich).

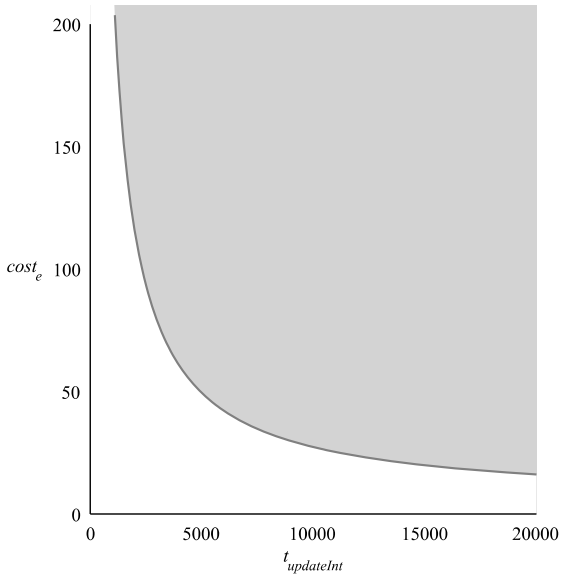

Abbildung 103: $t_{updateInt}$ und $cost_e$ Kombinationen, in denen das Replikationssystem günstiger (grauer Bereich) ist, gleich viel kostet (graue Linie) und teurer ist (weißer Bereich) (vgl. [84]–[86])

Die Grenzwerte der Funktion $cost_e(t_{updateInt})$ ergeben das Intervall, in dem ein Cloud-Standby-Ansatz auf Basis von Gesamtkosten sinnvoll ist:

$$cost_e^{min} = \lim_{t \to 60} cost_e\,(t) = \ 2989{,}97\ €/h$$

$$cost_e^{max} = \lim_{t \to \infty} cost_e\,(t) = \ 4{,}87\ €/h$$

Liegen die Ausfallkosten der Unternehmung unter den angenommenen Werten für die Serverkosten, Ausfallzeiten, etc. bei mehr als 2989,97 € pro Stunde, so sollte auf jeden Fall ein Replikationssystem eingesetzt werden. Bei solch hohen Kosten bietet es sich jedoch an, auf einen Hot-Standby-Ansatz zurückzugreifen. Es können ohne weitere Kosten zwei Systeme parallel betrieben werden. Liegen die Kosten unter 4,87 € pro Stunde, ist es unter den hier getätigten Annahmen nicht sinnvoll, ein Replikationssystem einzusetzen. Egal wie groß das Replikationsintervall gewählt wird, die Nutzung eines einfachen, ungesicherten Systems ist aus Kostensicht – nicht jedoch aus Verfügbarkeitssicht – sinnvoller.

Verhältnis der Verfügbarkeit zum Update-Intervall

Nutzt man die Werte aus Tabelle 10, so lassen sich die Verfügbarkeitsfunktionen α_1 und α_2 in Abhängigkeit von $t_{updateInt}$ berechnen. Allein durch die Einführung des Replikationssystems die Gesamtverfügbarkeit des Systems merklich. Aus einer Verfügbarkeit von 99,972% wird eine Verfügbarkeit von 99,999%:

$$\alpha_1^{min} = \lim_{t_{updateInt} \to \infty} \alpha_1\left(t_{updateInt}\right) = 0{,}9999930832$$

$$\alpha_1^{max} = \lim_{t_{updateInt} \to 60} \alpha_1\left(t_{updateInt}\right) = 0{,}9999993433$$

$$\overline{\alpha}_2 = \alpha_2\left(t_{updateInt}\right) = 0{,}999719299193525$$

Da eine Ausfallzeit von $t_{error} > 0$ angenommen wurde, macht es aus Verfügbarkeitssicht selbst bei einem Update-Intervall, das gegen unendlich geht (α_1^{min}) Sinn, Cloud Standby einzusetzen. Im Vergleich zu dem System ohne Cloud Standby ($\overline{\alpha}_2$) ist die Verfügbarkeit um einiges höher.

Kostenneutrales Update-Intervall

Es soll nun das kostenneutrale Update-Intervall bestimmt werden, d.h. die Zeit $\overline{t}_{updateInt}$ in der das Standardsystem und das Replikationssystem die gleichen Kosten erzeugen. Hierzu wird angenommen, dass die Ausfallkosten bestimmt sind:

$$cost_e = 400€/\text{h}$$

Mit Hilfe dieser Ausfallkosten können nun die neuen Kostenfunktionen aufgestellt werden:

$$\gamma_{i,400}(t_{updateInt}) = \gamma_i(t_{updateInt}, 400), i \in \{1,2\}$$

Die Grenzwertbetrachtung ergeben die maximalen und minimalen Kosten:

$$\gamma_{1,400}^{min} = \lim_{t_{updateInt} \to \infty} \gamma_{1,400}(t_{updateInt}) = \ 41750{,}70\ € \text{ / Jahr}$$

$$\gamma_{1,400}^{max} = \lim_{t_{updateInt} \to 60} \gamma_{1,400}(t_{updateInt}) = \ 49057{,}02\ € \text{ / Jahr}$$

Mit der Funktion $\gamma_{2,400}(t_{updateInt})$ können die Kosten für die Nutzung des Systems ohne Replikation ermittelt werden. Diese Kosten sind unabhängig von t und damit konstant. Trägt man die Kosten in ein Schaubild (Abbildung 104) ab, so wird deutlich, dass sich die Kosten $\gamma_{1,400}$ mit steigendem Updateintervall reduzieren und irgendwann mit $\gamma_{2,400}$ schneiden:

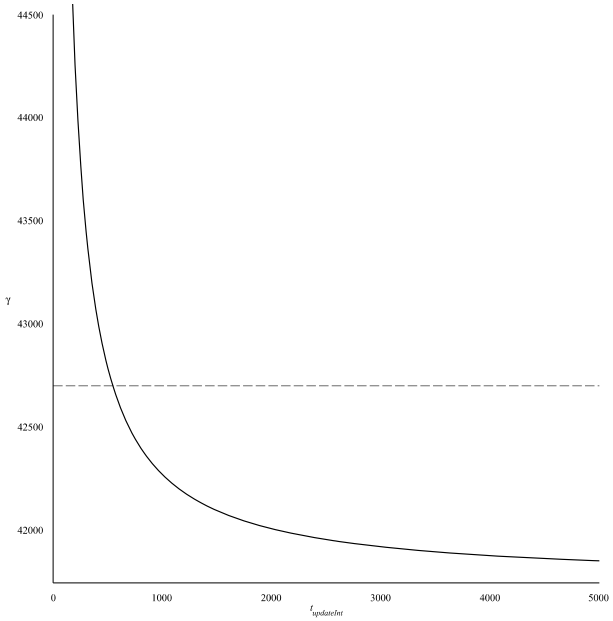

Abbildung 104: Kostenentwicklung bei steigendem Backup-Intervall (schwarze Kurve: $\gamma_{1,400}$, graue gestrichelte Gerade: $\gamma_{2,400}$, vgl [84], [85])

Durch Lösen der Gleichung

$$\gamma_{1,400}(t_{updateInt}) = \gamma_{2,400}(t_{updateInt})$$

nach $t_{updateInt}$ lässt sich das Update-Intervall bestimmen, das ohne monetären Mehraufwand gewählt werden kann:

$$\overline{t}_{updateInt} = 546{,}63\ min$$

Mit einem Intervall von 546 Minuten, also etwas weniger als einem Update täglich (alle 9 Stunden), kann im gegebenen Beispiel das System ohne höhere Kosten verfügbarer gemacht werden. Es ergibt sich hieraus folgende Änderung in der Verfügbarkeit:

$$\alpha_1(546) - \alpha_2(546) = 0{,}0002792$$

Das heißt, das System ist um 0,027 Prozentpunkte verfügbarer. Dies mag auf den ersten Blick nicht viel erscheinen, aber auf die angenommene Zeitspanne von 10 Jahren ergibt sich hieraus, dass das System 1440 Minuten oder einen Tag länger verfügbar ist und damit auch ein Aufstieg von Verfügbarkeitsklasse 3 in Verfügbarkeitsklasse 4:

$$\alpha_2(546) = 0{,}999719299193525$$

$$\alpha_1(546) = 0{,}999998503921401$$

7.4 Zusammenfassung

In diesem Kapitel wurde ein Anwendungsfall vorgestellt, auf dessen Basis eine experimentelle und simulative Evaluation durchgeführt wurde. Bei der experimentellen Evaluierung wurde das im Anwendungsfall beschriebene verteilte System zunächst ohne Cloud Standby analysiert. Dabei hat sich gezeigt, dass die Deploymentzeit mit steigender Datenmenge über die Zeit zunimmt. Diesem Ansteigen der Deploymentzeit und damit auch einer Erhöhung eines möglichen RTO kann mit Cloud Standby entgegengewirkt werden. Die Analyse des Systems mit Cloud Standby hat gezeigt, dass sich die Deploymentzeit reduzieren lässt und die Hypothese aus Kapitel 1 wurde somit bestätigt. Zusätzlich zeigten die durchgeführten Experimente, dass

durch die Nutzung der modellbasierten Deployment-Methode die Deploymentzeit im Vergleich zu einem seriellen Deployment reduziert werden kann und dass es mit Hilfe dieser Methode möglich wird das verteilte System anbieterunabhängig zu deployen.

Da die Absicherung mit Cloud Standby mit Kosten verbunden ist, wurden diese im Rahmen der experimentellen Evaluierung für den Anwendungsfall bestimmt und dann im Zuge der simulativen Evaluierung unter Berücksichtigung von Ausfallkosten näher untersucht. Die simulative Evaluierung hat gezeigt, dass der Einsatz von Cloud Standby immer eine Erhöhung der Verfügbarkeit mit sich bringt. Es wurde außerdem gezeigt, dass sich im Vergleich eines Systems ohne Cloud Standby dieses über lange Zeit bezahlt macht, da (bei Berücksichtigung der Ausfallkosten) bei den gleichen Kosten mit Cloud Standby eine höhere Verfügbarkeit erreicht werden kann.

8. Fazit

Kleine und Mittelständische Unternehmen (KMU) sehen sich in ihrem Alltag immer wieder Gefahren ausgesetzt. So hat sich in der Vergangenheit gezeigt, dass selbst zuverlässige Anbieter mit vielen eigenen Rechenzentren mit Ausfällen zu kämpfen haben [88] und auch große Unternehmen plötzlich den Betrieb einstellen müssen [33]. Hierdurch kann es zu einer Unterbrechung der Geschäftsprozesse und zum Datenverlust kommen. Laut Studien führen zwar 94 % der KMUs in Deutschland regelmäßig Datensicherungen durch [19], jedoch sichern gerade einmal 50% ihre kritischen Prozesse und die daran beteiligten Systeme mit einem Notfallsystem bei einem anderen Anbieter ab [122]. Rund 52 % der Unternehmen geben an, für die Absicherung ihrer Systeme eine zu geringe Ausstattung an Budget, IT-Ressourcen oder Wissen zu haben [122]. Die gleichen KMUs beziffern im Durchschnitt ihre Ausfallkosten mit $25.000 pro Tag [122]. Eine mögliche Lösung, um hohen Kosten bei der Bereithaltung von sporadisch genutzten Ressourcen zu begegnen, ist die Nutzung von öffentlichen Cloud Ressourcen [80], [94].

Hierzu wurde in Kapitel 4 eine neue Methode entwickelt, die aus einem Notfallwiederherstellungsprozess, einem Aktualisierungsprotokoll und einer Entscheidungsunterstützung zur Konfiguration des Prozesses besteht. Die Methode zur Notfallwiederherstellung baut dabei auf bestehenden Datensicherungslösungen und einer Methode zum anbieterunabhängigen Deployment eines verteilten Systems in der Cloud auf. Eine solche Deployment-Methode wurde in Kapitel 5 entwickelt. Sie besteht aus einer Beschreibungssprache, einem Deploymentprozess, einem Deploymentprotokoll und einem Deploymentalgorithmus. Die beiden neuen Methoden werden gemeinsam Cloud Standby genannt und wurden, wie in Kapitel 6 beschrieben, prototypisch implementiert.

Da die Absicherung mit einem Standby-System immer das Ziel hat, bei einem Notfall das Notfallsystem rechtzeitig wieder in Betrieb zu nehmen, wurde hierauf in der Evaluation in Kapitel 7 ein besonderes Augenmerk gelegt. Es wurde gezeigt, dass sich durch die Nutzung von Cloud Standby allgemein die Deploymentzeit und damit das kleinstmögliche RTO im konkreten Anwendungsfall bis um den Faktor 20 reduzieren lassen. Die Einsparung ergibt sich zum einen aus der Methode zur Notfallwiederherstellung selbst und zum anderen aus der Planung des Deployments mittels des Deploymentalgorithmus. Im Anwendungsfall war es so möglich ein RTO von ca. 30 Minuten mit ca. 3% zusätzlichen Kosten zu erreichen. Eine Langzeitbetrachtung der Kosten ergab außerdem, dass bei angenommenen 400€ Ausfallkosten pro Stunde[45] das Notfallsystem alle 9 Stunden ohne Mehrkosten aktualisiert werden konnte.

Für den Notfall bedeutet dies, dass nur die Startzeit der virtuellen Maschinen, nicht jedoch der Datentransfer zu Gewicht schlägt. Zusätzlich wurde im Rahmen der Evaluation gezeigt, dass es mit der Deployment-Methode möglich ist, ein verteiltes System auf 7 verschiedenen Clouds bei 4 verschiedenen Anbietern abzusichern.

Diese Ergebnisse sind im Folgenden nochmals zusammengefasst und werden im Anschluss einer kritischen Betrachtung unterzogen. Weiterhin wird ein Ausblick auf aufbauende wissenschaftliche Fragestellungen sowie mögliche Erweiterungen technischer und konzeptioneller Art geben. Im Anhang befindet sich die Umsetzung der Entscheidungsunterstützung mit Maple, eine Sprachreferenz der Beschreibungssprache und das Listing des Metamodells als XML.

[45] Diese Zahl ergibt sich aus den in Kapitel 1 dargestellten Ausfallkosten eines typischen KMU von $25.000 pro Tag [122] und der sehr konservativen Ableitung von 400€ pro Stunde.

8.1 Ergebnisse

Mit Cloud Standby soll es möglich sein, die Dauer des Deployments im Notfall zu reduzieren und ein anbieterunabhängiges Deployment zu ermöglichen, sodass beim Ausfall eines Anbieters das verteilte System bei einem anderen Anbieter deployt werden kann. Im Folgenden werden also die Ergebnisse zur *Reduktion des RTO* und die *Anbieterunabhängigkeit* präsentiert.

