HAW Hamburg

Grundseminar

Analyse abstrakter Architekturmodelle in verteilten Systemen

Bearbeiter:
Tom Schöner (2182801)

Betreuung:
Prof. Dr. Kai von Luck
Prof. Dr. Tim Tiedemann
Prof. Dr. Ulrike Steffens
Prof. Dr. Stefan Sarstedt

31. August 2017

Inhaltsverzeichnis

1 Vorwort

Diese schriftliche Ausarbeitung ist als Prüfungsleistung des Master-Studiengangs Informatik an der HAW im Grundseminar entstanden. Sie dient der Einarbeitung in das Thema Softwarearchitektur im Kontext von verteilten Systemen. Die Architektur von Modellen und dessen Komponenten sowie die Analyse von Softwarearchitekturen sind Bestandteil dieser Arbeit.

2 Einleitung

Software- und Systemarchitektur beschreiben in der Informatik den fundamentalen Baustein für die Beschreibung und Organisation von simplen sowie komplexen Softwaresystemen. Die frühe Ausarbeitung einer der Problemstellung angemessenen Softwarearchitektur, wenn auch nur als grobes abstraktes Modell, unterstützt dessen spätere Implementation. Im Allgemeinen steigt die Komplexität eines Systems über seinen Entwicklungszeitraum an und erschwert die Wartbarkeit. Die richtige Architektur, bzw. Module der Architektur in verteilten Systemen zu identifizieren ist daher keine triviale Aufgabe und umso bedeutender. Diese Ausarbeitung beleuchtet einige dieser Modelle als ganze sowie ausgewählte Module innerhalb der Architektur. Ferner werden auch Themen des Cloud-Computings näher betrachtet.

2.1 Motivation

Softwarearchitektur ist ein wichtiger Aspekt in Hinblick auf die Langlebigkeit und, im kommerziellen Bereich, Erfolg von Software-Projekten. IoT-Geräte und Smartphones sind Beispiele der vernetzten Welt und auch ein zentraler Bestandteil aktueller fortlaufender Forschung. Unter Anderem wird an der HAW in Hamburg im Bereich *Smart-Living* intensiv geforscht. Bestimmte Dienste müssen immer erreichbar sein und auch bei erhöhter Anfragelast funktionieren. Die Modellierung derart verteilter Systeme ist demgemäß für eine korrekte Interaktion ausschlaggebend und wird im Folgenden näher beleuchtet.

2.2 Softwarearchitektur

> „Die Softwarearchitektur ist die grundlegende Organisation eines Systems, dargestellt durch dessen Komponenten, deren Beziehungen zueinander und zur Umgebung sowie die Prinzipien, die den Entwurf und die Evolution des Systems bestimmen.“ (Ralf Reussner, p. 1, gekürzt)

Die Softwarearchitektur beschreibt demnach eine abstrahierte *High-Level*-Sichtweise auf Komponenten und dessen interne und externe Beziehungen eines Systems. Ralf Reussner beschreibt die Komponenten eines Systems später im Kapitel über die Architekturbeschreibung als Blackbox. Der Vergleich

mit der Blackbox verdeutlicht nochmals die Abstraktionsebene, denn für die Modellierung einer Softwarearchitektur ist die interne Logik und Prozessabwicklung einzelner Komponenten zu vernachlässigen. Das Zusammenspiel liegt hier im Fokus. Im Folgenden werden zwei Kategorien von Architekturen kurz erläutert: monolithische und verteilte Systeme.

2.3 Monolithische Systeme

Monolithische Systeme beschreiben in sich geschlossene, alleinstehende und zentralisierte Systeme. Sie können sowohl *Low-* als auch *High-Level* Systeme beschreiben, wobei *Low-Level* verstärkt die Hardwareebene betrachtet. In Kontrast zu verteilten Systemen (siehe Abschnitt 2.4) werden sie auf nur einer physischen oder auch virtuellen Maschine ausgeführt. Installation und Wartung monolithischer Systeme ist im Allgemeinen im Vergleich zu verteilten Systemen nicht mit so hoher Komplexität verbunden. Die von der äußersten Schicht einfach gehaltene Architektur bündelt somit alle Komponenten in einer Instanz.
Anforderungen wie die Skalierung bestimmter einzelner Komponente des Systems, bspw. aufgrund temporärer erhöhter Anforderungen, zeigen die Schwachstellen dieser Architektur. Die vertikale Skalierung[4] ist eine mögliche Antwort auf dieses Problem, allerdings keine langfristige. Physische Ressourcen haben bekanntlich ihre Grenzen und weitere Probleme wie *Single Point of Failure* werden hierdurch nicht gelöst.

