hochschule mannheim

Ein Proxy zur Erzeugung, Aufzeichnung und Wiedergabe von Netzwerk-Unterbrechungen in einer Testumgebung für verteilte Systeme

Jonathan Arns

Bachelor-Thesis

zur Erlangung des akademischen Grades Bachelor of Science (B.Sc.)

Studiengang Unternehmens- und Wirtschaftsinformatik

Fakultät für Informatik
Hochschule Mannheim

15.11.2021

Betreuer
Prof. Dr. Oliver Hummel, Hochschule Mannheim
Prof. Thomas Smits, Hochschule Mannheim

Arns, Jonathan:
Ein Proxy zur Erzeugung, Aufzeichnung und Wiedergabe von Netzwerk-Unterbrechungen in einer Testumgebung für verteilte Systeme / Jonathan Arns. —
Bachelor-Thesis, Mannheim: Hochschule Mannheim, 2021. 65 Seiten.

Arns, Jonathan:
A proxy to create, record and replay network partitions in a test environment for distributed systems / Jonathan Arns —
Bachelor Thesis, Mannheim: University of Applied Sciences Mannheim, 2021. 65 pages.

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ich bin damit einverstanden, dass meine Arbeit veröffentlicht wird, d. h. dass die Arbeit elektronisch gespeichert, in andere Formate konvertiert, auf den Servern der Hochschule Mannheim öffentlich zugänglich gemacht und über das Internet verbreitet werden darf.

Mannheim, 15.11.2021

Jonathan Arns

Zusammenfassung

Verteilte Systeme sind heute verbreiteter denn je zuvor. In diesen Systemen sind unzuverlässige Netzwerke und Partitionstoleranz-Bugs ein reales Problem. Aufgrund eines Mangels an geeigneten Test- und Debugging-Tools sind viele dieser Bugs außerdem schwierig zu reproduzieren und zu debuggen.

Diese Arbeit stellt einen proxy-basierten Replay-Mechanismus für Netzwerk-Unterbrechungen in verteilten Systemen vor. Dieser ermöglicht es, Netzwerk-Unterbrechungen zu erzeugen, aufzuzeichnen und die Aufzeichnungen als Replays am laufenden System abzuspielen. Auf diese Weise können Partitionstoleranz-Bugs aufgezeichnet und zum Debugging deterministisch reproduziert werden.

Zur Evaluation des Konzepts wurde im Rahmen dieser Arbeit ein Prototyp für einen proxy-basierten Replay-Debugger entwickelt. Der Prototyp wurde an einem realen Startup-System und an Simulationen bekannter Partitionstoleranz-Bugs aus Redis und MongoDB getestet. Die Experimente zeigen, dass ein proxy-basierter Replay-Mechanismus in der Lage ist, reale Bugs zuverlässig zu reproduzieren und somit ein effektives Werkzeug beim Debugging von verteilten Systemen sein kann.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Verteilte Systeme sind heute integral für viele Anwendungsfälle in der Informatik von Social Media über Streaming-Anbieter bis hin zu Machine-Learning.

Verteilte Systeme bringen Herausforderungen und potentielle Fehler mit sich, die in alleinstehenden Prozessen meist nicht auftreten. Eine davon ist, dass in verteilten Systemen oft einzelne Komponenten des Systems fehlschlagen, anstelle des ganzen Systems. Beispielsweise kann ein einzelner Knoten in einem System abstürzen oder das Netzwerk zeitweise ausfallen. Um die Verfügbarkeit und Korrektheit von Systemen zu garantieren, ist es wichtig, verteilte Systeme zu entwickeln, die sich von Fehlern in einzelnen Komponenten selbstständig erholen und weiter funktionieren können. [1]

Eine häufige Annahme, die problematisch ist, ist dass das Netzwerk, welches zur Kommunikation verwendet wird, fehlerfrei ist [2, 3]. In der Realität sorgen Netzwerk-Unterbrechungen und andere Fehler dafür, dass die Kommunikation innerhalb verteilter Systeme grundsätzlich ist [1, 4]. Verteilte Systeme müssen also auch in der Gegenwart von Netzwerk-Unterbrechungen die Integrität des jeweiligen Systems und seiner Daten garantieren. Obwohl das Problem an sich bereits intensiv erforscht wurde, führen Netzwerk-Unterbrechungen weiterhin zu schwerwiegenden Fehlern in vielen wichtigen Produktionssystemen [5]. Dazu trägt unter anderem ein Mangel an Test- und Debugging-

Tools für diesen Zweck bei [6, 7], der eine Folge der rasanten Entwicklung moderner Cloud-Infrastruktur ist [8]. Zwar gibt es bereits Tools, die effektiv darin sind, neue Bugs zu finden, diese sind aber nicht in der Lage, gefundene Bugs deterministisch zu reproduzieren [6]. Das ist sowohl für das Erstellen von Regressionstests als auch beim Debugging ein Problem.

Record-and-Replay ist ein Debugging-Ansatz, welcher es ermöglicht, selbst komplexe Bugs deterministisch zu reproduzieren. Existierende Replay-Debugger für verteilte Systeme sind allerdings wenig ausgereift und beschränken sich auf Systeme in bestimmten Programmiersprachen [9, 10]. Diese Arbeit stellt ein Konzept und einen Prototypen für einen auf Netzwerk-Unterbrechungen spezialisierten Replay-Debugger vor, welcher programmiersprachenagnostisch Bugs in verteilten Systemen reproduzieren kann.

Der hier verfolgte Ansatz für ein solches Tool ist eine Testumgebung, welche einen Proxy verwendet, um Netzwerk-Unterbrechungen zu erzeugen. Dieser Proxy zeichnet zusätzlich den gesamten Netzwerk-Verkehr des Systems inklusive Metadaten auf, um die erzeugten Netzwerk-Unterbrechungen später in einem Replay gleich wiedergeben zu können. Eine grafische Oberfläche stellt den Netzwerk-Verkehr und die Log-Ausgaben des System-under-Test (SUT) gemeinsam chronologisch sortiert dar, um dem Nutzer bei der Identifikation von Bugs zu helfen.

Zu Anfang der Arbeit werden zunächst die technischen Grundlagen über verteilte Systeme, Netzwerk-Unterbrechungen und relevante Technologien erläutert. Im Anschluss folgt eine Darstellung des aktuellen Stands der Technik, sowie offener Forschungsfragen und noch Herausforderungen im Forschungsgebiet, aus denen die Hypothese abgeleitet wird. Kapitel 4 knüpft daran an, indem im Detail die Idee und der daraus entwickelte Prototyp beschrieben wird.

Für die empirische Evaluation des Prototypen werden erst alle entworfenen Experimente beschrieben, bevor in Kapitel 6 die Ergebnisse zusammengefasst werden. Im Anschluss werden alle Ergebnisse und deren Einschränkungen diskutiert. Basierend auf den Gesamtergebnissen der Arbeit werden Empfehlungen für eine Weiterentwicklung den Konzepts gegeben.

Kapitel 2

Grundlagen

Dieses Kapitel fasst die für diese Arbeit relevanten Grundlagen zu verteilen Systemen, Netzwerk-Unterbrechungen und adjazenten Themen zusammen.

2.1 Verteilte Systeme

Gunawi et. al. [4] definieren ein verteiltes System als ein System, in dem Hardware- oder Software-Komponenten, die auf oder an vernetzten Rechnern laufen, untereinander kommunizieren und ihre Aktionen ausschließlich mittels Nachrichten über das Netzwerk koordinieren. Van Steen et. al. [2] fügen dieser Definition hinzu, dass die jeweils autonomen Komponenten gemeinsam einem Nutzer des Systems gegenüber wie ein einziges System erscheinen. Ein Nutzer kann dabei sowohl ein Mensch als auch eine Applikation sein.

Moderne verteilte Systeme sind heute allerdings nur noch selten direkt auf physikalisch vernetzen Rechnern deployed. Stattdessen wird Software immer öfter Cloud-Native entwickelt und betrieben. Cloud-Native entwickelte Systeme sind hoch-verteilte Systeme, die auf virtualisierter Hardware in Containern in der Cloud laufen. Klassischerweise monolithische oder wenig verteilte Systeme werden nach der Cloud-Native-Philosophie in viele Microservices unterteilt, die in einem DevOps Workflow durch CI/CD Pipelines automatisiert getestet und deployed werden. Diese Microservices werden in elasti-

schen orchestrierungs-Plattformen deployed, und können somit automatisch und nach Bedarf horizontal skaliert werden. [11] Diese Systeme werden unter der Annahme entwickelt, dass Hardware nicht beständig ist und jederzeit fehlschlagen kann [12].

2.2 Netzwerk-Unterbrechungen

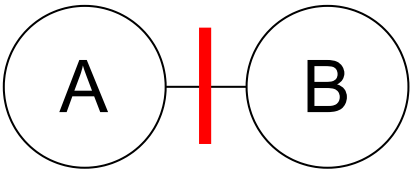
Eine Netzwerk-Unterbrechung in einem verteilten System ist eine Situation, in der ein Teil des Netzwerks ausfällt, sodass Teile des Systems, die eigentlich miteinander kommunizieren können sollten, sich nicht mehr erreichen können. Netzwerk-Unterbrechungen können zu fatalen Folgen, wie dem Verlust oder der Korruption von Daten, führen. Sie können außerdem in ganz unterschiedlichen Größenordnungen auftreten, sowohl zwischen geographisch unterschiedlich gelegenen Data-Centern als auch zwischen Rechnern im gleichen Data-Center. [5]

Zwar gibt es keine zuverlässigen Zahlen dazu, wie oft Netzwerk-Unterbrechungen auftreten, auch deshalb, weil Netzwerk nicht gleich Netzwerk ist und die Unterschiede auch im Bezug auf die Häufigkeit und Dauer von Netzwerk-Unterbrechungen sehr groß sind. Aus anekdotischen Berichten und internen Studien großer Unternehmen lässt sich trotzdem schließen, dass Netzwerk-Unterbrechungen nicht nur ein akademisches Problem sind, sondern auch in der Realität auftreten und Schaden anrichten. Aus diesem Grund ist es wichtig, verteilte Systeme so zu entwerfen, dass sie mit Netzwerk-Unterbrechungen umgehen können, beispielsweise durch die Nutzung eines Algorithmus für verteilten Konsens (siehe Kapitel 2.3). [3]

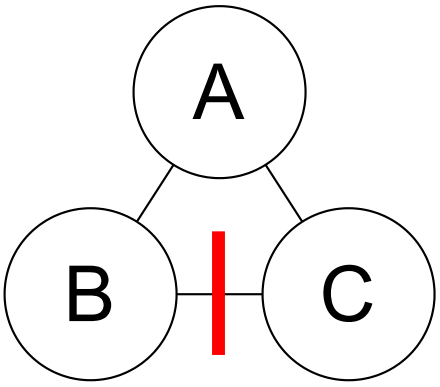
2.2.1 Arten von Netzwerk-Unterbrechungen

Netzwerk-Unterbrechungen lassen sich in drei Kategorien einteilen. Vollständige Netzwerk-Unterbrechungen wie in Abbildung 2.1a trennen ein verteiltes System in zwei komplett alleinstehende Teile. Sie sind die häufigste Art von Netzwerk-Unterbrechung und können sowohl zwischen verschiedenen Data-

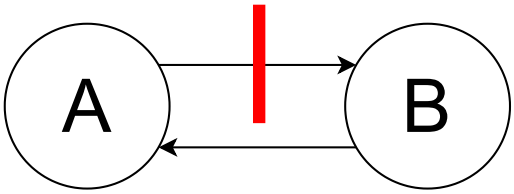
Centern als auch zwischen Teilen eines Systems im gleichen Data-Center auftreten. Nicht vollständige Netzwerk-Unterbrechungen wie in Abbildung 2.1b trennen einen Teil des Systems von einem anderen, während ein dritter Teil des Systems weiterhin beide anderen Teile erreichen kann. Sie entstehen, indem zwei Data-Center voneinander getrennt werden, die jedoch weiterhin für einem drittes Data-Center erreichbar sind. Einseitige Netzwerk-Unterbrechungen wie in 2.1c lassen Nachrichten nur in eine Richtung fließen. Diese Art von Netzwerk-Unterbrechung tritt selten auf und wird ausgelöst durch fehlerhafte Konfigurationen oder Hardware-Fehler. [5]



(a) Vollständige Netzwerk-Unterbrechung: Das System ist in zwei alleinstehende Teile getrennt. [5]



(b) Nicht vollständige Netzwerk-Unterbrechung: Teile des Systems sind voneinander getrennt, aber nicht alle Knoten des Systems sind betroffen. [5]



(c) Einseitige Netzwerk-Unterbrechung: Nachrichten können nur in eine Richtung fließen. [5]

Abbildung 2.1: Arten von Netzwerk-Unterbrechungen

2.2.2 Das CAP-Theorem

Das von Eric Brewer [13] entwickelte CAP-Theorem besagt, dass jedes verteilte System, welches Daten zwischen seinen Knoten teilt, nur zwei der drei Eigenschaften Konsistenz (Consistency), Erreichbarkeit (Availability) und Partitionstoleranz haben kann.

Konsistenz bedeutet, dass jeder Knoten des Systems jeden Client-Request korrekt beantwortet. Korrekt kann in diesem Kontext beispielsweise auch bedeuten, dass eine Anfrage nicht bearbeitet wird und eine entsprechende Meldung zurückgegeben wird. Generell sollte ein konsistentes System aber niemals falsche Daten liefern. Erreichbarkeit bedeutet, dass ein System jede Anfrage von einem Client beantwortet. Partitionstoleranz bedeutet, dass ein System in der Gegenwart von Netzwerk-Unterbrechungen funktioniert. [14]

Da Netzwerke in der Regel nicht zuverlässig sind und Netzwerk-Unterbrechungen in der Realität auftreten [3], gibt es bei der Entwicklung verteilter Systeme wie Datenbanken, die vom CAP-Theorem betroffen sind, nur zwei sinnvolle Möglichkeiten: CP und AP. Mit CP wird die Konsistenz der Daten sichergestellt, mit AP die Verfügbarkeit. In der Realität befindet sich zwischen den beiden Extremen allerdings ein Spektrum und Systeme wie MongoDB bietet beispielsweise die Möglichkeit, über verstellbare Konsistenz für jede Operation einzeln einzustellen, wie konsistent die Daten sein müssen [15]. Mit geringerer Konsistenz geht dabei eine entsprechend höhrere Verfügbarkeit einher, eine Anfrage mit geringeren Anforderungen an Konsistez hat also eine höhere Chance, erfolgreich zu sein.

2.2.3 Partitionstoleranz-Bugs

Die Bugs, mit denen sich diese Arbeit befasst, sind Bugs in der Partitionstoleranz von verteilten Systemen, sie werden also durch Netzwerk-Unterbrechungen ausgelöst. Die Effekte von Partitionstoleranz-Bugs sind laut einer Studie von Yuan et. al. [7] in über 79% der Fälle katastrophal. Dabei lösen beispielsweise über 26% der untersuchten Bugs den Verlust von Daten, über 13% Stale Reads und über 5% Dirty Reads aus.

Stale und Dirty Reads sind Leseoperationen gegen eine Datenstruktur, bei denen der Client falsche Daten erhält. Konkret werden bei einem Stale Read, dargestellt in Abbildung 2.2, Daten gelesen, die nicht mehr aktuell sind. In Systemen mit eventual Consistency werden Stale Reads in Kauf genommen, um die Verfügbarkeit der Daten zu erhöhen, in streng konsistenten Systemen hingegen sollten sie niemals auftreten [5]. Ein Dirty Read, dargestellt in Abbildung 2.3, ließt Daten aus einer Schreiboperation, die später zurück gerollt wird. Eine Schreiboperation, die zurückgerollt wird, ist im Endeffekt als wäre sie niemals passiert, sie sollte also auch niemals gelesen werden können. [16]

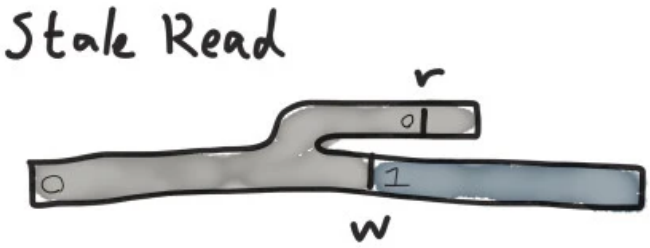


Abbildung 2.2: Visualisierung eines Stale Reads als Leseoperation in einer verzweigten Transaktions-Historie. Ist 'w' eine Schreiboperation, die einen Wert überschreibt, welcher später trotzdem von 'r' gelesen wird, dann ist 'r' ein Stale Read. [16]

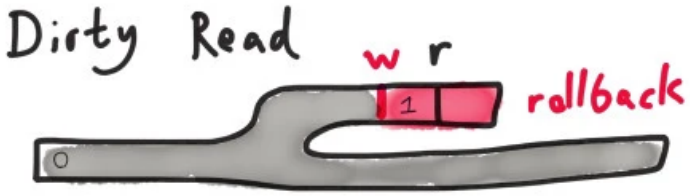


Abbildung 2.3: Visualisierung eines Dirty Reads als Leseoperation in einer verzweigten Transaktions-Historie. Ist 'w' eine Schreiboperation, die später zurückgerollt wird, dann ist 'r' ein Dirty Read. [16]

2.3 Verteilter Konsens

Die Problemstellung des verteilten Konsens ist, trotz der Existenz von Fehlern und Asynchronität in einem verteilten System zuverlässig einheitliche Entscheidungen zwischen Knoten des Systems zu treffen. Alle Entscheidungen im Sinne von verteiltem Konsens müssen dabei final, also unveränderlich sein. Außerdem muss garantiert sein, dass irgendwann eine Entscheidung getroffen wird. Der Entscheidungsprozess darf also nicht stagnieren. Ein Algorithmus, der das Problem des verteilten Konsens löst, erlaubt es, zuverlässige verteilte Systeme aus unzuverlässigen Komponenten zu bauen. [17] Da heutige Netzwerke, die zur Kommunikation innerhalb verteilter Systeme zwingend notwendig sind, fast immer unzuverlässig sind [3], wird die Umsetzung verteilter Systeme wie MongoDB und Kubernetes erst durch verteilten Konsens in der Praxis möglich [17].