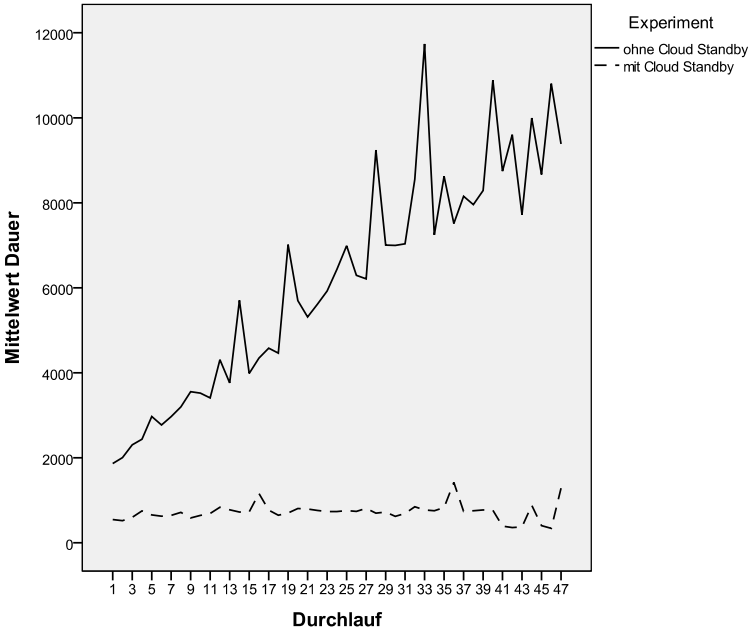

Abbildung 105: Vergleich der Deploymentzeiten über die Zeit (vgl. [86])

Reduktion des RTO – Betrachtet man, wie in Kapitel 7.2.3 geschehen, die Deployment-Dauer eines mit diesem Ansatz abgesicherten Systems,

so lässt sich feststellen, dass mit steigender Datensicherungsgröße der Unterschied der Deploymentzeiten zunimmt (siehe Abbildung 105). Wird Cloud Standby eingesetzt, so bleibt die Deploymentzeit konstant. Das verteilte System deployt also nicht nur zunehmend schneller, sondern die Zeit für das Deployment lässt sich im Voraus abschätzen. Durch die Reduktion der Deploymentzeit lässt sich damit auch ein niedrigeres RTO einhalten. Die Reduktion der Deploymentzeit setzt sich dabei teilweise aus der Notfallwiederherstellung und der Deployment-Methode zusammen. In Abbildung 106 ist der Einfluss der Modellierung von Abhängigkeiten zwischen Komponenten auf die Deploymentzeit nochmals dargestellt. Im hier untersuchten Anwendungsfall lässt sich die Deploymentzeit im Vergleich zu einem sequenziellen Deployment um ca. 30% verkürzen.

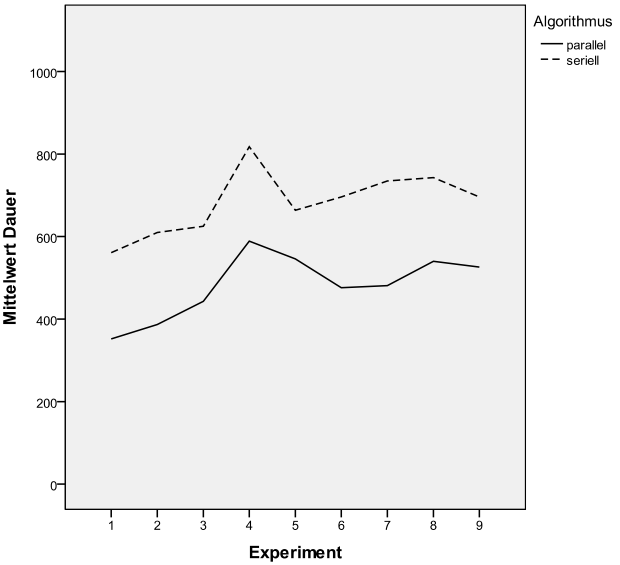

Abbildung 106: Einfluss des parallelen Deployments auf das RTO

Anbieterunabhängigkeit – Um ein verteiltes System anbieterunabhängig abzusichern, muss die gewählte Deployment-Methode auch eine Anbieterunabhängigkeit unterstützen. Es wurde in Kapitel 7.2.3 gezeigt, dass sich das im Anwendungsfall beschriebene verteilte System bei 4 verschiedenen Anbietern (Amazon Webservices, Google, HP und Rackspace) starten lässt (siehe Abbildung 107). Weitere Anbieter werden von der in Kapitel 6.3 vorgestellten Implementierung unterstützt. Der Fokus dieser Evaluierung wurde jedoch nur auf die größten Anbieter gelegt und es konnte gezeigt werden, dass das Deployment hier möglich ist.

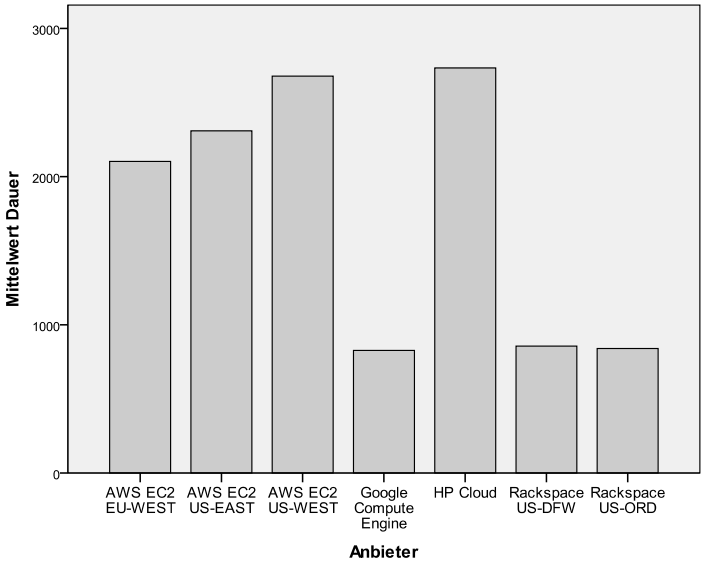

Abbildung 107: Vergleich der Deploymentzeiten des initialen Deployments bei verschiedenen Anbietern

Da eine Absicherung mittels eines Standby-Systems immer mit zusätzlichen Kosten verbunden ist, werden im Folgenden die Kosten der Absicherung dargelegt[46]. Die *zusätzlichen Kosten* ergeben sich aus dem periodischen Starten von virtuellen Maschinen bei einem zweiten Cloud-Anbieter. In diesem Fall sind stets Mehrkosten zu erwarten. Möchte man analysieren, wann sich ein Standby-System monetär lohnt, so sind immer auch die Ausfallkosten von Interesse, da diese *langfristige Einsparungen* darstellen.

Zusätzliche Kosten – Wie in der Einleitung beschrieben, sind mit Notfallsystemen auch zusätzliche Kosten verbunden. Diese sind, ohne Einbeziehung der Ausfallkosten, reiner Overhead. Für den Anwendungsfall liegt der Overhead für ein RTO im Bereich [20;50] min zwischen 2-3% der Kosten für das Hosting des Primärsystems (siehe Abbildung 108). Geht man außerdem von Speicherkosten in Höhe von – im schlimmsten Fall – 1.000€ pro Jahr aus, liegen die zusätzlichen Kosten bei ca. 5%.

[46] Hierbei wird sich auf die Laufzeitkosten der virtuellen Maschinen und nicht auf die Speicherkosten konzentriert, da letztere wie bereits in Kapitel 7.2.3 dargelegt marginal sind.

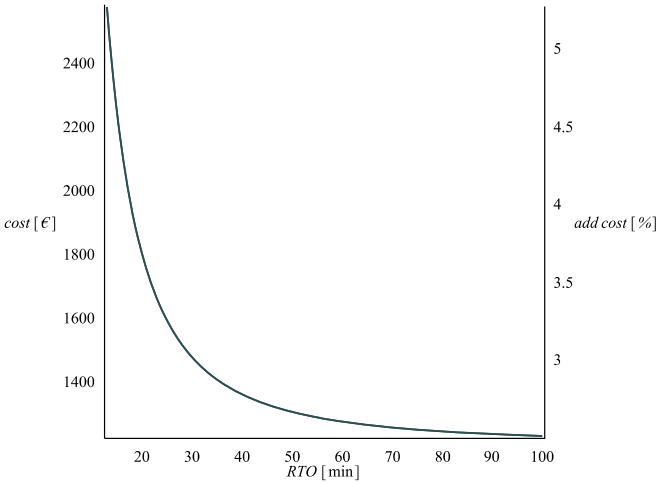

Abbildung 108: Absolute Kosten (cost) und prozentuale zusätzliche Kosten (add cost) für Cloud Standby in Bezug auf das RTO

Langfristige Einsparungen – Berücksichtigt man in einer Langzeitbetrachtung die Ausfallwahrscheinlichkeit des Primärproviders und die Ausfallkosten der Geschäftsprozesse, so lässt sich bestimmen, in welchem Bereich der in dieser Arbeit vorgestellte Ansatz günstiger ist als die Inkaufnahme eines Ausfalls. Aus Abbildung 109 lässt sich entnehmen, dass bei geringen oder sehr hohen stündlichen Ausfallkosten die Absicherung mit dem hier vorgestellten Ansatz nicht sinnvoll ist.

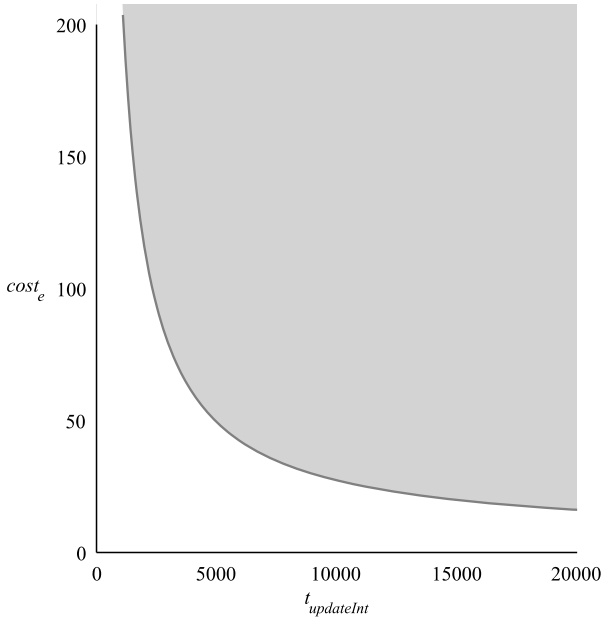

Abbildung 109: $t_{updateInt}$ und $cost_e$ Kombinationen, in denen das Replikationssystem günstiger (grauer Bereich) ist, gleich viel kostet (graue Linie) und teurer ist (weißer Bereich) (vgl. [84]–[86])

Aus Entscheidersicht könnte es aber auch interessant sein, in welcher Ausfallkosten-/RTO-Kombination der Einsatz von Cloud Standby sinnvoll ist. Dies ist in Abbildung 110 nochmals dargestellt.

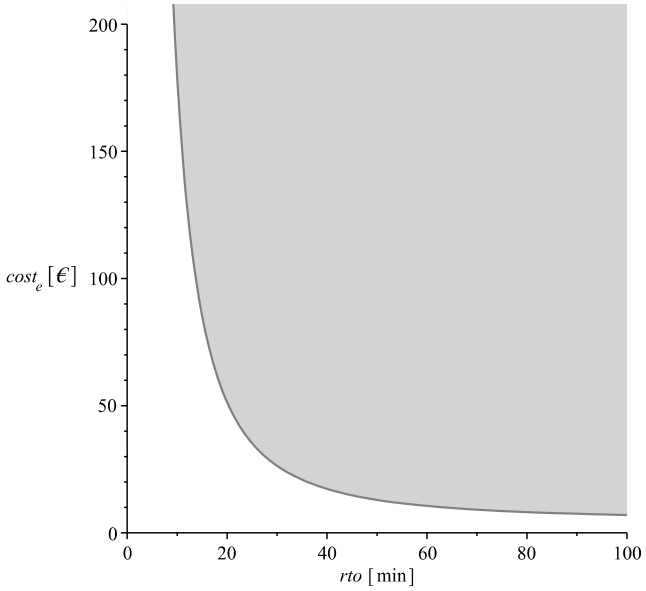

Abbildung 110: Vergleich von RTO und Ausfallkosten ($cost_e$) mit Kombinationen, in denen sich der Einsatz von Cloud Standby lohnt (grau)

Diese Ergebnisse zeigen, dass es mit Cloud Standby möglich wird, ein verteiltes System anbieterunabhängig in der Cloud abzusichern, und sich durch die Absicherung die Deploymentzeit verringern lässt. Die Kosten für die Absicherung können mit den in dieser Arbeit vorgestellten Methoden geschätzt werden. Außerdem kann mit der Methode zur Entscheidungsunterstützung zur Wahl des Update-Intervalls zudem angegeben werden, in welchen Fällen sich der Einsatz von Cloud Standby lohnt und in welchen nicht. Im Folgenden werden diese Aspekte nochmals kritisch betrachtet.

8.2 Kritische Betrachtung

Die Evaluierung und die daraus folgenden Ergebnisse haben gezeigt, dass Cloud Standby zu einer Erhöhung der Verfügbarkeit eines verteilten Systems führt und im Anwendungsfall langfristig günstiger ist als ein System ohne Absicherung. Trotz der beschriebenen Vorteile von Cloud Standby, ergeben sich jedoch auch Grenzen der Anwendbarkeit, die im Folgenden aufgezeigt und diskutiert werden sollen.

Notfallwiederherstellung – Die in dieser Arbeit vorgestellte Methode Cloud Standby dient zur Wiederherstellung eines verteilten Systems im Notfall. Aus diesem Grund wurde ein Standby-Ansatz gewählt, der ein Notfallsystem bei einem Cloud-Anbieter vorhält. Wie bereits in Kapitel 4.2.2 beschrieben, gibt es neben Notfällen auch eine Reihe anderer Fehler, die von Cloud Standby nicht betrachtet werden. Hierzu zählen alltägliche Fehler, wie z.B. der Ausfall einzelner Server oder globale Katastrophen wie z.B. ein Atomkrieg. Alltägliche Fehler werden nicht vom Notfallmanagement, sondern im Rahmen des Störungsmanagements gelöst und bei Fehlern, die zu einer Kategorie jenseits von Notfällen gehören, gibt es eigene Verfahren wie das Krisen- oder Katastrophenmanagement. Oftmals liegt jedoch der Weg aus einer Krise oder gar Katastrophe gar nicht beim KMU, sondern bei übergeordneten Institutionen wie einem Land oder der Staatengemeinschaft.

Warm Standby – Die Evaluierung hat gezeigt, dass bei der Absicherung eines Systems, bei dem die BIA geringe Ausfallkosten ergeben hat, der Einsatz von Cloud Standby nicht sinnvoll ist. Hier überwiegen die Kosten von Cloud Standby den Nutzen, der durch die Absicherung gezogen werden kann. In diesem Fall sollte auf eine Warm-Standby-Absicherung komplett verzichtet und stattdessen ein Cold-Standby gewählt werden. Gleiches gilt, wenn ein Ausfall sehr teuer ist. In diesem Fall ist die Nutzung eines Hot-Standby-Ansatzes statt eines Warm-Standby-Ansatzes wie Cloud Standby zu empfehlen.