2.4 Dezentrale, verteilte Systeme

Auch bei verteilten Systemen lässt sich eine Unterscheidung zwischen *Low-* und *High-Level* vornehmen. Bedeutende etablierte Architekturen für verteilte Systeme sind: Client-Server, Cluster, Grid oder auch Peer-to-peer (siehe Abschnitt 3.1).
In Hinblick auf Cloud-Computing bieten verteilte Systeme im Bereich Skalierbarkeit und Erreichbarkeit gegenüber monolithisch aufgebauten Systemen einen Vorteil. Als einfaches Beispiel kann hierfür ein HTTP-Server mit mehreren Instanzen betrachtet werden (siehe Abbildung 1), welche über einen Load-Balancer mittels *Round-Robin* angesteuert werden. Eine Skalierung ist in diesem Szenario durch ein Hinzufügen einer weiteren Instanz möglich und auch fehlerhafte Instanzen sollten bei richtiger Konfiguration die Erreichbarkeit nur vermindern aber nicht zu einem Stillstand des Systems führen (*Fail*

Tolerance).

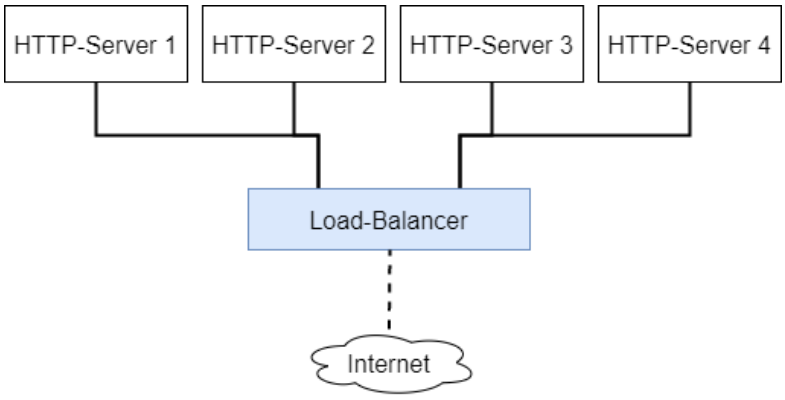

Abbildung 1: HTTP-Server Instanzen mit vorgeschalteten Load-Balancer

Allgemeiner ist jedoch die erhöhte Komplexität und die Latenz ein Nachteil, welche jeweils der Aufteilung der System-Komponenten auf mehrere physische oder virtuelle Instanzen zu verschulden ist.

3 Architekturmodelle und Komponenten

Zunächst wird das Peer-to-peer Modell (siehe Abschnitt 3.1) als mögliche Implementation für ein verteiltes System erläutert. Die beiden folgenden Abschnitte Service-Discovery und CAP Theorem befassen sich weniger mit einem Modell als Ganzen, sondern vielmehr mit für verteilte Systeme essentielle Prinzipien und Komponenten.
Fehling et al. beschreibt in seiner Arbeit über das Cloud-Computing noch weitere Architekturen und Pattern.

3.1 Peer-to-peer

Reine Peer-to-peer Systeme sind im Allgemeinen „verteilte Systeme ohne jegliche zentrale Kontrolleinheit oder hierarchische Organisation, in denen jeder *Peer* Software mit gleichgestellter Funktionalität ausführt“ (Stoica et al., 2003, p. 1) Die sogenannten Peers sind innerhalb des Netzes gleichgestellt, es gibt jedoch auch Protokoll-Implementationen die eine Gruppierung der Peers nach ihrer Qualifikation vornehmen. Peer-to-peer Systeme können von

der reinen Form abweichen und einen zentralen Server für die Verwaltung verwenden oder auch dynamisch mehrere zentrale Server und somit ein hierarchisches System bestimmen (Hybride Peer-to-peer Systeme). Im Vergleich zum Client-Server Modell bietet das Peer-to-peer Modell aufgrund seiner dynamischen Natur gute Skalierungseigenschaften[5]. Beispiele für solche Systeme sind: Dateiaustauschbörsen, Instant Messaging, Internettelefonie oder auch verteiltes Rechnen. Durch die dezentrale Nutzung von Ressourcen ist es möglich Rechenleistung, Speicher oder auch Bandbreite untereinander aufzuteilen.