2.3.1 Raft

Raft ist ein Algorithmus für verteilten Konsens. Raft repliziert ein Log zwischen den Knoten eines verteilten Systems, dabei können Log-Einträge beispielsweise Datenbank-Transaktionen oder andere Events sein. In Raft hat jeder Knoten immer einen von drei Zuständen: Follower, Leader oder Candidate. Die möglichen Zustandsübergänge sind in Abbildung 2.4 abgebildet. Follower sind Knoten, die den Zustand des Leaders replizieren. Auf sie kann lesend, aber nicht schreibend zugegriffen werden. Jedes Raft-Cluster hat immer genau einen Leader. Dieser ist der einzige Knoten, der Schreiboperationen von außen durchgeführen kann. Der Leader wird von den anderen Knoten des Clusters gewählt. Candidates sind Knoten, für die eine Wahl zum Leader läuft. Raft verwendet Heartbeat-Requests vom Leader zu den anderen Knoten und Timeouts auf Followers und Candidates, um Probleme wie Netzwerk-Unterbrechungen zu entdecken und Wahlen für einen neuen Leader zu starten. Jeder Knoten hält einen Zähler für den sog. Term, bzw. die aktuelle Amtszeit, in der sich das Cluster befindet. Mit jeder neuen Election wird dieser Zähler erhöht. Knoten folgen keinem Leader, der einen geringeren

Term hat als sie selbst. Außerdem kann ein Knoten pro Term nur für einen Leader stimmen. So stellt Raft sicher, dass ein Cluster, in dem der Leader nicht mindestens die Hälfte des Clusters erreichen kann, nach einer gewissen Zeit immer einen neuen Leader wählt. Das ist wichtig, weil alle Entscheidungen in Raft Mehrheitsentscheidungen sind und ein Leader, der weniger als die Hälfte des Clusters erreichen kann, somit keine Entscheidungen treffen, also auch keine Daten schreiben kann. [18]

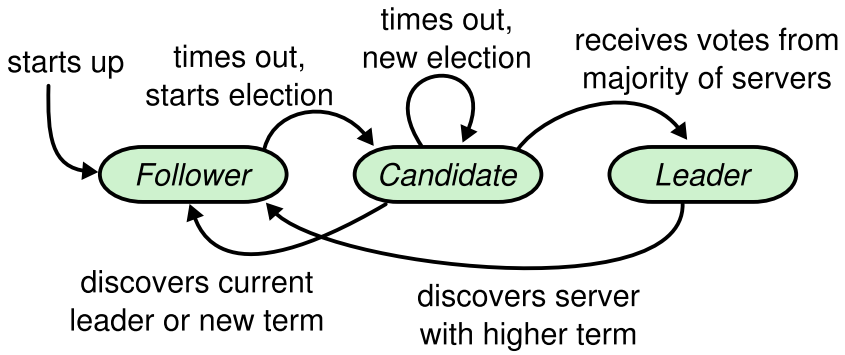


Abbildung 2.4: Zustände und Zustandsübergänge von Knoten in Raft. Ein Follower wird zum Candidate und startet eine Wahl, wenn er zu lange keine Nachrichten vom Leader empfängt. Erhält ein Candidate Stimmen von einer absoluten Mehrheit der Knoten im Cluster, wird er zum neuen Leader. Ein Leader bleibt in der Regel bestehen, bis er beispielsweise abstürzt oder durch eine Netzwerk-Unterbrechung vom einer Mehrheit des Clusters getrennt wird. [18]

Raft hat viele Details, auf die in dieser groben Darstellung des Algorithmus nicht eingegangen werden kann. Der Artikel, in dem Ongaro und Ousterhout [18] den Algorithmus vorstellen, bietet einen vollständigen Überblick über diese Feinheiten.

2.3.2 Linearisierbarkeit

Die von Herlihy und Wing [19] eingeführte Linearisierbarkeit ist eine Korrektheitseigenschaft für nicht-blockierende Datenstrukturen. Sie ist ein Werkzeug zur Entscheidung, ob eine Reihe von beobachteten Operationen an einer Datenstruktur unter den Regeln der jeweiligen Datenstruktur in der beob-

achteten Reihenfolge möglich sein. Linearisierbarkeit wird unter Anderem verwendet, um das Verhalten von verteilten Systemen auf Korrektheit zu prüfen. Raft unterstützt die Implementierung von Systemen mit linearisierbarer Semantik [18].

Kapitel 3

Stand der Technik

Im Umgang mit Fehlern in verteilten Systemen werden eine Reihe verschiedener Ansätze mit unterschiedlichen Zielen verfolgt. Failure Testing und Chaos Engineering sind zwei ähnliche Methoden, die sich darauf verlassen, Fehler mit Hilfe von Tests zu finden, mit dem Unterschied, dass Failure Testing in einer Testumgebung und Chaos Engineering in der Produktionsumgebung eingesetzt werden. Daneben gibt es Log-analyse und Tracing-Systeme, die vor unter anderem dazu dienen, aufgetretene Fehler nachzuvollziehen. Zu diesem Zweck gibt es einige weitere, oft experimentelle Debugging-Tools. Diese reichen von Tools, die einem klassischen Debugger ähneln, bis hin zu Tools, die sich vor allem auf die Visualisierung von Systemen und Abläufen konzentrieren. [20]

Nachfolgend werden die genannten Methoden und jeweils wichtige Tools weiter erläutert. Außerdem wird betrachtet, welche Rolle Proxies als Technologie heute in dem Anwendungsbereich einnehmen und welche weiteren Möglichkeiten sie in adjazenten Bereichen bieten. Schlussendlich wird daraus in Kombination mit den noch offenen Herausforderungen des Forschungsfeldes die zentrale Hypothese dieser Arbeit abgeleitet.

3.1 Failure Testing

Failure Testing ist ein Ansatz zum Testen der Fehlertoleranz verteilter Systeme. Die Grundidee ist immer, als Client mit dem System zu interagieren und gleichzeitig die Umgebung des Systems zu manipulieren, beispielsweise mit Netzwerk-Unterbrechungen. Failure Tests werden als automatisierte Integrationstests über eine Programmierschnittstelle definiert und umfassen in der Regel den Aufbau und Abbau des Systems, eine Reihe definierter Interaktionen mit dem System und zufällige oder deterministische Fehler-Injektionen in die Systemumgebung. Zusätzlich wird oft nach den Tests die Einhaltung von Invarianten des Systems anhand eines Modells geprüft, um in zufallsgetriebenen Tests automatisch Bugs identifizieren zu können. [21]

Majumdar et. al. [22] zeigen, dass zufallsgetriebenes Failure-Testing effektiv für Partition-Toleranz Bugs ist, obwohl die Menge an potentiell fehlerhaften Abläufen in realen Systemen gewaltig ist. Diese Effektivität zeigt sich auch in der Realität an Tools wie Jepsen[1]. Jepsen ist ein Failure-Testing-Framework in der Sprache Clojure, welches speziell auf das Verhalten von verteilten Systemen bei Netwerk-Unterbrechungen u.Ä. ausgerichtet ist. Jepsen hat sich auf dem Gebiet zu einem Industriestandard für das Testen von verteilten Datenbanken entwickelt [23] und wird für Projekte wie MongoDB, etcd, VoltDB, Hazelcast, Elasticsearch und CockroachDB verwendet und hat in jedem dieser Systeme bereits reale Bugs aufgedeckt [24].

Bei Jepsen-Tests werden auf einem Kontrollknoten eine Reihe von Client-Prozessen ausgeführt, die Requests an das SUT senden. Gleichzeitig werden mit Hilfe von sog. Nemesis Prozessen zufallsbasiert verschiedene Fehler in das System eingeführt. Jepsen greift dazu per SSH auf die einzelnen Knoten des SUT zu und verwendet die in Linux integrierten Möglichkeiten zum Filtern von Netzwerk-Paketen (iptables) zur Erzeugung von Netzwerk-Unterbrechungen. Neben Netzwerk-Unterbrechungen ist Jepsen in der Lage, verschiedene andere Fehler zu erzeugen, wie beispielsweise Unterschiede zwischen den Systemuhren der einzelnen Knoten. Am Ende eines Tests wird verifiziert, ob alle durchgeführten Operationen unter den Regeln des SUT

[1] https://github.com/jepsen-io/jepsen

linearisierbar sind. Sind sie das nicht, hat Jepsen einen Bug gefunden. [25]

Failure-Testing eignet sich vor allem gut für Komponenten mit hohen Anforderungen an Fehlertoleranz und Korrektheit, wie verteilte Datenbanken, Message-Broker und verteilte Dateisysteme. Failure-Testing ist heute der effektivste Ansatz, um Bugs in solchen Systemen zu finden [23] und der hohe Aufwand Failure-Tests einzurichten, lohnt sich hier besonders, da diese Systeme in der Regel eine große Menge von Nutzern haben, die sich auf deren Zuverlässigkeit verlassen.

3.2 Chaos Engineering

Chaos Engineering verfolgt das Ziel, die Resilienz und Verfügbarkeit von großen, heterogenen verteilten Systemen in der Produktions-Umgebung zu verbessern. Dazu werden direkt in der Produktions-Umgebung Experimente mit Fehlern wie Abstürzen von virtuellen Maschinen und Netwerk-Unterbrechungen durchgeführt, um Schwachstellen zu identifizieren und zu verifizieren, damit das System im Ernstfall mit derartigen Fehlern umgehen kann. Basiri et. al. [26], die Chaos Engineering, wie es bei Netflix eingesetzt wird, vorstellen, definieren den Ablauf eines Experiments dabei in vier Schritten.

1. Definiere den Ruhezustand des Systems anhand vorhandener Ausgaben.
2. Hypothese: Der Ruhezustand bleibt sowohl in der Kontrollgruppe als auch in der Versuchsgruppe während des Experiments erhalten.
3. Injiziere Ereignisse wie Server-Abstürze und Netzwerk-Unterbrechungen in der Versuchsgruppe von Services.
4. Versuche die Hypothese anhand von Unterschieden zwischen Kontroll- und Versuchsgruppe zu widerlegen.

Bekannt gemacht wurde die Methode durch Netflix [23], unter anderem mit dem dort entwickelten Chaos Monkey. Dieser ist ein Service, der automatisiert während der normalen Arbeiteszeiten Fehler im Produktionssystem

erzeugt. Die injizierten Fehler sind dabei Abstürze von internen Services, Abstürze von virtuellen Maschinen und Netzwerk-Unterbrechungen und erhöhte Netzwerk-Latenz für einzelne Nachrichten. Netflix stellt heraus, dass als Folge dessen Entwickler ihre Services resilient gegen derartige Fehler bauen und die Verfügbarkeit des Systems sich dadurch deutlich verbessert. Zusätzlich zu automatisierten Tools wie dem Chaos Monkey wird beim Chaos Engineering auf manuelle Experimente gesetzt, die teilweise sehr große Störungen wie Netzwerk-Unterbrechungen zu mehreren Data-Centern auf einmal umfassen. Erweisen sich manuelle Experimente als effektiv, werden diese wiederum nach und nach automatisiert. [26]

Chaos Engineering eignet sich besonders gut für sehr große, heterogene Systeme bei großen Organisationen wie Netflix, die in der Lage sind, spezialisierte Chaos Engineers einzustellen oder auszubilden. Dort hat sich die Methode aus der Not heraus, die Qualität immer größerer, heterogener Systeme sichern zu müssen, als Best Practice und fester Teil der Engineering-Kultur etabliert. [23]

3.3 Log-Analyse und Tracing

Log Analyse ist ein leichtgewichtiger Black-Box Ansatz zur Fehleridentifikation und -Analyse, der sich für Systeme eignet, die nicht modifiziert werden können [20]. Es werden so viele Logs wie möglich aus den verschiedenen Teilen eines Systems aggregiert und gemeinsam analysiert. Aufgrund der großen Menge an Logs sind diese allein oft nicht direkt einem Menschen nützlich. Stattdessen werden Data-Mining-Techniken eingesetzt, um relevante Informationen aus den aggregierten Logs zu gewinnen [27].

Eine deutlich schwergewichtigere Lösung als die Black-Box-Log-Analyse ist das Tracing eines verteilten Systems. Die Idee ist, jede Aktion, die im Zusammenhang mit einem User-Request passiert, aggregiert aufzuzeichnen, um so den gesamten Weg eines Requests durch das System zeigen zu können. Dabei können verschiedene Informationen aufgezeichnet werden, unter anderem Log-Nachrichten und RPC-Aufrufe, bzw. interne Nachrichten, je nachdem,

was das Ziel des konkreten Tracing Systems ist. Im Gegensatz zur einfachen Log-Analyse erfordert Tracing Eingriffe in das verteilte System, um die notwendigen Informationen zu generieren und mit Trace-IDs zu kennzeichnen. Die erzeugten Traces können dafür in der Regel gut visualisiert werden und bieten Entwicklern in dieser Form direkt nützliche Einsichten in das System. [28]

Tracing ist eine vielseitige Methode, die je nach Implementierung unterschiedliche Use-Cases bedienen kann. Neben verteiltem Profiling zur Identifikation von Bottlenecks im System und der Diagnose von Problemen im Ruhezustand des Systems zählt Anomalieerkennung in verteilten Systemen zu diesen Use-Cases. Dazu gehört sowohl die Erkennung als auch das Debugging von selten auftretenden Problemen, die beispielsweise durch Netzwerk-Unterbrechungen ausgelöst werden können. [29]

Ähnlich wie Chaos Engineering wird Tracing von Organisationen mit großen, heterogenen Systemen in Produktions-Umgebungen eingesetzt. Mit Dapper [28] hat Google eine eigene Tracing Lösung geschaffen, die sich vor allem für das Profiling von Systemen und die Diagnose von Problemen im Ruhezustand des Systems eignet [29]. Magpie [30] ist ein Tracing System, welches sich insbesondere zur Annomalieerkennung eignet, sowohl für Probleme in der Korrektheit als auch der Performance von verteilten Systemen [29].

3.4 Debugger

Neben Werkzeugen wie Tracing, die ebenfalls zum Debugging von verteilten Systemen dienen, gibt es eine Reihe weiterer Tools, die konkret diesem Zweck dienen, oft mit einem speziellen Fokus auf entweder eine bestimmte Art von Systemen oder eine bestimmte Klasse von Fehlern. Die Tools lassen sich in zwei grobe Kategorien einteilen: Visuelle Debugger und klassische Debugger.

3.4.1 Klassische Debugger

Die Tools in dieser Kategorie versuchen, das klassische, interaktive Debugging von nicht-verteilten Systemen mit Debuggern wie GDB auf verteilte Systeme zu übertragen. Die Herausforderung dabei ist, dass klassische Debugger darauf ausgelegt sind, genau einen Prozess zu kontrollieren, in verteilten Systemen aber per Definition mehrere Prozesse, oft auf verschiedenen Rechnern, existieren. Zusätzlich zu Kontrollfluss und Zustandsänderungen in einzelnen Prozessen müssen durch die Verteiltheit des Systems ebenfalls Nachrichtenflüsse zwischen den Prozessen beobachtet werden. Achar et. al. [31] stellen heraus, dass interaktives Debugging im klassischen Sinne nicht in allen verteilten Systemen möglich ist. Stattdessen muss ein System gewisse Einschränkungen im Bezug auf Zustandsänderungen und die Trennung zwischen lokalen und geteilten Zuständen einhalten. Mit GoTcha [31] stellen sie einen vollständigen interaktiven Debugger für Systeme mit dem Global Object Tracker (GoT) Programmiermodell vor. Konkret funktioniert GoTcha nur für Systeme, die eine GoT Implementierung in Python namens Spacetime[2] verwenden. Damit ist GoTcha zwar für solche Systeme ein mächtiges Tool, in seiner Anwendbarkeit aber eingeschränkt.

Ein anderer Ansatz, der interaktives Debugging ermöglichen kann, welcher auch in nicht-verteilten Systemen Anwendung findet [32], um besonders flüchtige Bugs zu reproduzieren, ist Replay-Debugging. Dieser Ansatz wurde erstmals 1997 auch auf verteilte Systeme angewandt [9], damals im Modul für verteilte Applikationen des Compilers der Programmiersprache Ada95. Damit konnten Abläufe in verteilten Systemen aufgezeichnet werden, um diese zu Debugging-Zwecken auf einzelne Knoten des Systems wiederzugeben.

Ein moderneres Beispiel für die Idee ist liblog [10], ein Replay-Debugger für C und C++ Applikationen, der in der Lage ist, Netzwerk-Verkehr aufzuzeichnen. Der Mechanismus hinter liblog ist, für eine Applikation alle Aufrufe der Standardbibliothek libc, inklusive ihrer Ergebnisse, sowie alle über ein Netzwerk empfangenen Nachrichten zu loggen, sodass diese später deterministisch in einem Replay wiedergegeben werden können.