Legacy Systeme – Auch gibt es verteilte Systeme, bei denen sich von ihrer Struktur her eine Absicherung mit Cloud Standby anbietet, und andere, für die es keinen Sinn macht. Existiert eine sehr komplexe Legacy-Anwendung bereits seit Jahren, so könnte es schwer sein, dieses verteilte System mit der in dieser Arbeit vorgestellten Beschreibungssprache zu modellieren. In diesem Fall würde es sich eventuell anbieten nur Teile der Anwendung mit Cloud Standby abzusichern (einzelne zentrale Verteilte Systeme) oder Cloud Standby erst im Zuge eine Neuinstallation einzuführen. Die Portierung einer bestehenden Anwendung in das Modell hängt dabei auch von den äußeren Rahmenbedingungen ab. Ist das Deployment beispielsweise über Skripte oder Konfigurationsmanager bereits automatisiert, so fällt es leicht, ein solches System auch mit der hier vorgestellten Beschreibungssprache zu modellieren. Liegen bisher alle nötigen Informationen nur als implizites Wissen oder in Form von textuellen Beschreibungen vor, sogestaltet sich dies schwierig. Der Grund dafür hängt jedoch nicht mit dem hier vorgestellten Ansatz zusammen, sondern umfasst das Thema der Deploymentautomatisierung und des IT-Management allgemein. Hierzu gibt es speziell über Standards wie ITIL bereits ein großes Feld an Lösungen, auf die zurückgegriffen werden kann. Dadurch wird auch die Hürde für die Einführung von Cloud Standby reduziert.

Hochverfügbare Systeme – Wenn man sich aus dem in dieser Arbeit fokussierten Feld der Kleinen und Mittelständischen Unternehmen wegbewegt, so gibt es verteilte Systeme, die bereits mit etablierten Fehlertoleranzsystemen ein verteiltes System über Anbietergrenzen betreiben. Diese Systeme sind von Grund auf so gebaut, dass sie mit dem Ausfall eines ganzen Anbieters umgehen können. In diesem Fall bietet Cloud Standby keinen Mehrwert. Cloud Standby konzentriert sich auf kleine bis mittelgroße verteilte Systeme, die bei einem einzelnen Anbieter laufen und deren Absicherung gegen Ausfälle.

Bestehende Datensicherungsmethoden – Ein weiteres Problem bei der Einführung von Cloud Standby könnte die bereits vorhandene Datensicherungsmethode bzw. deren technische Realisierung sein. Damit die in der Evaluierung dargestellten Deploymentzeiteinsparungen möglich sind, muss die Datensicherungsmethode dazu fähig sein, die Datenrücksicherungen inkrementell auszuliefern. Sollte sie dies nicht, könnten und werden teilweise alle Daten auf einmal zurückgeliefert und der Deploymentprozess würde somit unkontrolliert andauern, was zu einer Verletzung des RTO führt. Dies ist besonders dann der Fall, wenn sich viele Daten auf dem System befinden, wie z.B. bei Datenbanken. Inwieweit Datenbanken mit Cloud Standby abgesichert werden können, hängt also davon ab, mit welcher Methode die Datensicherung durchgeführt wird und die Daten bei einer Rücksicherung ausgeliefert werden. Sollte die Datensicherungsmethode außerdem zu viel Zeit für eine Datensicherung oder Datenrücksicherung benötigen, könnte dies mit dem gewählten Update-Intervall kollidieren. Hier sollten in den künftigen Arbeiten weitere Untersuchungen durchgeführt werden, um zu sehen, welche Datensicherungsmethoden sich mit welchen Sicherungsintervallen anbieten und welche nicht.

Kosten – Bei den in dieser Arbeit vorgestellten Methoden zur Berechnung von Kosten ergibt sich gleich eine ganze Reihe von Problemen. So ist die Bewertung der Kritikalität der Prozesse eine elementare Größe bei der Entscheidung, ob ein Standby-System eingesetzt werden soll oder nicht. Aus dieser Kritikalität der Prozesse ergeben sich zudem die Ausfallkosten für den Prozess (oder umgekehrt). Diese und weitere Fragestellungen entspringen dem Gebiet der Wirtschaftswissenschaften und es gibt in dieser Domäne eine Vielzahl von Verfahren für die Durchführung dieser Bewertungen, z.B. im Risiko-Management. Solche Untersuchungen sind die Grundlage für die vorliegende Arbeit, die keinen Anspruch erhebt, in diesem Feld einen Beitrag zu leisten. Alle

hier vorgestellten Berechnungsmethoden dienen dazu, dem Nutzer von Cloud Standby eine Unterstützung bei der Konfiguration zu geben oder die Nutzbarkeit von Cloud Standby generell zu belegen. Gerade bei der Langzeitbetrachtung könnten Effekte auftauchen, die heute nicht vorhersehbar sind. Langfristig kommt es zu eine Veränderung der Preise, Angebote und so weiter. Hier erhebt diese Arbeit nicht den Anspruch, die Zukunft vorherzusagen, sondern gibt einem technischen Nutzer ein Werkzeug an die Hand, das ihn bei Konfigurations-Entscheidungen unterstützt. Damit besonders Effekte wie die Erhöhung von Preisen nicht so sehr ins Gewicht fallen, wird in dieser Arbeit der Ansatz gewählt, ein verteiltes System mit Cloud Standby Absicherung dem ohne Absicherung gegenüberzustellen. Da bei beiden Szenarien Kostensteigerungen zu erwarten sind, fallen diese nicht stark ins Gewicht. In den zukünftigen Arbeiten könnte jedoch die hier vorgestellten Berechnungsmethoden so erweitert werden, dass Preise nicht durch Konstanten, sondern durch Funktionen ausgedrückt werden.

Vertrauen und Sicherheit – Soll wie im Rahmen dieser Arbeit vorgeschlagen Public Cloud Computing für die Absicherung genutzt werden, so stellen sich unweigerlich auch Fragen bezüglich der Datensicherheit als auch damit einhergehende Vertrauensfragen. Bei der Wahl des Notfall-Cloud-Anbieters muss darauf geachtet werden, dass dieser Anbieter vertrauenswürdig ist und für das zu sichernde System notwendige Sicherheitsgarantien bietet. Diese Fragen stellen sich nicht nur beim Cloud Computing, sondern sind klassische Fragen des Outsourcings. Ebenso wie bei der Wahl des Rechenzentrums für die Primär-Cloud muss sich das KMU bei der Notfall-Cloud Gedanken machen, welchem Anbieter es seine Daten anvertraut und welchem nicht.

Im Folgenden werden die oben diskutierten und darüber hinaus gehenden möglichen Erweiterungen von Cloud Standby vorgestellt.

8.3 Ausblick

Diese Arbeit beschreibt eine Methode, mit der ein verteiltes System in der Cloud abgesichert werden kann. Dabei ergeben sich an vielen Stellen noch weitere Fragen, die unter anderem aus den Annahmen und den bereits diskutierten möglichen Problemen resultieren:

Zusammenhang von RTO/RPO – Der bereits in Kapitel 4 und 8.2 diskutierte Zusammenhang zwischen dem Aktualisierungsintervall und dem daraus resultierenden RTO sowie dem in der Datensicherungsmethode hinterlegten Sicherungsintervall (RPO) und der Sicherungsmethodik sollte in Zukunft für einzelne Anwendungsfälle genauer untersucht werden. Hier ist speziell der Anwendungsfall von großen Datenbanken interessant, außerdem die Untersuchung, welche Arten von Datensicherungslösungen für diese Anwendungsfälle zur Verfügung stehen, und inwieweit diese mit Cloud Standby kompatibel sind.

Berechnung der Kosten – Aus der bereits in Kapitel 4.5 und 8.2 diskutierten Frage nach der Berechnung der Ausfallkosten und dem Umgang mit sich ändernden Preisen in der Zukunft ergibt sich eine Reihe von weiteren Fragen. Diese Fragen sind jedoch unabhängig von der vorliegenden Arbeit und stellen sich außerdem in vielen anderen Kontexten. Als künftige Arbeit könnten jedoch die Anforderungen und Annahmen dieser Arbeit als Grundlage dienen, neue Verfahren im Bereich der Wirtschaftswissenschaften zu entwickeln, um so beispielsweise eine genauere Berechnung der Ausfallkosten zu ermöglichen.

Anbieterwahl – Wie bereits in Kapitel 7.2.3 beschrieben, ist die Deploymentzeit bei verschiedenen Anbietern sehr unterschiedlich. Diese Beobachtung sollte in zukünftigen Arbeiten weiter untersucht werden. Mit dem Wissen hierrüber und mittels weiterer Informationen, wie der Reputation der Anbieter könnte dem Modellierer so eine Entscheidungsunterstützung zur Wahl der Notfall-Cloud gegeben werden.

Testen – Im Notfall ist das Unternehmen, das Cloud Standby implementiert hat, darauf angewiesen, dass das Notfallsystem den Betrieb korrekt übernehmen kann. Da sich bei der Modellierung des Systems Fehler einschleichen oder sich Umstände über die Zeit ändern können, so sollte das Notfallsystem regelmäßig getestet werden. Im Notfallmanagement sind solche Tests im Rahmen von Übungen vorgesehen. In künftigen Arbeiten sollte hierzu eine Methode entwickelt werden, die Tests automatisch und ohne Risiko für das Primärsystem durchführt.

Im Bereich der Modellierung ergeben sich diverse Themen, die in der Zukunft angegangen werden könnten.

Erweiterung der Deploymentsprache – Die Deploymentsprache umfasst bereits eine große Anzahl von Elementen. Je nach Anwendungsfall könnte jedoch eine Erweiterung der Sprache an der einen oder anderen Stelle sinnvoll sein. Dies ist vor allem dann zu empfehlen, wenn die Sprache für außergewöhnlichere Cloud-Anbieter oder Konfigurationsmanager genutzt werden soll. Auch wäre es möglich, die Sprache in anderen Kontexten als der Notfallwiederherstellung einzusetzen und disbezüglich zu optimieren. Aufgrund der modularen Struktur ist dies problemlos möglich.

Skalierung – Die Beschreibungssprache erlaubt es im laufenden Betrieb Instanzen hinzuzufügen oder abzuschalten. Als eine zukünftige Arbeit wäre das Hinzufügen einer Skalierungskomponente in die Implementierung der Sprache denkbar, die mit dem Wissen der Absicherung eigene Skalierungsentscheidungen treffen kann.

Hilfe bei der Modellierung – In der aktuellen Version der Deploymentsprache hat der Modellierer viele Freiheiten. Dies ist zunächst einmal gut, aber birgt auch die Gefahr, dass dieser dadurch überlastet und schlechte Entscheidungen bei der Modellierung trifft. Um dies zu verhindern, könnte die Beschreibungssprache noch um weitere Policys erweitert werden, die auch semantisch prüfen, ob eine Modellierungs-

entscheidung sinnvoll ist oder nicht. Zusätzlich könnten weitere Verfahren zur Entscheidungsunterstützung, wie das Service Feature-Modelling, genutzt werden, um Entscheidungen, wie z.B. die Wahl der passenden föderierten Elemente, zu unterstützen. Zu diesem Thema gibt es bereits von Wittern et al erste Ansätze [133], die weiter ausgebaut werden könnten.

Automatische Erstellung von Modellen – Ist ein verteiltes System gegeben, so können mit den passenden Verfahren automatisch die Komponenten auf einem Image oder dem laufenden Server erfasst werden. Hierzu gibt es bereits bestehende Ansätze, wie der von Menzel et al. [95] mit dem sich der gesamte Software-Teil eines Modells automatisch erstellen lassen könnte.

Grafische Repräsentation – Der aktuelle Fokus der Beschreibungssprache liegt darin, das Deployment zu automatisieren und gestaltet sich daher sehr technisch. Dadurch, dass jedoch diese technische Basis gelegt ist, besteht die Möglichkeit, das Modell auch zur Kommunikation von Experten zu nutzen. Hierfür müsste es, ähnlich wie das bei UML oder anderen Auszeichnugssprachen geschehen ist, den Metamodell-Elementen eine grafische Repräsentation zugewiesen werden. So könnte beispielsweise dem Element „Cloud“ eine Wolke zugewiesen werden und die Relation „hat virtuelle Maschine“ darüber repräsentiert werden, dass die virtuelle Maschine in die Wolke gezeichnet wird.

Erweiterung des Editors – Mit der EMF Beschreibungssprache ist bereits die Basis gelegt, eine große Anzahl von nützlichen Werkzeugen zu schaffen. So könnten die zuvor genannten Hilfen bei der Modellierung auch direkt in die Oberfläche des Editors integriert werden. Außerdem wäre es möglich, mit der großen Anzahl von Frameworks aus dem Eclipse-Umfeld den Editor so zu erweitern, dass aus dem Baum-Editor ein Grafischer Editor wird. Hierzu ist es notwendig, jedem der Elemente eine grafische Repräsentation zu geben.

Evolution der Modelle über die Zeit – Im Laufe der Zeit können sich die Strukturen der verteilten Systeme ändern. Zwar ist es problemlos möglich, ein Modell in mehreren Versionen zu erstellen und vorzuhalten, wobei es jedoch sinnvoll, wenn es hierzu die Unterstützung durch ein Vorgehensmodell und Werkzeug gebe. Da die Modelle auf XML basieren, sind die Änderungen des Systems leicht zu erkennen und statt des kompletten Systems deployt werden.

Modell-Repositories – Wird Cloud Standby von einer Vielzahl von Anwendern genutzt, so könnte es sinnvoll sein, Modellteile in Repositories zu sammeln. Auf diese Weise könnten beispielsweise auch unerfahrene Anwender von dem Expertenwissen anderer profitieren und z.B. nur noch die Struktur eines verteilten Systems modellieren, während ein anderer Nutzer die Infrastruktur und Föderation modelliert hat. Auch hierzu gibt es bereits Verfahren, wie das von Schuster [120], die das gemeinsame Erstellen von Dokumenten vereinfachen.

Weitere mögliche künftige Arbeiten könnten sich damit beschäftigen, Cloud Standby in bestehende Standardisierungsbemühungen zu integrieren.

Integration in TOSCA – Wie bereits in Kapitel 3.2.2 dargelegt ist eines der großen Probleme von TOSCA, dass es sehr allgemein gehalten ist. Nachdem bereits erste Aktivitäten in diese Richtung stattgefunden haben, wäre es sinnvoll, auch in Zukunft den Standard TOSCA zu erweitern und damit besser nutzbar zu machen. Sowohl die Prozesse als auch die Beschreibungssprache dieser Arbeit könnten in TOSCA umgesetzt werden und somit als Vorlage für die anbieterunabhängige Modellierung von verteilten Systemen dienen.

Integration in ITIL – Der in Kapitel 1 vorgestellte Notfallmanagementprozess wird in dieser Art auch bei ITIL genutzt. Auch ist es für das Aktualisieren von Cloud Standby Modellen sinnvoll, ein funktionierendes IT-Management zu haben. Wird in dem bestehenden IT-Management das Modell, wie auch in der Vergangenheit die Notfall-

pläne aktualisiert, kann sicher gegangen werden, dass keine Änderungen verloren gehen. Um diesen Prozess zu unterstützen, könnten spezielle Prozesse entwickelt werden, die die Eigenheiten von Cloud Standby fest in ITIL verankern und so eine bessere Integration und Automatisierung ermöglichen.