Die direkte Kommunikation der Peers untereinander ist besonders herauszustellen. Direkt miteinander verbundene Peers werden Nachbarn genannt, indirekt verbundene Peers sind in dem Modell auch vorgesehen, wie in Abbildung 2 zu sehen ist. Das Peer-to-peer Netzwerk ist hier ein ungerichteter Graph, bei dem die Peers die Knoten und deren Verbindungen die Kanten darstellen. Die Netzwerke bauen normalerweise auf physischen Netzwerken auf und weisen eine davon unabhängige Topologie[5] auf. Sind die Peers nicht direkt miteinander verbunden, müssen Routing-Protokolle die Anfrage über verbundene Peers durchreichen. Peers besitzen Client- so wie Server Funktionalität, sie sind demnach berechtigt Anfragen anzunehmen und auch zu versenden.

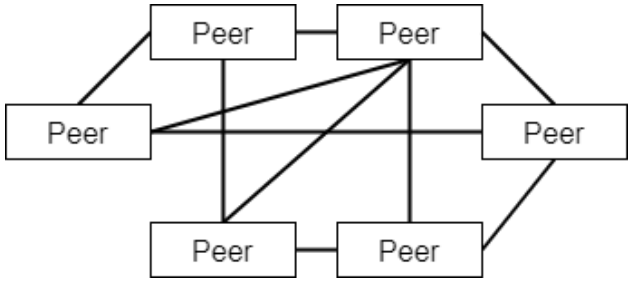

Abbildung 2: Peer-to-peer Architektur mit 6 Peers.

Für Peer-to-peer Architekturen gibt es kein standardisiertes Protokoll, sondern nur Protokolle, die unter diese Kategorie fallen. Das 2003 entwickelte lookup-Protokoll Chord[7] versucht beispielsweise die Problematik der effizienten und dynamischen Identifizierung der Peers mittels Hash-Tables zu lösen. Das Chord Protokoll verfügt nur über eine Operation: Anhand eines gegeben Schlüssels einen anderen Peer identifizieren. Hierfür muss nicht jeder Peer das gesamte Netzwerk an Peers kennen, sondern nur Routing-

Informationen zu einigen wenigen Peers. Somit muss in einem N-Peer System jeder Peer nur $\log(N)$ andere Peers kennen. Der Routing-Algorithmus kann den gewünschten Peer über Anfragen an bekannte Peers auflösen. Das dynamische Verhalten, bedingt durch das fortlaufende Hinzufügen und Entfernen von Knotenpunkten, ist hierbei die eigentliche Herausforderung. Chord aktualisiert seine internen verteilten Hash-Tabellen hierbei automatisch und schafft somit eine hohe *Availabilty* (siehe auch Abschnitt 3.3). Der Algorithmus funktioniert bereits, sobald jeder Knotenpunkt nur eine korrekte Information über das Netzwerk enthält, wenn auch mit deutlichen Geschwindigkeitsverlust. Der genaue Algorithmus soll an dieser Stelle nicht weiter beleuchtet werden. Des Weiteren wird auch ein Vergleich von Chord mit üblichen DNS aufgestellt, dessen Service in ähnlicher Weise auch von Chord durchgeführt werden könnte.

Das Chord Protokoll verdeutlicht die Schwierigkeiten bei der Kommunikation von Komponenten in dynamischen, verteilten Systemen. Die Verwendung eines systemunspezifischen Protokolls[1] ist für die Wartung und das Verständnis von Systemen nicht zu vernachlässigen. Protokolle bieten ein standardisiertes Verfahren, an dem sich ein System orientiert. Auch ohne das System zu kennen ist es somit möglich, Entscheidungen und Aussagen über Teilgebieten der Architektur zu treffen.