[2] `https://github.com/Mondego/spacetime`

3.4.2 Visuelle Debugger

Ein Problem des klassischen interaktiven Debugging-Ansatzes ist, alle relevanten Informationen verständlich darzustellen. Das liegt daran, dass nicht nur die Zustände der einzelnen Knoten, sondern als zusätzliche Ebene auch die Kommunikation zwischen den Knoten relevant ist. Die Problematik wird unter anderem dadurch deutlich, dass liblog beispielsweise gleich mehrere GBD Instanzen als Frontend für Replays benötigt [10].

Um das Problem zu lösen haben sich mit der Zeit eine Reihe von Debugging-Tools für verteilte Systeme entwickelt, die die Zustände einzelner Prozesse abstrahieren und sich darauf konzentrieren, Abläufe und Zustände auf Systemebene zu visualisieren. Ein gemeinsames Konzept der verschiedenen Lösungen ist die Verwendung von Zeit-Raum-Diagrammen zur Visualisierung von Events und Nachrichtenflüssen zwischen den einzelnen Knoten. Ein Beispiel hierfür ist in Abbildung 3.1 dargestellt. Diese zeigt ein interaktives Zeit-Raum-Diagramm aus dem Tool ShiViz [33], welches zusätzliche Detailinformationen zu Events bietet.

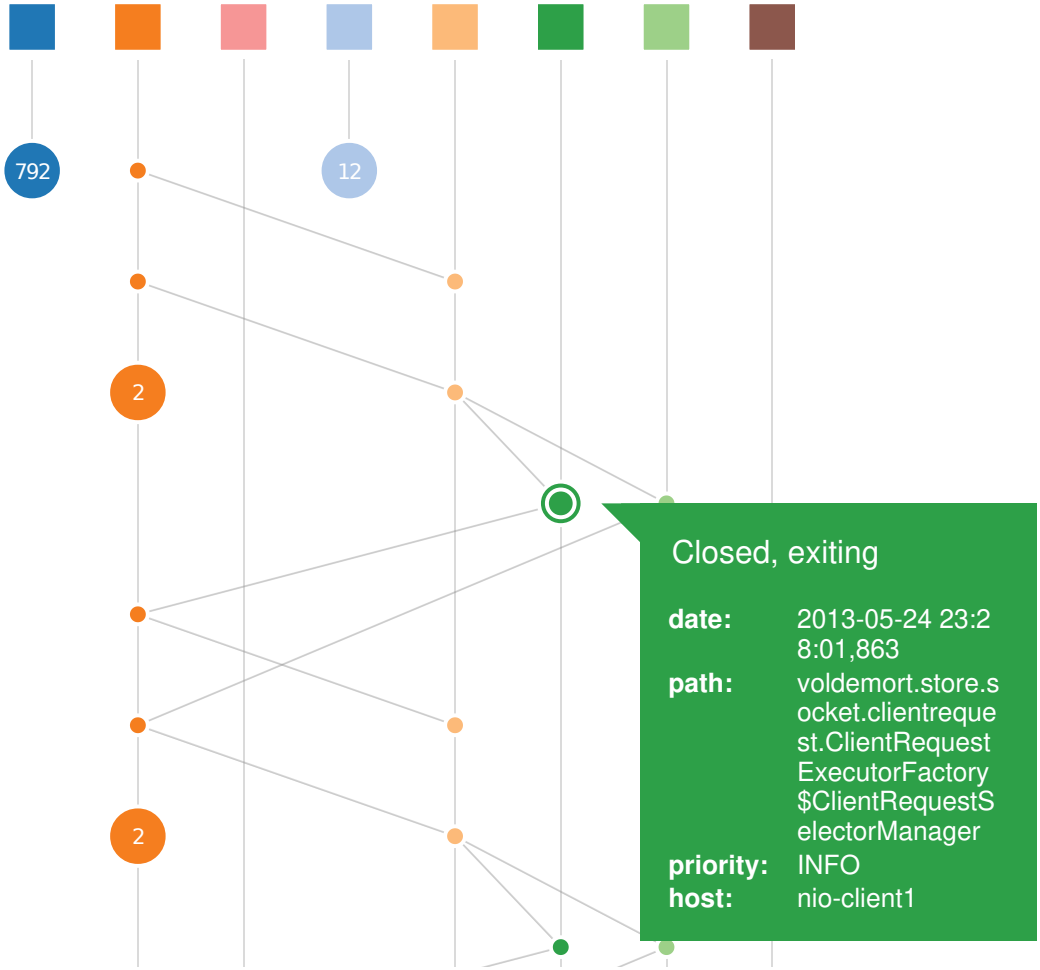


Abbildung 3.1: Interaktives Raum-Zeit-Diagramm in Shiviz [33]

Ein sehr mächtiger visueller Debugger ist Oddity [34]. Das Tool bietet interaktives Debugging, das in vielen Aspekten klassischen Debuggern ähnelt, allerdings nur auf Systemebene. Oddity erlaubt es, die Ausführung von einzelnen Knoten des SUT zu pausieren und den Erhalt einzelner Nachrichten und Events an den Knoten zu kontrollieren, alles in einer graphischen Oberfläche, die den Zustand des Systems inklusive anstehender Nachrichten und Events darstellt. Ein Screenshot dieser Darstellung ist in Abbildung 3.2 zu sehen. Oddity erlaubt es außerdem, den Systemzustand auf einen früheren Zustand zurückzusetzen und so mehrere Kontrollflüsse in einer Session zu debuggen. Zusätzlich zu der grafischen Oberfläche, die live den Systemzustand

abbildet, kann Oddity auch Zeit-Raum-Diagramme für Abläufe des Systems generieren.

Bei all diesen Vorzügen hat Oddity zwei große Einschränkungen. Einerseits funktioniert das Tool nur für Systeme, die als eine Menge von Event-Handlern implementiert sind. Andererseits erfordert Oddity erheblichen Aufwand zur Einrichtung, da eine auf das jeweilige SUT zugeschnittene Implementierung einer Oddity-API erforderlich ist, die direkt mit dem SUT interagieren kann. Die erste Einschränkung ist nicht inherent und Woos et. al. [34] schreiben, dass in Zukunft weitere Programmiermodelle unterstützt werden können. Der Einrichtungsaufwand dagegen ist tiefer in der Architektur des Tools verankert und ist zur Umsetzung der vollen Funktionalität notwendig.

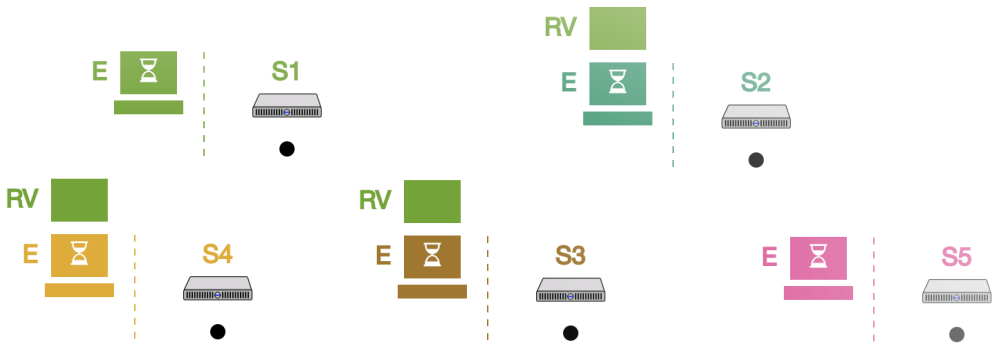


Abbildung 3.2: Visualisierung eines Systemzustands in Oddity [34]

Neben Tools wie ShiViz und Oddity, welche mindestens in ihren jeweiligen akademischen Umfeldern erfolgreich eingesetzt werden [35, 36], gibt es im Bereich der Visualisierung von verteilten Systemen auch Forschung zum Einsatz von Augmented Reality. Mit debugAR stellen Reipschläger et. al. [37] einen Prototypen vor, der demonstriert, wie Augmented Reality verwendet werden könnte, um die vielen Dimensionen verteilter Systeme zu visualisieren. Abbildung 3.3 zeigt ein Rendering von debugAR. Dargestellt ist ein interaktives Zeit-Raum-Diagramm ähnlich der Oberfläche von ShiViz.

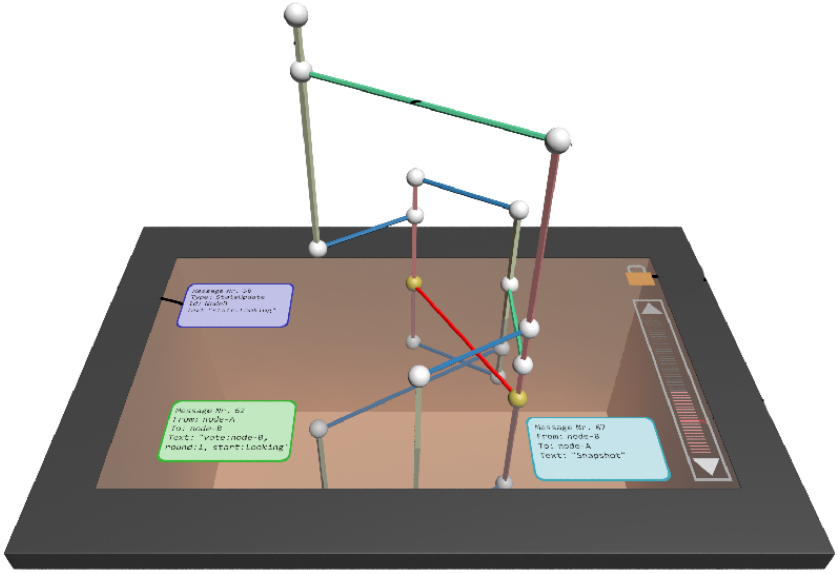


Abbildung 3.3: Rendering einer 3D Visualisierung von Nachrichtenflüssen in debugAR [37]

3.5 Verwendung von Proxies

Proxies finden in verteilten Systemen unter anderem Verwendung, um die Verfügbarkeit von Systemen zu verbessern [4], aber auch zum Testen und beim Debugging können Proxies eingesetzt werden. Der bei Shopify entwickelte Toxiproxy[3] ist ein für das Failure-Testing von verteilten Systemen entwickelter TCP-Proxy, welcher Netzwerk-Unterbrechungen und andere Fehler des Netzwerks simulieren kann.

In anderen Feldern werden Proxies außerdem erfolgreich an verschiedenen Stellen eingesetzt, um Replays von Netzwerk-Verkehr abzuspielen. Beispielsweise werden proxy-basierte Replays für reproduzierbare Performance-Tests von nicht-verteilten Systemen verwendet [38]. Das Web-Security-Testing-Tool Burp Suite[4] verwendet ebenfalls einen Proxy, um Requests abzufangen, aufzuzeichnen und gegen ein Angriffsziel wiedergeben zu können.

[3] `https://github.com/Shopify/toxiproxy`
[4] `https://portswigger.net/burp`

Auch Debugger, die auf dieses Prinzip setzen, gibt es bereits. Mugshot [39] und ähnliche Tools [40] sind Replay-Debugger für Browser-Applikationen, die einen Proxy einsetzen, um Requests aufzuzeichnen und in Replays deterministisch wiederzugeben. Speziell für das Debugging verteilter Systeme wurde ein proxy-basierter Replay-Mechanismus hingegen noch nicht eingesetzt.

3.6 Offene Fragen und Herausforderungen

Fehlertoleranz und Debugging in verteilten Systemen sind aktuelle Forschungsfelder, die sich schnell weiterentwickeln und dementsprechend noch viele Möglichkeiten für Innovationen bieten. Die folgenden sind derzeit offene Probleme, die weiteren Entwicklungen bedürfen.

Alvaro und Tymon [23] stellen als großen aktuellen Mangel heraus, dass sowohl Failure-Testing als auch Chaos Engineering in ihrer Anwendung so komplex sind, dass die Techniken ausschließlich den Entwicklern zur Verfügung stehen, die sich voll darauf spezialisieren. Sie schlagen unter anderem eine weitgehende Automatisierung von Failure-Testing in der Zukunft vor. Laut Balalaie et. al. [21] fehlt der Industrie außerdem ein Failure-Testing-Framework, das es ermöglicht, effektiv Bugs aufzudecken und diese Bugs mit deterministischen Testfällen zu reproduzieren. Jepsen beispielsweise agiert ausschließlich zufallsbasiert und kann Bugs oft nicht zuverlässig reproduzieren. Es ist zudem wichtig, dass ein solches Framework cross-Plattform und programmiersprachen-agnostisch ist, um vielfältig einsetzbar zu sein. Heute verwenden einige große Projekte selbst entwickelte Test-Frameworks, die sich auf ein konkretes System beschränken und durch den getrennten Entwicklungsaufwand nicht mit dem Funktionsumfang von Tools wie Jepsen mithalten können [21].

Im Bereich des Debuggings von verteilten Systemen stellt sich unter anderem noch die Frage danach, wie unterschiedliche Aspekte von verteilten Systemen am besten visualisiert werden können [34]. Auch ist noch offen, wie interaktive Debugger mit sehr großen Systemen skalieren können, sowohl im Sinne der Visualisierung als auch der Performance [31].

Im Kontext ihres Prädikat-Checkers D3S stellen Liu et. al. [41] insbesondere Replay-Debugging als Teil ihrer Zukunftsvision heraus, um mit D3S identifizierte Bugs einfach zu debuggen. In der Praxis haben sich seitdem Failure-Testing und Chaos-Engineering gegenüber Prädikat-Checkern zur Identifikation von Bugs durchgesetzt. Trotzdem könnte sich ein programmiersprachen-agnostischer Replay-Debugger, welcher durch Failure-Testing gefundene Bugs deterministisch wiedergeben kann, als sehr hilfreich erweisen und ist eine Entwicklung, die es noch zu machen gilt. In Anbetracht dessen, dass Proxies sich in anderen Bereichen als effektives Werkzeug für Replays von Netzwerk-Verkehr erwiesen haben (siehe Kapitel 3.5), scheint es möglich, dass ein proxy-basierter Replay-Mechanismus der Schlüssel für die Entwicklung eines solchen Tools sein könnte.

Diese Arbeit stellt die Hypothese auf, dass ein proxy-basierter Replay-Mechanismus ein effektives Tool beim Debugging von Partitionstoleranz-Bugs in verteilten Systemen ist. Ziel ist es, einen Prototypen zu entwickeln, anhand dessen diese Hypothese evaluiert werden kann. Insbesondere soll empirisch untersucht werden, ob durch Partitionstoleranz-Bugs mit Hilfe eines Proxys in einer Testumgebung deterministisch reproduziert werden können.

Kapitel 4

Ein Proxy zum Debuggen von Netzwerk-Unterbrechungen

Die grundlegende Idee dieser Arbeit ist, einen Proxy als Kernstück eines Replay-Debuggers für verteilte Systeme zu verwenden. Dieses Kapitel beschreibt die Anforderungen und die Umsetzung des im Rahmen dieser Arbeit entwickelten Prototypen ditm[1] (Debugger in the middle), anhand dessen das Konzept evaluiert wird.

Grundsätzlich ähnelt ditm dem in Kapitel 3.5 beschriebenen Toxiproxy, der als Failure-Testing-Tool konzipiert ist, in der Hinsicht, dass auch ditm die Möglichkeit bietet, mittels eines Proxys Netzwerk-Unterbrechungen zu erzeugen. Im Gegensatz zu Failure-Testing-Tools wie dem Toxiproxy geschieht das mit ditm allerdings, während der Nutzer mit dem System interagiert, anstatt in programmatisch vordefinierten Testfällen. Ditm ähnelt in der Hinsicht mehr einem Debugger, als einem Test-Framwork. Um gefundene Bugs dennoch zuverlässig und mit geringem Aufwand reproduzieren zu können, soll ditm Situationen aufzeichnen und zum späteren Zeitpunkt wieder abspielen können, insbesondere mit den gleichen Netzwerk-Bedingungen wie während der Aufzeichnung.

[1] https://github.com/JonathanArns/ditm

4.1 Anforderungen

Ditm ist ein Prototyp, der primär dem Zweck dient, die in Kapitel 3.6 aufgestellte Hypothese zu untersuchen. Die Anforderungen reflektieren diesen Umstand, indem beispielsweise ausschließlich HTTP als Netzwerk-Protokoll unterstützt wird, um die Entwicklung einfach zu halten.

4.1.1 Funktionale Anforderungen

Tabelle 4.1 beschreibt die funktionalen Anforderungen an das System ditm.

Tabelle 4.1: Funktionale Anforderungen an ditm

ID	Anforderung	Beschreibung
FA1	Netzwerk-Unterbrechungen	Das System soll kontrolliert Netzwerk-Unterbrechungen zwischen Knoten des zu testenden Systems erzeugen können.
FA2	Aufzeichnung	Das System soll den gesamten Netzwerkverkehr des zu testenden Systems aufzeichnen können. Insbesondere sollen auch erzeugte Netzwerk-Unterbrechungen aufgezeichnet werden.
FA3	Replay	Das System soll anhand einer durch das System erstellten Aufzeichnung die Situation in der Aufzeichnung am laufenden Testsystem wiederherstellen und vor allem inklusive Netzwerk-Unterbrechungen wieder abspielen können. Dabei sollen immer genau die Teile des Netzwerkverkehrs geblockt werden, die auch in der Aufzeichnung vom System geblockt wurden. Nicht mehr, nicht weniger und keine anderen.