Anhang

Im Anhang dieser Arbeit werden die Umsetzung der Entscheidungsunterstützung aus Kapitel 4.5 und eine Referenz der Beschreibungssprache aus Kapitel 5.3 präsentiert

A.1 Umsetzung der Entscheidungsunterstützung in Maple

Wie in Kapitel 4 dargelegt wurde die Entscheidungsunterstützung aus Kapitel 4.5 prototypisch in dem Computer Algebra System Maple umgesetzt. Im Folgenden werden die implementierten Algorithmen erläutert.

Zunächst muss bei der Entscheidungsunterstützung die stationäre Verteilung berechnet werden. Diese Berechnung ist in Abbildung 111 dargestellt.

```
steadyStateVector := proc( P::Matrix )
  # Lokale Variablen definieren
  local n, Q, e, QT, b;

  # Dimension der Matrix P bestimmen
  n := Dimension( P )[1];

  # Q = P - I
  Q := P - IdentityMatrix( n );

  # e ist ein Vektor, der nur aus Einsen besteht
  e := <seq(1, i=1..n)>;

  # Vektor e an Q hängen und transponieren
  QT := Transpose( <Q | e> );

  # b ist der Einheitsvektor mit einer Eins an der Stelle n+1
  b := UnitVector( n+1, n+1 );

  # Lineares Gleichungssystem QT*pi = b lösen
  return LinearSolve(QT,b):
end proc:
```

Abbildung 111: Algorithmus zur Berechnung der Stationären Verteilung

Im Anschluss müssen aus den Aufenthaltsdauern die Rückkehrwahrscheinlichkeiten berechnet werden. Dies ist in Abbildung 112 dargestellt.

```
calcLambda := proc( times)
  # Lokale Variablen definieren
  local duration, lam;

  # duration[n] ist die Aufenthaltsdauer in S_n
  duration[1] := times[1];
  duration[2] := times[2];
  duration[3] := times[3];
  duration[4] := min(duration[1], rto(times[2]));
  duration[6] := duration[1];
  duration[7] := times[4];
  duration[5] := duration[7] - duration[6] - duration[4];

  # Rückkehrwahrscheinlichkeiten berechnen
  lam := [seq(duration[i] / (duration[i] + 1), i = 1..7)];
return lam;
end proc:
```

Abbildung 112: Algorithmus zur Berechnung der Rückkehrwahrscheinlichkeiten

Aufbauend auf diesen Algorithmen können dann die Gesamtkosten (Abbildung 113) und die Gesamtverfügbarkeit (Abbildung 114) berechnet werden.

```
evaluateCost := proc( Ptemp, times, cost, epsilon, server)
  # Lokale Variablen definieren
  local lambda, c, s, totalcost, duration;
  c_P1 := cost[1]; c_P2 := cost[2]; c_e := cost[3];

  # Rückkehrwahrscheinlichkeiten bestimmen
  lambda := calcLambda(times);

  # Stationäre Verteilung bestimmen
  s := steadyStateVector(Ptemp(lambda, epsilon)) :

  # Gesamtkosten aus der stationären Verteilung berechnen
  totalcost := s[1]·(c_P1·server + c_e) + s[2]·(c_P1·server) + s[3]·(c_P1·server + c_P1·server)
     + s[4]·(c_P2·server + c_e) + s[5]·(server·c_P2) + s[6]·(c_P2·server + c_P1·server) + s[7]
     ·c_e :
return totalcost·24·tage_jahr :
end proc:
```

Abbildung 113 Algorithmus zur Berechnung der Gesamtkosten

```
evaluateAvail := proc( Ptemp, times, epsilon)
  # Lokale Variablen definieren
  local s, lambda, availxx;

  # Rückkehrwahrscheinlichkeiten bestimmen
  lambda := calcLambda(times);

  # Stationäre Verteilung bestimmen
  s := steadyStateVector(Ptemp(lambda, epsilon)) :

  # Gesamtverfügbarkeit aus der stationären Verteilung berechnen
  availxx := 1 − s[7]−s[4]−s[1];
return availxx :
end proc:
```

Abbildung 114: Algorithmus zur Berechnung der Gesamtverfügbarkeit

Diese Algorithmen bilden die Grundlage aller in dieser Arbeit durchgeführten Untersuchungen.

A.2 Referenz der Beschreibungssprache

Die in Kapitel 5.3 entwickelte und abstrakt beschriebene Beschreibungssprache wird im Folgenden detailliert vorgestellt. Hierzu werden für jedes

Element die Generalisierungen, Attribute und Assoziationen beschrieben. Außerdem werden zusätzlich zu den in Kapitel 5.3 beschriebenen Paketen das Paket „Grundlegende Elemente“ eingeführt, das die Aufgabe hat, globale Attribute wie z.B. den Namen eines Elements zu definieren. Die anderen Pakete sind analog zu der Beschreibung in Kapitel 5.3 aufgebaut.

A.2.1 Grundlegende Elemente

Die grundlegenden Elemente sind Basistypen, die bei allen anderen Elementen des Metamodells gleich sind. So lassen sich später die einzelnen Elemente leichter nutzen.

Basis Element (BasicElement)

Das abstrakte Element „Basis Element“ dient ähnlich wie das „UML::Element“ dazu, dass jedes der am Metamodell beteiligten Elemente direkt oder indirekt von ihm abgeleitet wird. Hierdurch können später alle Elemente des Modells über den Typ „Basis Element“ nutzbar gemacht werden.

Generalisierungen

keine

Attribute

keine

Assoziationen

keine

Benanntes Element (NamedElement)

Das abstrakte Element „Benanntes Element“ dient ähnlich wie das „UML::NamedElement“ dazu, dass alle Elemente, die von ihm abgeleitet werden einen Namen besitzen.

Generalisierungen

„Basis Element“ (aus Paket „Grundlegende Elemente“)

Attribute

- name: String

  Der Name eines Elements dient zur Beschreibung, nicht zur eindeutigen Identifikation. Bei der Modellierung sollte darauf geachtet werden, dass jedem Element ein eindeutiger Name zugewiesen wird. Dies ist nicht zwingend notwendig, erleichtert jedoch die Lesbarkeit des Modells.

Assoziationen

keine

A.2.2 Software

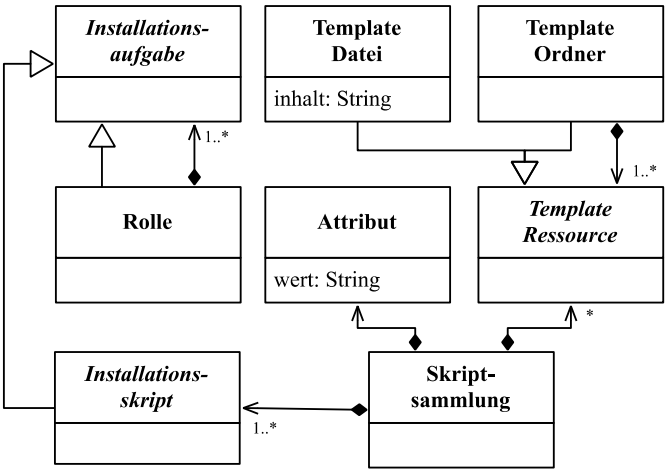

Abbildung 115: MOF Klassendiagramm des Software-Pakets

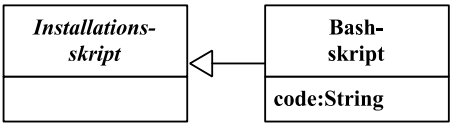

Abbildung 116: Konkretisierung des abstrakten Installationsskripts in der Implementierung

Installationsaufgabe (InstallationTask)

Das Element „Installationsaufgabe“ beschreibt eine Tätigkeit, die auf einer virtuellen Maschine ausgeführt wird, um eine Konfiguration durchzuführen oder neue Software zu installieren. Es ist Teil des Kompositum-Entwurfsmusters [43] und dient dazu, Baumstrukturen mit Skripten als

Blätter aufzubauen. Die Installationsaufgabe ist eine abstrakte Metaklasse und muss zur Modellierungszeit durch ein Skript oder eine Rolle konkretisiert werden.

Generalisierungen

- „Benanntes Element“ (aus dem Paket „Grundlegende Elemente“)

Attribute

keine

Assoziationen

keine

Installationsskript (InstallationScript)

Mit dem abstrakten Element „Installationsskript“ wird die Sammlung der auf dem virtuellen Server ausführbaren Installationsbefehle bezeichnet. Konkrete Ausprägungen werden in der Implementierung festgelegt und können im einfachsten Fall Bash Skripte sein. Es ist jedoch auch möglich, bestehende Skripte von Konfigurationsmanagern zu integrieren. Dies erfolgt in der konkreten Implementierung und wird an dieser Stelle nicht weiter behandelt.

Generalisierungen

- Aufgabe (aus dem Paket Installation)

Attribute

keine

Assoziationen

Keine

Bash Skript (BashScript)

Das „Bash Skript (BashScript)“ ist eine Konkretisierung des Bash Scripts, das direkt per SSH auf den Instanzen ausgeführt werden kann. Siehe dazu Abbildung 116.

Generalisierungen

- „Installationsskript“ (aus dem Paket „Software“)

Attribute

- code: String

  In dem Attribut ist der Quellcode des Bash Skripts hinterlegt. Er ist direkt per SSH ausführbar.

Assoziationen

keine

Skriptsammlung (ScriptCollection)

Das abstrakte Element „Skriptsammlung“ dient zum Zusammenfassen von Skripten zu logischen Gruppen. Allen Skripten einer Skriptsammlung stehen zur Laufzeit außerdem die Daten aus Template- oder Attribut-Objekten (s.u.) zur Verfügung.

Generalisierungen

- „Benanntes Element“ (aus dem Paket „Grundlegende Elemente“)

Attribute

keine

Assoziationen

- skripte: Skriptinstallation [1..*]

  Mit der Skripte-Kompositions-Beziehung werden die Skripte definiert, die Teil der Skriptsammlung sind. Um eine Typsi-

cherheit zu erreichen, kann diese Beziehung in der konkreten Implementierung auch in den nicht-abstrakten Elementen zwischen z.B. SkriptSammlungBash und SkriptBash erfolgen.

- attribute: Attribut [*]

  Mit dieser Kompositionsbeziehung können einer Skriptsammlung Attribute zugewiesen werden, die dann zur Laufzeit allen Skripten zur Verfügung stehen.

- templateRessourcen [*]

  Über diese Beziehung können Template-Ressourcen und damit ganze Template-Baumstrukturen mit Ordnern und Dateien abgelegt werden.

Rolle (Role)

Mit dem Element Rolle können mehrere Skripte und Rollen zusammengefasst und in eine Ausführungsreihenfolge gebracht werden.

Generalisierungen

- aufgabe (aus dem Paket Installation)

Attribute

keine

Assoziationen

- installationsAufgaben: Aufgabe [*]

  Die Installationsaufgaben sind eine geordnete Liste von Skripten und Rollen, die so nacheinander zur Ausführung kommen. Mittels einer Tiefensuche kann der Installationsbaum zur Laufzeit aufgelöst und die Skripte, die die Blätter des Baums darstellen, ausgeführt werden.

Attribut (Attribute)

Über das Attribut können Variablen der Skriptsammlung zugewiesen werden. Jedem Skript aus der Sammlung stehen diese Variablen zur Laufzeit zur Verfügung.

Generalisierungen

- „Benanntes Element“ (aus dem Paket „Grundlegende Elemente“)

Attribute

- wert: String

  Das Attribut „Wert“ beinhaltet eine beliebige Zeichenkette, die zur Laufzeit in dem Installationsskript als Variable zur Verfügung steht.

Assoziationen

keine

Template Datei (TemplateFile)

Unter einem Template wird eine Datei verstanden, die beim Deployment zur Verfügung steht. Es kann sich hierbei um eine Konfigurationsdatei, eine Webserver-Startseite oder ähnliches handeln. Beim Deployment werden Variablen (Attribute) innerhalb der Datei mit konkreten Werten ersetzt. So ist es z.B. möglich, eine neue Instanz beim Deployment einem bestehenden Loadbalancer zuzuordnen oder eine instanz-spezifische Webserver-Startseite zu erstellen. Zum Beschreiben des Inhalts der beim Deployment zur Verfügung stehenden Datei wird das Attribut „inhalt“ verwendet. In der Baumstruktur, die durch die Template Ressource-Objekte (s.u.) gebildet wird, sind die Template Dateien die Blätter.

Generalisierungen

- TemplateRessource (aus dem Paket Installation)

Attribute

- inhalt: String

  Gibt den Inhalt der Template Datei wieder.

Assoziationen

keine

Template Ordner (TemplateFolder)

Der „TemplateOrdner“ hat innerhalb des Kompositum-Entwurfsmusters [43] (zur Erstellung einer Baumstruktur) die Rolle des Kompositums. Er dient dazu, ähnlich wie in einem Dateisystem, die Ordnerpfade für die Strukturierung von Template Dateien zu ermöglichen.

Generalisierungen

- TemplateRessource (aus dem Paket Installation)

Attribute

keine

Assoziationen

- templateRessource: TemplateRessource [1..*]

  Die Kompositionsbeziehung erlaubt es dem TemplateOrdner, weitere TemplateOrdner oder TemplateDateien zuzuweisen und so die Baumstruktur aufzubauen.

Template Ressource (TemplateResource)

Das abstrakte Element „Template Ressource“ stellt das gemeinsame Interface für die „Template Datei“ und dem „Template Ordner“ dar. Es ist Teil

des Kompositum-Entwurfsmusters [43], hat die Rolle der Komponente und dient dazu, Baumstrukturen mit Template Dateien als Blätter aufzubauen.

Generalisierungen

keine

Attribute

keine

Assoziationen

keine

Template Datei (TemplateFile)

Unter einem Template wird eine Datei verstanden, die beim Deployment zur Verfügung steht. Es kann sich hierbei um eine Konfigurationsdatei, eine Webserver-Startseite oder ähnliches handeln. Beim Deployment werden Variablen (Attribute) innerhalb der Datei mit konkreten Werten ersetzt. So ist es z.B. möglich, eine neue Instanz beim Deployment einem bestehenden Loadbalancer zuzuordnen oder eine instanz-spezifische Webserver-Startseite zu erstellen. Zum Beschreiben des Inhalts der beim Deployment zur Verfügung stehenden Datei wird das Attribut „inhalt“ verwendet. In der Baumstruktur, die durch die Template Ressource-Objekte (s.u.) gebildet wird, sind die Template Dateien die Blätter.

Generalisierungen

- TemplateRessource (aus dem Paket Installation)

Attribute

- inhalt: String

  Gibt den Inhalt der Template Datei wieder.

Assoziationen

keine

Template Ordner (TemplateFolder)

Der „TemplateOrdner“ hat innerhalb des Kompositum-Entwurfsmusters [43] (zur Erstellung einer Baumstruktur) die Rolle des Kompositums. Er dient dazu, ähnlich wie in einem Dateisystem, die Ordnerpfade für die Strukturierung von Template Dateien zu ermöglichen.