3.2 Service-Discovery

Im Kontext von Cloud-Computing soll an dieser Stelle das Prinzip der Service-Discovery erläutert werden. Ähnlich zu Peer-to-peer in Abschnitt 3.1 muss auch hier eine Verbindung zu anfangs unbekannten Instanzen aufgebaut werden. Service-Discovery ist keine Architektur, sondern eine der internen Komponenten, aus denen sich letztendlich die Cloud-Architektur zusammensetzt. Als Service-Discovery wird die automatisierte Erkennung von Services in einem Netzwerk bezeichnet. In anderen Kontexten kann auch die automatische Erkennung von Hardware unter den Begriff des Service-Discovery fallen. In cloudbasierten Applikationen haben die Services dynamisch zugewiesene IPs, wodurch für ihre Identifikation eine einfache Konfigurationsdatei mit Routing-Informationen nicht mehr ausreicht. Des Weiteren ist es bei Microservices nicht unüblich mehrere Instanzen desselben Services durch horizon-

[1] Systemunspezifisch in dem Sinne, dass es als generelle Lösung für Peer-to-peer Architekturen angewendet werden kann

tale Skalierung zu verwenden.

Es gibt zwei erprobte Pattern für Service-Discovery: *client-side* discovery und *server-side* discovery. Bei *client-side* discovery (siehe Abbildung 3)[2] gibt es eine Service Registry Instanz, die jeder Client und Service kennt. Die Service Registry kann bei einer Anfrage des Clients in seiner Datenbank passende Services raussuchen und diese weitereichen. Der Client ermittelt daraufhin mittels Load-Balancing den konkreten Service. Zu diesem Zeitpunkt ist dem Client der Service samt seiner IP bekannt.

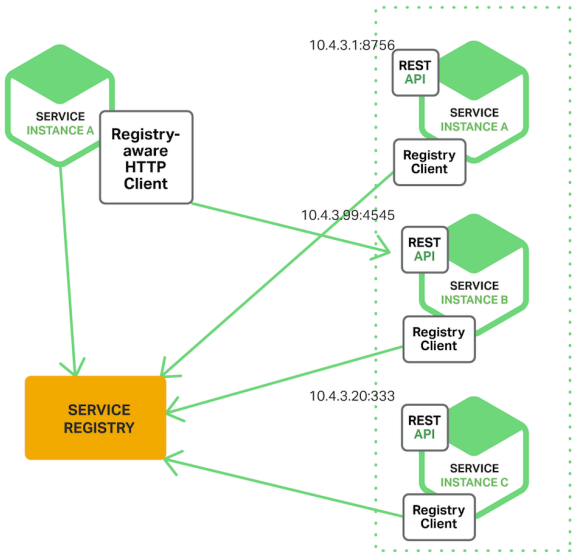

Abbildung 3: Client-side Service Discovery mit drei dynamisch hinzugefügten Service Instanzen

Bei server-side discovery (siehe Abbildung 4) kennt der Client die Service Registry nicht direkt, sondern tätigt seinen Request an einen Load-Balancer. Dass es mehrere Instanzen des Services gibt ist dem Client somit gänzlich unbekannt, aber auch nicht relevant. Der Load-Balancer übernimmt nun die Aufgabe den Request weiter zu verarbeiten und ähnlich zur client-side discovcery die Service Registry zu nach den Services zu befragen. Ein Beispiel für einen solchen Load-Balancer ist der Cloud-Load Balancer von AWS.

[2] https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/, Aufruf am: 25.08.2017

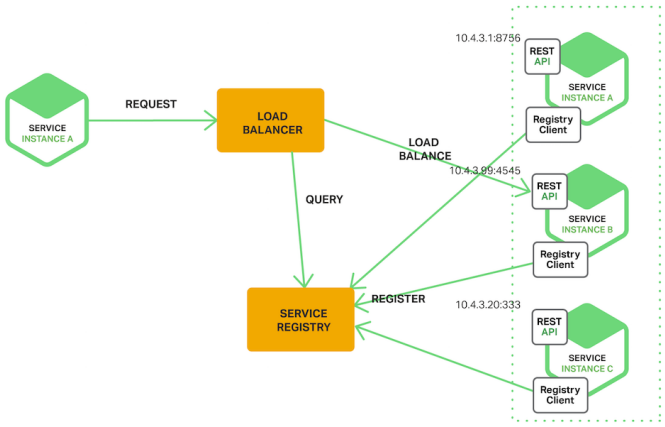

Abbildung 4: Server-side Service Discovery mit drei dynamisch hinzugefügten Service Instanzen

Die Service-Registry sollte aufgrund ihrer hohen Ressourcenauslastung aus einem Cluster von Servern bestehen, welcher ein Replikationsprotokoll für die Konsistenz verwendet. Das Open-Source Projekt Eureka[3] von Netflix beschreibt eine solche Service Registry.