FA4	Zufällige Unterbrechungen	Das System soll Netzwerk-Unterbrechungen zufällig erzeugen können.
FA5	Kontrollierte Unterbrechungen	Das System soll dem Nutzer die Möglichkeit geben, Netzwerk-Unterbrechungen feingranular zu steuern.
FA6	Log-Aggregation	Das System soll zusätzlich zum Netzwerkverkehr auch die Log-Ausgaben des zu testenden Systems aufzeichnen und aggregieren können.
FA7	Log-Matching	Das System soll die aufgezeichneten Nachrichten und Logs in chronologischer Reihenfolge gemeinsam anzeigen können.
FA8	Volume-Snapshots	Das System soll für zustandsbehaftete Systeme Snapshots der Docker-Volumes erstellen und zu einem späteren Zeitpunkt wiederherstellen können, um deterministisch Replays für zustandbehaftete Systeme zu ermöglichen.
FA9	Nachrichten von außen	Das System soll dem Nutzer Schnittstelle bieten, über die Nachrichten reproduzierbar von außen in das zu testende System gesendet werden können, um Prozesse im zu testenden System anzustoßen.
FA10	Responses Blocken	Das System soll in der Lage sein, nicht nur Requests auf dem Hinweg zum Server zu blockieren, sondern optional auch erst die Antwort auf dem Rückweg.

4.1.2 Nicht-funktionale Anforderungen

Tabelle 4.2 beschreibt die nicht-funktionalen Anforderungen an das System ditm.

Tabelle 4.2: Nicht-funktionale Anforderungen an ditm

ID	Anforderung	Beschreibung
NFA1	Portabilität	Das System soll vollständig in Docker lauffähig und mittlels docker-compose konfigurierbar sein.
NFA2	Proxy Architektur	Das System soll auf einen proxy-basierten Replay-Mechanismus setzen, um die in Kapitel 3.6 beschriebene Hypothese untersuchen zu können.
NFA3	HTTP	Das System soll mit HTTP als Netzwerkprotokoll arbeiten und grundlegend alle verteilten Systeme, die ausschließlich über HTTP kommunizieren, unterstützen.
NFA4	Usability	Das System soll einfach über eine grafische Oberfläche bedienbar sein.
NFA5	Echtzeit	Requests und Logs einer laufenden Aufzeichnung sollen in Echtzeit angezeigt werden.
NFA6	Unverändertes SUT	Das SUT soll für die Verwendung des Systems nicht angepasst werden müssen.

4.2 Architektur

Das Herzstück des Systems ist entsprechend der Idee ein eigens entwickelter HTTP-Proxy, welcher den Netzwerk-Verkehr in einem verteilten System inklusive Metadaten aufzeichnet und Nachrichten wahlweise nicht weiter leiten kann, um Netzwerk-Unterbrechungen zu simulieren. Dieser Proxy läuft in ei-

nem Docker-Container direkt neben dem SUT und ist auch Teil des gleichen Docker-Netzwerks wie das SUT. Außer dem Proxy hat ditm eine weitere, selbst entwickelte Komponente, ein Log-Driver-Plugin für Docker, welches Container-Logs an den Proxy sendet. Die Datenflüsse zwischen Komponenten des Systems sind in Abbildung 4.1 im Kontext eines SUT mit zwei Knoten dargestellt.

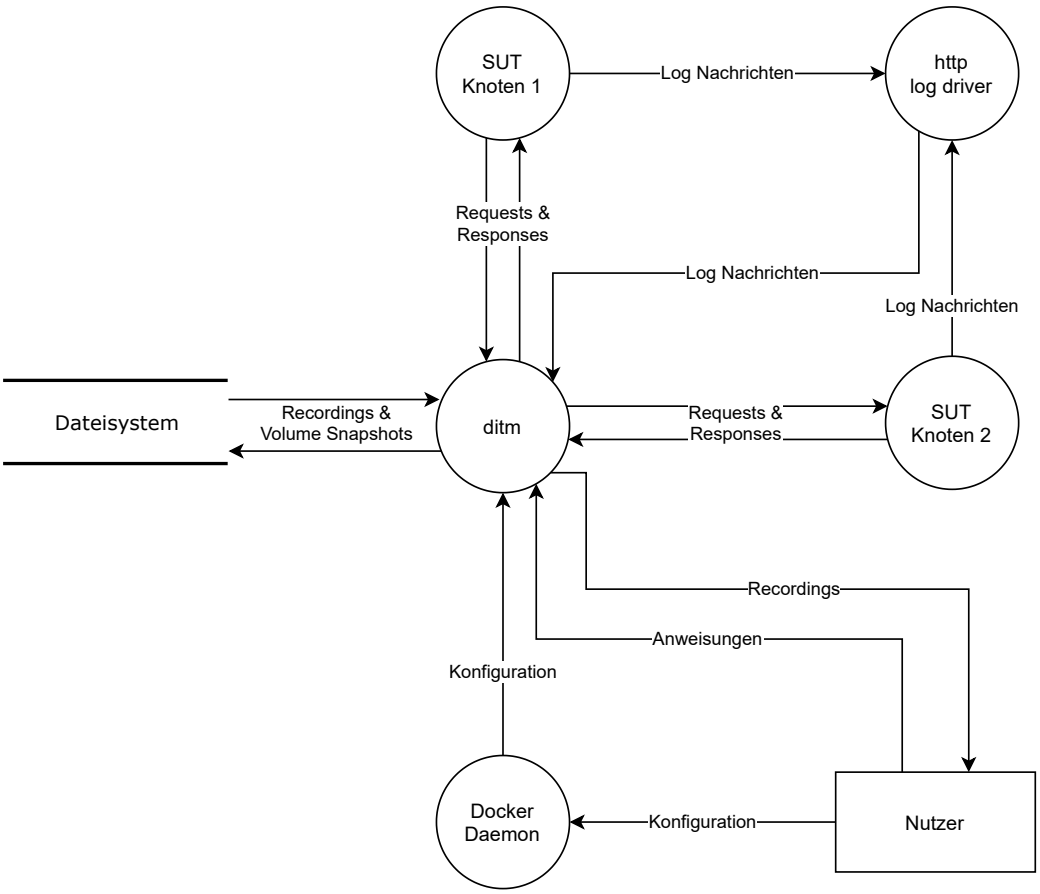


Abbildung 4.1: Datenflussdiagramm für ditm

4.2.1 Datenmodell

Das Datenmodell von ditm ist einfach gehalten. Das Top-Level-Datenobjekt ist das Recording, dieses enthält drei chronologisch geordnete Listen, eine mit den Log-Nachrichten des SUT, eine mit den aufgezeichneten Requests

und eine mit BlockConfigs, die die konfigurierten Netzwerk-Unterbrechungen enthalten. Alle Listen enthalten die jeweiligen Objekte für alle Knoten des SUT, ohne diese weiter voneinander zu trennen. Sollte eine gefilterte Liste benötigt werden, die nur Objekte für bestimmte Knoten enthält, wird diese dynamisch erzeugt. Recordings enthalten außerdem eine Referenz auf einen Volume-Snapshot, sofern für das Recording einer existiert.

Die eigentlichen Request-Datenobjekte enthalten neben dem Request-Body eine Menge Metadaten, die teilweise vom Request selbst stammen und teilweise durch ditm erzeugt wurden, beispielsweise ob der Request geblockt wurde oder nicht. Log-Nachrichten umfassen neben der eigentlichen Nachricht ebenfalls wichtige Metadaten, wie den Namen des Containers, der die Nachricht geloggt hat. In Abbildung 4.2 ist das beschriebene Datenmodell in einem Klassendiagramm dargestellt. Die Implementierungen des Matcher-Interfaces fehlen im Klassendiagramm, da sich hier keine Unterschiede zwischen den einzelnen Implementierungen zeigen würden und diese gesondert in Kapitel 4.3.4 behandelt werden. Die Methoden der Klasse Proxy wurden ebenfalls aus Platzgründen weggelassen und da diese hier konkret von geringerem Interesse sind. Ein Proxy nimmt im Prototypen die Rolle eines Gottobjekts ein, dieses Design-Pattern wurde für den Prototypen solideren Patterns vorgezogen, um die Entwicklung einfach zu halten und Erweiterungen, bzw. Änderungen des Datenmodells währenddessen zu erleichtern.

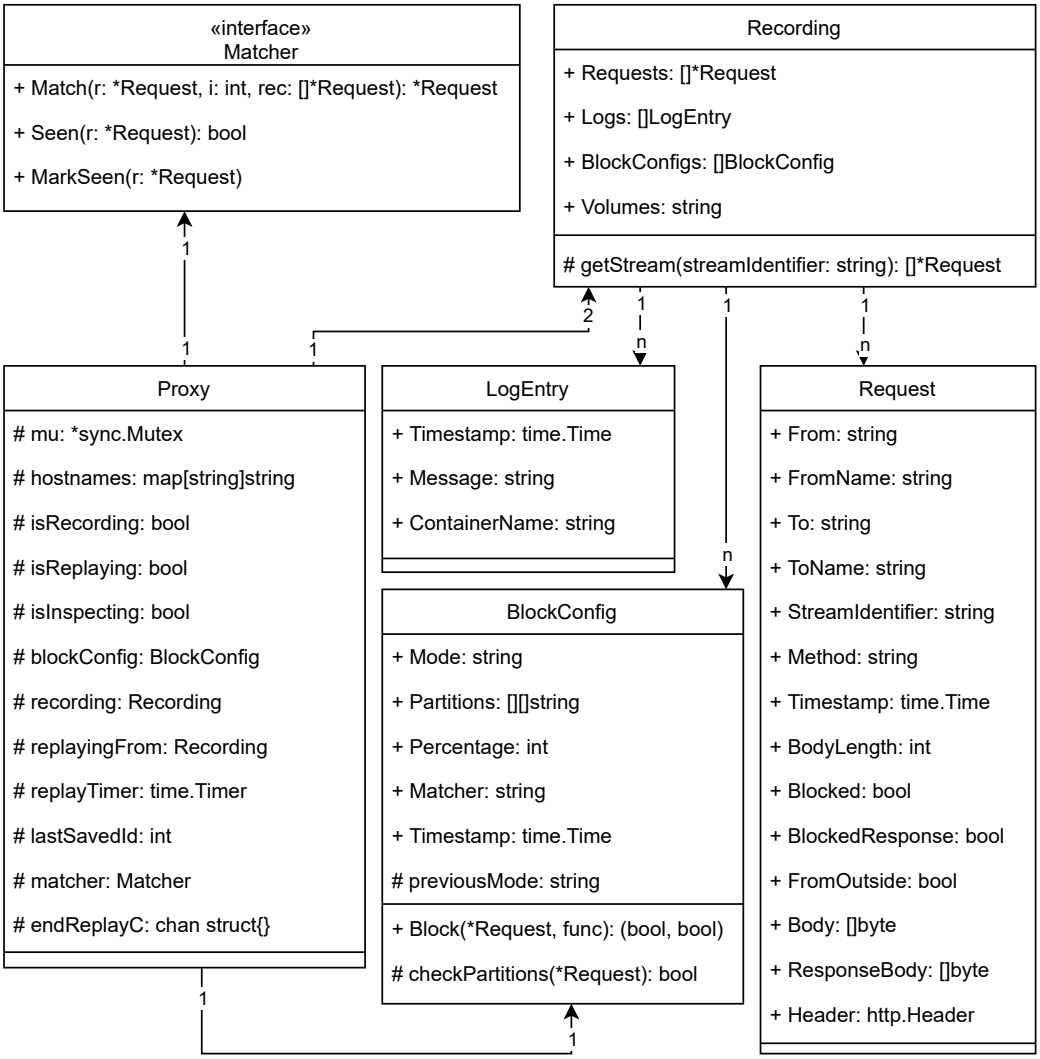


Abbildung 4.2: Klassendiagramm für ditm ohne Proxy-Methoden und Matcher-Implementierungen

4.2.2 Weg eines Requests

Um eine konkrete Vorstellung von der Arbeitsweise des Proxys zu vermitteln, wird im folgenden beispielhaft der Weg eines Requests durch das System beschrieben. In diesem Kontext wird der Ursprung des Requests als Client und das Ziel des Requests als Server bezeichnet, beide sind Knoten des SUT. Abbildung 4.3 stellt zunächst den groben Ablauf auf Systemebene

dar. Wichtig ist zu bemerken, dass der Proxy bereits beim ersten Erhalten des Requests die komplette Analyse durchführt und nicht nur entscheidet, ob der Request geblockt werden soll, sondern die Entscheidung auch direkt für die Response trifft, ohne die Response jemals gesehen zu haben. Tatsächlich wird die Response gar nicht von ditm betrachtet, stattdessen wird die Analyse vollständig am Request durchgeführt. Die einzige Interaktion des Proxys mit der Response ist, diese zur Anzeige im Frontend aufzuzeichnen, sie zu blocken falls vorher festgelegt oder sie ansonsten direkt weiter zu leiten. Der Grund dafür ist, dass fast alle für ditm relevanten Metadaten bereits aus dem Request abgeleitet werden können. Zwar könnten die Länge der Response und die Response-Header möglicherweise ebenfalls von Interesse sein, der Kompromiss wird hier aber in Kauf genommen, da das die Implementierung erheblich vereinfacht und auch so genügend Metadaten vorliegen, um verschiedene Algorithmen zur Entscheidung, ob ein Request im Replay geblockt werden soll, zu evaluieren. Die verschiedenen Ansätze, die diese Arbeit betrachtet, werden in Kapitel 4.3.4 behandelt.

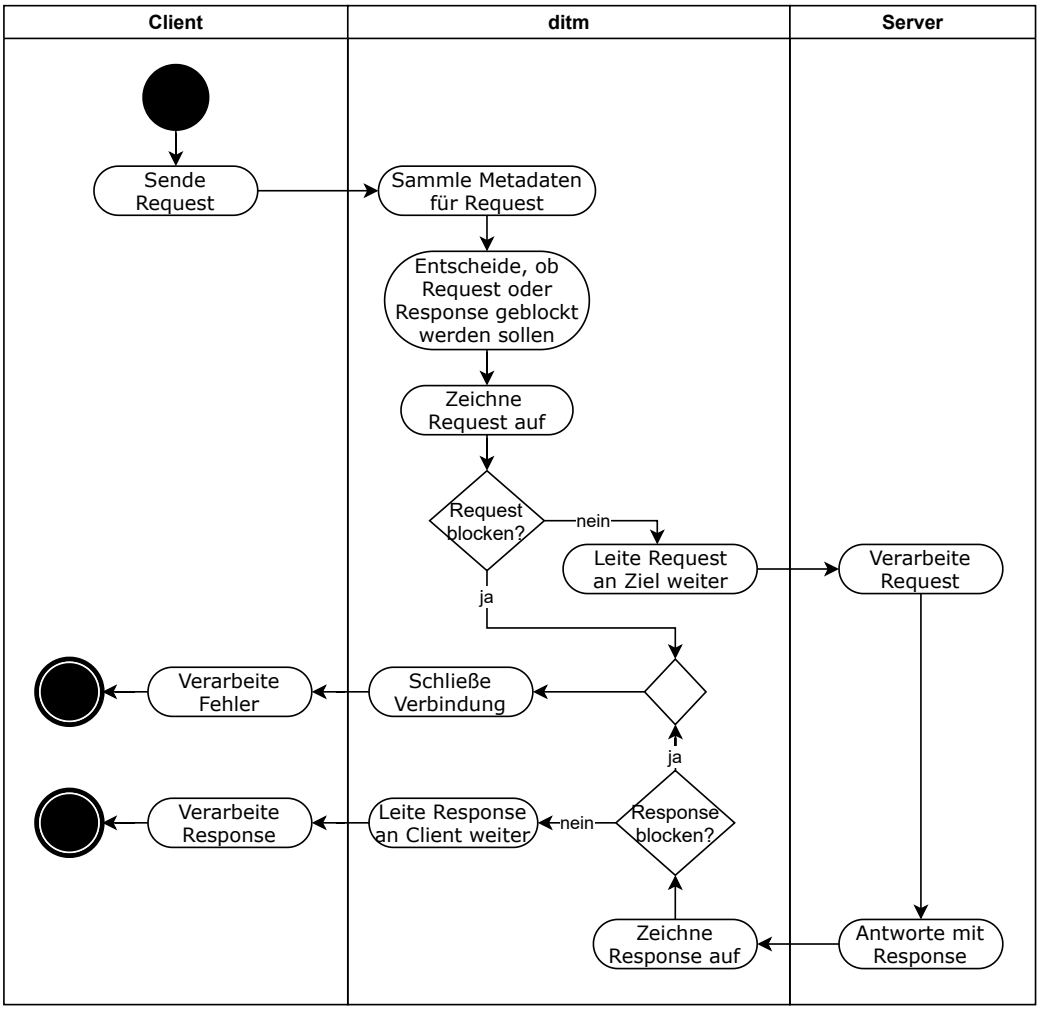


Abbildung 4.3: Aktivitätsdiagramm der Interaktion von ditm mit Client und Server eines Requests

4.3 Implementierung

Als Programmiersprache wird für die Implementierung auf Go gesetzt. Go bietet sich an, da die Standardbibliothek umfassende, robuste APIs für den Umgang mit Netzwerken bietet und die Sprache insgesamt eine schnelle Entwicklungsgeschwindigkeit erlaubt, ohne die Performance zu vernachlässigen.

4.3.1 Datenspeicherung

Als Speicherformat für ditm wurden Dateien im JSON-Format gewählt. So wird jedes Recording-Datenobjekt in einer eigenen Datei serialisiert gespeichert. Dieses Format wurde einer Datenbank aus mehreren Gründen vorgezogen. Zum einen wird Dateispeicher ohnehin für die Volume-Snapshots benötigt, die als Zip-Dateien gespeichert werden, es spart also an Komplexität für den Prototypen, für die restlichen Daten ein ähnliches Speichermodel zu wählen. Zum anderen bieten Dateien den Vorteil, dass der Nutzer die Möglichkeit hat, diese selbst einzeln zu verwalten. Es ist dem Nutzer also möglich, die Dateien einer einzelnen Aufzeichnung und des zugehörigen Snapshots getrennt zu sichern oder einfach mit anderen Nutzern über einen generischen Filesharing-Dienst zu teilen.