Generalisierungen

- TemplateRessource (aus dem Paket Installation)

Attribute

keine

Assoziationen

- templateRessource: TemplateRessource [1..*]

  Die Kompositionsbeziehung erlaubt es dem TemplateOrdner, weitere TemplateOrdner oder TemplateDateien zuzuweisen und so die Baumstruktur aufzubauen.

Template Ressource (TemplateResource)

Das abstrakte Element „Template Ressource“ stellt das gemeinsame Interface für die „Template Datei“ und dem „Template Ordner“ dar. Es ist Teil des Kompositum-Entwurfsmusters [43], hat die Rolle der Komponente und dient dazu, Baumstrukturen mit Template Dateien als Blätter aufzubauen.

Generalisierungen

keine

Attribute

keine

Assoziationen

keine

A.2.3 Infrastruktur

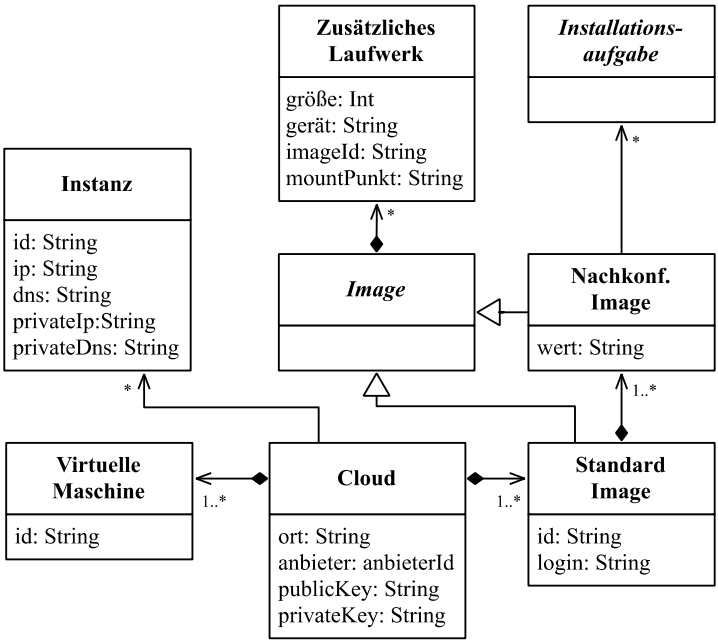

Abbildung 117: MOF Klassendiagramm des Infrastruktur-Pakets (vgl. [84])

Cloud (Cloud)

Das Element „Cloud“ steht für das Rechenzentrum, in dem die IaaS-Ressourcen zur Verfügung stehen. Die Cloud ist über eine API zugreifbar und verwaltbar.

Generalisierungen

- „Benanntes Element“ (aus dem Paket „Grundlegende Elemente“)

Attribute

- ort: String

  Der Ort beschreibt den Standort des Rechenzentrums. Der Standort kann für die Auswahl des Rechenzentrums zur rechtssicheren Speicherung von Daten oder für die Einhaltung eines Mindestabstands zum primären Rechenzentrum bei der Absicherung von Bedeutung sein.

- anbieter: anbieterId

  Die eindeutige Anbieterkennung wird benutzt, um die Cloud des Anbieters eindeutig zu identifizieren und zur Laufzeit mittels externer Bibliotheken die passenden Schnittstellen zum Zugriff zu nutzen. In der konkreten Implementierung muss der Datentyp anbieterId mit Hilfe des «enumeration» Datentyps umgesetzt werden.

- publicKey: String

  Der öffentliche Schlüssel ist Teil des Identifikationsverfahrens, mit dessen Hilfe der Nutzer die Dienste des Cloud-Anbieters in Anspruch nehmen kann. Der öffentliche Schlüssel könnte dabei ein Benutzername oder Teil einer Public-Key-Authentifizierung sein.

- privateKey: String

  Ähnlich wie der öffentliche Schlüssel ist der private Teil der Authentifizierung des Benutzers gegenüber dem Rechenzentrumsanbieter. Er kann ebenfalls Teil einer Public-Key-Authentifikation oder ein simples Passwort sein.

Assoziationen

- virtuelleMaschinen: virtuelleMaschine [1..*]

  Die Komposition beschreibt die virtuellen Maschinen, die in dem Rechenzentrum zur Verfügung stehen. Beispiele hierfür sind „kleine Server mit 1 Kern“ oder „große Server mit 4 Kernen“.

- standardImages: StandardImage [1..*]

  Einfache Images werden von den Rechenzentrumsanbietern zur Verfügung gestellt oder wurden bereits vom Benutzer hochgeladen. Sie stehen beim Anstarten des virtuellen Servers als Auswahl bei diesem Rechenzentrum zur Verfügung.

- instanzen: Instanz [0..*]

  Alle am verteilten System beteiligten Instanzen des Cloud-Anbieters sind in dieser Assoziation zusammengefasst.

Virtuelle Maschine (VirtualMachine)

Um eine Software ausführen zu können, muss im IaaS Umfeld virtuelle Hardware zur Verfügung stehen. Diese Virtuelle Hardware wird im Rahmen dieser Arbeit als „Virtuelle Maschine“ bezeichnet.

Generalisierungen

- „Benanntes Element“ (aus dem Paket „Grundlegende Elemente“)

Attribute

- id: String

  Die ID wird genutzt, um die virtuellen Maschinen-Typen des Anbieters eindeutig zu identifizieren und bei der Ausführung auf genau diesen Typ zu verweisen.

Assoziationen

keine

Image (Image)

Ein „Image“ ist eine Sammlung von Software-Artefakten, die auf einer virtuellen Maschine ausgeführt werden können. Dabei besteht das Image immer aus einem Betriebssystem inklusive Software- und Konfigurationseinstellungen. Das abstrakte Image-Element dient als gemeinsame Schnittstelle für „Standardisierte Images“ und „Nachkonfigurierte Images“.

Generalisierungen

- „Benanntes Element“ (aus dem Paket „Grundlegende Elemente“)

Attribute

keine

Assoziationen

- zusätzlicheLaufwerke: zusätzlichesLaufwerk [0..*]

  Standardmäßig ist einer virtuellen Maschine nur ein Laufwerk über das Image zugewiesen. Es ist im Betriebssystem als „Gerät“ (z.B. /dev/hda) verfügbar. Mit der Assoziation zusätzlicher Laufwerke können weitere Festplatten hinzugefügt werden, die andere Geräte-Bezeichnungen tragen (z.B. /dev/hdg).

Standard Image (StandardImage)

Das „Standard Image“ wird von einem Cloud-Anbieter zur Verfügung gestellt und kann so direkt als Instanz gestartet werden. Es sind keine Nachkonfigurationen erforderlich, um zum gewünschten Endzustand zu kommen. Standard Images werden entweder vom Cloud-Anbieter zur Verfügung gestellt und können nicht modifiziert werden oder es sind benutzer-

definierte Images, die entweder aus bestehenden Images modifiziert oder als Ganzes hochgeladen wurden.

Generalisierungen

- „Image“ (aus dem Paket „Infrastruktur“)

Attribute

- id: String

  Die ID wird genutzt, um das Image, das vom Anbieter bereitgestellt wird, eindeutig zu identifizieren und bei der Ausführung dieses zu instanziieren.

Assoziationen

- nachkonfigurierteImages: nachkonfiguriertesImage [0..*]

  Jedem Standard Image können noch weitere Installationsaufgaben zugewiesen werden, die dann beim Deployment ausgeführt werden. Das Bündel aus Standard Image und Installationsaufgaben wird als Nachkonfiguriertes Image bezeichnet.

Nachkonfiguriertes Image (ConfiguredImage)

Genügen die Standard Images nicht den Anforderungen beim Deployment oder die Software auf einem Image ändert sich immer wieder, sodass es unpraktikabel wäre, immer wieder ein neues Standard Image zu erstellen und abzuspeichern, kann Software über das Nachkonfigurierte Image direkt beim Deployment nachinstalliert werden.

Generalisierungen

- „Image“ (aus dem Paket „Infrastruktur“)

Attribute

keine

Assoziationen

- installationsAufgaben: installationsAufgabe [1..*]

  Über die Assoziation installationsAufgaben wird eine geordnete Liste von Installationsaufgaben definiert, die nach dem Starten des Betriebssystems ausgeführt werden müssen, um den gewünschten Endzustand des Images zu erreichen.

Zusätzliches Laufwerk (AdditionalDrive)

Soll einer virtuellen Maschine zur Laufzeit mehr als eine Festplatte zur Verfügung stehen, so können zusätzliche Laufwerke hinzugefügt werden. Je nach Anbieter können diese auf Basis von bestehenden Images oder als leere Festplatte erstellt werden.

Generalisierungen

- „Benanntes Element“ (aus dem Paket „Grundlegende Elemente“)

Attribute

- größe: Integer

  Die Größe repräsentiert die Speicherkapazität der Festplatte, die durch das Element „Zusätzliches Laufwerk“ repräsentiert wird.

- gerät: String

  Die Bezeichnung des Gerätenamens, unter dem das Laufwerk dem Betriebssystem zur Verfügung stehen soll (z.B. /dev/hdg).

- imageId: String

  Die optionale ID eines Images, die den Inhalt des Laufwerks widerspiegelt.

- mountPunkt: String

  Der optionale Laufwerksbuchstabe oder Pfad, in dem das Laufwerk dem Betriebssystem zugreifbar gemacht werden

kann. Die Aktion des Mountens muss je nach Cloud-Anbieter als Nachkonfiguration durchgeführt werden, sobald das Betriebssystem läuft.

Assoziationen

keine

Instanz (Instance)

Die „Instanz“ ist die Laufzeitrepräsentation der virtuellen Maschine und des standardisierten Images. Auf einer Instanz läuft auf unterster Ebene immer ein Betriebssystem und darauf aufbauend verschiedene Softwarepakete, mit denen die Instanz mit der Außenwelt kommunizieren kann (z.B. SSH). Das Instanz-Element wird nicht modelliert, sondern zur Laufzeit automatisch erstellt und aktualisiert.

Generalisierungen

- „Benanntes Element“ (aus dem Paket „Grundlegende Elemente“)

Attribute

- id: String

  Die Instanz hat bei dem zugehörigen Anbieter eine Kennung, mit der sie sich eindeutig identifizieren lässt. Diese Kennung wird in der ID gespeichert.

- ip: String

  Die IP-Adresse der Instanz, mit der sie über das Internet zugänglich ist. Soll später z.B. per SSH weitere Software installiert werden, so wird diese IP genutzt, um die Verbindung herzustellen.

- dns: String

  Alternativ zur IP kann auch der Domänenname genutzt werden, der sich zu der IP auflösen lässt. Auch dieser ist öffentlich und dient als Zugangspunkt aus dem Internet.

- privateIp: String

  Optional kann einer Instanz auch noch eine private IP-Adresse zugewiesen werden. Diese ist dann nicht aus dem Internet, sondern nur innerhalb des Cloud-Netzwerks verfügbar. Sie kann aus Sicherheitsgründen genutzt werden, um zu gewährleisten, dass keine Daten außerhalb des Rechenzentrums transportiert werden.

- privateDns: String

  Der optionale nicht-öffentliche Domänenname, der alternativ zur privaten IP genutzt werden kann.

Assoziationen

- standardImage: StandardImage

  Das für die laufende Instanz gewählte Image des Cloud-Anbieters.

- virtuelleMaschine: VirtuelleMaschine

  Die für die laufende Instanz gewählte virtuelle Maschine des Cloud-Anbieters.

A.2.4 Föderation

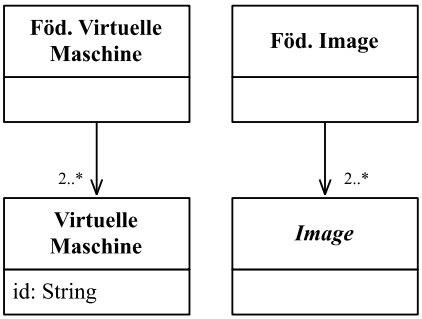

Abbildung 118: MOF Klassendiagramm des Föderation-Pakets (vgl. [84])

Föderierte Virtuelle Maschine (FederatedVirtualMachine)

Die „Föderierte Virtuelle Maschine“ ist die gemeinsame Schnittstelle für die virtuellen Maschinen aller an der föderierten Cloud beteiligten Anbieter. Die virtuellen Maschinen repräsentieren hierbei die nicht-funktionalen Eigenschaften der laufenden Instanz. Diese Eigenschaften werden nicht explizit modelliert, sondern sind dem Modellierer bekannt und er nimmt die Zuweisung zu der föderierten Virtuellen Maschine auf deren Basis vor. Eine föderierte Virtuelle Maschine repräsentiert also immer ein gewisses Spektrum an nicht-funktionalen Eigenschaften, die von ihr garantiert werden.

Generalisierungen

- „Benanntes Element“ (aus dem Paket „Grundlegende Elemente“)

Attribute

keine

Assoziationen

- virtuelleMaschinen: virtuelleMaschine [2..*]

  Über die Assoziation virtuelleMaschinen ist der Föderierten Virtuellen Maschine von jedem Cloud-Anbieter mindestens eine virtuelle Maschine zugewiesen. Die Einhaltung dieser Restriktion obliegt dem Modellierer, andernfalls kann am Ende das System nicht auf allen Clouds deployt werden. Für die Absicherung auf verschiedenen Anbietern ist außerdem zu beachten, dass mindestens zwei verschiedene Anbieter an der föderierten Cloud Infrastruktur beteiligt sind.

Föderiertes Image (FederatedImage)

Ähnlich wie das Image steht das „Föderierte Image“ für die funktionalen Eigenschaften der Instanz. Der Modellierer stellt sicher, dass in dem Element von allen Anbietern ein Image verfügbar ist und dass die funktionalen Eigenschaften aller Images identisch sind, d.h. das gleiche Betriebssystem und die gleiche Software installiert und konfiguriert sind.

Generalisierungen

- „Benanntes Element“ (aus dem Paket „Grundlegende Elemente“)

Attribute

keine

Assoziationen

- virtuelleImages: virtuellesImage [2..*]

  Ähnlich wie bei der Föderierten Virtuellen Maschine stellt die Assoziation virtuelleImages sicher, dass mindestens zwei anbieterspezifische, funktional gleiche Images in der föderierten Cloud zur Verfügung stehen.