3.3 CAP Theorem

Das Zusammenspiel von Komponenten wird oftmals von elementaren Prinzipien kontrolliert und beeinflusst. Für verteilte Systeme ist besonders bei der Modellierung von genannten Systemen bei der Datenverarbeitung das CAP Theorem bedeutend und wird daher im Folgenden erläutert:

Das CAP Theorem beschreibt den unvermeidlichen Zusammenhang zwischen:

- ***C****onsistency* (Konsistenz)
- ***A****vailability* (Verfügbarkeit)
- ***P****artition-tolerance* (Ausfalltoleranz)

Es sei demnach unmöglich innerhalb eines asynchron aufgebauten Netzwerk-Models auf Daten mit Lese- und Schreibvorgängen zuzugreifen, auf das alle

[3] https://github.com/Netflix/eureka, Aufruf am 22.08.2017

oben aufgeführten Punkte *Consistency, Availability* und *Partition-tolerance* zutreffen[2]. In Bezug auf Web-Services, wenn auch auf andere Gebiete anwendbar, sorgen atomare Operationen mit einer totalen Ordnung für *Consistency*. Jede Anfrage unter den gleichen Bedingungen sollte das gleiche Ergebnis erzeugen, demnach ist ein deterministisches System wünschenswert.

Availability bedeutet für verteilte Systeme, dass eine Anfrage auch in einer Antwort resultiert. Wenn auch nur als schwache Definition anzusehen, muss jeder hierfür zuständige Algorithmus terminieren. Für verteilte Datenbanken kann Datenabstraktion die *Availability* steigern[6]. Sei eine Datenbank mit vier Knoten gegeben, daher einen Master und drei Slave-Knoten im Read-only Modus. Wird nun eine große Datenmenge mit beispielsweise 2GB im Master hinterlegt, muss diese auch auf die drei Knoten repliziert werden. Selbst bei einer Übertragungsrate von 100MBit/s dauert dies mindestens 8 Minuten:

$$2000\text{MB} \cdot \left(\frac{100\text{MB}}{8\text{s}}\right)^{-1} \cdot 3 = 480\text{s} \tag{1}$$

Lässt sich nun durch Datenabstraktion die Größe mit einem Faktor 20 auf 100MB abbilden, so verringert sich auch die Replikationsdauer bei linearer Annahme um diesen Faktor auf 24 Sekunden. Mit dieser Methode können die Daten schneller von den replizierten Knoten abgefragt werden, wodurch letztendlich die *Availability* profitiert.

Partition-tolerance erlaubt es einem Netzwerk eine beliebige Anzahl an Nachrichten von einer *Node* zur anderen zu verschicken. Wenn ein Netzwerk partitioniert wurde, gehen alle aus diesem Netzwerk gesendeten Nachrichten verloren[2]. Für *Consistency* bedeutet dies, dass auch bei verlorenen Nachrichten atomar verlaufen muss. In Verbindung mit *Availability* muss eine Anfrage auch terminieren.

4 Analyse von Architekturmodellen

Für die Bewertung und Analyse von Architekturmodellen müssen bestimmte Kriterien aufgestellt und festgelegt werden, wie Architektur beschrieben werden sollen. Hierbei ist die Bewertung immer relativ zu der individuellen Problemstellung zu sehen. Die Benutzung einer klar definierten und standardisierten Beschreibungssprache für Architekturen wie ISO/IEC DIS 25961 (auch ANSI/IEEE 1471-2000) schafft einen konzeptionellen Rahmen[5]. Der Standard definiert die Beschreibungssprache als eine Menge von Modellen,

wie UML. Zudem gibt es viele unterschiedliche Ansichten, welche Standpunkte der Architektur zusammengefasst werden sollten. Die „4+1“-Standpunktmenge von Kruchten beinhaltet folgende Kategorien[3]:

- **Logischer Standpunkt**: Funktionalität des Systems für den Endnutzer, Objekte und Komponenten
- **Prozess-Standpunkt**: Programmablauf durch Prozesse, Threads und Tasks und deren Kommunikation
- **Physischer/ Deployment Standpunkt**: Verteilung der Softwarekomponenten auf physikalischer Ebene
- **Entwicklungs-Standpunkt**: Aus der Sicht eines Entwicklers, Softwaremanagement

Der fünfte Standpunkt zeigt nur Anwendungsfälle oder Anwendungsszenarien auf. Generell ist dieser grobe Überblick ein sinnvoller Startpunkt, um Softwarearchitekturen zu analysieren, da es die verschiedenen Bereiche sinnvoll voneinander getrennt beleuchtet.

Ein mögliches Tool zur Analyse von bestehenden Architekturen ist *SonarGraph*[4]. Es wird als statisches Codeanalyse-Tool unter Anderem für Monitoring, Metriken oder das Einhalten festgelegter Regeln verwendet. Die Abhängigkeiten eines Systems können mittels eines Graphen visuell dargestellt werden. Auf diese Weise lassen sich beispielsweise Zyklen erkennen und entfernen.

5 Fazit

Softwarearchitekturen bilden das Fundament eines Systems. Die Analyse bestehender Systeme ist nicht trivial und zeitaufwendig. Daher sollte die noch abstrakte Architektur vor der Implementierungsphase ausführlich entwickelt und untersucht werden. Es ist jedoch eher die Norm als die Ausnahme, dass sich Anforderungen und Spezifikationen von Projekten oft unvorhersehbar ändern. Analysetools wie SonarGraph können bei diesem Prozess helfen und einen Überblick verschaffen. Die komplexe und individuelle Natur von Softwarearchitekturen, bedingt durch Faktoren wie Programmiersprache, asynchrone oder parallele Berechnungen, Anzahl der Komponenten oder auch

[4] https://www.hello2morrow.com/products/sonargraph, Aufruf am 20.08.2017

Abhängigkeiten der Komponenten untereinander, erschwert diesen Prozess jedoch erheblich und die Tools stoßen an ihre Grenzen.

6 Weiterführende akademische Arbeit

Für die weitere Arbeit im Bereich verteilte Systeme und Cloud-Computing sollte ein konkreteres Themengebiet herausgefiltert werden. Es bietet sich an Architekturmodelle wie Peer-to-peer oder Grid auf neue Anforderungen hin zu untersuchen. Wie verhalten sich genannte Systeme bei starker Skalierung und wie robust sind diese im Fehlerfall? Inwiefern lassen sich diese Prinzipien auf das Cloud-Computing anwenden? Beispielsweise könnte die Integration einer Peer-to-peer Architektur in die Service-Discovery untersucht werden. Die Orchestrierung von Komponenten in Cloud-Architekturen bietet einen weiteren Ansatzpunkt.

Literatur

[1] Christoph Fehling, Frank Leymann, Ralph Retter, Walter Schupeck, and Peter Arbitter. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer, 2014. doi: 10.1007/978-3-7091-1568-8.

[2] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002. ISSN 0163-5700. doi: 10.1145/564585.564601. URL http://doi.acm.org/10.1145/564585.564601.

[3] P. B. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12 (6):42–50, Nov 1995. ISSN 0740-7459. doi: 10.1109/52.469759.

[4] M. Michael, J. E. Moreira, D. Shiloach, and R. W. Wisniewski. Scale-up x scale-out: A case study using nutch/lucene. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–8, March 2007. doi: 10.1109/IPDPS.2007.370631.

[5] Wilhelm Hasselbring (Hrsg.) Ralf Reussner. *Handbuch der Softwarearchitektur*, volume 2. dpunkt Verlag, 2009.

[6] Steven P. Reiss and Qi Xin. Building dynamic, long-running systems. In *Proceedings of the 4th International Workshop on Software Engineering for Systems-of-Systems*, SESoS '16, pages 19–24, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4172-1. doi: 10.1145/2897829.2897831. URL http://doi.acm.org/10.1145/2897829.2897831.

[7] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, Feb 2003. ISSN 1063-6692. doi: 10.1109/TNET.2002.808407. URL http://ieeexplore.ieee.org/document/1180543/.

Abbildungsverzeichnis