4.3.2 Log-Driver

Um die Log-Ausgaben des SUT in ditm aggregieren zu können, muss ditm aus seinem Container heraus auf die Logs der anderen Container zugreifen. Leider bietet Docker von sich aus keine einfache Möglichkeit dies zu konfigurieren, stattdessen muss auf Dockers Plugin-API zurück-gegriffen werden. Immerhin bietet Docker eine spezielle Plugin-API für Log-Driver, die es ermöglicht, Log-Nachrichten durch einen eigenen Driver beliebig weiter zu verarbeiten. Ditm verwendet ein einfaches, eigenes Log-Driver-Plugin, welches Nachrichten als HTTP Request an eine beliebige URL sendet. Ditm selbst bietet in seiner API einen Endpunkt an, welcher diese Log-Nachrichten empfangen und verarbeiten kann.

4.3.3 Frontend

Als Frontend bietet ditm dem Nutzer eine Web-Oberfläche, in der laufende sowie vergangene Aufzeichnungen tabellarisch visualisiert werden können. Des Weiteren ist der Proxy vollständig über die GUI kontrollierbar. Events einer laufenden Aufzeichnung werden in Echtzeicht per Server-Sent-Events

(SSE) an das Frontend gesendet und dort dargestellt. Das Frontend selbst ist eine einfache, serverseitig gerenderte HTML-Seite mit dem notwendigen JavaScript-Code, um SSEs zu empfangen und zu verarbeiten. Die Seite ist mittels Bootstrap gestyled und in Tabs aufgeteilt.

4.3.4 Request Matcher

Die Logik zur Entscheidung, ob ein Request, bzw. eine Response, während eines Replays geblockt werden soll, ist ein Kernstück des Proxys, welches es ermöglicht, Aufzeichnungen zuverlässig wiederzugeben. Das Problem lässt sich darauf herunterbrechen, genau den Request im ursprünglichen Recording zu identifizieren, dem der aktuelle Request während des Replays entspricht. Das liegt daran, dass sich der Proxy während eines Replays immer exakt wie während der Aufnahme verhalten soll.

Der erste Schritt im Prozess, Requests aus Recording und Replay zu matchen, ist immer die Requests sowohl im Recording als auch im Replay nach dem StreamIdentifier des aktuellen Requests zu filtern, sodass zwei kleinere Request-Streams entstehen. Der StreamIdentifier wird zusammengesetzt aus den Namen des Clients und des Servers für den jeweiligen Request, ein so gefilterter Stream stellt also den gesamten Verkehr zwischen genau zwei Knoten des SUT dar. Der Rest des Matchings ist über das in Listing 4.1 dargestellte Matcher-Interface abstrahiert, um leicht mehrere Implementierungen austauschen zu können.

Listing 4.1: Das Matcher-Interface in ditm

```
1 type Matcher interface {
2     Match(r *Request, i int, rec []*Request) *Request
3     Seen(r *Request) bool
4     MarkSeen(r *Request)
5 }
```

Die Methode Match erwartet als Parameter den aktuellen Request, dessen Index im gefilterten Stream des Replays und den gesamten gefilterten Stream des Recordings und gibt einen Pointer auf den Request im Recording zurück, der das beste Match darstellt. Die Methoden Seen und MarkSeen dienen

dazu, Requests zu markieren, falls sie bereits im aktuellen Replay vorgekommen sind, bzw. abzufragen, ob ein Request bereits markiert wurde. Es wird also nach jedem Aufruf von Match der zurückgegebene Request markiert. Eine Methode um Requests zu entmarkieren wird nicht benötigt, da für jedes Replay ein neuer Matcher initialisiert wird. Insgesamt wurden vier Implementierungen für das Interface mit verschiedenen Algorithmen für Match erstellt, die im Folgenden dargestellt und später gegeneinander evaluiert werden. Alle vier Implementierungen verwenden eine Hashtabelle zur Umsetzung der Seen und MarkSeen Funktionalität und ignorieren bei jedem Aufruf von Match bereits markierte Requests, sowie Requests, die von außerhalb des SUT stammen.

Zählend

Mit Abstand der einfachste Request-Matcher ist der zählende Matcher. Dieser bestimmt Matches allein aufgrund der Position von Requests in den jeweiligen Streams. Es wird also immer der Request als Match bestimmt, welcher im Recording den Index i hat, wobei i der zweite Parameter der Match-Methode ist. Ist der entsprechende Request bereits markiert oder von außerhalb des SUT, wird kein Match zurück gegeben, was bei allen Matchern dazu führt, dass ditm den Request nicht manipuliert. Der zählende Matcher ist nicht als ernsthafter Versuch gemeint, einen guten Matcher zu entwickeln. Vielmehr soll er als Vergleichsgrundlage für die anderen Matcher dienen.

Exakt

Alle weiteren Matcher, der exakte eingeschlossen, verwenden ein Punktesystem, in dem alle Requests des Recordings mit dem Request, der gematcht werden soll, verglichen werden und dabei für verschiedene Eigenschaften Punkte zugewiesen bekommen. Am Ende wird der Request mit der höchsten Punktzahl als Match zurück gegeben. Der Exakte Matcher vergibt einen Punkt für jede der Folgenden Bedingungen, die erfüllt ist.

- Die Requests stehen an gleicher Stelle in ihren jeweiligen Streams.

- Die URIs der Requests sind gleich.
- Die URIs der Requests sind gleich lang.
- Die Bodies der Requests sind gleich.
- Die Bodies der Requests sind gleich lang.

Sei i der Index im Replay, j der Index im Recording, req der aktuelle Request im Replay und r der aktuelle Request im Recording, dann wird der Score für r also wie in Listing 4.2 berechnet.

Listing 4.2: Scoring Code für den Exakten Matcher

```
score := 0.0
if i == j {
    score += 1
}
if req.To == r.To {
    score += 1
}
if len(req.To) == len(r.To) {
    score += 1
}
if string(req.Body) == string(r.Body) {
    score += 1
}
if len(req.Body) == len(r.Body) {
    score += 1
}
```

Dieser Matcher sollte sich in Situationen mit gemischten Request Reihenfolgen bereits deutlich besser verhalten als der einfache zählende Matcher, da er neben der Reihenfolge auch andere Eigenschaften der Requests betrachtet. Trotzdem ist es auch hier vorstellbar, dass der Matcher deutliche Probleme haben wird, sobald er auf eine Menge ähnlicher Requests trifft, die nur leicht gemischt sind.

Mix

Um dieses Problem noch weiter anzugehen, betrachtet der Mix-Matcher nicht nur, ob die Position der Requests absolut gleich ist, sondern den relativen Abstand der Requests zueinander, und bewertet Requests mit geringerem

Abstand besser. Alle anderen Eigenschaften werden weiterhin absolut miteinander verglichen. Um die Gewichtung zwischen dem relativen Abstand und den konstanten Komponenten des Scores anzupassen, werden diese wie in Listing 4.3 mit der Länge des Recordings als Faktor multipliziert. Das führt dazu, dass Gleichheit in anderen Eigenschaften höher gewichtet wird, als die Position eines Requests und diese nur dann mit ins Gewicht fällt, wenn es mehrere Requests im Recording gibt, die ansonsten gleich gut passen.

Listing 4.3: Scoring Code für den Mix-Matcher

```
1  faktor := float64(len(rec)) // Laenge des Recordings
2  score := 0.0
3  score -= math.Abs(float64(j - i)) // Abstand der Requests
4  if req.To == r.To {
5      score += 1 * faktor
6  }
7  if len(req.To) == len(r.To) {
8      score += 1 * faktor
9  }
10 if string(req.Body) == string(r.Body) {
11     score += 1 * faktor
12 }
13 if len(req.Body) == len(r.Body) {
14     score += 1 * faktor
15 }
```

Heuristisch

Der heuristische Matcher unterscheidet sich vom Mix-Matcher nur an zwei Stellen. Anstatt die Längen der URIs und Bodies absolut miteinander zu vergleichen, wird hier wie bei der Position die Differenz zwischen den beiden genommen und eine kleinere Differenz besser bewertet. Die Werte werden weiterhin mit der Länge des Recordings als Faktor multipliziert. Um die Tatsache auszugleichen, dass die Differenz selbst größere Werte annehmen kann als die konstanten Werte, wird sie allerdings wie in Listing 4.4 zusätzlich durch 10 dividiert.

Listing 4.4: Scoring Code für den Heutristischen Matcher

```
1  faktor := float64(len(rec))
2  score := 0.0
3  score -= math.Abs(float64(j - i))
4  if req.To == r.To {
```

```
5       score += 1 * faktor
6   }
7   score -= math.Abs(
8       float64(len(req.To)-len(r.To))
9   ) * faktor / 10
10  if string(req.Body) == string(r.Body) {
11      score += 1 * faktor
12  }
13  score -= math.Abs(
14      float64(len(req.Body)-len(r.Body))
15  ) * faktor / 10
```

Obwohl der heuristische Matcher komplexer ist als der Mix-Matcher und insgesamt feinere Unterschiede zwischen Requests berücksichtigen kann, ist nicht von vornherein zu sagen, ob das von Vorteil ist. Unter anderem wird auch diese Frage in der Evaluation von ditm empirisch angegangen. Es ist durchaus denkbar, dass es auch von der jeweiligen Situation abhängt, welcher Matcher sich besser verhält.

Zeitbasiert

Der zeitbasierte Matcher funktioniert grundlegend anders als die anderen Matcher und passt nicht direkt in das Design des Matcher-Interfaces. Er bricht insbesondere mit der Idee, Requests einander genau zuzuordnen, um zu entscheiden, welche Requests geblockt werden.

Der zeitbasierte Matcher wurde für Situationen entwickelt, in denen selbst bei genau gleichen äußeren Bedingungen das SUT in Aufzeichnung und Replay unterschiedliche Requests sendet. Die Raft-Implementierung, welche in Kapitel 5.3.1 beschrieben wird, ist ein Beispiel für ein System, welches inherent Zufallskomponenten verwendet um zu bestimmen, welche Requests wann gesendet werden. Die Idee hier ist, Requests nicht ein genaues Match zuzuordnen, sondern stattdessen die BlockConfig zu ermitteln, mit welcher der Request aufgrund seiner zeitlichen Position im Replay behandelt werden sollte. Sofern ausschließlich Konfigurationen ohne Zufallskomponente verwendet wurden, kann dies ein effektiver Weg sein, Replays zu erzeugen, die zwar vielleicht nicht eins zu eins der Aufzeichnung entsprechen, aber trotzdem Bugs zuverlässig reproduzieren.

Zur Umsetzung werden alle Änderungen an der BlockConfig mit Zeitstempel im Recording gespeichert. Anstelle eines gematchten Requests gibt der Matcher einen neuen Dummy Request zurück, der lediglich die Information enthält, wie der aktuelle Request behandelt werden soll. Dieser Workaround ist notwendig, um in das Matcher-Interface zu passen. Auch die Funktionalität des Matchers, Requests zu identifizieren, die bereits im Replay aufgetaucht sind, wird hier über die im Replay vergangene Zeit implementiert, sodass Requests als bereits gesehen gelten, sobald der Zeitpunkt überschritten ist, an dem sie in der Aufzeichnung vorkamen. Einzige Ausnahme hierbei sind die Requests von außerhalb des SUT. Diese werden wie bei den anderen Matchern erst markiert, wenn sie tatsächlich gesendet wurden, da es wichtig ist, diese während des Replays vollständig wiederzugeben.

4.4 Nutzung

Im folgenden wird kurz die für ditm notwendige Konfiguration und die Bedienungsoberfläche von ditm dargestellt.

4.4.1 Konfiguration

Ditm wird am einfachsten über docker-compose konfiguriert, dort werden Umgebungsvariablen und Docker-Volumes für das SUT und für ditm selbst gesetzt. Außerdem muss für den vollen Funktionsumfang der HTTP-Log-Driver ebenfalls in docker-compose konfiguriert werden. Ohne diesen fehlt die Aggregation der Log-Nachrichten. Vor der ersten Verwendung muss der Driver außerdem installiert werden. Dies ist möglich, indem das Repository `https://github.com/JonathanArns/http-log-driver` gecloned und darin der Befehl **make all** ausgeführt wird. Zu beachten ist, dass in docker-compose nur alle zum Starten nötigen Einstellungen geschehen, alles andere ist Teil der Benutzeroberfäche. Eine beispielhafte Konfiguration ist in Listing 4.5 zu sehen.

Listing 4.5: Beispiel für ditms Konfiguration mit docker-compose

```
version: "3.9"

services:
  ditm:
    image: ditm
    hostname: ditm
    environment:
      # Liste der SUT Hostnames
      CONTAINER_HOST_NAMES: target,target2
    volumes:
      # enthaelt alle Volumes der SUT-Container
      - ./volumes:/volumes
      - ./recordings:/recordings
      - ./snapshots:/snapshots
    ports:
      - "8000:80"    # Browser UI
      - "5000:5000"  # Proxy
    depends_on:
      - target
      - target2

  target:
    image: <image>
    hostname: target
    environment:
      - HTTP_PROXY=ditm:5000
    volumes:
      - ./volumes/target:/volume
    logging:
      driver: jonathanarns/http-log-driver
      options:
        endpoint: http://localhost:8000/log

  target2:
    image: <image>
    hostname: target2
    environment:
      - HTTP_PROXY=ditm:5000
    volumes:
      - ./volumes/target2:/volume
    logging:
      driver: jonathanarns/http-log-driver
      options:
        endpoint: http://localhost:8000/log
```

4.4.2 Bedienung

Abbildung 4.4 zeigt die Hauptansicht von ditms Web-Frontend, in der die Nachrichten und Log-Ausgaben des SUT gemeinsam tabellarisch dargestellt werden. Requests haben dabei jeweils einen farbigen Hintergrund, der in rot oder grün anzeigt, ob ein Request durch den Proxy geblockt wurde oder nicht. Blaue Requests sind Client-Requests von außerhalb des SUT und werden nie geblockt. Ein Pfeil in der letzten Spalte der Tabelle zeigt außerdem, ob ein Request auf dem Hinweg, oder nur die Response auf dem Rückweg geblockt wurde. Requests und Log-Ausgaben, die vom selben Knoten stammen, sind außerdem in der Spalte ‘From‘ farblich gleich gekennzeichnet.

Neben der Hauptansicht, in der Volume-Snapshots erstellt und Recordings gesteuert werden können, bietet die GUI außerdem Ansichten, in denen existierende Recordings oder existierende Volume-Snapshots aufgelistet werden. Die Recordings können dort zur Anzeige geladen oder zum Start eines Replays verwendet werden. Die Volume-Snapshots können ebenfalls aus der Liste geladen werden. Im Partitions-Menu der GUI kann eingestellt werden, welche Requests der Proxy blocken soll, sowohl zufallsbasiert, als auch anhand frei definierbarer Unterteilungen des Netzwerks.

Home
Recordings
Volumes
Partitions
Start Recording
Save Volumes
Back to live
Timestamp
From
To
Body
Response-Body
Flow
10:01:06.759709 outside GET http://target-b/api/account/1 could not get a <-->
10:01:06.761276 ab-target-b 2021/10/09 10:01:06 /api/account/1
10:01:06.761591 target-b GET http://target-a:8080/api/account/1 -\->
10:01:06.762015 ab-target-b 2021/10/09 10:01:06 [handler:GetAccount] Get "http://target-a:8080/api/account/1
10:01:07.810119 outside GET http://target-b/api/account/1 {"account":1,"ac <-->
10:01:07.810424 ab-target-b 2021/10/09 10:01:07 /api/account/1
10:01:07.810970 target-b GET http://target-a:8080/api/account/1 {"status":200," <-->
10:01:07.811611 ab-target-a 2021/10/09 10:01:07 /api/account/1
10:01:07.811956 ab-target-a map[1:0xc000096360]
10:01:07.812188 ab-target-a 2021/10/09 10:01:07 [db:Acquire:172.25.0.4:32820] waiting for lock for ID:1
10:01:07.812420 ab-target-a 2021/10/09 10:01:07 [db:Acquire:172.25.0.4:32820] locking for ID:1
10:01:07.812635 ab-target-a 2021/10/09 10:01:07 [db:Release:172.25.0.4:32820] unlocking for ID:1
10:01:08.881043 outside GET http://target-b/api/account/1 could not get a <-->
10:01:08.881321 target-b GET http://target-a:8080/api/account/1 {"status":200," <-\-
10:01:08.881360 ab-target-b 2021/10/09 10:01:08 /api/account/1
10:01:08.881565 ab-target-a 2021/10/09 10:01:08 /api/account/1
10:01:08.882049 ab-target-a 2021/10/09 10:01:08 [db:Acquire:172.25.0.4:32820] waiting for lock for ID:1
10:01:08.882330 ab-target-a 2021/10/09 10:01:08 [db:Acquire:172.25.0.4:32820] locking for ID:1
10:01:08.882416 target-b GET http://target-a:8080/api/account/1 -\->">

Abbildung 4.4: Screenshot der Hauptansicht in ditms Frontend

Kapitel 5

Versuchsanordnungen

Dieses Kapitel beschreibt die Versuche, mit denen die in Kapitel 3.6 aufgestellte Hypothese evaluiert wird. Konkret sollen die folgenden Fragen über den zuvor beschriebenen Prototypen ditm beantwortet werden.