A.2.5 Verteiltes System

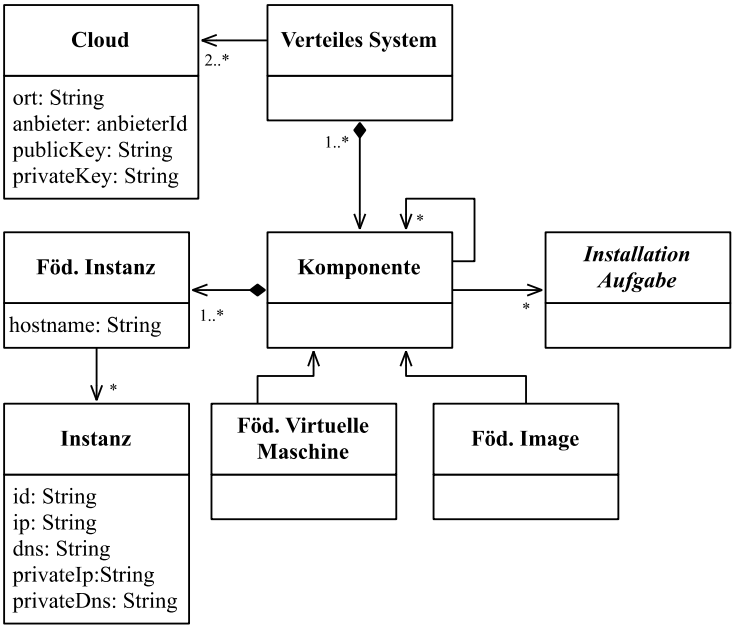

Abbildung 119: MOF Klassendiagramm des Verteiltes-System-Pakets (vgl. [84])

Verteiltes System (DistributedSystem)

Das Verteilte System repräsentiert das reale Verteilte System, das zu deployen ist. Soll eine elastische Anwendung im Cloud Computing deployt werden, so setzt sie häufig auf traditionelle verteilte Systeme oder Cluster-Anwendungen auf [83]. Durch geschicktes Zusammenschalten solcher Cluster-Anwendungen als Komponenten ergibt sich ein verteiltes System, das elastisch ist und die Vorteile von IaaS Cloud Computing optimal ausnutzen kann. Soll das verteilte System wie im Rahmen dieser Arbeit auf einen anderen IaaS Anbieter abgesichert werden, so muss vor dem

Deployment festgelegt werden, auf welcher der möglichen Clouds das verteilte System gestartet wird.

Generalisierungen

- „Benanntes Element“ (aus dem Paket „Grundlegende Elemente“)

Attribute

keine

Assoziationen

- clouds: Cloud [2..*]

  Über die Clouds-Assoziation wird eine geordnete Liste von Cloud-Anbietern hinterlegt. Der erste Anbieter der Liste ist die Primärcloud, und das System wird hier standardmäßig gestartet. Alle weiteren stellen die Notfallsysteme dar. Im Rahmen dieser Arbeit wird speziell für die Evaluation davon ausgegangen, dass nur ein einzelnes Notfallsystem existiert, theoretisch können jedoch mit allen vorgestellten Prozessen und Protokollen noch weitere Notfallsysteme vorgehalten werden.

Komponente (Component)

Die Komponente ist nach Tanenbaum [123] ein funktional abgeschlossener Teil des verteilten Systems. In der Praxis sind Komponenten häufig wieder verteilte Anwendungen oder Cluster-Anwendungen, die z.B. als Cloud-Anwendung zur Verfügung stehen [83]. Das Komponenten-Element stellt dabei eine logische Gruppierung von Föderierten Instanzen dar, die je nach Status des Gesamtsystems auf der Primär- oder Notfallcloud betrieben werden.

Generalisierungen

- „Benanntes Element“ (aus dem Paket „Grundlegende Elemente“)

Attribute

keine

Assoziationen

- installationAufgaben: Installationsaufgabe [0..*]

  Einer Komponente können optional Installationsaufgaben zugewiesen werden. Soll eine Software für eine Komponente auf allen Anbietern installiert werden, so macht es Sinn, diese Zuweisung auf Komponenten-Ebene vorzunehmen. So lässt sich die Komplexität des Modells reduzieren. Es wäre jedoch auch möglich, die gleiche Konfiguration des Verteilten Systems zu erreichen, indem man die Zuweisungen einzig über die Nachkonfigurierten Images vornimmt.

- föderierteVirtuelleMaschine: FöderierteVirtuelleMaschine

  Der Komponente ist genau eine Föderierte Virtuelle Maschine zugewiesen. Diese beschreibt die nicht-funktionalen Eigenschaften der Instanzen der Komponente.

- föderiertesImage: FöderiertesImage

  Außerdem ist der Komponente exakt ein Föderiertes Images zugeordnet. Es repräsentiert die funktionalen Eigenschaften unabhängig von den einzelnen Anbietern.

- benötigt: Komponente

  Komponenten können einander bedingen. Das heißt beim Deployment könnte z.B. der Applikationsserver eine Laufende Datenbank voraussetzen. Über die „benötigt-Beziehung“ kann ein Abhängigkeitsgraph modelliert werden, der beim Deployment beachtet wird. Es ist vom Modellierer darauf zu achten,

dass der Graph zyklenfrei ist. Beim Abhängigkeitsgraphen muss es sich also um einen azyklisch gerichteten Graph handeln [124].

- föderierteInstanzen: FöderierteInstanz [1..*]

  Eine Komponente besteht aus verschiedenen Instanzen, die anbieterunabhängig betrieben werden können. Diese Föderierten Instanzen sind analog zu den anderen Föderierten Elementen die logischen Gruppierungen der realen Instanz-Elemente, wie sie von den verschiedenen Anbietern zur Verfügung gestellt werden. Die Anzahl der Föderierten Instanzen bestimmt maßgeblich die Performanz der Komponente, und die Skalierung erfolgt auf Basis des Hinzuschaltens oder Entfernens Föderierter Instanzen. Diese Skalierung ist jedoch nicht Teil dieser Arbeit. Über das Metamodell wird lediglich die Möglichkeit geschaffen, eine Skalierung durchzuführen, um so auch elastische verteilte Systeme zu ermöglichen.

Föderierte Instanz (FederatedInstance)

Die Föderierte Instanz ist die Abstraktion der Instanz in der föderierten Cloud. Zur Laufzeit aggregiert die föderierte Instanz die Instanzen, die parallel laufen und die gleiche Aufgabe bei verschiedenen Anbietern haben.

Generalisierungen

- „Benanntes Element“ (aus dem Paket „Grundlegende Elemente“)

Attribute

- hostname: String

  Der Hostname beschreibt die URI, unter der die Instanz aus dem Internet zugänglich sein soll. Bei einem Notfall kann so

z.B. über den DNS-Server eine andere Instanz diesem Hostname zugewiesen werden und so vom Primärsystem auf das Notfallsystem umgeschaltet werden.

Assoziationen

- instanzen: Instanz [0..*]

  Zur Laufzeit ist eine Föderierte Instanz immer auch einer Instanz der Anbieter zugewiesen, bei der das verteilte System gerade läuft.

- datenrücksicherungsaufgaben: Datenrücksicherungsaufgabe [0..*]

  Jede Instanz kann eine Datenrücksicherungsaufgabe haben, die am Ende des Deploymentprozesses den letzten Stand der Geschäftsdaten auf der Instanz einspielt.

A.2.6 Datenrücksicherung

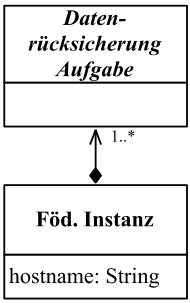

Abbildung 120: Metamodell des Datenrücksicherung-Pakets

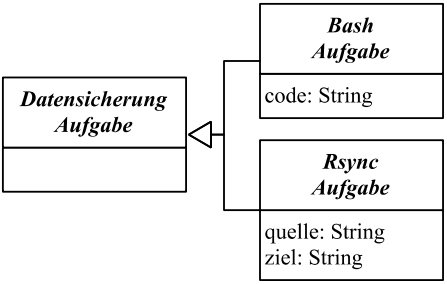

Abbildung 121: Konkretisierung der abstrakten Datensicherung Aufgabe in der Implementierung

Datenrücksicherungsaufgabe (RecoverTask)

Der Datenrücksicherungsaufgabe kommt eine zentrale Rolle im Cloud Standby Ansatz zu. Hier wird definiert, wie genau der letzte Stand der Datenrücksicherung wieder eingespielt werden soll. Die Ausführung der Datenrücksicherungsaufgaben ist der letzte Schritt im Deployment-Prozess.

Generalisierungen

- „Benanntes Element“ (aus dem Paket „Grundlegende Elemente“)

Attribute

keine

Assoziationen

Keine

Rsync Aufgabe (RsyncTask)

Neben dem Installationsskript wurde auch das abstrakte Element „Datensicherung Aufgabe“ konkretisiert. In der Implementierung wird Rsync[47] zum Rückspielen der Datensicherungen eingesetzt und Bash Skripte zur Erledigung zusätzlicher Aufgaben, wie dem Starten eines Datenbank-Import. Es wurde daher das neue Element „Rsync Aufgabe“ und „Bash Aufgabe“ als Konkretisierung der „Datensicherung Aufgabe“ erstellt (siehe Abbildung 121).
Als Attribute hat die „Rsync Aufgabe (RsyncTask)“ die Quelle und das Ziel der Datensicherung, wobei die Quelle in der Regel einen über das Netzwerk angebundenen Server darstellt, auf dem die Datensicherungen zur Verfügung stehen.

Generalisierungen

- „Datensicherung Aufgabe“ (aus dem Paket „Datensicherung“)

Attribute

- quelle: String

  Die Quelle wird dem Rsync-Befehl übergeben und beschreibt, wo die Daten für den Abgleich mit dem lokalen Dateisystem zu finden sind.

- ziel: String

  Der Zielordner auf der Instanz, in dem die Datensicherung eingespielt werden soll.

Assoziationen

Keine

[47] https://rsync.samba.org/

A.2.7 EMF Ecore Metamodell

Alle zuvor erläuterten Elemente des Metamodells wurden im Rahmen der Arbeit als EMF Metamodell umgesetzt. Der Vollständigkeit halber folgt nun ein Listing der Beschreibungssprache, mit dessen Hilfe Cloud Standby Modelle erstellt und verifiziert werden können[48].

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
name="model" nsURI="http://model/1.0" nsPrefix="model">
  <eClassifiers xsi:type="ecore:EClass" name="CloudStandby">
    <eStructuralFeatures xsi:type="ecore:EReference"
name="distributedSystem" lowerBound="1"
        eType="#//DistributedSystem" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference"
name="installation" lowerBound="1"
        eType="#//Installation" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference"
name="infrastructure" lowerBound="1"
        eType="#//Infrastructure" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference"
name="federation" lowerBound="1"
        eType="#//Federation" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Component">
    <eStructuralFeatures xsi:type="ecore:EReference"
name="federatedInstance" lowerBound="1"
        upperBound="-1" eType="#//FederatedInstance" contain-
ment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference"
name="federatedVirtualMachine"
```

[48] Das Metamodell ist auch zum Download verfügbar unter: https://github.com/alexlenk/CloudStandby/blob/master/org/cloudstandby/model/model/model.ecore

```
        lowerBound="1" eType="#//FederatedVirtualMachine"/>
    <eStructuralFeatures xsi:type="ecore:EReference"
name="federatedImage" lowerBound="1"
        eType="#//FederatedImage"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
name="name" lowerBound="1" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference"
name="requires" upperBound="-1"
        eType="#//Component"/>
    <eStructuralFeatures xsi:type="ecore:EReference"
name="installationTask" upperBound="-1"
        eType="#//InstallationTask"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass"
name="FederatedInstance">
    <eStructuralFeatures xsi:type="ecore:EAttribute"
name="hostname" lowerBound="1"
        eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference"
name="instance" upperBound="-1"
        eType="#//Instance" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference"
name="backupJob" upperBound="-1"
        eType="#//BackupJob" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass"
name="FederatedVirtualMachine">
    <eStructuralFeatures xsi:type="ecore:EReference"
name="virtualMachine" lowerBound="2"
        upperBound="-1" eType="#//VirtualMachine"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
name="name" lowerBound="1" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass"
name="InstallationTask" abstract="true">
    <eStructuralFeatures xsi:type="ecore:EAttribute"
name="name" lowerBound="1" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="StandardImage"
eSuperTypes="#//Image">
```

```
    <eStructuralFeatures xsi:type="ecore:EReference"
name="configuredImage" upperBound="-1"
        eType="#//ConfiguredImage" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
name="id" lowerBound="1" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
name="login" lowerBound="1" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
name="jsonDescription" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Image" ab-
stract="true">
    <eStructuralFeatures xsi:type="ecore:EAttribute"
name="name" lowerBound="1" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass"
name="FederatedImage">
    <eStructuralFeatures xsi:type="ecore:EReference"
name="image" lowerBound="2" upperBound="-1"
        eType="#//Image"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
name="name" lowerBound="1" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Cloud">
    <eStructuralFeatures xsi:type="ecore:EReference"
name="standardImage" lowerBound="1"
        upperBound="-1" eType="#//StandardImage" contain-
ment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference"
name="virtualMachine" lowerBound="1"
        upperBound="-1" eType="#//VirtualMachine" contain-
ment="true"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
name="name" lowerBound="1" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
name="location" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
```

```
    <eStructuralFeatures xsi:type="ecore:EAttribute"
name="providerId" lowerBound="1"
        eType="#//ProviderId"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
name="publicKey" lowerBound="1"
        eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
name="privateKey" lowerBound="1"
        eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass"
name="VirtualMachine">
    <eStructuralFeatures xsi:type="ecore:EAttribute"
name="name" lowerBound="1" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
name="id" lowerBound="1" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
name="jsonDescription" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass"
name="DistributedSystem">
    <eStructuralFeatures xsi:type="ecore:EReference"
name="cloud" lowerBound="1" upperBound="-1"
        eType="#//Cloud"/>
    <eStructuralFeatures xsi:type="ecore:EReference"
name="Component" lowerBound="1"
        upperBound="-1" eType="#//Component" contain-
ment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference"
name="cloudStandbyReplication"
        eType="#//CloudStandbyReplication" contain-
ment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Role" eSuper-
Types="#//InstallationTask">
    <eStructuralFeatures xsi:type="ecore:EReference"
name="installationTask" upperBound="-1"
        eType="#//InstallationTask"/>
  </eClassifiers>
```

```
  <eClassifiers xsi:type="ecore:EClass"
name="InstallationScript" abstract="true"
      eSuperTypes="#//InstallationTask"/>
  <eClassifiers xsi:type="ecore:EClass" name="BashScript"
eSuperTypes="#//InstallationScript">
    <eStructuralFeatures xsi:type="ecore:EAttribute"
name="code" lowerBound="1" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"
        defaultValueLiteral=""/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass"
name="ScriptCollection">
    <eStructuralFeatures xsi:type="ecore:EReference"
name="attribute" upperBound="-1"
        eType="#//Attribute" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference"
name="templateResource" upperBound="-1"
        eType="#//TemplateResource" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
name="name" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference"
name="installationScript" lowerBound="1"
        upperBound="-1" eType="#//InstallationScript" con-
tainment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass"
name="BashScriptCollection" eSuper-
Types="#//ScriptCollection"/>
  <eClassifiers xsi:type="ecore:EClass" name="Attribute">
    <eStructuralFeatures xsi:type="ecore:EAttribute"
name="name" lowerBound="1" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
name="value" lowerBound="1" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Federation">
    <eStructuralFeatures xsi:type="ecore:EReference"
name="federatedImage" lowerBound="1"
        upperBound="-1" eType="#//FederatedImage" contain-
ment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference"
name="federatedVirtualMachine"
```

```
        lowerBound="1" upperBound="-1"
eType="#//FederatedVirtualMachine" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass"
name="Infrastructure">
    <eStructuralFeatures xsi:type="ecore:EReference"
name="cloud" lowerBound="2" upperBound="-1"
        eType="#//Cloud" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Installation">
    <eStructuralFeatures xsi:type="ecore:EReference"
name="scriptCollection" upperBound="-1"
        eType="#//ScriptCollection" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference"
name="role" upperBound="-1" eType="#//Role"
        containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="TemplateFile"
eSuperTypes="#//TemplateResource">
    <eStructuralFeatures xsi:type="ecore:EAttribute"
name="content" lowerBound="1"
        eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Instance">
    <eStructuralFeatures xsi:type="ecore:EAttribute"
name="id" lowerBound="1" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
name="ip" lowerBound="1" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
name="modelIdentifier" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
name="dns" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
name="privateIp" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
name="privateDns" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
```

```
    <eStructuralFeatures xsi:type="ecore:EReference"
name="cloud" lowerBound="1" eType="#//Cloud"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EEnum" name="ProviderId">
    <eLiterals name="AmazonEC2" literal="aws-ec2"/>
    <eLiterals name="RackspaceCloudserversUS" value="1" lit-
eral="rackspace-cloudservers-us"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass"
name="ConfiguredImage" eSuperTypes="#//Image">
    <eStructuralFeatures xsi:type="ecore:EReference"
name="installationTask" upperBound="-1"
        eType="#//InstallationTask"/>
    <eStructuralFeatures xsi:type="ecore:EReference"
name="additionalDrive" upperBound="-1"
        eType="#//AdditionalDrive" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass"
name="TemplateResource" abstract="true">
    <eStructuralFeatures xsi:type="ecore:EAttribute"
name="uri" lowerBound="1" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="TemplateDir"
eSuperTypes="#//TemplateResource">
    <eStructuralFeatures xsi:type="ecore:EReference"
name="templateResource" lowerBound="1"
        upperBound="-1" eType="#//TemplateResource" contain-
ment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="BackupJob" ab-
stract="true">
    <eStructuralFeatures xsi:type="ecore:EAttribute"
name="period" lowerBound="1"
        eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EInt" default-
ValueLiteral="5"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
name="name" lowerBound="1" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="RsyncJob" eSu-
perTypes="#//BackupJob">
```