1. Was sind die Grenzen von ditms Request-Matching?
2. Wie gut funktioniert ditms Log-Matching, bzw. die chronologische Ordnung von Log-Nachrichten relativ zu Requests?
3. Lassen sich Partitionstoleranz-Bugs deterministisch mit ditms Replay-Mechanismen reproduzieren?

5.1 Request-Matching

Ziel dieses Versuchs ist es, die erste Frage zu beantworten. Der zeitbasierte Matcher wird in diesem Versuch nicht evaluiert, da er aufgrund seiner Funktionsweise schon per Design für die zufallsbasierten Netzwerk-Unterbrechungen, die für diese Experimente verwendet werden, ungeeignet ist. Stattdessen wird er nur mit den Versuchen in 5.3 evaluiert.

Das Test System ist ein einfacher HTTP-Server mit zwei Endpunkten. Einer davon löst eine große Menge an Requests aus, der andere verarbeitet

diese. Es wird für den Versuch ein Cluster aus zwei identischen Knoten verwendet, wovon einer die Rolle eines API Gateways einnimmt, über das von außen der Versuch ausgelöst wird, und der andere die eines dahinter verborgenen Services. Da ditm den Nachrichtenfluss zwischen zwei Knoten des SUT immer einzeln betrachtet, genügen zwei Knoten, um die Grenzen des Request-Matchings abzustecken.

Die Requests, die der Gateway-Knoten versendet, sind anhand eines Query-Parameters eindeutig identifizierbar. Das System versendet diese in einer Schleife und bietet neben der Anzahl zu versendender Requests verschiedene andere Konfigurationsmöglichkeiten, die den Strom von Requests beeinflussen. Die Requests können in vollständig fester oder in unterschiedlich stark gemischter Reihenfolge erfolgen. Wie stark eine Request-Reihenfolge gemischt ist, wird dabei darüber definiert, um wie viele Stellen ein Request beim Mischen maximal von seiner ursprünglichen Position verschoben werden kann. Sei dies der Mischungsgrad, dann entspricht ein Mischungsgrad von 0 immer einem vollständig geordneten Nachrichten-Fluss. Für jeden anderen Wert wird der Fluss vor jeder Ausführung neu zufällig gemischt, sodass ditm die Requests während eines Replays also in anderer Reihenfolge erhält, als während der Aufzeichnung. Die aufgeführten Reihenfolgen stellen beispielhaft dar, wie ein Fluss von zehn Requests bei unterschiedlichen Mischungsgraden angeordnet werden könnte.

```
(0): 0 1 2 3 4 5 6 7 8 9
(1): 1 0 2 4 3 6 5 7 9 8
(2): 0 3 2 1 5 6 4 9 7 8
(3): 4 2 1 0 7 3 9 5 6 8
```

Requests können außerdem optional asynchron erfolgen, was ebenfalls dazu führt, dass sie in stark gemischter Reihenfolge und nahezu gleichzeitig gesendet werden. Zusätzlich besteht die Möglichkeit, entweder einfache GET-Requests zu senden, dieselben GET-Requests mit einem Zeitstempel als zusätzlichem Parameter, oder POST-Requests mit einem Zeitstempel und einem zusätzlichen Füller im Request-Body. Der Füller hat eine von drei festen Längen, somit dient die Länge des Request-Bodys als weiteres Merkmal

für ditm, um Requests identifizierbar zu machen. Die Zeitstempel dienen genau umgekehrt dazu, Requests schwieriger identifizierbar zu machen, indem ditm durch die Zeitstempel die Query-Parameter, bzw. die Bodies der Requests nicht mehr einfach direkt miteinander vergleichen kann, um Requests zu identifizieren.

Zuletzt gibt es die Möglichkeit, für den verwendeten HTTP-Client (net/http in Go) die Wiederverwendung von TCP Verbingungen mit Keep-Alive zu deaktivieren. Dies ist notwendig, da der Client sonst in bestimmten Situationen Retries sendet, was durch diese Einstellung verhindert werden kann.

Für die Evaluation werden einfache GET-Requests, GET-Requests mit Zeitstempel, GET-Requests ohne Keep-Alive, GET-Requests mit Zeitstempel und ohne Keep-Alive und POST-Requests verwendet. Für jede Art von Request werden Experimente mit fünf unterschiedlichen Konfigurationen zur Reihenfolge der Requests durchgeführt. Diese sind asynchron, synchron mit fester Reihenfolge und synchron mit gemischter Reihenfolge mit Mischungsgraden von 1, 2 und 3. Zusätzlich wird jedes Experiment einmal mit zehn Requests und einmal mit 100 Requests durchgeführt.

In jedem Experiment werden zehn voneinander unabhängige Aufzeichnungen erstellt. Von jeder Aufzeichnung wird dann mit jedem der vier Request-Matcher ein Replay durchgeführt. Für jedes Replay wird anhand von dazu bestimmten Log-Ausgaben des SUT bestimmt, wie viele Requests ditm falsch behandelt hat. Aus diesem Wert wird dann pro Request-Matcher über die zehn erzeugten Replays der Durchschnitt und der Median der Anzahl der falsch behandelten Requests berechnet. Diese Werte bilden das Ergebnis des jeweiligen Experiments. Alle Experimente zum Request-Matching werden vollständig automatisiert durchgeführt und ausgewertet.

5.2 Log-Matching

Für die Evaluation des Log-Matchings wird das gleiche SUT verwendet wie für die Experimente zum Request-Matching. Da die chronologische Darstellung von Logs zwischen den Requests leider nicht so einfach automatisch aus-

wertbar ist, werden hierfür zusätzliche Experimente von Hand durchgeführt und ausgewertet. Es werden insgesamt vier Experimente durchgeführt, bei denen eine Aufzeichnung mit jeweils 10 und 100 Requests einmal synchron und einmal asynchron erstellt wird. Das Ergebnis sind hierbei jeweils zwei Werte: die maximale Entfernung einer Log-Nachricht von ihrer korrekten Position, also der größte vorgekommene Fehler und die Gesamtzahl an Nachrichten, die an falscher Stelle eingeordnet wurden, also die Anzahl vorgekommener Fehlern. Da das Log-Matching im Replay exakt gleich funktioniert wie in der Aufzeichnung, sind für die Evaluation des Log-Matchings keine Replays notwendig.

5.3 Debugging

Mit dem im Folgenden beschriebenen Versuch soll versucht werden, die tatsächlichen Debugging-Möglichkeiten von ditm empirisch zu evaluieren. Eine empirische Evaluation eines Debuggers ist grundsätzlich schwierig, da es beim Debugging vor allem darum geht, Entwicklern beim Verständnis eines komplexen Problems zu helfen, welches sie noch nie zuvor gesehen haben. Es bräuchte also im Optimalfall eine große Menge relevanter, bekannter Bugs und Entwickler, denen diese Bugs vollständig unbekannt sind, um zu evaluieren, wie effektiv ein Debugger tatsächlich ist.

Eine solche Studie würde den Rahmen dieser Arbeit bei weitem sprengen. Stattdessen wird hier evaluiert, ob ditm tatächlich in der Lage ist, Bugs zu finden und deterministisch in Replays zu reproduzieren. Das ist die Funktionalität, die im Kern dieser Arbeit steht und die essenziell für das angedachte Debugging-Konzept von ditm ist.

Dazu werden Systeme mit bekannten Partitionstoleranz-Bugs verwendet. Die Bugs sollen mit Hilfe von ditm erst in einer Aufzeichnung nachweislich erzeugt werden und dann wiederholbar in Replays wiedergegeben werden.

5.3.1 Systeme

Es werden insgesamt drei verschiedene Systeme verwendet, in denen jeweils mindestens ein Bug im Zusammenhang mit Netzwerk-Unterbrechungen bekannt ist.

Bank

Das erste System ist eine kleine Bank-Simulation mit einem zentralen Server, der Kontodaten transaktionell verwaltet und einem API-Gateway, welches die Funktionalitäten des Systems kapselt. Es werden insgesamt vier Operationen unterstützt, das Anlegen eines Kontos, das Abfragen eines Kontostands und das Ein- und Auszahlen von Geld auf einem Konto.

Raft

Das zweite System ist eine Implementierung des Raft-Konsens-Algorithmus in der Form eines Append-Logs. Die Implementierung basiert auf einem auf Github zugänglichem Hobby-Projekt[1], das Raft mit Threads als Knoten umsetzt. Für diese Arbeit wurde die Implementierung angepasst, um als verteiltes System zu laufen und über HTTP zu kommunizieren. Obwohl es sich um eine minimale Implementierung handelt, ist diese im Sinne des Raft-Algorithmus vollständig und bietet damit alle nötige Funktionalität, um daran Bugs aus realen verteilten Systemen, die ähnliche Algorithmen verwenden, nachstellen zu können. Dazu zählen Systeme wie MongoDB, Redis und VoltDB. Die Bugs, welche hier nachgestellt werden, stammen ursprünglich aus MongoDB und Redis.

Startup

Das dritte und letzte System ist das System des Startups Spenoki[2]. Konkret handelt es sich um eine SaaS-Lösung für das Sustainability Reporting von

[1] https://github.com/eileen-code4fun/SimpleRaft
[2] https://www.spenoki.de/

Mittelständischen Unternehmen. Das System setzt auf eine verteilte, serviceorientierte Architektur und besteht aus derzeit fünf Services. Im Kontext von ditm kann einer der fünf Services nicht verwendet werden, da dieser von AWS-Diensten wie S3 abhängig ist, es wird hier also ein Cluster aus vier Knoten verwendet. Die anderen Services sind von dem fehlenden Service nicht abhängig und können auch in dieser kleineren Konstellation problemlos funktionieren.

5.3.2 Bekannte Bugs

An dieser Stelle sind die bekannten Bugs in den jeweiligen Systemen beschrieben, die hier mit Hilfe von ditm untersucht werden sollen.

B1: Doppelte Buchung

Das Banksystem wurde speziell für dieses Szenario entwickelt und enthält einen Bug, welcher dazu führt, dass Ein- und Auszahlungen doppelt auf einem Konto verbucht werden können. Konkret tritt dieses Problem auf, weil das API-Gateway automatisch Retries für fehlgeschlagene Operationen sendet und die Ein- und Auszahlung nicht idempotent sind. Um den Bug zu reproduzieren, muss ein Ein- oder Auszahlungsrequest an das API-Gateway gesendet werden und die Response auf den Request zwischen Bankserver und Gateway von einer Netzwerk-Unterbrechung geblockt werden.

B2: Dirty Reads in MongoDB

Jepsen hat in MongoDB Version 2.6.7 einen Bug entdeckt, der im Fall gewisser Netzwerk-Unterbrechungen für eine kurze Zeit Dirty Reads zulässt. Wenn der Leader-Knoten von der Mehrheit des Clusters getrennt wird, tritt dieser nach einem Timeout selbst zurück und nimmt keine Schreiboperationen mehr entgegen. Wird allerdings vor diesem Timeout ein Wert über diesen Leader geschrieben, kann dieser von dort auch gelesen werden, bevor er später zurückgerollt wird. Das liegt daran, dass in MongoDB damals neue

Schreiboperationen unter allen Konsistenzeinstellungen sofort vom Leader lesbar waren, noch bevor sie an eine Mehrheit des Clusters repliziert wurden. [16]

Dieser Bug wird hier originalgetreu mittels der in Kapitel 5.3.1 beschriebenen Raft-Implementierung simuliert.

B3: Election-Endlosschleife in MongoDB

Alquraan et. al. [5] nennen als Beispiel für Partitionstoleranz-Bugs bei nicht vollständigen Netzwerk-Unterbrechungen einen Bug in MongoDB, welcher zu einer Schleife führt, in der endlos neue Leader gewählt werden. MongoDB verwendet einen Verwaltungsknoten, um unentschiedene Elections zu entscheiden. Tritt in einem Cluster mit zwei Replica-Knoten und einem Verwaltungsknoten eine Netzwerk-Unterbrechung wie in Abbildung 5.1 auf, führt das dazu, dass Knoten A und B sich abwechselnd zum Leader wählen lassen, da sie jeweils nie den Leader erreichen können, der Verwaltungsknoten aber immer an beiden Elections teilnehmen kann und den alten Leader dann über den neuen Leader informiert. Dieser Bug manifestiert sich vollständig ohne Interaktionen durch einen Client von außen und der Effekt verschwindet wieder, sobald die Netzwerk-Unterbrechung geheilt ist.

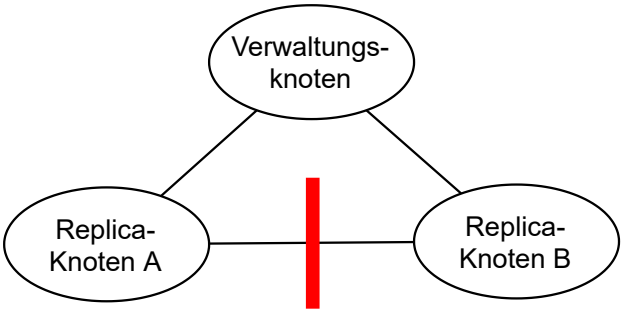


Abbildung 5.1: Visualisierung einer nicht vollständigen Netzwerk-Unterbrechung (rot), die in MongoDB eine Election-Endlosschleife auslösen konnte

Im Gegensatz zu MongoDB verwendet Raft keinen Verwaltungsknoten. Der beschriebene Bug kann dennoch realitätsnah in einem Raft-Cluster nach-

gestellt werden, da in Raft jeder Knoten die Rolle einnehmen kann, die in MongoDB der Verwaltungsknoten für diesen Bug spielt. Dazu müssen in Raft zwei Sicherheitsmechanismen ausgeschaltet werden:

- dass Knoten nur für einen Leader gleichzeitig stimmen dürfen und
- dass Knoten keine Leader akzeptieren, die einen geringeren Term haben, als sie selbst.

B4: Split-Brain in Redis-Raft

Wenn ein leader-basiertes System wie Raft mehr als einen Leader-Knoten gleichzeitig hat, wird dies als Split-Brain Situation bezeichnet. Split-Brain-Situationen führen unter anderem zu Dirty Reads. Um diese Situationen zu vermeiden hat Raft verschiedene Sicherheitsmaßnahmen, wie dass jeder Knoten pro Term nur für einen Leader stimmen darf [18]. Eine Maßnahme, dessen Fehlen diesen konkreten Bug ermöglicht, ist, dass das Entfernen eines Knotens aus dem Cluster eine Mehrheitsentscheidung aller Knoten ist [42]. Der Leader sollte also nicht in der Lage sein, einen Knoten zu entfernen, ohne eine Bestätigung der Aktion von mindestens der Hälfte des Clusters zu erhalten.

Einer frühen Version der Raft-Implementierung Redis-Raft, welche optinal in Redis verwendet werden kann, fehlte diese Einschränkung. In dieser Version ist es möglich, eine Split-Brain Situation zu erzeugen, indem der Leader durch eine Netzwerk-Unterbrechung vom Rest des Clusters getrennt wird und dann auf einmal angewiesen wird, alle anderen Knoten aus dem Cluster zu entfernen. Das führt dazu, dass der alte Leader effektiv zu einen getrennten Cluster mit einem Knoten wird und dort weiter als Leader agiert, während die anderen Knoten einen neuen Leader wählen und gemeinsam weiter als Cluster funktionieren. Das passiert, weil die anderen Knoten durch die Netzwerk-Unterbrechung nie von ihrer Entfernung informiert werden und diese auch nicht bestätigen. [42]

B5: Unerreichbare Daten

Das System von Spenoki verwendet ein eigenes, hierarchie-basiertes Authorisierungssystem. Dieses erlaubt es, Lese- und Updateoperationen in jedem Service lokal zu authorisieren, für das Anlegen neuer Daten ist aber ein Request an den zentralen Usermanagement-Service notwendig. Eine frühe Version dieses Authorisierungssystems hatte einen Bug, der dazu führte, dass neu geschriebene Daten für Nutzer unzugänglich würden, sollte der Usermanagement-Service während der Schreiboperation unerreichbar sein. Das lag daran, dass die Daten in diesem Fall einfach ohne die zur Authorisierung notwendige ID geschrieben wurden, die vom Usermanagement-Service kommen sollte.

In neueren Versionen des Systems schlagen stattdessen Schreiboperationen in diesem Fall ganz fehl, um inkonsistente Datenstände zu verhindern. Der Bug wurde hier in eine aktuelle Version des Systems wieder eingebaut, anstatt die alte Version des Systems zu verwenden, da dies einfacher war, als kompatible alte Versionen aller Teile des Systems zu finden und zu konfigurieren.

Kapitel 6

Ergebnisse

Im Folgenden werden die Ergebnisse der in Kapitel 5 Experimente dargestellt.

6.1 Request-Matching

Die Ergebnisse der Experimente zur Evaluation des Request-Matchings sind gruppiert nach Art der Requests, sodass pro Tabelle der Einfluss von Zufall in der Reihenfolge der Requests auf das Matching erkennbar wird. Jede Zeile in den Tabellen stellt die Ergebnisse für eine bestimmte Konfiguration des in Kapitel 5.1 beschriebenen Versuchs dar, jeweils für die vier verschiedenen Request-Matcher, die in den Versuchen evaluiert werden. Für jedes Experiment wird sowohl der Durchschnitt als auch der Median über zehn individuelle Durchführungen des jeweiligen Versuchs angegeben.

Tabelle 6.1: Ergebnisse zum Request-Matching von einfachen GET-Requests: Anzahl der im Replay falsch behandelten Requests pro Anzahl Requests im Recording, beides als Durchschnitt und Median über zehn Durchführungen des Versuchs mit dem jeweiligen Matcher.