```
    <eStructuralFeatures xsi:type="ecore:EAttribute"
name="local" lowerBound="1" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
name="remote" lowerBound="1"
        eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass"
name="CloudStandbyReplication">
    <eStructuralFeatures xsi:type="ecore:EAttribute"
name="updateInterval" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass"
name="AdditionalDrive">
    <eStructuralFeatures xsi:type="ecore:EAttribute"
name="name" lowerBound="1" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
name="size" lowerBound="1" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
name="device" lowerBound="1"
        eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
name="imageId" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
name="mountPoint" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="BashJob" eSu-
perTypes="#//BackupJob">
    <eStructuralFeatures xsi:type="ecore:EAttribute"
name="code" lowerBound="1" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
</ecore:EPackage>
```

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[1] M. B. Anderson, „Which costs more: prevention or recovery“, *Manag. Nat. Disasters Environ. Wash. DC World Bank*, 1991.

[2] Apache, „jclouds“. [Online]. Verfügbar unter: http://jclouds.apache.org/. [Zugegriffen am: 06. Jan. 2014].

[3] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, und others, „Above the clouds: A berkeley view of cloud computing“, *EECS Dep. Univ. Calif. Berkeley Tech Rep UCBEECS-2009-28*, 2009.

[4] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, und I. Stoica, „A view of cloud computing“, *Commun. ACM*, Bd. 53, Nr. 4, S. 50–58, 2010.

[5] W. Arnold, T. Eilam, M. Kalantar, A. Konstantinou, und A. Totok, „Pattern based SOA deployment“, *Serv.-Oriented Comput. 2007*, S. 1–12.

[6] AWS Inc., „Amazon Web Services, Cloud Computing: Compute, Storage, Database“, 2013. [Online]. Verfügbar unter: https://aws.amazon.com/. [Zugegriffen am: 30. Mai 2013].

[7] N. Ayari, D. Barbaron, L. Lefevre, und P. Primet, „Fault tolerance for highly available internet services: concepts, approaches, and issues“, *Commun. Surv. Tutor. IEEE*, Bd. 10, Nr. 2, S. 34–46, 2008.

[8] barcoo, „Barcode Scanner für Preisvergleich & Testberichte“. [Online]. Verfügbar unter: http://www.barcoo.com/. [Zugegriffen am: 30. Mai 2013].

[9] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, und A. Warfield, „Xen and the art of virtualization“, *ACM SIGOPS Oper. Syst. Rev.*, Bd. 37, Nr. 5, S. 164–177, 2003.

[10] R. E. Barlow und F. Proschan, *Mathematical theory of reliability*. Siam, 1996.

[11] R. E. Barlow und F. Proschan, „Statistical theory of reliability and life testing: probability models“, DTIC Document, 1975.

[12] C. Baun, M. Kunze, J. Nimis, und S. Tai, „Cloud Computing: Web-Based Dynamic IT Services“, 2011.

[13] D. Bermbach und S. Tai, „Eventual consistency: How soon is eventual? An evaluation of Amazon S3's consistency behavior“, in *Proceedings of the 6th Workshop on Middleware for Service Oriented Computing*, 2011, S. 1.

[14] L. Bernardin, P. Chin, P. DeMarco, K. O. Geddes, D. E. G. Hare, K. M. Heal, G. Labahn, J. P. May, J. McCarron, und M. B. Monagan, *Maple 16 Programming Guide*. Maplesoft, Waterloo, 2012.

[15] S. K. Bose, S. Brock, R. Skeoch, und S. Rao, „CloudSpider: Combining Replication with Scheduling for Optimizing Live Migration of Virtual Machines across Wide Area Networks“, 2011, S. 13–22.

[16] R. Böttcher, „IT-Servicemanagement mit ITIL V3“, *Hann. Heise Verl.*, 2008.

[17] E. Brandtzæg, P. Mohagheghi, und S. Mosser, „Towards a domain-specific language to deploy applications in the clouds“, in *CLOUD COMPUTING 2012, The Third International Conference on Cloud Computing, GRIDs, and Virtualization*, 2012, S. 213–218.

[18] S. Brodsky, *Mastering XMI: Java programming with the XMI toolkit, XML, and UML*. New York; Chichester: Wiley, 2000.

[19] F. Büllingen und A. Hildebrand, „IT-Sicherheitsniveau in kleinen und mittleren Unternehmen - Studie im Auftrag des Bundesministeriums für Wirtschaft und Technologie“, Bundesministerium für Wirtschaft und Technologie (BMWi), Sep. 2012.

[20] Bundesamt für Sicherheit in der Informationstechnik, „BSI-Standard 100-4 - Notfallmanagement - Version 1.0“. 2008.

[21] Bundesamt für Sicherheit in der Informationstechnik, *Hochverfügbarkeitskompendium Band B: Bausteine*. 2013.

[22] Bundesamt für Sicherheit in der Informationstechnik, *Hochverfügbarkeitskompendium Band G: Einführung und methodische Grundlagen*. 2013.

[23] Bundesamt für Sicherheit in der Informationstechnik, „ITIL und Informationssicherheit“. 2005.

[24] Canonical Ltd, „Ubuntu Juju“. [Online]. Verfügbar unter: https://juju.ubuntu.com/. [Zugegriffen am: 04. Mai 2014].

[25] M. C. Caraman, S. A. Moraru, S. Dan, und C. Grama, „Continuous Disaster Tolerance in the IaaS clouds“, in *Optimization of Electrical and Electronic Equipment (OPTIM), 2012 13th International Conference on*, 2012, S. 1226–1232.

[26] M. Catan, R. Di Cosmo, A. Eiche, T. Lascu, M. Lienhardt, J. Mauro, R. Treinen, S. Zacchiroli, G. Zavattaro, und J. Zwolakowski, „Aeolus: Mastering the Complexity of Cloud Application Deployment“, in *Service-Oriented and Cloud Computing*, Bd. 8135, K.-K. Lau, W. Lamersdorf, und E. Pimentel, Hrsg. Springer Berlin Heidelberg, 2013, S. 1–3.

[27] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, und R. E. Gruber, „Bigtable: A distributed storage system for structured data“, *ACM Trans. Comput. Syst. TOCS*, Bd. 26, Nr. 2, S. 4, 2008.

[28] R. Di Cosmo, M. Lienhardt, R. Treinen, S. Zacchiroli, und J. Zwolakowski, „Optimal provisioning in the cloud“, Technical report, Aeolus project, 2013.

[29] R. Di Cosmo, S. Zacchiroli, und G. Zavattaro, „Towards a formal component model for the cloud“, in *Software Engineering and Formal Methods*, Springer, 2012, S. 156–171.

[30] R. Cowan, „Tortoises and hares: choice among technologies of unknown merit“, *Econ. J.*, Bd. 101, Nr. 407, S. 801–814, 1991.

[31] S. Crosby, R. Doyle, M. Gering, M. Gionfriddo, S. Hand, M. Hapner, D. Hiltgen, M. Johanssen, J. Leung, und F. Machida, „Open virtualization format specification“, *Stand. Technol. No DSP0243 DMTF Specif. Distrib. Manag. Task Force*, 2009.

[32] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, und A. Warfield, „Remus: High Availability Via Asynchronous Virtual Machine Replication“, in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, 2008, S. 161–174.

[33] K. Eichenwald, *Verschwörung der Narren*. Bertelsmann, 2006.

[34] T. Eilam, M. H. Kalantar, A. V. Konstantinou, und G. Pacifici, „Reducing the complexity of application deployment in large data centers“, in *2005 9th IFIP/IEEE International Symposium on Integrated Network Management, 2005. IM 2005.*, 2005, S. 221–234.

[35] T. Eilam, M. H. Kalantar, A. V. Konstantinou, G. Pacifici, J. Pershing, und A. Agrawal, „Managing the configuration complexity of distributed applications in internet data centers“, *Commun. Mag. IEEE*, Bd. 44, Nr. 3, S. 166–177, 2006.

[36] Ensembl, „Genome Browser“. [Online]. Verfügbar unter: http://www.ensembl.org/index.html. [Zugegriffen am: 28. Juni 2013].

[37] N. Ferry, F. Chauvel, A. Rossini, B. Morin, und A. Solberg, „Managing multi-cloud systems with CloudMF“, in *Proceedings of the Second Nordic Symposium on Cloud Computing & Internet Technologies*, 2013, S. 38–45.

[38] N. Ferry, A. Rossini, F. Chauvel, B. Morin, und A. Solberg, „Towards Model-Driven Provisioning, Deployment, Monitoring, and Adaptation of Multi-cloud Systems“, 2013, S. 887–894.

[39] A. Flissi, J. Dubus, N. Dolet, und P. Merle, „Deploying on the Grid with DeployWare“, 2008, S. 177–184.

[40] Forbes, „The Battle For The Cloud Is On -- Who Will Win? - Forbes“. [Online]. Verfügbar unter:

http://www.forbes.com/sites/groupthink/2013/05/22/the-battle-for-the-cloud-is-on-who-will-win/. [Zugegriffen am: 16. Jan. 2014].

[41] N. Fuqua, *Reliability engineering for electronic design*, Bd. 34. CRC Press, 1987.

[42] S. Gai, *Cisco unified computing system (UCS): a complete reference guide to the data center virtualization server architecture*. Indianapolis, IN: Cisco Press, 2010.

[43] E. Gamma, *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. Bonn; Reading, Mass. [u.a.]: Addison-Wesley, 1996.

[44] S. Ghemawat, H. Gobioff, und S.-T. Leung, „The Google file system“, präsentiert auf ACM SIGOPS Operating Systems Review, 2003, Bd. 37, S. 29–43.

[45] Google, „Drive“. [Online]. Verfügbar unter: http://www.google.com/drive/apps.html. [Zugegriffen am: 30. Mai 2013].

[46] Google Developers, „What Is Google App Engine? - Google App Engine“. [Online]. Verfügbar unter: https://developers.google.com/appengine/docs/whatisgoogleappengine. [Zugegriffen am: 30. Mai 2013].

[47] J. Gray und D. Bitton, „Disk shadowing“, in *VLDB*, 1988, Bd. 88, S. 331–338.

[48] J. Gray und D. P. Siewiorek, „High-availability computer systems“, *Computer*, Bd. 24, Nr. 9, S. 39–48, 1991.

[49] O. Haase, *Kommunikation in verteilten Anwendungen: Einführung in Sockets, Java RMI, CORBA und Jini*. Oldenbourg Verlag, 2008.

[50] S. M. Hawkins, D. C. Yen, und D. C. Chou, „Disaster recovery planning: a strategy for data security“, *Inf. Manag. Comput. Secur.*, Bd. 8, Nr. 5, S. 222–230, 2000.

[51] A. Hiles, *The definitive handbook of business continuity management*. Wiley, 2010.

[52] B. Hill, *Working with Microsoft Office 365: Running Your Small Business in the Cloud*. O'Reilly Media, Inc., 2012.

[53] M. Huber und G. Huber, „Prozess-und Projektmanagement für ITIL", *Wiesb. Vieweg Teubner Verl.*, 2011.

[54] W. Hummer, F. Rosenberg, F. Oliveira, und T. Eilam, „Testing Idempotence for Infrastructure as Code", in *Middleware Conference*, 2013.

[55] IBM Corporation, „IBM SmartCloud Provisioning", 2013. [Online]. Verfügbar unter: http://public.dhe.ibm.com/common/ssi/ecm/en/ibd03003usen/IBD03003USEN.PDF. [Zugegriffen am: 16. Apr. 2014].

[56] IBM Corporation, „IBM UrbanCode Deploy", 2013. [Online]. Verfügbar unter: http://public.dhe.ibm.com/common/ssi/ecm/en/rad14132usen/RAD14132USEN.PDF. [Zugegriffen am: 16. Apr. 2014].

[57] J. A. Illik, *Verteilte Systeme: Architekturen und Software-Technologien*. expert Verlag, 2007.

[58] K. Jackson, *OpenStack Cloud Computing Cookbook*. Packt Publishing, 2012.

[59] D. Jayasinghe, C. Pu, F. Oliveira, F. Rosenberg, und T. Eilam, „AESON: A Model-Driven and Fault Tolerant Composite Deployment Runtime for IaaS Clouds", in *Proceedings of the 2013 IEEE International Conference on Services Computing*, 2013, S. 575–582.

[60] R. Jhawar, V. Piuri, und M. Santambrogio, „A comprehensive conceptual system-level approach to fault tolerance in Cloud Computing", 2012, S. 1–5.

[61] R. Jhawar, V. Piuri, und M. Santambrogio, „Fault Tolerance Management in Cloud Computing: A System-Level Perspective“, *IEEE Syst. J.*, Bd. 7, Nr. 2, S. 288–297, Juni 2013.

[62] B. W. Johnson, „An introduction to the design and analysis of fault-tolerant systems“, in *Fault-tolerant computer system design*, 1996, S. 1–87.

[63] K. R. Joshi, M. A. Hiltunen, W. H. Sanders, und R. D. Schlichting, „Automatic model-driven recovery in distributed systems“, in *Reliable Distributed Systems, 2005. SRDS 2005. 24th IEEE Symposium on*, 2005, S. 25–36.

[64] M. H. Kalantar, F. Rosenberg, J. Doran, T. Eilam, M. D. Elder, F. Oliveira, E. C. Snible, und T. Roth, „Weaver: Language and runtime for software defined environments“, *IBM J. Res. Dev.*, Bd. 58, Nr. 2, S. 1–12, 2014.

[65] G. Katsaros, A. Lenk, M. Menzel, R. Skipp, und J. Rake-Revelant, *Open Datacenter Alliance: Service Orchestration with TOSCA White Paper*. 2014.

[66] G. Katsaros, M. Menzel, A. Lenk, J. R. Revelant, R. Skipp, und J. Eberhardt, „Cloud Application Portability with TOSCA, Chef and Openstack“, in *Proceedings of the 2014 IEEE International Conference on Cloud Engineering*, 2014, S. 295–302.

[67] K. Keeton, D. Beyer, E. Brau, A. Merchant, C. Santos, und A. Zhang, „On the road to recovery: restoring data after disasters“, in *ACM SIGOPS Operating Systems Review*, 2006, Bd. 40, S. 235–248.

[68] A. Kivity, Y. Kamay, D. Laor, U. Lublin, und A. Liguori, „kvm: the Linux virtual machine monitor“, in *Proceedings of the Linux Symposium*, 2007, Bd. 1, S. 225–230.

[69] M. Klems, S. Tai, L. Shwartz, und G. Grabarnik, „Automating the delivery of IT Service Continuity Management through cloud service orchestration“, in *Network Operations and Management Symposium (NOMS), 2010 IEEE*, 2010, S. 65–72.

[70] G. Klett, K.-W. Schröder, und H. Kersten, *IT-Notfallmanagement mit System*. Springer DE, 2011.

[71] D. E. Knuth, „Backus normal form vs. backus naur form“, *Commun. ACM*, Bd. 7, Nr. 12, S. 735–736, 1964.

[72] A. V. Konstantinou, T. Eilam, M. Kalantar, A. A. Totok, W. Arnold, und E. Snible, „An architecture for virtual solution composition and deployment in infrastructure clouds“, in *Proceedings of the 3rd international workshop on Virtualization technologies in distributed computing*, 2009, S. 9–18.

[73] T. Kurze, M. Klems, D. Bermbach, A. Lenk, S. Tai, und M. Kunze, „Cloud federation“, in *CLOUD COMPUTING 2011, The Second International Conference on Cloud Computing, GRIDs, and Virtualization*, 2011, S. 32–38.

[74] L. Lamport, „How to make a multiprocessor computer that correctly executes multiprocess programs“, *Comput. IEEE Trans. On*, Bd. 100, Nr. 9, S. 690–691, 1979.

[75] T. A. Lascu, J. Mauro, und G. Zavattaro, „Automatic Component Deployment in the Presence of Circular Dependencies“, in *10th International Symposium on Formal Aspects of Component Software, FACS 2013*, 2013.

[76] T. . Lascu, J. Mauro, und G. Zavattaro, „A Planning Tool Supporting the Deployment of Cloud Applications“, *Tools Artif. Intell. ICTAI 2013 IEEE 25th Int. Conf. On*, S. 213–220, Nov. 2013.

[77] A. Lenk, „Ein offenes Framework zur Erstellung kommerzieller Cloud Angebote auf Basis bestehender Applikationen“, Diplomarbeit, Karlsruher Institut für Techologie (KIT), 2009.

[78] A. Lenk, C. Danschel, M. Klems, D. Bermbach, und T. Kurze, „Requirements for an IaaS deployment language in federated Clouds“, in *Service-Oriented Computing and Applications (SOCA), 2011 IEEE International Conference on*, 2011, S. 1–4.

[79] A. Lenk, G. Katsaros, M. Menzel, J. Rake-Revelant, R. Skipp, E. Castro-Leon, und Gopan V P, „TIOSA: Testing VM Interoperabil-

ity at an OS and Application Level - A hypervisor testing method and interoperability survey“, in *ICE2 2014: IEEE International Conference on Cloud Engineering*, Boston, 2014.

[80] A. Lenk, M. Klems, J. Nimis, S. Tai, und T. Sandholm, „What’s inside the Cloud? An architectural map of the Cloud landscape“, in *Software Engineering Challenges of Cloud Computing, 2009. CLOUD’09. ICSE Workshop on*, 2009, S. 23–31.

[81] A. Lenk, M. Menzel, J. Lipsky, S. Tai, und P. Offermann, „What Are You Paying For? Performance Benchmarking for Infrastructure-as-a-Service Offerings“, in *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, 2011, S. 484–491.

[82] A. Lenk, M. Menzel, D. Müller, J. Rake-Revelant, R. Bederke, R. Skipp, M. Tadikonda, S. Govindan, G. P.V, und G. Katsaros, „Open Datacenter Alliance: Implementing the Open Data Center Alliance Virtual Machine Interoperability Usage Model“, präsentiert auf FORECAST, 2013.

[83] A. Lenk, J. Nimis, T. Sandholm, und S. Tai, „An Open Framework to Support the Development of Commercial Cloud Offerings based on Pre-Existing Applications“, präsentiert auf Annual International Conference on Cloud Computing and Virtualization (CCV 2010), 2010, S. 297–305.

[84] A. Lenk und F. Pallas, „Cloud Standby System and Quality Model“, *Int. J. Cloud Comput. IJCC*, Bd. 1, Nr. 2, S. 48–59, 2013.

[85] A. Lenk und F. Pallas, „Modeling Quality Attributes of Cloud-Standby-Systems“, in *Service-Oriented and Cloud Computing*, Springer, 2013, S. 49–63.

[86] A. Lenk und S. Tai, „Cloud Standby: Disaster Recovery of Distributed Systems in the Cloud“, in *Service-Oriented and Cloud Computing*, Bd. 8745, M. Villari, W. Zimmermann, und K.-K. Lau, Hrsg. Springer Berlin Heidelberg, 2014, S. 32–46.

[87] B. Liljas, „How to calculate indirect costs in economic evaluations“, *Pharmacoeconomics*, Bd. 13, Nr. 1, S. 1–7, 1998.

[88] Z. Li, M. Liang, L. O'Brien, und H. Zhang, „The Cloud's Cloudy Moment: A Systematic Survey of Public Cloud Service Outage", *ArXiv Prepr. ArXiv13126485*, 2013.

[89] G. S. Lynch, *Single point of failure: The 10 essential laws of supply chain risk management*. John Wiley and Sons, 2009.

[90] B. Mahr, „Die Informatik und die Logik der Modelle", *Inform.-Spektrum*, Bd. 32, Nr. 3, S. 228–249, 2009.

[91] Marion Schneider, „Die Konsum-Revolution! barcoo durchbricht 10 Millionen-Marke bei Downloads | barcoo blog - Wissen was Du kaufst!". [Online]. Verfügbar unter: http://www.barcoo.com/blog/2013/04/15/barcoo-durchbricht-10-mio-marke-bei-downloads. [Zugegriffen am: 11. Juni 2014].

[92] E. M. Maximilien, A. Ranabahu, R. Engehausen, und L. Anderson, „IBM altocumulus: a cross-cloud middleware and platform", in *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, 2009, S. 805–806.

[93] E. M. Maximilien, A. Ranabahu, R. Engehausen, und L. C. Anderson, „Toward cloud-agnostic middlewares", in *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications - OOPSLA '09*, Orlando, Florida, USA, 2009, S. 619.

[94] P. Mell und T. Grance, „The NIST definition of cloud computing", *NIST Spec. Publ.*, Bd. 800, S. 145, 2011.

[95] M. Menzel, M. Klems, H. A. Le, und S. Tai, „A configuration crawler for virtual appliances in compute clouds", in *Cloud Engineering (IC2E), 2013 IEEE International Conference on*, 2013, S. 201–209.

[96] Microsoft, „Determining cost of availability". [Online]. Verfügbar unter: http://technet.microsoft.com/en-us/library/bb432646.aspx. [Zugegriffen am: 06. Feb. 2014].

[97] R. Mietzner und F. Leymann, „Towards Provisioning the Cloud: On the Usage of Multi-Granularity Flows and Services to Realize a Unified Provisioning Infrastructure for SaaS Applications“, in *2008 IEEE Congress on Services - Part I*, Honolulu, HI, USA, 2008, S. 3–10.

[98] R. Mietzner, T. Unger, und F. Leymann, „Cafe: A generic configurable customizable composite cloud application framework“, *Move Meaningful Internet Syst. OTM 2009*, S. 357–364, 2009.

[99] S. Nadgowda, P. Jayachandran, und A. Verma, „I2Map: Cloud Disaster Recovery Based on Image-Instance Mapping“, in *Middleware 2013*, Springer, 2013, S. 204–225.

[100] S. Nelson, *Pro data backup and recovery*. Apress, 2011.

[101] S. Nelson-Smith, *Test-driven infrastructure with Chef*, 1st ed. Sebastopol, CA: O'Reilly, 2011.

[102] NetApp, „NetApp Private Storage for Amazon Web Services (AWS)“. [Online]. Verfügbar unter: http://www.netapp.com/us/media/tr-4133.pdf. [Zugegriffen am: 07. Apr. 2014].

[103] B. C. Neuman, „Readings in distributed computing systems“, *Chapter Scale Distrib. Syst. IEEE Comput. Soc. Los Alamitos CA*, S. 463–89, 1994.

[104] K. Neumann und M. Morlock, *Operations Research*, 2. Aufl. München: Hanser, 2002.

[105] OASIS, „Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0“. 18. März 2013.

[106] Object Management Group, Inc. (OMG), „Meta Object Facility (MOF) Core Specification Version 2.4.1“. 2011.

[107] Object Management Group, Inc. (OMG), „OMG MOF 2 XMI Mapping Specification Version 2.4.1“. 2013.

[108] Object Management Group, Inc. (OMG), „Unified Modeling Language (UML): Infrastructure, Version 2.0“. 2005.

[109] Object Management Group, Inc. (OMG), „Unified Modeling Language (UML), Superstructure Specification Version 2.4.1“. 2011.

[110] Openstack, „Heat“. [Online]. Verfügbar unter: https://wiki.openstack.org/wiki/Heat. [Zugegriffen am: 02. Aug. 2014].

[111] Opscode, „Chef“. [Online]. Verfügbar unter: http://www.opscode.com/chef/. [Zugegriffen am: 30. Mai 2013].

[112] M. Pokharel, S. Lee, und J. S. Park, „Disaster Recovery for System Architecture Using Cloud Computing“, 2010, S. 304–307.

[113] M. Pokharel, Seulki Lee, und Jong Sou Park, „Disaster Recovery for System Architecture Using Cloud Computing“, in *Applications and the Internet (SAINT), 2010 10th IEEE/IPSJ International Symposium on*, 2010, S. 304–307.

[114] Rackspace, „Rackspace Homepage“. [Online]. Verfügbar unter: http://www.rackspace.com/. [Zugegriffen am: 07. Feb. 2014].

[115] S. Rajagopalan, B. Cully, R. O'Connor, und A. Warfield, „SecondSite: disaster tolerance as a service“, in *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, 2012, S. 97–108.

[116] Salesforce, „Salesforce1 Platform: Trusted Application Development Platform - Salesforce.com“. [Online]. Verfügbar unter: http://www.salesforce.com/platform/overview/. [Zugegriffen am: 07. Feb. 2014].

[117] A. Sampaio und N. Mendonca, „Uni4Cloud: an approach based on open standards for deployment and management of multi-cloud applications“, in *Proceedings of the 2nd International Workshop on Software Engineering for Cloud Computing*, 2011, S. 15–21.

[118] S. Sanfilippo und P. Noordhuis, *Redis*. 2010.

[119] K. Schmidt, *High availability and disaster recovery*. Springer, 2006.

[120] N. Schuster, *Coordinating Service Compositions: Model and Infrastructure for Collaborative Creation of Electronic Documents*. KIT Scientific Publishing, 2013.

[121] D. Steinberg, *EMF: Eclipse Modeling Framework*, 2nd ed., Rev. and updated. Upper Saddle River, NJ: Addison-Wesley, 2009.

[122] Symantec, „2011 SMB Disaster Preparedness Survey - Global Results“, 2011.

[123] A. S. Tanenbaum und M. van Steen, *Verteilte Systeme-Prinzipien und Paradigmen (2. Aufl.).* Pearson Studium, 2008.

[124] K. Thulasiraman und M. N. Swamy, *Graphs: theory and algorithms*. Wiley-Interscience, 2011.

[125] C. Ullenboom, *Java ist auch eine Insel*, Bd. 8. Galileo Press, 2003.

[126] Verbraucherzentrale Bundesverband, „Was ist die ‚Ampelkennzeichnung‘.“ [Online]. Verfügbar unter: http://www.vzbv.de/mediapics/was_ist_die_ampel.pdf. [Zugegriffen am: 28. Juni 2013].

[127] P. Verissimo und L. Rodrigues, *Distributed systems for systems architects*, Bd. 1. Springer, 2001.

[128] VMware, „vCloud Suite: Infrastructure-as-a-Service (IaaS) & Cloud Computing | United States“. [Online]. Verfügbar unter: http://www.vmware.com/products/vcloud-suite/. [Zugegriffen am: 07. Feb. 2014].

[129] K.-H. Waldmann, *Stochastische Modelle: eine anwendungsorientierte Einführung*. Berlin: Springer, 2003.

[130] M. Wallace und L. Webber, *The disaster recovery handbook: A step-by-step plan to ensure business continuity and protect vital operations, facilities, and assets*. Amacom, 2010.

[131] T. White, *Hadoop: The definitive guide*. O'Reilly Media, Inc., 2012.

[132] M. E. Whitman, H. J. Mattord, und A. Green, *Principles of incident response and disaster recovery*. Cengage Learning, 2013.

[133] E. Wittern, A. Lenk, S. Bartenbach, und T. Braeuer, „Feature-based Configuration of Vendor-independent Deployments on IaaS", in *18th IEEE International Enterprise Distributed Object Computing Conference (EDOC)*, 2014.

[134] T. Wood, E. Cecchet, K. K. Ramakrishnan, P. Shenoy, J. Van der Merwe, und A. Venkataramani, „Disaster recovery as a cloud service: Economic benefits & deployment challenges", in *2nd USENIX Workshop on Hot Topics in Cloud Computing*, 2010.

[135] T. Wood, H. A. Lagar-Cavilla, K. Ramakrishnan, P. Shenoy, und J. Van der Merwe, „PipeCloud: using causality to overcome speed-of-light delays in cloud-based disaster recovery", in *Proceedings of the 2nd ACM Symposium on Cloud Computing*, 2011, S. 17.