Shuffle	Anzahl Requests		Heurist. Matcher		Exakter Matcher		Mix Matcher		Zählender Matcher	
	avg	med	avg	med	avg	med	avg	med	avg	med
no	13.3	13.5	0.8	0.0	0.8	0.0	0.8	0.0	1.7	0.0
no	133.6	133.0	2.9	0.0	0.3	0.0	2.9	0.0	15.2	0.0
1	13	13.0	1.7	1.5	2.2	2.0	1.7	2.0	2.6	2.5
1	133.2	134.0	22.7	23.0	39.1	38.0	23.6	25.0	34.6	36.5
2	13.5	14.0	2.5	2.5	3.0	3.0	2.5	3.0	3.3	4.0
2	131.5	130.0	22.5	21.5	40.3	42.5	21.4	20.0	45.2	46.0
3	13.3	14.0	2.0	2.0	3.5	3.0	2.3	2.5	4.5	4.0
3	135.1	137.0	25.8	27.0	42.2	39.0	24.8	27.5	46.7	47.0
async	13	14.0	1.5	1.5	3.4	4.0	1.4	1.0	5.5	5.0
async	113.1	111.5	7.2	6.5	30.7	31.0	7.0	6.0	68.2	67.0

Die Tabellen 6.1 und 6.2 stellen die Ergebnisse der Experimente dar, in denen durch Retries vom HTTP-Client mehr Requests gesendet wurden, als die im Versuchsaufbau speziefizierten zehn oder 100. Die zusätzlichen Requests sind in jedem Aufruf des SUT unterschiedlich, auch zwischen Recordings und deren Replays.

In diesen Experimenten ist zu beobachten, dass selbst bei ungemischten Request-Flüssen mit allen Matchern Fehler in manchen Replays passieren. Trotzdem ist zu erkennen, dass der heuristische und der Mix-Matcher sich hier insgesamt deutlich besser verhalten als die anderen beiden. In den Versuchen mit Zeitstempeln, deren Ergebnisse in Tabelle 6.2 dargestellt sind, machen der heuristische und der Mix-Matcher in vielen Konfigurationen des Versuchs ein Vielfaches der Fehler, die sie in den Versuchen ohne Zeitstempel in Tabelle 6.1 machen.

Tabelle 6.2: Ergebnisse zum Request-Matching von GET-Requests mit Zeitstempel: Anzahl der im Replay falsch behandelten Requests pro Anzahl Requests im Recording, beides als Durchschnitt und Median über zehn Durchführungen des Versuchs mit dem jeweiligen Matcher.

Shuffle	Anzahl Requests		Heurist. Matcher		Exakter Matcher		Mix Matcher		Zählender Matcher	
	avg	med	avg	med	avg	med	avg	med	avg	med
no	13.6	14.0	0.7	0.0	0.9	0.0	1.2	0.0	2.1	0.0
no	134.7	136.5	19.6	0.0	19.0	0.0	9.4	0.0	19.9	0.0
1	13.8	14.0	3.3	3.0	3.0	2.5	3.8	4.0	3.0	3.5
1	135.4	137.5	37.0	37.0	36.2	35.0	36.7	36.5	38.1	39.5
2	13.4	13.0	3.7	4.0	4.3	4.5	3.2	3.5	3.4	3.0
2	134.1	133.5	42.0	41.0	43.1	45.0	40.0	37.5	42.9	42.0
3	13.2	13.0	4.1	4.0	5.3	6.0	5.0	6.0	4.8	5.0
3	134.5	134.5	48.8	47.0	48.3	47.5	50.8	51.5	49.1	48.0
async	13.7	13.5	5.0	5.0	5.6	5.0	6.1	6.0	6.2	6.5
async	113.4	108.0	60.9	60.0	65.2	65.0	65.0	65.0	65.6	66.5

Tabelle 6.3: Ergebnisse zum Request-Matching von GET-Requests mit no-keep-alive Header: Anzahl der im Replay falsch behandelten Requests pro Anzahl Requests im Recording, beides als Durchschnitt und Median über zehn Durchführungen des Versuchs mit dem jeweiligen Matcher.

Shuffle	Anzahl Requests		Heurist. Matcher		Exakter Matcher		Mix Matcher		Zählender Matcher	
	avg	med	avg	med	avg	med	avg	med	avg	med
no	10	10.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
no	100	100.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	10	10.0	0.0	0.0	1.7	2.0	0.0	0.0	4.2	3.5
1	100	100.0	0.0	0.0	22.5	22.5	0.0	0.0	40.2	41.0
2	10	10.0	0.0	0.0	2.5	2.0	0.0	0.0	3.6	4.0
2	100	100.0	0.0	0.0	25.4	25.5	0.0	0.0	48.8	50.5
3	10	10.0	0.0	0.0	2.2	2.0	0.0	0.0	5.0	5.0
3	100	100.0	0.0	0.0	25.4	25.0	0.0	0.0	52.2	53.0
async	10	10.0	0.0	0.0	2.5	2.5	0.0	0.0	7.3	7.5
async	100	100.0	0.0	0.0	22.4	21.5	0.0	0.0	68.2	65.5

Tabelle 6.3 zeigt, dass ditm problemlos mit stark gemischten Requests umgehen kann, sofern diese keine weiteren zufälligen Komponenten enthalten. Auch hier verhalten sich der Heuristische und der Mix-Matcher am besten, die beiden anderen Matcher haben deutliche Probleme.

Tabelle 6.4: Ergebnisse zum Request-Matching von GET-Requests mit Zeitstempel no-keep-alive Header: Anzahl der im Replay falsch behandelten Requests pro Anzahl Requests im Recording, beides als Durchschnitt und Median über zehn Durchführungen des Versuchs mit dem jeweiligen Matcher.

Shuffle	Anzahl Requests		Heurist. Matcher		Exakter Matcher		Mix Matcher		Zählender Matcher	
	avg	med	avg	med	avg	med	avg	med	avg	med
no	10	10.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
no	100	100.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	10	10.0	3.4	3.0	4.3	4.0	3.7	4.0	3.5	3.5
1	100	100.0	40.7	39.0	43.8	45.0	40.4	40.0	43.7	43.0
2	10	10.0	5.3	5.5	4.9	5.0	5.0	4.5	4.4	4.0
2	100	100.0	50.6	51.5	48.7	48.0	50.9	52.5	48.4	48.0
3	10	10.0	4.6	5.0	4.3	4.0	4.5	4.5	5.7	6.0
3	100	100.0	48.5	50.0	50.8	51.0	49.9	50.0	50.4	48.5
async	10	10.0	5.0	5.0	5.7	6.0	4.9	4.5	7.7	7.5
async	100	100.0	61.5	60.5	60.0	61.0	62.4	62.5	68.7	69.0

Bei gemischten Request-Flüssen, in denem zusätzliche zufällige Komponenten auftreten, die das Request-Matching zusätzlich erschweren, machen alle Request-Matcher Fehler. Weiterhin sind der heuristische und der Mix-Matcher mit Abstand am besten, einen klaren Favoriten gibt es hier aber zwischen den beiden nicht, wie Tabelle 6.4 zeigt.

Tabelle 6.5: Ergebnisse zum Request-Matching von POST-Requests mit Zeitstempel und Füller im Body: Anzahl der im Replay falsch behandelten Requests pro Anzahl Requests im Recording, beides als Durchschnitt und Median über zehn Durchführungen des Versuchs mit dem jeweiligen Matcher.

Shuffle	Anzahl Requests		Heurist. Matcher		Exakter Matcher		Mix Matcher		Zählender Matcher	
	avg	med	avg	med	avg	med	avg	med	avg	med
no	10	10.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
no	100	100.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	10	10.0	0.7	1.0	3.1	2.5	0.8	1.0	4.0	4.0
1	100	100.0	8.0	8.5	42.0	41.0	7.3	6.5	46.4	45.5
2	10	10.0	1.3	1.0	3.8	3.5	1.5	1.0	3.9	4.0
2	100	100.0	23.6	22.5	44.5	44.5	24.9	25.0	46.6	46.5
3	10	10.0	2.3	2.0	4.4	5.0	3.2	3.0	4.3	5.0
3	100	100.0	31.2	31.5	46.6	45.5	27.4	25.5	50.2	49.5
async	10	10.0	5.1	5.0	5.4	6.0	4.4	4.5	7.0	7.0
async	100	100.0	59.6	59.0	58.9	60.0	58.6	58.5	66.7	67.5

Auch in den Experimenten in Tabelle 6.5, die als zusätzliches Differenzierungsmerkmal drei unterschiedliche Längen der Request-Bodies über einen weiteren Query-Parameter einführen, erzielt kein Matcher selbst in den am leichtesten gemischten Experimenten perfekte Ergebnisse. Trotzdem ist eine deutlich bessere Performance im Vergleich zu Requests ohne den zusätzlichen Parameter in Tabelle 6.4 zu erkennen. Vor allem in leicht gemischten Experimenten machen der heuristische und der Mix-Matcher weniger als halb so viele Fehler.

6.2 Log-Matching

Die Ergebnisse für das Log-Matching sind in Tabelle 6.6 zusammengefasst. Es ist anzumerken, dass die Anzahl der Fehler und auch der maximale Fehler bei den asynchronen Experimenten Schätzwerte sind, da durch die asynchrone

Natur der Experimente bei manchen Logs nicht eindeutig festzustellen ist, ob sie an richtiger Stelle stehen oder nicht.

Tabelle 6.6: Ergebnisse der Log-Matching-Experimente: Der maximale Fehler beschreibt, wie weit entfernt eine Log-Nachricht maximal von ihrer korrekten Position eingeordnet war, die Anzahl an Fehlern, wie viele Log-Nachrichten insgesamt falsch eingeordnet waren.

Anzahl Requests	Anzahl Logs	async	max. Fehler	Anzahl Fehler
10	18	nein	1	1
100	178	nein	1	8
10	19	ja	15	17
100	165	ja	121	162

Eine interessante Beobachtung ist, dass in dem längeren der beiden synchronen Experimente die Fehler vorwiegend eng zusammen in Clustern von zwei bis drei Fehlern direkt hintereinander auftreten. Bei den asynchronen Experimenten stehen sogar fast alle Logs ganz am Ende der Aufzeichnung und fast alle Requests gemischt mit nur sehr wenigen Logs am Anfang.

6.3 Debugging

Alle der beschriebenen Bugs lassen sich mit Hilfe von ditm in Recordings und in Replays reproduzieren. Da insbesondere für die Reproduktion der Bugs in Raft teilweise eine Reihe von Schritten notwendig ist, wurden hierfür Python-Skripts geschrieben, die ditm effektiv als Failure-Testing-Tool verwenden und so ein Recording mit dem jeweiligen Bug erstellen. Tabelle 6.7 gibt einen Überblick darüber, welche Bugs anhand dieser Recordings mit welchem Matching-Mechanismus zuverlässig reproduzierbar sind.

*Tabelle 6.7: Überblick darüber, mit welchen Matchern welche Bugs zuverlässig in Replays reproduzierbar sind. *Nur in einem Cluster mit drei Knoten.*

Bug	Heuristisch	Zeitlich
B1: Doppelte Buchung	ja	nein
B2: Dirty Reads in MongoDB	nein	ja
B3: Election Endlosschleife in MongoDB	ja*	ja
B4: Split-Brain in Redis-Raft	nein	ja
B5: Unnerreichbare Daten	ja	ja

Außer der doppelten Buchung sind alle Bugs mit dem zeitbasierten Matcher reproduzierbar. Im Gegensatz dazu hat der heuristische Matcher, welcher hier stellvertretend für die anderen drei Matcher mit aufgeführt ist, Probleme mit dem raft-basierten System. Zwar kann der heuristische Matcher den Bug 3 in einem Cluster mit drei Knoten zuverlässig reproduzieren, wird das gleiche Experiment in einem Cluster mit fünf Knoten durchgeführt, ist aber auch dieser Bug nur noch mit dem zeitbasierten Matcher reproduzierbar.

Eine wichtige Beobachtung bei Bug 5 ist, dass ditm grundsätzlich keine token-basierte Authentifizierung während Replays unterstützt. Als Workaround wurde hier die Lebensdauer der Access-Token des SUT auf mehrere Monate gesetzt, um ditm zu erlauben, diese während eines Replays wiederzuverwenden. Mit dieser Einstellung hatte ditm keine Probleme, den Bug deterministisch zu reproduzieren.

Kapitel 7

Diskussion

In dieser Arbeit wurde ein neuartiger, proxy-basierter Replay-Mechanismus für das Debugging von verteilten Systemen mit Fokus auf Netzwerk-Unterbrechungen entwickelt. Die Kernidee dabei ist ein Proxy, der den gesamten Nachrichten-Verkehr des verteilten Systems kontrolliert, aufzeichnet und anhand der Aufzeichnungen Replays von Situationen abspielen kann. Für diese Wiedergabe wurden zwei verschiedene Ansätze entwickelt, einerseits das Konzept, Nachrichten in Aufnahme und Wiedergabe zueinander zu matchen und andererseits das Konzept, Netzwerk-Unterbrechungen anhand des zeitlichen Ablaufs einer Situation wiederzugeben. Im Folgenden werden die Ergebnisse der empirischen Evaluation des Prototypen diskutiert.

Die Experimente zum Log-Matching, welches zur Visualisierung der Abläufe im SUT beiträgt, zeigen demonstativ, dass das Log-Matching bei hoch asynchronen Prozessen mit vielen schnellen Logs und Requests an seine Grenzen kommen kann und dann effektiv unbrauchbar wird. Ganz im Gegenteil dazu stehen die Ergebnisse der synchronen Experimente. Zwar gibt es auch hier durchaus Fehler, diese sind aber selten genug und vor allem klein genug, so dass das Log-Matching hier erheblich zur Effektivität von ditm als Debugger beitragen kann. Die Tatsache, dass sich die Fehler, die vorgekommen sind, eng zusammen in Gruppen aufgetreten sind, deutet darauf hin, dass in der Logging-Pipeline möglicherweise ein Buffer nicht schnell genug abgearbeitet wird und sich die Log-Nachrichten so anstauen können. Es ist außerdem

naheliegend, dass nicht der ditm-Proxy selbst den Flaschenhals bildet, sondern bereits der Docker-Daemon oder noch wahrscheinlicher der verwendete HTTP-Log-Driver, da die Zeitstempel, welche zur Sortierung der Tabelle von Requests und Logs verwendet werden, in diesem Teil des Systems generiert werden.

Im Bezug auf das Request-Matching lässt sich sagen, dass die zwei komplexesten Matcher, der heuristische und der Mix-Matcher, wie erwartet in allen Request-Matching-Experimenten die besten Ergebnisse erzielen. Zwischen den beiden besteht aber kein relevanter Unterschied, der in diesen Ergebnissen erkennbar wird. Vor allem die Experimente ohne Zufallskomponenten in den Requests, deren Ergebnisse in Tabelle 6.3 dargestellt sind, deuten darauf hin, dass Request-Matching als Replay-Mechanismus funktionieren kann. Trotzdem liefert keiner der Matcher perfekte Ergebnisse in allen Situationen und es ist davon auszugehen, dass es auch in realen Situationen zu Fehlern kommen kann. Es ist allerdings zu beachten, dass die Request-Machting-Experimente spezifisch so konzipiert sind, dass sie schwierige Situationen für ditm erzeugen, um die Grenzen des Ansatzes zu finden. Diese liegen vor allem bei Requests, die in zufälliger Reihenfolge gesendet werden und zusätzlich selbst Zufallskomponenten enthalten. Des Weiteren können die Request-Matcher nicht gut mit Systemen umgehen, die unter gleichen Bedingungen unterschiedliche Requests senden.

Das Konzept des Request-Matchings wurde unter der Annahme entwickelt, dass verteilte Systeme in der Regel unter gleichen Bedingungen unterschiedliche Requests senden. Die Debugging-Experimente an dem raft-basierten System haben demonstriert, dass dies nicht immer der Fall ist und dass Request-Matching in diesem Fall, wie aufgrund der Request-Matching-Exprimente erwartet, nicht als Replay-Mechanismus für Bugs funktioniert. Vor allem für Systeme wie verteilte Datenbanken wie MongoDB, die Raft o.Ä. verwenden, eignet sich Request-Matching damit nicht.

Die Debugging-Experimente zeigen auch, dass der einfachere zeitbasierte Mechanismus hingegen effektiv darin ist, Bugs in Replays zu reproduzieren. Dass sich der erste Bug mit dem Mechanismus nicht reproduzieren lässt, liegt daran, dass der Mechanismus nicht in der Lage ist, ditms zufallsgetriebene

Netzwerk-Unterbrechungen in Replays zu reproduzieren und ditm nicht die Möglichkeit bietet, einseitige Netzwerk-Unterbrechungen gezielt zu steuern. Mit der Addition dieses Features könnte der zeitbasierte Ansatz alle fünf der getesteten Bugs deterministisch reproduzieren.

Die Ergebnisse der Untersuchung bestätigen also die Hypothese, dass es möglich ist, mit Hilfe eines Proxys durch Netzwerk-Unterbrechungen ausgelöste Bugs deterministisch in Replays zu reproduzieren, ohne das SUT zu instrumentieren oder anderweitig zu verändern. Im Gegensatz zu existierenden Replay-Debuggern für verteilte Systeme ist dieser Ansatz agnostisch gegenüber der Technologie des SUT und damit prinzipiell deutlich breiter einsetzbar als bisherige Lösungen.

Für sich allein hat ein Tool wie ditm allerdings trotzdem keinen großen Wert, da der Aufwand, einen komplexen Bug erstmals in einer Aufzeichnung zu erzeugen und zu identifizieren deutlich zu groß ist. Ein interaktives Tool wie ditm ist grundsätzlich nicht gut dazu geeignet, neue Bugs zu finden, da der potentielle Suchraum in realen Systemen zu groß ist, als dass ein Mensch diesen händisch durchsuchen könnte. Vielmehr ist der Wert von ditm, aufgezeichnete Bugs schnell und zuverlässig lokal zu reproduzieren. Dadurch erhalten Entwickler, die komplexe Bugs in verteilten Systemen untersuchen, schnelles Feedback zu Veränderungen, das es ihnen erlaubt, Hypothesen schnell zu überprüfen und somit effizienter zu arbeiten.

7.1 Einschränkungen

Eine große Einschränkung für die Ergebnisse dieser Arbeit ist, dass der entwickelte Prototyp ausschließlich für Systeme funktioniert, die HTTP-basiert kommunizieren. Das ist insofern einschränkend, dass viele Systeme wie verteilte Datenbanken in der Regel eigene, TCP-basierte Protokolle verwenden. Insbesondere der zeitbasierte Replay-Mechanismus sollte sich allerdings problemlos auch in einem Ethernet- oder TCP-Proxy anwenden lassen, da der Inhalt oder protokoll-spezifische Metadaten von Nachrichten nicht betrachtet werden müssen.

Es ist weiterhin zu beachten, dass diese Arbeit nicht empirisch untersucht, ob Replay-Debugging auf kognitiver Ebene eine effektive Methode zum Debugging von verteilten Systemen ist. Anekdotisch kann hier allerdings hinzugefügt werden, dass ditm sich bei der Entwicklung der in Kapitel 5 verwendeten Systeme als Debugger bewährt hat. Mit ditms Hilfe konnte unter Anderem eine Diskrepanz zwischen der Dokumentation und dem tatsächlichen Verhalten der Standardbibliothek von Go festgestellt werden. Der Mangel in der Dokumentation wurde auf dem entsprechenden Issue Tracker festgehalten[1] und kann somit in Zukunft verbessert werden.

7.2 Ausblick

Die Vision, die durch die Ergebnisse dieser Arbeit unterstützt wird, ist einen Replay-Debugger wie ditm mit einem Failure-Testing-Tool wie Jepsen zu kombinieren, so dass Jepsen für jeden identifizierten Bug ein Recording erstellt, welches dann debuggt werden kann. Eine große Herausforderung dabei ist, Jepsen dazu zu bringen, Aufzeichnungen zu erstellen, anhand derer ein deterministisches Replay möglich ist. Es ist denkbar, dass für die Symbiose der beiden Techniken ein neues Failure-Testing-Tool notwendig ist, welches die Anforderungen des Replay-Debuggings von vorneherein mit einbezieht.

Um ein Tool wie ditm in der Realität einsetzen zu können, wäre außerdem eine neue Implementierung des Konzepts notwendig, die andere Netzwerk-Protokolle als HTTP unterstützt, vorzugsweise mindestens TCP. Eine mögliche, sehr nutzerfreundliche Variante wäre über einen Docker-Network-Driver. Eine entsprechende Implementierung wurde im Rahmen dieser Arbeit bereits angefangen, aber für einen Prototypen als zu aufwändig befunden. Aus technischer Sicht spricht allerdings nichts gegen das Prinzip und die grundlegende Proxy-Funktionalität konnte in dem angefangenen Projekt bereits demonstriert werden. Für eine neue Implementierung sollte außerdem ein Konzept enwickelt werden, wie mit token-basierter Authentifizierung und ähnlichen Situationen umgegangen werden kann, in denen es nicht ausreicht, Client-

[1] https://github.com/golang/go/issues/48438

Requests im Replay einfach zu wiederholen.

Des Weiteren gibt es viele Möglichkeiten, das Interaktionsmodell und die GUI von ditm zu erweitern. Für eine reale Verwendung erscheint mindestens eine Visualisierung von Abläufen als Zeit-Raum-Diagramm wie in Oddity [34] oder ShiViz [33] sinnvoll.

Zwei weitere Features, die ein Tool wie ditm zu einem erheblich mächtigeren Debugger machen könnten, sind ein Diff-Tool und ein Editor für Recordings. Das Diff-Tool sollte sowohl Unterschiede zwischen verschiedenen Replays der gleichen Aufzeichnung, sowie zwischen Aufzeichnung und Replay identifizieren können, um schnell den Effekt von Änderungen im Code des SUT beim Debugging feststellen zu können. Ein Editor für Recordings, der es erlaubt, Netzwerk-Unterbrechungen zu verändern, zu entfernen und hinzuzufügen, wäre ein weiteres Werkzeug für Entwickler, um schnell Hypothesen zu überprüfen, für die sonst die jeweilige Situation komplett händisch reproduziert werden müsste.

Kapitel 8

Fazit

Im Rahmen dieser Arbeit wurde erforscht, ob ein Proxy ein geeignetes Werkzeug ist, um Netzwerk-Unterbrechungen deterministisch in einer Testumgebung für verteilte Systeme zu reproduzieren. Zu diesem Zweck wurde ein Prototyp für einen proxy-basierten Replay-Debugger namens ditm entwickelt, der das Konzept für HTTP-basierte Systeme implementiert. Ditm ermöglicht es, Netzwerk-Unterbrechungen zu erzeugen, Netzwerk-Verkehr und persistierte Zustände in verteilten Systemen aufzuzeichnen und anhand der Aufzeichnungen Replays durchzuführen, die zuverlässig Partitionstoleranz-Bugs reproduzieren. Dazu erfordert ditm keine Veränderung des jeweiligen Systems. Zur Bedienung bietet ditm eine GUI, in der die aufgezeichneten Requests gemeinsam mit den Log-Ausgaben des Systems dargestellt werden und mit Hilfe von docker-compose ist ditm leicht zu konfigurieren und vollständig mit einem Kommando zu starten.

Die Fähigkeit von ditm, Partitionstoleranz-Bugs deterministisch zu reproduzieren wurde mit fünf verschiedenen Bugs in drei verschiedenen Systemen evaluiert, darunter das Produktionssystem des Startups Spenoki. Die Ergebnisse zeigen, dass das Prinzip grundsätzlich funktioniert, auch wenn ditm noch unter gewissen Kinderkrankheiten leidet. Beispielsweise fehlt die Möglichkeit, während Replays korrekt mit token-basierter Authentifizierung umzugehen, wofür es noch eine sinnvolle Lösung zu finden gilt. Weitere nächste Schritte für den realen Einsatz eines Tools wie ditm als Debugger sind, das Sys-

tem für TCP als Netzwerkprotokoll zu implementieren und ein Frontend zu entwickeln, das die aggregierten Events des SUT als Zeit-Raum-Diagramm virtualisieren kann.

Die Zukunftsvision ist eine Integration mit einem Failure-Testing-Tool wie Jepsen, sodass gefundene Bugs automatisch aufgezeichnet werden und für Replay-Debugging zur Verfügung stehen. Für diese Vision gibt diese Arbeit dem Forschungsfeld einen einfachen Replay-Mechanismus für verteilte Systeme an die Hand, der in Kombination mit der nächsten Generation von Failure-Testing-Tools das Debugging verteilter Systemen effizienter machen kann.

Literatur

[1] Maarten Van Steen und Andrew S Tanenbaum. *Distributed systems*. Leiden, The Netherlands, 2017.

[2] Maarten van Steen und Andrew S Tanenbaum. "A brief introduction to distributed systems". In: *Computing* 98.10 (2016), S. 967–1009.

[3] Peter Bailis und Kyle Kingsbury. "The network is reliable: An informal survey of real-world communications failures". In: *Queue* 12.7 (2014), S. 20–32.

[4] George F Coulouris, Jean Dollimore, Tim Kindberg und Gordon Blair. *Distributed systems: concepts and design*. Boston, USA: Pearson Education, Inc., 2012. ISBN: 978-0-13-214301-1.

[5] Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta und Samer Al-Kiswany. "An Analysis of Network-Partitioning Failures in Cloud Systems". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Okt. 2018, S. 51–68. ISBN: 978-1-939133-08-3.

[6] Armin Balalaie und James A. Jones. "Towards a Library for Deterministic Failure Testing of Distributed Systems". In: *Proceedings of the ACM Symposium on Cloud Computing*. SoCC '19. Santa Cruz, CA, USA: Association for Computing Machinery, 2019, S. 486. ISBN: 9781450369732. DOI: 10.1145/3357223.3366026.

[7] Ding Yuan u. a. "Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems". In: *11th USENIX Symposium on Operating Systems Design*

and Implementation (OSDI 14). Broomfield, CO: USENIX Association, Okt. 2014, S. 249–265. ISBN: 978-1-931971-16-4.

[8] Haryadi S. Gunawi u. a. "Why Does the Cloud Stop Computing? Lessons from Hundreds of Service Outages". In: *Proceedings of the Seventh ACM Symposium on Cloud Computing*. SoCC '16. Santa Clara, CA, USA: Association for Computing Machinery, 2016, S. 1–16. ISBN: 9781450345255. DOI: 10.1145/2987550.2987583.

[9] Daniel Neri, Laurent Pautet und Samuel Tardieu. "Debugging Distributed Applications with Replay Capabilities". In: *Proceedings of the Conference on TRI-Ada '97*. TRI-Ada '97. St. Louis, Missouri, USA: Association for Computing Machinery, 1997, S. 189–195. ISBN: 0897919815. DOI: 10.1145/269629.269649.

[10] Dennis Michael Geels, Gautam Altekar, Scott Shenker und Ion Stoica. "Replay debugging for distributed applications". In: *Proceedings of the 2006 USENIX Annual Technical Conference*. 2006, S. 289–300.

[11] Nane Kratzke und Peter-Christian Quint. "Understanding cloud-native applications after 10 years of cloud computing - A systematic mapping study". In: *Journal of Systems and Software* 126 (2017), S. 1–16. ISSN: 0164-1212. DOI: 10.1016/j.jss.2017.01.001.

[12] Dennis Gannon, Roger Barga und Neel Sundaresan. "Cloud-Native Applications". In: *IEEE Cloud Computing* 4.5 (2017), S. 16–21. DOI: 10.1109/MCC.2017.4250939.

[13] Eric A Brewer. "Towards robust distributed systems". In: *PODC*. Bd. 7. 10.1145. Portland, OR. 2000, S. 343477–343502.

[14] Seth Gilbert und Nancy Lynch. "Perspectives on the CAP Theorem". In: *Computer* 45.2 (2012), S. 30–36. DOI: 10.1109/MC.2011.389.

[15] William Schultz, Tess Avitabile und Alyson Cabral. "Tunable Consistency in MongoDB". In: *Proc. VLDB Endow.* 12.12 (Aug. 2019), S. 2071–2081. ISSN: 2150-8097. DOI: 10.14778/3352063.3352125.

[16] Kyle Kingsbury. *Jepsen: MongoDB stale reads*. 20. Apr. 2015. URL: https://aphyr.com/posts/322-jepsen-mongodb-stale-reads (besucht am 25. 10. 2021).

[17] Heidi Howard. "Distributed consensus revisited". Diss. 2019. DOI: 10.17863/CAM.38840.

[18] Diego Ongaro und John Ousterhout. "In Search of an Understandable Consensus Algorithm". In: *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, Juni 2014, S. 305–319. ISBN: 978-1-931971-10-2.

[19] Maurice P. Herlihy und Jeannette M. Wing. "Linearizability: A Correctness Condition for Concurrent Objects". In: *ACM Trans. Program. Lang. Syst.* 12.3 (Juli 1990), S. 463–492. ISSN: 0164-0925. DOI: 10.1145/78969.78972.

[20] Ivan Beschastnikh, Patty Wang, Yuriy Brun und Michael D Ernst. "Debugging Distributed Systems: Challenges and Options for Validation and Debugging". In: *Queue* 14.2 (März 2016), S. 91–110. ISSN: 1542-7730. DOI: 10.1145/2927299.2940294.

[21] Armin Balalaie. "Towards a Library for Deterministic Failure Testing of Distributed Systems THESIS". Magisterarb. UNIVERSITY OF CALIFORNIA, IRVINE, 2020.

[22] Rupak Majumdar und Filip Niksic. "Why is Random Testing Effective for Partition Tolerance Bugs?" In: *Proc. ACM Program. Lang.* 2.POPL (Dez. 2017). DOI: 10.1145/3158134.

[23] Peter Alvaro und Severine Tymon. "Abstracting the Geniuses Away from Failure Testing". In: *Commun. ACM* 61.1 (Dez. 2017), S. 54–61. ISSN: 0001-0782. DOI: 10.1145/3152483.

[24] Kyle Kingsbury. *Jepsen Analyses*. 16. Feb. 2017. URL: https://jepsen.io/analyses (besucht am 29. 09. 2021).

[25] Kyle Kingsbury et. al. *Jepsen*. URL: https://github.com/jepsen-io/jepsen (besucht am 24. 10. 2021).

[26] Ali Basiri u. a. "Chaos Engineering". In: *IEEE Software* 33.3 (2016), S. 35–41. DOI: 10.1109/MS.2016.60.

[27] Wei Xu, Ling Huang, Armando Fox, David Patterson und Michael Jordan. "Experience Mining Google's Production Console Logs". In: *Proceedings of the 2010 Workshop on Managing Systems via Log Analysis and Machine Learning Techniques*. SLAML'10. Vancouver, BC, Canada: USENIX Association, 2010, S. 5.

[28] Benjamin H. Sigelman u. a. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Techn. Ber. Google, Inc., 2010. URL: https://research.google.com/archive/papers/dapper-2010-1.pdf.

[29] Raja Sambasivan, Rodrigo Fonseca, Ilari Shafer und Gregory Ganger. *So, you want to trace your distributed system? Key design insights from years of practical experience*. Parallel Data Laboratory, Carnegie Mellon University. Apr. 2014.

[30] Paul Barham, Austin Donnelly, Rebecca Isaacs und Richard Mortier. "Using Magpie for request extraction and workload modelling." In: *Proceedings of the 6th USENIX Symposium on Operating Systems Desing and Implementation (OSDI 04)*. Bd. 4. 2004, S. 18–18.

[31] Rohan Achar, Pritha Dawn und Cristina V. Lopes. "GoTcha: An Interactive Debugger for GoT-Based Distributed Systems". In: *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2019. Athens, Greece: Association for Computing Machinery, 2019, S. 94–110. ISBN: 9781450369954. DOI: 10.1145/3359591.3359733.

[32] Nima Honarmand und Josep Torrellas. "Replay debugging: Leveraging record and replay for program debugging". In: *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 2014, S. 455–456. DOI: 10.1109/ISCA.2014.6853229.

[33] Ivan Beschastnikh, Patty Wang, Yuriy Brun und Michael D. Ernst. "Debugging Distributed Systems". In: *Commun. ACM* 59.8 (Juli 2016), S. 32–37. ISSN: 0001-0782. DOI: 10.1145/2909480.

[34] Doug Woos, Zachary Tatlock, Michael D. Ernst und Thomas E. Anderson. "A Graphical Interactive Debugger for Distributed Systems". In: *ArXiv* abs/1806.05300 (2018).

[35] Ivan Beschastnikh, Perry Liu, Albert Xing, Patty Wang, Yuriy Brun und Michael D. Ernst. "Visualizing Distributed System Executions". In: *ACM Trans. Softw. Eng. Methodol.* 29.2 (März 2020). ISSN: 1049-331X. DOI: 10.1145/3375633.

[36] Ellis Michael, Doug Woos, Thomas Anderson, Michael D. Ernst und Zachary Tatlock. "Teaching Rigorous Distributed Systems With Efficient Model Checking". In: *Proceedings of the Fourteenth EuroSys Conference 2019*. EuroSys '19. Dresden, Germany: Association for Computing Machinery, 2019. ISBN: 9781450362818. DOI: 10.1145/3302424.3303947.

[37] Patrick Reipschläger, Burcu Kulahcioglu Ozkan, Aman Shankar Mathur, Stefan Gumhold, Rupak Majumdar und Raimund Dachselt. "DebugAR: Mixed Dimensional Displays for Immersive Debugging of Distributed Systems". In: *Extended Abstracts of the 2018 CHI Conference on Human Factors in Computing Systems*. CHI EA '18. Montreal QC, Canada: Association for Computing Machinery, 2018, S. 1–6. ISBN: 9781450356213. DOI: 10.1145/3170427.3188679.

[38] Antonia Bertolino, Guglielmo De Angelis und Antonino Sabetta. "VCR: Virtual Capture and Replay for Performance Testing". In: *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. 2008, S. 399–402. DOI: 10.1109/ASE.2008.58.

[39] James Mickens, Jeremy Elson und Jon Howell. "Mugshot: Deterministic Capture and Replay for Javascript Applications". In: *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*. NSDI'10. San Jose, California: USENIX Association, 2010, S. 11.

[40] Ravi Netravali u. a. "Mahimahi: Accurate Record-and-Replay for HTTP". In: *2015 USENIX Annual Technical Conference (USENIX ATC 15)*.

Santa Clara, CA: USENIX Association, Juli 2015, S. 417–429. ISBN: 978-1-931971-225.

[41] Xuezheng Liu u. a. "D3S: Debugging Deployed Distributed Systems". In: *NSDI*. 2008.

[42] Kyle Kingsbury. *Jepsen: Redis-Raft 1b3fbf6*. 23. Juni 2020. URL: `https://jepsen.io/analyses/redis-raft-1b3fbf6` (besucht am 04. 11. 2021).

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis