HAW HAMBURG

Bachelorarbeit

Johann Arne Laur

Evaluierung von Konsensalgorithmen in verteilten Speichersystemen

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department Computer Science

Johann Arne Laur

Evaluierung von Konsensalgorithmen in verteilten Speichersystemen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Angewandte Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Martin Becke
Zweitgutachter: Prof. Dr. Michael Koehler-Bußmeier

Eingereicht am: 10. Dezember 2024

Johann Arne Laur

Thema der Arbeit

Evaluierung von Konsensalgorithmen in verteilten Speichersystemen

Stichworte

Konsensalgorithmen, Raft, Verteilte Datenbanken, Pirgoue

Kurzzusammenfassung

Die Arbeit ist im Kontext der Konsensprobleme und der Konsensbildenden Algorithmen einzuordnen. In dem verteilten Speichersystem *rqlite* untersucht und vergleicht die Arbeit die Effizienz sowie die Probleme von Raft und Pirogue. Mit einem teilfaktoriellen Versuchsplan werden die Auswirkungen der Skalierung, Heartbeat Timeout, Netzwerkpartitionen, Witnesses (Pirogue) und der Ausfälle für beide Algorithmen untersucht. Pirogue kann zu einer besseren Verfügbarkeit, schnelleren Entscheidungen des Quorums und zu einem besseren Umgang mit Netzwerkproblemen führen.

Johann Arne Laur

Title of Thesis

Evaluation of consensus algorithms in distributed storage systems

Keywords

consensus algorithms, raft, distributed databases, pirogue

Abstract

The thesis is to be categorised in the context of consensus problems and consensus algorithms. In a distributed storage system, it analyses and compares the efficiency and the problems of Raft and Pirogue. The effects of scaling, heartbeat timeout, network partitions, witnesses (pirogue) and node failures for both algorithms are analysed using a partial factorial test plan. Pirogue can lead to a better availability of the cluster and also to faster decisions by the quorum and better handling of specific network problems.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1 Einleitung

1.1 Motivation

Eine Verteilung von Systemkomponenten kann sich nicht nur geplant, sondern ebenso mit wachsenden Unternehmensstrukturen durch Akquisition ergeben [27]. Einzelne Standorte verfügen über eigene Infrastruktur und eigene lokale Informationssysteme, die Herausforderungen für die Integration bereithalten [27]. Insbesondere die Verwaltung eines gemeinsamen Datenbestandes erfordert in der Praxis oftmals eine komplexe Administrierung [23]. Um Datenkonsistenz zu gewährleisten, ergibt sich die Notwendigkeit für ein automatisiertes, skalierbares und fehlertolerantes Verfahren, das eine konsistente Sicht auf die Daten gewährleistet. Diese Probleme sind in der Informatik bekannt: Speichersysteme wie auch verteilte Datenbanken stehen oftmals vor der Herausforderung, aus verschiedenen Schreiboperationen einen konsistenten Datenbestand anbieten zu müssen [23]. Auch wenn einzelne Prozesse im Netzwerk beliebige Eingaben liefern, muss eine einheitliche, konsistente Ausgabe geschaffen werden - insbesondere bei Prozessfehlern [19]. Hieraus ergibt sich die Problemklasse der Konsensprobleme, mit der sich diese Arbeit beschäftigt [19]. Konkret lässt sich das Problem auf einem ungerichteten Graphen aus n Knoten (und Prozessen), die jeweils den vollständigen Graphen kennen, definieren [19]. Dieses Übereinstimmungsproblem in einem verteilten System (*agreement problem* [19]) kann ein Konsensalgorithmus wie Raft annähernd lösen [20]. Die gleiche Problemklasse in einem fehlertoleranten, verteilten System kann Pirogue annähernd lösen [22]. Beide Algorithmen lösen ein gleiches Problem in einer gleichen Umgebung durch ein Quorum, mit dem Abstimmungen zur Ergebnisfindung durchgeführt werden. Dies schafft eine gemeinsame Basis für einen sinnvollen Vergleich in dieser Arbeit. Interessant gestaltet sich der Vergleich unter anderem durch unterschiedliche Arten, die Menge an abstimmungsberechtigten Servern zu bilden (Quorum).

1.2 Zielsetzung

Das Ziel dieser Arbeit ist die Evaluation und Diskussion von konsensbildenden Algorithmen. Gegenstand der Untersuchung sind dabei der stark verbreitete Algorithmus *Raft* sowie eine moderne Weiterentwicklung von Raft: *Pirogue*. Hervorzuheben bezüglich der Auswertung sind die Parameter der Größe des Netzwerks (Skalierbarkeit), verschiedener Konfigurationen, aber besonders auch das fehlertolerante Verhalten bei verschiedenen Ausfällen. Die verschiedenen Faktoren sollen nicht in allen Kombinationen, sondern mit einer aussagekräftigen Teilmenge experimentell überprüft werden (siehe 3.2.2). Die Optimierungen aus Pirogue sollen daher reproduziert und geprüft werden, um daraus eine fundierte Implementierungsempfehlung für ein Datenspeicherungssystem abzuleiten. Die verschiedenen Modelle dieser Konsistenz sind dabei zunächst zu definieren (siehe 2.2). Um die Evaluierung realitätsnah zu gestalten, verwenden die Experimente eine Raft-Implementierung, die an einer relationalen Datenbank ansetzt. Die Implementierung von Pirogue findet im selben Szenario statt. In [22] nehmen die Autoren von Pirogue bereits einige grundlegende Analysen vor, um ihren Algorithmus zu präsentieren. Die Arbeit zielt daher nicht nur auf eine Reproduktion der Ergebnisse dieses Papers ab, sondern wertet beide Algorithmen mit umfassenderen Metriken in einem konkreten Implementierungskontext aus.

1.3 Gliederung

Zum tieferen Verständnis der definierten Ziele dieser Arbeit sollen zunächst Grundbegriffe aus dem Forschungskontext der verteilten Systeme und der Konsensprobleme eingeführt und erläutert werden. Dies umfasst die Definition zentraler Konzepte und die Beschreibung der genutzten oder damit verwandten, relevanten Algorithmen. Es soll insgesamt ein Verständnis über die Problemklasse der Konsensprobleme aufgebaut, die Art der Algorithmen und ihre Besonderheiten beschrieben werden. Darüber hinaus wird ein kurzer Überblick geschaffen, mit welchen Alternativen, Algorithmen oder Ansätzen ähnliche, jedoch unterschiedliche Probleme in der Informatik lösbar sind und wodurch sich die Algorithmen von ähnlichen abgrenzen.

Anschließend werden kurz einige maßgebliche Entscheidungen bei der Implementierung der Protokolle und dem Aufbau der Versuchsumgebung beschrieben. Der Hauptteil gliedert sich in einen Methoden- und einen Ergebnisteil. Zu dem methodischen Vorgehen

gehört die Aufstellung des Versuchsplans mit seinen Faktoren und Stufen. Neben der Beschreibung der Metriken und der Art ihrer Erfassung, befindet sich im Hauptteil eine Erläuterung des Deployments und der Durchführung der Versuche.

Abschließend beinhaltet die Arbeit eine Auswertung, um daraufhin die Algorithmen Raft und Pirogue vergleichen zu können. Das Fazit soll die Vor- und Nachteile beider Algorithmen beleuchten, den geeigneten Einsatz in einem verteilten Speichersystem diskutieren und einen Ausblick auf mögliche sich anschließende Arbeiten bieten.

2 Grundlagen

2.1 Verteiltes System

Üblicherweise wird ein verteiltes System als eine Menge unabhängiger Rechner definiert, die einem Nutzer jedoch als ein abgeschlossenes, kohärentes System erscheinen [26]. Anwendungen kommunizieren unter anderem durch Nachrichten über Rechnernetze und formen damit insgesamt ein verteiltes System [24]. Dahinter verbirgt sich bereits ein wichtiges Merkmal, das zu klassischen Problemen dieser Art von Systemen führt: Dadurch, dass die Rechner unabhängig voneinander sind und Prozesse nicht wie in einem klassischen monolithischen System über Schnittstellen der gewählten Programmiersprache und -umgebung beziehungsweise nicht über die Schnittstellen des Betriebssystems kommunizieren können, ergeben sich Herausforderungen in der Prozesskommunikation. Hieraus kann bereits abgeleitet werden, dass zum Beispiel Synchronisationsprobleme bei Prozessen auf einem Rechner unter Umständen in einem solchen System durch die Verteilung bereits komplett neu zu lösen sind.

Ein wenig präziser wird dieser Aspekt von einer weiteren möglichen Definition aufgegriffen, die ein verteiltes System als eine Ansammlung an Rechnern, die ein gemeinsames Problem lösen, begreift, die dabei jedoch keinen gemeinsamen (Arbeits-)speicher besitzen [3]. Somit wird besonders deutlich, dass das verteilte System nicht nur Mehraufwand in der Kommunikation mit sich bringt, sondern zugleich auch gezielte Strategien erfordert, um persistente oder flüchtige Daten konsistent (siehe Kapitel 2.2 für Details bzgl. Konsistenzmodellen) und auf allen beteiligten Rechnern zu halten. Zu den weiteren Zielen eines verteilten Systems gehört die Verteilungstransparenz. Wie auch die weiteren Beschreibungen dieses Abschnittes bildet [26] dabei die Grundlage. Weitere Quellen sind gegebenenfalls explizit referenziert. Die Verteilungstransparenz konkretisiert die Kohärenz von verteilten Systemen: Das System soll die Verteilung verbergen. Die folgende Tabelle aus [26] zählt dessen Teilaspekte auf. Das Verständnis über diese Systemziele hilft zudem dabei, Designentscheidungen späterer Konsensprotokolle zu verstehen. Ebenso ist

zu betonen, dass keines der Ziele vollständig erfüllt sein muss, um ein nahezu optimales System zu entwickeln.

Transparenz	**Beschreibung**
Zugriff	Verbirgt Unterschiede in der Datendarstellung und die Art und Weise, wie auf eine Ressource zugegriffen wird
Ort	Verbirgt, wo sich eine Ressource befindet
Migration	Verbirgt, dass eine Ressource an einen anderen Ort verschoben werden kann
Relokation	Verbirgt, dass eine Ressource an einen anderen Ort verschoben werden kann, während sie genutzt wird
Replikation	Verbirgt, dass eine Ressource repliziert ist
Nebenläufigkeit	Verbirgt, dass eine Ressource von mehreren konkurrierenden Benutzern gleichzeitig genutzt werden kann
Fehler	Verbirgt den Ausfall und die Wiederherstellung einer Ressource

Tabelle 2.1: Transparenztypen nach Tanenbaum [26]

Informationssysteme müssen zudem ständig erweiterbar und skalierbar sein, um wechselnden Anforderungen der Nutzer gerecht zu werden oder sich an anderweitige Rahmenbedingungen anpassen zu lassen. Daher sind Problemstellungen bei dem Entwurf von Algorithmen und Protokollen stets mit dem Gedanken der Skalierbarkeit im Hinterkopf zu betrachten, sodass das Gesamtsystem auch im Falle einer Funktionserweiterung oder bei ansteigender Nutzung noch effizient arbeiten kann. Zuletzt muss hervorgehoben werden, dass der Begriff des verteilten Systems weder über die Art der Verteilung noch die Rechenleistung eine Aussage trifft.

2.2 Konsistenz

Eine Verteilung der Systemkomponenten kann sich aus unterschiedlichen Gründen ergeben. Häufige Architekturentscheidungen sind dabei bessere Skalierbarkeit oder Ausfallsicherheit, damit das System nicht von einzelnen Knoten und damit auch deren Datenbestand abhängig ist. Somit benötigen auch Prozesse auf anderen Rechnern die Daten eines anderen und eine Replikation des allgemeinen Datenbestandes kann erforderlich sein [26]. In jedem Fall führt das Halten von Datenkopien durch Replikation zu der Herausforderung, dass bei Änderungen an einem Datensatz eine Konsistenz der Daten über alle Kopien (Replikate) hinweg gewährleistet werden muss. Dies kann durch verschiedene Konsistenzmodelle erreicht werden, die unterschiedliche Garantien und Kompromisse in

Bezug auf Latenz, Verfügbarkeit sowie auch Effizienz bieten. Je nach Entscheidung sind Modelle einfacher anzuwenden oder komplizierter in der Implementierung und Einhaltung.
Allgemein lassen sich die Konsistenzmodelle in Datenzentrische- und Client-zentrische Modelle unterteilen. Aus einem Konsistenzmodell folgt demnach eine Erwartungshaltung für einen Rechner, bezüglich der Daten, die er nach Schreib- oder Leseoperationen erhält. Ein Konsistenzmodell etabliert ein im System stets geltendes Regelsystem über die Handhabung des (verteilten) Speichers. Zu betrachten sind dabei Datensätze und eine Menge an Operationen. Bei einzelnen Datenelementen greifen hingegen die hier nicht weiter relevanten Kohärenzmodelle [26].

Das strengste und zugleich am simpelsten zu beschreibende Modell ist die strikte Konsistenz: Dieses Modell setzt voraus, dass jeder Prozess zu jedem Zeitpunkt seine geschriebenen Daten auch lesen kann. Dasselbe gilt für alle weiteren Prozesse auf beliebigen Rechnern im System [23]. In einem Rechnernetzwerk gibt es bereits physikalisch betrachtet keine Möglichkeit, Latenzen zu vermeiden. Außerdem muss jeder beteiligte Computer über die exakt selbe Zeit verfügen, sodass auch mögliche Konflikte bei parallelen Schreiboperationen in sich anschließenden Leseoperationen keinerlei Inkonsistenzen aufweisen oder zumindest eine zeitabhängige Konfliktlösung ohne weitere Verzögerung vorliegt [17].
Grundsätzlich ist kein Modell optimal und die Implementierungsentscheidung hängt vom betrieblichen Kontext ab. Beispielsweise kann ein Wetterbericht [26], der für gewöhnlich keine drastischen Veränderungen in Echtzeit aufweist, auch einen sehr lockeren Ansatz wählen, bei dem Clients unter Umständen noch einen leicht veralteten Datensatz erhalten.
Dieses Kapitel soll einige besonders erwähnenswerte Ansätze beleuchten, die sich dabei auf die Definitionen von Tanenbaum aus [26] stützen.

2.2.1 Daten-zentrische Konsistenzmodelle

Zum Verständnis der Konsensprotokolle sind die Auffassungen von Konsistenz in einem verteilten System essenziell. Die Protokolle zielen auf einen Konsens über die Anordnung von (Schreib-)Operationen ab, aus dem dann nach Ausführung auf allen Clients ein konsistenter Zustand folgt.

Sequentielle Konsistenz

Alle Lese- und Schreiboperationen an einem Speicher werden in einer spezifischen Reihenfolge durchgeführt. Jede gültige Reihenfolge dieser Operationen ist grundsätzlich zulässig, jedoch betrachten alle Prozesse auf allen Rechnern dann dieselbe Reihenfolge [26]. Unabhängig von jeglicher Zeit sind sich die Prozesse über die Reihenfolge von (atomaren) Schreiboperationen auf den Datenobjekten einig. Keine verschiedenen Prozesse erhalten gleichzeitig unterschiedliche Werte für einen spezifischen (atomaren) Lesezugriff [26, 28]. Dies veranschaulicht Abbildung 2.1.

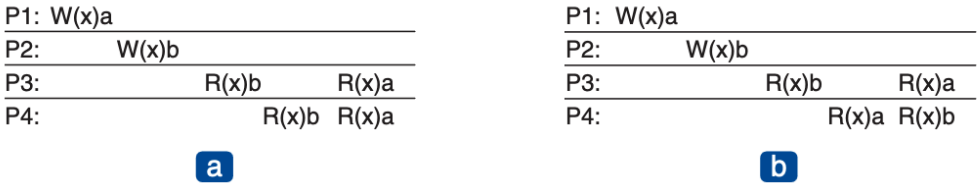


Abbildung 2.1: Sequenziell konsistenter Datenspeicher (a); nicht sequenziell konsistent (b) aus [26]

Kausale Konsistenz

Zum Erzielen von kausaler Konsistenz müssen alle Rechner Datenänderungen in der Reihenfolge verarbeiten, in der sie ursprünglich getätigt wurden [26]. Sofern also ein kausaler Zusammenhang besteht und sich zwei Schreiboperationen bedingen, muss diese Reihenfolge stets auch so beibehalten werden. Andere Operationen dürfen in beliebiger Ordnung über das Netzwerk im System übertragen und an den Knotenpunkten verarbeitet werden. An diesem Punkt ergibt sich somit eine Differenz zu der sequentiellen Konsistenz. Unabhängige Ereignisse können zueinander so gesehen ein inkonsistentes Verhalten an den Tag legen. Dies wirkt sich positiv auf die Netzwerkbelastung und die Latenzzeiten aus, da nicht jede Operation sofort an alle Knoten weitergeleitet werden muss. Als Beispiel kann ein soziales Netzwerk angeführt werden, bei dem die Reihenfolge der Posts und Kommentare konsistent bleiben muss, während andere Aktionen wie das Liken eines Beitrags keine strikte Reihenfolge erfordern – oder die Kommentare eines Posts in der Reihenfolge gleich bleiben müssen, ein gesamter Post aber auch verzögert und unabhängig von anderen Beiträgen eintreffen darf.

Linearisierbarkeit

Dieser Ansatz ist am wichtigsten für die Protokolle im Hauptteil der Arbeit. Er kommt nahe an die utopische, strikte Konsistenz heran und bietet eine Verschärfung bisheriger Daten-zentrischer Modelle. Für Systeme, in denen Datensätze möglichst immer synchron gehalten werden sollen und somit beispielsweise ein Konsensprotokoll zum Einsatz kommt, beschreibt dieses Modell das Verhalten am besten. Die folgende Definition stammt aus [9]. Ereignisse sind linearisierbar, sofern die Antwort von Ereignis 0 (e_0) vor dem Aufruf von Ereignis 1 (e_1) liegt. Hieraus folgt eine partielle, irreflexive Ordnung ($<$). Kann diese Ordnung um weitere Ereignisse erweitert werden, sodass sie weiterhin serialisierbar ist und diese Reihe an Ereignissen eine Übermenge zur bisherigen Ordnung bildet, so ist sie linearisierbar. Zum einfachen Verständnis lässt sich annehmen, dass die Ereignisse sich alle durch jeweils Aufruf (*inv*) und Antwort (*res*) sortieren lassen.

$$e_0 <_H e_1, \quad \text{if } res(e_0) \text{ precedes } inv(e_1) \text{ in } H \tag{2.1}$$

$$complete(H') \text{ is equivalent to some legal sequential history } S \tag{2.2}$$

$$<_H \quad \subseteq \quad <_S \tag{2.3}$$

2.2.2 Client-zentrische Konsistenzmodelle

Dieser Abschnitt hebt sich etwas von den Zielen eines Konsensalgorithmus ab, da die angesprochenen Ansätze nicht mehr auf eine (mehr oder wenige) systemweite Ordnung und Datenkonsistenz setzen, sondern einen einzelnen Client in den Vordergrund rücken. Aus Sicht eines Clients ergeben sich weitere Modelle, die zum Verständnis und zur Abgrenzung gegenüber der systemweiten Ansätze kurz beschrieben werden sollen. Ein einfaches Beispiel für ein Client-zentrisches Modell nennt sich *Write-Follows-Reads*. Ein Datenspeicher ist auf diese Weise konsistent, wenn Schreiboperationen, die sich auf ein Datenobjekt beziehen, dieses bereits vorher gelesen (und in ihren lokalen Datenspeicher geladen) haben. Eine erneute Operation auf einem Datenelement arbeitet somit nie auf einem älteren Wert als zuvor gelesen, sondern immer mindestens auf dem bereits bekannten (im Optimalfall natürlich immer dem aktuellen Wert). Ein weiteres mögliches Modell *Read-your-Writes* lässt sich in seiner Funktionsweise ebenso vom Namen gut ableiten: Ein Prozess liest dabei immer mindestens seine selbst geschriebenen Werte. Ein bereits geschriebener Wert verschwindet nicht wieder für einen Prozess und bleibt konsistent [26].

An diesen beiden Fällen wird sichtbar, wie aus diesen Sichten auf einen Datenspeicher kein Begriff von Konsistenz für ein Gesamtsystem folgen muss.

2.3 Fehlertoleranz

Das Systemziel der Fehlertoleranz gliedert sich in eine Reihe an weiteren Unterzielen und stellt insgesamt ein recht großes Forschungsgebiet der Informatik dar. In dieser Arbeit soll daher nur auf einen kleinen Auszug eingegangen werden, um die Ziele der später betrachteten Protokolle oder auch deren Grenzen zu verstehen. Dazu beitragen soll der folgende Überblick der Fehlermodelle aus [26]. Die Darstellung wurde leicht gekürzt, indem Untertypen zusammengefasst wurden, da dies für ein ausreichendes Verständnis über die Unterschiede genügt.

Ausfalltyp	**Beschreibung**
Absturzausfall (Crash Failure)	Ein Server arbeitet bis zu einem bestimmten Punkt korrekt, danach stellt er seinen Dienst vollständig ein.
Dienstausfall (Omission Failure)	Ein Server erfüllt seine Aufgaben nicht korrekt (ein- oder ausgehende Anfragen).
Zeitbedingter Ausfall (Timing Failure)	Die Antwortzeit eines Servers liegt außerhalb des festgelegten Zeitintervalls.
Antwortfehler (Response Failure)	Die Antwort eines Servers ist falsch (Wertfehler oder Zustandsübergangsfehler).
Byzantinischer oder zufälliger Ausfall (Arbitrary/Byzantine Failure)	Der Server verhält sich unvorhersehbar oder sendet widersprüchliche Informationen an unterschiedliche Teile des Systems.

Tabelle 2.2: Fehlerarten nach Tanenbaum [26]

Auch wenn bei einem Absturz meist nur ein Neustart hilft, können Dienstausfälle durch Kommunikationsprobleme oder auch in häufigen Fällen durch Softwarefehler entstehen [26]. An dieser Stelle sei ebenso kurz auf das Halteproblem verwiesen, das eine eindeutige Erkennung als Dienstausfall unmöglich macht [30]. Abstürze von Prozessen können in einem verteilten System jederzeit unerwartet passieren, wobei Redundanzen, Replikationen der Daten (Duplikation) oder Abstimmungsverfahren vor der Durchführung von Operationen Schaden abfedern können [7]. Schwieriger zu behandeln und speziell zu identifizieren sind Fehler byzantinischer Art. Ein Prozess sendet dabei zufällig oder willentlich

falsche Antworten, die jedoch nicht direkt als Fehler erkennbar sind [26]. Byzantinische Fehler spielen kurz in Kapitel 2.13.1 eine Rolle, darüber hinaus behandeln Protokolle wie Raft und Pirogue jedoch vorrangig die anderen genannten Typen.

2.4 Heartbeat

Um dem Ziel der soeben erläuterten Fehlertoleranz gerecht zu werden, können Heartbeats ein essenzieller Baustein sein. Die folgenden Erklärungen zur Fehlererkennung basieren dabei auf [26]. Das Erkennen von Fehlern jeglicher Art ist durch verschiedene Strategien und Maßnahmen sicherzustellen. Diese Strategie kann passiv oder aktiv sein. Im zweiten Fall rücken Heartbeats in den Vordergrund, bei denen Prozesse in regelmäßigen Zeitabständen ein Lebenszeichen von sich geben. Dies kann eine minimale Nachricht an einen möglichen Hauptprozess sein, die diesem den fehlerfreien Zustand bestätigt. Grundsätzlich baut dieses Verfahren, wie in der Praxis für gewöhnlich umgesetzt, auf einen Zeitüberschreitungsmechanismus auf. Sollte der überwachende Prozess nach einer vordefinierten Zeit keinen weiteren Heartbeat von einem Prozess erhalten, kann dieser von einem Ausfall ausgehen und Konsequenzen einleiten (zum Beispiel Wiederherstellungsmechanismen oder das Ausschließen aus dem Quorum einer Konsensbildung). Ein Nachteil dieser Methode können *False Positives* sein, sofern ein Client für die Bearbeitung einer Anfrage sehr lange braucht und somit der Timeout für einen Heartbeat überschritten wird. Auch in komplexeren oder schlicht deutlich größeren Netzen kann der Nachrichtenverkehr der Heartbeats zu Problemen führen. Nicht nur können Heartbeats durch Netzwerkprobleme zu einer verfälschten Fehlererkennung führen, zugleich wächst der Kommunikationsaufwand stark bei einer Vielzahl beteiligter Rechner. Eine Erweiterung durch *Gossiping* wäre hier denkbar. Rechner teilen somit nur ihren direkten Nachbarn im Netzwerk ihre Lebenszeichen mit. Bei Ausfällen kann ein Rechner dies wiederum an seine Nachbarn melden oder selbst Maßnahmen erreichen. Dieser Ansatz reduziert zugleich die Zentralisierung, sodass sich auch ein Ausfall des für die Heartbeat-Überwachung verantwortlichen Prozesses weniger kritisch kennzeichnet. Mit dem gleichen Ziel präsentiert sich *FUSE*, ein Verfahren, das Heartbeats über einen minimalen Spannbaum im Cluster verteilt. Alternativ lassen sich Fehlerdetektoren einsetzen. Diese beobachten die Zustände ihrer Komponenten oder Prozesse und führen bei Bedarf Maßnahmen durch. Auffälligkeiten neben einem Absturz sind auch ungewöhnlich hohe oder niedrige Ressourcenanforderungen. Eine mögliche Maßnahme ist eine Recovery-Prozedur der Komponente.

2.5 CAP-Theorem

Das CAP-Theorem beschreibt ein grundlegendes Trilemma der Informatik bezüglich der Softwarearchitektur in (verteilten) IT-Systemen. Hinter dem Akronym verbergen sich drei von einem System erwartete Fähigkeiten: Consistency, Availability und Partition Tolerance [28, 8]. Die Verfügbarkeit zielt auf das Ziel ab, dass eine Ressource im Falle einer Anfrage auch eine Antwort bereitstellt [28]. Die dritte Eigenschaft dient der Funktionsfähigkeit des Systems trotz einer Aufspaltung in mehrere Komponenten durch beispielsweise Netzwerkprobleme [8]. Die Kernaussage des Theorems ist die Unmöglichkeit, alle genannten Ziele gleichzeitig vollständig zu implementieren [28, 8]. Eine besonders starke Anforderung an die Konsistenz führt dabei in jedem Fall zu einer Vernachlässigung der weiteren beiden Eigenschaften. Auch das CAP-Theorem zeigt somit die Grenzen der zu untersuchenden Protokolle auf, wie sich in Kapitel 2.9 zeigen wird. Ein monolithisches System einer Datenbank kann zum Beispiel (mangels Verteilung) konsistente Daten garantieren, allerdings bei einem Ausfall keine Antworten mehr liefern und so gesehen auch nicht partitionstolerant operieren [28].

2.6 Logische Uhren

Das Schaffen einer Ordnung von Ereignissen in verteilten Systemen gilt als komplex. Mittlerweile existieren eine Vielzahl an Zeitprotokollen mit verschiedenen Einsatzzwecken, Vor- und Nachteilen. Ein Zeitstempel wirkt für uns im Alltag sehr eindeutig, kann jedoch im Kontext von verteilten Systemen viele Probleme mitbringen [13, 26]. Eine maßgebende Herausforderung stellt die stets vorhandene Verzögerung bei der Kommunikation zwischen den Rechnern dar. Logische Uhren nach Lamport sind einfache ganzzahlige Zähler, die jeder Knoten als Zustand enthält. Verschiedene Einflüsse inkrementieren diese Zähler. Dazu gehört das Eintreffen oder die eigenständige Initiierung eines Ereignisses [13]. Nachrichten zwischen den Servern beinhalten stets die eigene logische Uhr als Zusatzinformation. Erhält ein Server anschließend einen höheren Zähler als sein eigener, kann er davon ausgehen, dass die logische Zeit des Kommunikationspartners weiter fortgeschritten ist, woraufhin die eigene Uhr auf den erhaltenen Stand angeglichen wird. Diese Form der Zeitbetrachtung synchronisiert Uhren nur, sofern notwendig, das heißt, sobald eine kausale Abhängigkeit durch einen Datenaustausch zweier Prozesse stattfindet. Zwei Ereignisse in einem Prozess definieren daher eine sogenannte *Happens-Before*-Beziehung, wenn die Uhr von $a < b$. Die übliche Darstellung dieser transitiven Relation ist $a \rightarrow b$.

Dasselbe gilt für unterschiedliche Prozesse, wenn a das Ereignis des Sendens einer Nachricht und b das Empfangen dieser repräsentiert [26].

2.7 Verteilte Datenbanken und Zustandsautomaten

Häufig bringt die Literatur verteilte Systeme in den Zusammenhang mit sogenannten *state machines* (im Folgenden als Zustandsautomaten bezeichnet [26]). Für Systeme, die Zustands-behaftet arbeiten, eignet sich diese Veranschaulichung. Es handelt sich bei den hierbei betrachteten Zustandsautomaten um deterministische Automaten (DEA), sodass die gleiche Abfolge an Operationen zu dem genau gleichen Zustand führt. Diese Eigenschaft wird für gewöhnlich als Grundbedingung zur Korrektheit des Algorithmus vorausgesetzt [5]. Angenommen eine Operation führt auf einem Rechner zu einem nicht-erwartbaren, zufälligen Zustand, so würde aus einer globalen Ordnung der Operationen (siehe Daten-zentrische Konsistenz in Kapitel 2.2.1) keine Datenkonsistenz mehr folgen müssen.

Eine formale Definition des Automaten $\mathcal{A}$ enthält eine Menge aller möglichen Zustände (Q), eine Menge an verfügbaren Eingaben (Eingabealphabet Σ), einen Startzustand (q_0), eine Funktion für Zustandsübergänge ($\delta : Q \times \Sigma \to 2^Q$) sowie eine Teilmenge der Zustände als zulässige Endzustände (F) [10]:

$$\mathcal{A} = (Q, \Sigma, q_0, \delta, F) \tag{2.4}$$

Jeder Server, der daher über dieselben Befehls-Logs verfügt, bildet abschließend auch die gleichen Zustände ab. In einer (verteilten) Datenbank lässt sich jeder Wert als Zustand des entsprechenden Feldes bezeichnen [14]. In Konsensalgorithmen spricht man daher für gewöhnlich von replizierten Zustandsautomaten, da eine Menge an identischen Automaten über die Abfolge ihrer Transitionen hinweg und somit über ihre Zustände konsistent gehalten werden.

2.8 Konsensprobleme

Aus einer naiven Sicht auf ein verteiltes System stellt sich die Frage, warum ein komplexes Protokoll notwendig ist, um Daten über alle Knoten hinweg konsistent zu halten. Ein Anwender könnte genauso seinen eigenen Rechner als *Single Point of Truth* deklarieren, ein Regelsystem etablieren und jeder weitere Knoten müsste schlicht noch, womöglich in einem zeitlichen Intervall oder auf eine beliebige Art ereignisgesteuert, die Daten des zentralen Hauptknotens übernehmen. Ein solches Leader-Follower-Prinzip hat seine Daseinsberechtigung, kann jedoch genauso schnell scheitern. Der *Single Point of Truth* wird schnell zu einem *Single Point of Failure*, das heißt ein durch Angreifer verursachter oder zufälliger Ausfall des Rechners wirkt sich kritisch auf das Gesamtsystem aus (singuläre Fehlerstelle) [26, 20]. Außerdem kann der Master eines solchen Systems kompromittiert werden, woraus eine schrittweise Durchseuchung erfolgen würde, sofern dem Master blind vertraut wird. Zuletzt lässt sich feststellen, dass sich dieser Master auch ohne bösartige Absichten irren und somit unwissentlich dem Datenbestand Falschinformationen zuführen kann. Die Darstellung des deterministischen Zustandsautomaten aus 2.7 lässt sich hier mit der des verteilten Systems aus 2.1 zusammenführen: Die einzelnen replizierten Automaten kommunizieren durch das Konsensprotokoll und erscheinen im Gesamtbild als ein einziger, zuverlässiger Automat [20]. Alle hier aufgeführten Verfahren folgen den gleichen Zielen eines zuverlässigen und verfügbaren Systems, dessen Funktionalität nicht durch einzelne Knoten des Serverclusters einschränkbar ist [20]. Teils werden die dazu beitragenden Protokolle auch als essenzieller Baustein zur Entwicklung eines solchen Systems genannt. Die Protokolle weisen eine erhöhte Komplexität durch die Vielzahl an nicht-funktionalen Anforderungen auf, denen sie unterliegen. Die verursachenden Probleme sind dabei überwiegend Abstürze, Netzwerkprobleme oder sogar byzantinische Fehler [22]. Das Auftreten von Fehlern erfordert erst komplexere Lösungen, ansonsten ist das Erzielen eines Konsenses in einem (naiven) verteilten System durch einfachen Nachrichtenaustausch relativ simpel [19]. Lösungen für Konsensprobleme können mit jeweils unterschiedlichen der in Kapitel 2.3 beschriebenen Fehler umgehen. Ein Ausfällen gegenüber fehlertolerantes Verfahren (CFT) legt daher eine Obergrenze f an tolerierbaren Ausfällen fest, bei der weiterhin eine Korrektheit zu garantieren ist [19]. Eine geeignete Verbildlichung bietet ein Parlament, das zwar gewisse bürokratische Mehraufwände sowie einiges an Komplexität mitbringt, jedoch gesellschaftlich verträglichere, allgemein akzeptierte Entscheidungen trifft [14].

2.9 Konsensbildende Algorithmen

Im Allgemeinen müssen diese Algorithmen nach [7] und [19] drei Bedingungen bei der Ausführung erfüllen:

- Terminierung: Jeder korrekte Prozess entscheidet sich irgendwann für einen Wert.
- Übereinstimmung: Alle Prozesse entscheiden sich für denselben Wert.
- Gültigkeit: Wenn alle Prozesse den gleichen Ausgangswert $v \in V$ wählen, dann entscheiden sich alle korrekten Prozesse für diesen Wert v.

Korrekte Prozesse sind hierbei sehr allgemein gehalten, da Konsensalgorithmen mit verschiedenen Fehlern umgehen müssen. Die Algorithmen müssen gleichermaßen mit Ausfällen der Knoten sowie auch mit jeglichen Kommunikationsfehlern umgehen können [22].

Das byzantinische Fehlermodell erfordert eine Verschärfung dieser Definition um *korrekte* Prozesse bei dem Übereinstimmungs- und Gültigkeitskriterium [19]. Ebenso lässt sich das Kriterium der Übereinstimmung für das *k-agreement*-Problem lockern: Es existiert nun $W \subset V$ mit $|W| = k$, sodass für alle Werte einer Entscheidung $v \in W$ gilt [19].

2.10 Raft

Das Protokoll verfolgt das Ziel durch von den Teilnehmern jeweils zu replizierende Logbücher. Darüber hinaus sichert Raft seinen Anwendern einen höheren Grad an Sicherheit zu als ein Algorithmus wie Paxos [20, 15]. Folgende Erklärungen basieren dabei auf [20]. Raft definiert besonders den Leader anders: Replikationsvorgänge der Logs gehen nur von dem Leader aus statt von allen Knoten (*strong leader*). Dieser erhält sämtliche Verwaltungsaufgaben der Logs, sodass ein Datenfluss nur vom Leader zu allen anderen Servern stattfindet. Abbildung 2.3 veranschaulicht den groben Aufbau und zeigt die interne Flussrichtung der Daten vom Konsensmodul über den Log-Speicher bis hin zur Anwendung auf dem Zustandsautomaten. Auch hier ist wichtig zu betonen, dass Logs nur einzufügen sind, jedoch keinerlei Updates oder Löschvorgänge auf diesen zulässig sind. Neben der Rolle des *Leaders* existieren *Follower* und *Candidates*, von denen ein Knoten je genau eine Rolle vertreten kann. Die initiale Rolle eines Servers ist der Follower, der Heartbeats von einem Leader erhält. Bleiben die Heartbeats aus, initiiert dieser Server eine neue

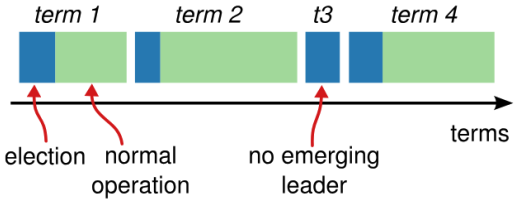


Abbildung 2.2: Verwendung der Terms in Raft [20]

Abstimmung und wechselt in die Rolle des *Candidates*, um immer die Existenz von genau einem Leader zu garantieren. Diese Bedingung wird ebenfalls durch den Einsatz von einer Art logischen Uhr, den *Terms*, erfüllt. Erhält der Candidate eine Log-Anfrage mit einem höheren Term, akzeptiert er den Sender als Leader. Im einfachsten Fall einer Wahl beantworten alle weiteren Teilnehmer die Anfrage des Candidates mit ihrer Zustimmung. Ein Server stimmt stets für die Wahlanfrage, die er zuerst erhält (*first-come-first-served*). Um eine Vielzahl an Konflikten durch veraltete Logs zu vermeiden, sendet ein *Candidate* in seiner Abstimmungsanfrage (`RequestVoteRPC`) seine Uhr (Term) mit. Eine Zustimmung wird nur erteilt, sofern der Empfänger keine höheren Terms anzubieten hat. Im schlechtesten Fall kommt es zu keiner Entscheidung und nach einem zufälligen kurzen Timeout startet eine Neuwahl. Ein neuer Leader kann Inkonsistenzen der Logs erkennen, woraufhin Follower alle Logs des Leaders übernehmen müssen.
Die Grundlage für bereits vom Leader abgesegnete Logs (commited) bilden ebenfalls die *Terms* (siehe 2.6) der Server. Die Terms werden im Vergleich zu den logischen Uhren nach Lamport allerdings hier nur bei einem neuen Wahlgang anstatt bei jedem eintreffenden Ereignis inkrementiert.
Der gesamte Prozess definiert sich wie folgt: Der Leader sendet Replikationsanfragen (`AcceptEntriesRPCs`) inklusive seines Terms, eines Commit-Index, des vorherigen Terms und dem Nachrichteninhalt an die *Follower*. Diese übernehmen die Einträge anknüpfend an den Index des vorherigen Logs (ggf. werden also Einträge überschrieben). Kleinere Terms als die des Followers führen zu einer Ablehnung der Anfrage. Die Funktionsweise der Terms als eine Art Zeit des Protokolls ist in Abbildung 2.2 visualisiert. Die Mehrheit der Teilnehmer entscheidend zum Commit einer Operation. Der Leader kontrolliert dabei die erfolgreiche Replikation auf Basis der Follower-Antworten. Nach dieser Form der Mehrheitsentscheidung kann der Leader seinen Commit-Index inkrementieren [20].

Als weiteres wesentliches Merkmal hebt sich Raft durch das Konzept des *Joint Consensus* aus, bei dem verschiedene Serverkonfigurationen jeweils mit einer Mehrheit betrieben

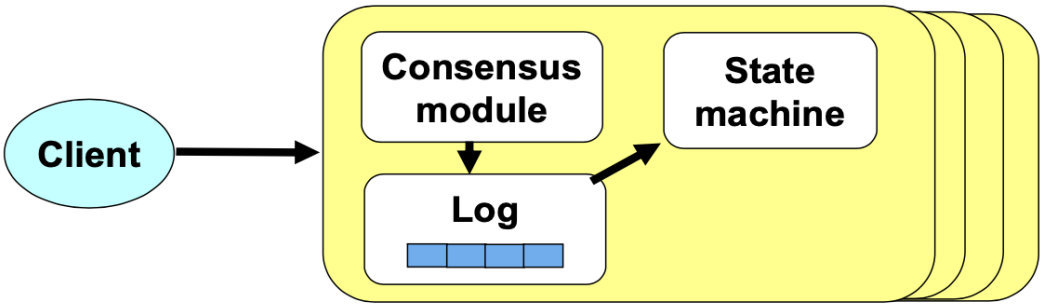


Abbildung 2.3: Grundlegende Raft-Architektur [20]

werden. Dieser Ansatz erlaubt Konfigurationsanpassungen, ohne dabei Ausfälle zu riskieren oder die Funktionsfähigkeit einzuschränken. Um trotzdem weiter Korrektheit zu gewährleisten sowie insgesamt Downtimes zu reduzieren, ist eine Mehrheit aus jeweils beiden Konfigurationen im Falle einer Transition notwendig. Die Aufgabe des Leaders darf weiterhin bei einem beliebigen Server des Clusters liegen [20].

Ein Konsensalgorithmus muss Abhilfe bei Ausfällen oder sehr großen Verzögerungen im Netzwerk schaffen. Raft stellt den Verlust von Einträgen durch die *Leader Completeness Property* sicher. Ein Server muss für einen erfolgreichen Wahlgang alle bereits als persistent festgeschriebenen Logs aufweisen. Diese Information teilt ein Knoten entsprechend mit, zudem sorgen die zuvor beschriebenen Terms für die Einhaltung dieser Eigenschaft. Knoten ohne Leader-Befugnisse stellen einen weniger kritischen Punkt dar – einzelne Verzögerungen oder Ausfälle sind wesentlich tolerierbarer. Der Leader ist mit einem Retry-Mechanismus versehen, der unbeantwortete Replikationsanfragen wiederholt sendet. Um hierdurch keine Inkonsistenzen zu erzeugen, müssen jegliche Operationen idempotent implementiert sein. Zur Sicherstellung verwendet das Protokoll Idempotency Keys (*unique serial numbers*) für die über das Netzwerk versendeten Befehle [20].

Final verzeichnet Raft nach eigener Aussage in [20] eine vergleichbare Performance mit verbreiteten Algorithmen und stellt darüber hinaus Optimierungsmöglichkeiten in Aussicht.

2.11 Pirogue

Pirogue setzt unter anderem an der Effizienz von Raft an. Zuvor skizzierte Ansätze zeichnen sich ebenfalls durch hochfrequente Kommunikation zwischen den Prozessen der Knoten aus. Ein Server kann im Normalfall Teil des Quorums zur Mehrheitsabstimmung sein oder darüber hinaus mit unterschiedlich vielen Privilegien Abstimmungen koordinieren, Commits durchführen und die Replikationsprozesse verwalten. Mindestens die Konsens-Komponente des Servers ist daher dauerhaft mit einer Vielzahl an Aufgaben beschäftigt.
Pirogue hebt dabei die Problematik vieler aktiver Server hervor, die an dem Konsens arbeiten. Eine Lösung soll ein stärkerer Fokus auf die tatsächlich aktiven Knoten bieten, wobei Prozesse sogar in einen Ruhezustand versetzt werden können, sofern keine Notwendigkeit besteht. Dieser Ansatz nennt sich *Dynamic Quorum* im Vergleich zu Rafts *Static Quorum* (feste Anzahl an konfigurierten Raft-Servern). Die Idee hierzu ist bereits älter, denn schon in [11] schlagen die Autoren ein dynamisches Abstimmungsverfahren vor, das in allen Fällen (mit mehr als vier Rechnern) eine bessere Verfügbarkeit aufweist als ein statisches. Ein dynamischer Ansatz soll vor allem eine bessere Verfügbarkeit erlauben und zu Pirogues minimalistischem Ziel beitragen, indem es für die Toleranz von n Knotenausfällen $n + 2$ Server benötigt [22] – eine geringere Anzahl im Vergleich zu statischen Konfigurationen (siehe Kapitel 2.3). Der Begriff der Dynamik meint in diesem Verfahren, dass sich das Quorum selbst anpasst und keine händische Intervention erfolgt oder erfolgen soll [11]. Um die in diesem Fall benötigten Metadaten über die Server im Cluster zu pflegen, setzt das Protokoll auf relativ einfache Datenstrukturen. Sogenannte *Cohort Sets* in Form von Bitmaps enthalten Informationen über die Zustände der teilnehmenden Server und sind genauso Bestandteil der Kommunikation zwischen diesen wie die Anfragen zum Austausch von Daten (Logs). Ein aktives *Cohort Set* definiert somit das Quorum (die Menge der abstimmenden Prozesse), sodass nur darin enthaltene Server stimmberechtigt sind. Pirogue-Cluster somit müssen somit nicht so stark skalieren, wie Raft-Cluster, um trotzdem Fehler zu tolerieren [22]. Auch der Einsatz von Witnesses spielt bei Pirogue eine Rolle und soll mit noch weniger Ressourcen vergleichbare Verfügbarkeiten bieten [22, 21].

2.11.1 Vergleich zu Raft

Die hier aufgezeigten Differenzen sind maßgeblich für spätere Erwartungen. Die Abbildung 2.4 veranschaulicht die Unterschiede im Umgang mit detektierten Ausfällen durch den Heartbeat. Während Raft seine Funktionsfähigkeit direkt überprüfen muss, schreibt der Pirogue-Leader die aktiven Server in das Cohort Set (hier C.S.) und berechnet basierend darauf die nötige Mehrheit.

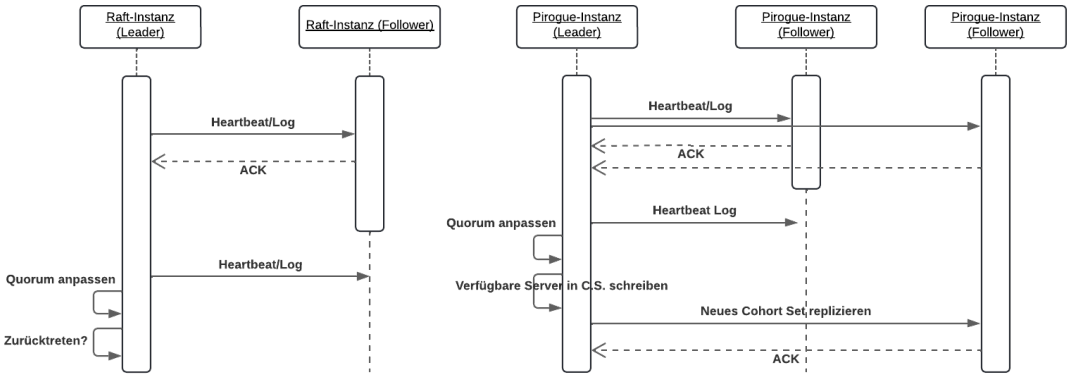


Abbildung 2.4: Reaktion auf möglichen Ausfall bei Raft (links) und Pirogue (rechts)

2.12 Commit-Protokolle

Nach den beschriebenen Konsensalgorithmen, einer Eingrenzung des Szenarios sowie umfassender Erklärungen zu den Systemzielen könnte ein Leser sich an dieser Stelle fragen, wozu die Komplexität der Algorithmen notwendig ist. In einfachen (monolithischen) relationalen Datenbanken kommen zwei- oder dreiphasige Commit-Protokolle zur Durchführung von Transaktionen zum Einsatz. Grundsätzlich ließen sich diese Konzepte auch auf eine verteilte Architektur übertragen und ggf. erweitern. Diese Protokollklasse weist jedoch wesentliche Unterschiede auf, auf die Zwecks einer Abgrenzung kurz eingegangen werden soll. Ein Zwei-Phasen-Commit-Protokoll nutzt ebenso einen Koordinator, der an ein Quorum an stimmberechtigten Prozessen einen Vorschlag sendet und diese zur Abstimmung vorstellt. Genau dann, wenn alle Prozesse diesem zustimmen, erfolgt ein Commit. Dies führt möglicherweise zu einem Abbruch (*Abort*) der Transaktion nach Ablehnung eines einzelnen Teilnehmers. Da das Protokoll weitere Operationen während der Durchführung der Abstimmung durch eine Sperre verhindert, kann der Ausfall des

Koordinators in einen Deadlock führen. Eine Weiterentwicklung bildet das Drei-Phasen-Commit-Protokoll, das eine *Pre-Commit*-Phase einführt. Durch Bestätigung eines Vorschlags können Ausfälle von Teilnehmern des Quorums oder des Koordinators erkannt werden. Diese Eigenschaften der Protokolle stammen aus [7]. Nach [19] lassen sich die Korrektheitsbedingungen (wie auch in Kapitel 2.9) hier in folgender Form aufstellen:

- Übereinstimmung: Keine zwei Prozesse entscheiden sich für verschiedene Werte (wie bei oben genannten Konsensproblemen).
- Gültigkeit:
  1. Wenn ein Prozess mit dem Wert *Abort* beginnt, dann ist *Abort* das einzig gültige Ergebnis.
  2. Wenn alle Prozesse mit *Commit* beginnen, dann ist dies das einzig valide Ergebnis (bei keinen Ausfällen).
- Terminierung: Wie bei 2.9 (stark), in schwacher Form entscheiden sich Prozesse irgendwann, sofern keine Ausfälle auftreten.

2.13 Weitere Konsensalgorithmen

2.13.1 Practical byzantine fault tolerance (PBFT)

Der letzte große Konsensalgorithmus kennzeichnet sich als weniger effizient und mit einem geringeren Durchsatz als vorherige Verfahren, löst dabei jedoch näherungsweise das Problem der byzantinischen Fehler. Diese Bezeichnung für Fehler stammt aus der Erzählung der *byzantinischen Generäle*. Dabei tauschen zwei Generäle über Boten Botschaften aus, allerdings wird einer der Boten auf dem Weg vom Feind abgefangen und mit einer manipulierten Botschaft versehen. Diese Analogie im Kontext der verteilten Systeme zielt auf das Vertrauen der Prozesse untereinander ab. Der Konsensalgorithmus arbeitet bei n Teilnehmern bei bis zu $\frac{n-1}{3}$ fehlerhaften Teilnehmern noch korrekt. Zudem heben die Autoren in [5] die Sicherheit des Mechanismus hervor, da dieser unter anderem explizit signierte Nachrichten und asynchrone Verschlüsselungstechniken verwendet. Ein Cluster enthält beispielsweise $|\mathcal{R}| = 3\mathbf{f} + 1$ Knoten, wobei $\mathbf{f}$ die Anzahl der zulässig fehlerhaften Knoten und $\mathcal{R}$ die Menge der Knoten bezeichnet [5].
Das Protokoll arbeitet in drei Phasen, um einen Datenkonsens zu erreichen. Neben einer

Prepare- und *Commit*-Phase gibt es noch eine weitere *Pre-Prepare*-Phase. Auch PBFT arbeitet durch den Einsatz einer Rolle *Primary* leaderbasiert, während die verbleibenden Teilnehmer *Backups* darstellen. Diese fungieren als Clients und senden ihre Anfragen an den Primary, der diese Anfragen daraufhin an alle weiteren Clients sendet (Multicast). Die Clients antworten nach erfolgreicher Durchführung dem initiierenden Client, der dabei auf mindestens $\mathbf{f}+1$ Antworten wartet. Die Multicast-Operation des Clients kennzeichnet den Übergang in die *Prepare*-Phase nach Erhalt einer *Pre-Prepare*-Nachricht des Primärknotens. Diese Konfiguration der Rollen bilden eine *View* mit einer (sequentiellen) Nummer v. In einer View wird der Primärknoten (*Primary*) mit dieser Nummer ausgewählt: $p = v \mod |\mathcal{R}|$. Innerhalb einer View soll stets eine totale Ordnung der Anfragen garantiert werden, daher sind zwei Vorbereitungsphasen notwendig (falls der *Primary* fehlerhaft arbeitet). Die Knoten verifizieren die Nachrichten der beiden Phasen, indem sie die zugehörigen Kennungen v der View, eine Sequenznummer i sowie einen Hash des Nachrichteninhaltes (*digest*) vergleichen [5]. Die Clients überwachen gleichzeitig den Primary durch einen internen Timer. Wird dort ein Timeout erreicht, geht der Client von einem Problem des Primarys aus und initiiert einen Wechsel in eine neue Konfiguration (View-Wechsel zu $v+1$).

2.13.2 Proof of Work & Blockchains

Konsensalgorithmen sind ein essenzieller Bestandteil von heutigen Blockchains. Unter anderem die populäre Kryptowährung Bitcoin baut auf einem Protokoll dieser Klasse auf [29]. Oftmals werden Konsensalgorithmen daher sogar nur im Kontext von Blockchains verstanden, gegenüber dieser Technologie soll sich die Arbeit jedoch abgrenzen. Dennoch soll für ein umfassendes Bild von Konsensalgorithmen ein kurzer Blick auf den stark verbreiteten *Proof of Work* geworfen werden, um die Unterschiede zu den hier im Vordergrund stehenden Algorithmen hervorzuheben.
Das wesentliche Ziel bei der Implementierung einer Blockchain ist der hohe Grad an Datenintegrität, den diese den Anwendern zusichert. Ungewollte Datenmanipulation zählt zu den größten Bedrohungen von IT-Systemen, wodurch die Nachfrage nach Technologien, die hierbei Abhilfe schaffen, besonders hoch ist [12]. Die Verteilung erhöht zudem die Ausfallsicherheit sowie die Vertraulichkeit, da einzelne Knoten im Netzwerk keinen *Single Point of Failure* darstellen [12]. Die Architektur der Technologie erfüllt somit die wesentlichen Schutzziele eines IT-Systems [6].
Der Konsensalgorithmus *Proof of Work* trägt zur Umsetzung bei, in dem Teilnehmer

des Protokolls eine anspruchsvolle Aufgabe bezüglich der Rechenleistung zu lösen haben, deren Lösung dann jedoch einfach von jedem weiteren Akteur verifizierbar ist [29, 7]. Eine solche Methode kann konkret die Berechnung eines gezielten Hashwertes sein, der zum Beispiel eine spezifische Anzahl (i.d.R. 40 [7]) an führenden Nullen enthält. Die häufige Berechnung eines Hashwertes ist dabei mit einem höheren Rechenaufwand verbunden [29]. Diese übliche Aufgabe stellt eher ein Ratespiel für die Rechner dar, anstelle eines Problems, das einen komplexen Algorithmus zur Lösung erfordert [7]. Ein hierbei erfolgreicher Knoten darf den nächsten Datenblock in die Blockchain einfügen, die weiteren Knoten verifizieren die erfolgreiche Berechnung. Das Lösen der Rechenaufgabe ist im allgemeinen Sprachgebrauch auch als *Mining* bekannt. Im Vergleich zu den bisher beschriebenen Algorithmen findet somit eine stärkere Dezentralisierung statt, da keine Prozesse mit der Rolle eines Koordinators mehr Abstimmungen koordinieren. Gerade bei Kryptowährungen schalten *Miner* ihre Rechner dem Netzwerk zu, die somit erst recht keinem zentralen Administrator unterliegen. Hier verwendete Konsensalgorithmen sind laut Lamport für den Einsatz in mäßig großen, oftmals privaten, von einer kleinen Administratorennmenge verwalteten Netzen, unter anderem bei verteilten Datenbanken, bestimmt [14]. Das Bitcoin-Netzwerk ist mit einer Vielzahl an Teilnehmern global über das Internet verteilt, sodass der Kommunikationsaufwand von Raft hier nicht praktikabel wäre (oder zumindest signifikant ineffizient) [4]. Ein weiter Aspekt, der hier diskutierte Verfahren aus der Blockchain ausschließt, ist die mangelnde Authentifizierung der Teilnehmer, sodass private Netze in der Regel nur auf diese zurückgreifen [4]. Auf den Punkt bringen lässt sich diese Differenz durch die beiden Oberbegriffe der nachrichtenbasierten sowie der nachweisbasierten Algorithmen [4]. Gegenstand dieser Arbeit sind die nachrichtenbasierten Verfahren. Ebenso sind die Finalität und das zugrundeliegende Konsistenzmodell unterschiedlich. Eine mit Raft beschlossene Operation gilt als final, sofern eine Mehrheit zugestimmt hat. Die Änderung (hier die Operation) ist von allen Akteuren zu lernen und irreversibel (dies meint nicht, dass keine inverse Operation daraufhin mehr durch eine Abstimmung beschließbar ist). Blöcke werden der Blockchain angefügt, sobald eine ausreichende Anzahl an nachfolgenden Blöcken durch den entsprechenden Konsensalgorithmus angehängt wurde. Einen abschließenden Überblick bietet die Tabelle 2.3 (Pirogue ist der Spalte mit Raft zuzuordnen):

	Paxos/Raft	**PBFT**	**Proof of Work**
Fehlertoleranz	CFT	BFT	BFT
Sicherheit	ja	ja	letztendliche Konsistenz bei Synchronizität
Authentisierung der Knoten nötig	ja	ja	nein
Durchsatz	sehr hoch	hoch	niedrig
Energieverbrauch	sehr gering	sehr gering	sehr hoch
Skalierbarkeit (große Knotenzahl)	schlecht	schlecht	gut
Formale Sicherheitsbeweise	ja	ja	nein (nur unter unreal. Annahmen)

Tabelle 2.3: Nachrichten- und Nachweisbasierte Verfahren nach [4]

3 Durchführung

Dieser Teil der Arbeit setzt sich aus der Implementierung von Pirogue nach [22], der Entwicklung einer Versuchsumgebung für beide Algorithmen und der Durchführung von Versuchen zusammen. Raft und Pirogue sollen in derselben Umgebung vergleichbar, ausführbar sein, sodass ein differenzierter Vergleich stattfinden kann.

3.1 Implementierungsentscheidungen

Während der Implementierung dienen Testdurchläufe in Form von Unit-Tests, die dabei den Netzwerkaspekt ausklammern, der Validierung. Auch verschiedene lokale Clusterkonfigurationen (1-5 Instanzen) wurden genutzt, um die Implementierungsschritte zu überprüfen. Da sich die Korrektheitsbedingungen dort gleichen, konnte eine Vielzahl an Unit-Tests aus Raft übernommen und um Fälle für Pirogue ergänzt werden. Diese Fälle betreffen besonders das dynamische Quorum, das heißt da, wo Raft die Beschlussfähigkeit nicht mehr feststellt, soll Pirogue sich nicht auf das statisch konfigurierte Quorum verlassen. Die folgenden Aspekte beschreiben konkrete Entscheidungen in der Umsetzung von Pirogue.

3.1.1 Raft

Implementierungen für Raft existieren in fast jeder Programmiersprache, auch in der hier verwendeten *golang* (https://go.dev/), sodass eine Eigenentwicklung keinen Vorteil bieten würde. Die wiederum eigene Entwicklung von Pirogue verwendet als Basis eine populäre Implementierung Rafts von HashiCorp (https://github.com/hashicorp/raft). Die Bibliothek kommt ebenfalls in *rqlite* (https://rqlite.io/) zum Einsatz, sodass die Experimente in einem praxisnahen Szenario durchführbar sind. Somit ist später ein Versuchsaufbau unter den gewünschten, nahezu realistisch gehaltenen Bedingungen umsetzbar.

3.1.2 Cohort Sets

Die verwendete Raft-Bibliothek identifiziert Server eindeutig über eine Server-ID. Diese besteht aus einem String, damit ggf. die Netzwerkadresse einen eindeutigen Identifizierer bieten kann. Die *Cohort Sets* aus Pirogue [22] und daher die Bitmaps benötigen somit einen (numerischen) Index. Die Server-ID muss kollisionsfrei auf eine Ganzzahl abgebildet werden, wozu sich eine Hashfunktion optimal eignet. Kryptografische Ansprüche bestehen nicht.

Roaring Bitmaps sind komprimierte Bitmaps, die somit nur einen Bruchteil des Speicherplatzes im Vergleich zu einer klassisch implementierten Bitmap verbrauchen. Eine einfache Funktion wie *DJB2* erfüllt in diesem Fall die Anforderungen. Die gehashten Server-IDs erstrecken sich jedoch über einen sehr großen Wertebereich für die Anwendung in Bitmaps. Die Bitmaps würden unkomprimiert zu viel Arbeitsspeicher verbrauchen und zu großen Ineffizienzen führen. In einem Test führte dies zu einem Memory Leak über mehrere GB je Pirogue-Prozess. Roaring Bitmaps finden in vielen modernen Systemen Verwendung, da sie eine effiziente Komprimierung anwenden. Die Bitmaps setzen auf eine Key-Value-Datenstruktur. Die Key-Value-Tuple bilden 32-Bit-Werte, deren Keys die gemeinsamen, vorderen (Most Signifikant) 16-Bits darstellen [16]. Die Datenstrukturen passen nun größentechnisch überwiegend in den Prozessorcache und können sehr schnell gelesen werden.

Vergleiche der Cohort Sets bei Leader-Wahlen sind relevant, um als *Candidate* ein aktuelles Cohort Set zu ermitteln. Durch Ausfälle kann es sein, dass noch nicht alle Server über das aktuelle Cohort Set verfügen, jedoch ansonsten *Follower* fälschlicherweise von Abstimmungen ausgeschlossen werden. Durch den Vergleich in der Stimmenauszählung des Candidate soll dieses Problem behoben werden [22]. Die Autoren liefern in [22] jedoch keine genauere Beschreibung dieses Verfahrens. Der Vergleich wird daher ebenfalls als eine Art Abstimmung interpretiert, bei der zudem einige Ausnahmen gelten. Der Candidate sammelt alle empfangenen Cohort Sets und führt nach jeder erhaltenen Stimme sowie nach Ablauf des Timeouts für die Wahl den Vergleich durch. Er zählt die Vorkommnisse eines Servers in den Cohort Sets. Enthalten eine Mehrheit der Abstimmenden ($\frac{n}{2}+1$) einen Server in ihren Cohort Sets, nimmt der Candidate diesen ebenfalls auf. Ein Server wird immer in das Cohort Set des Candidate aufgenommen, wenn er ebenfalls an der Abstimmung teilgenommen hat. Kann der Server auf eine Abstimmungsanfrage (`RequestVoteRPC`) korrekt antworten, darf der Candidate ihn wieder als aktiv betrachten und muss nicht auf einen weiteren Heartbeat warten. Zuletzt kann es dazu kommen,

dass ein Follower bereits ausgefallen oder nicht mehr erreichbar ist, der Candidate als nahezu einziger aber noch kein aktuelles Cohort Set erhalten hat (womöglich, da er bis kurz vor der Wahl, aber vor einem weiteren Heartbeat ebenfalls inaktiv war).

Replizierungsprozesse des Cohort Sets vom Leader dürfen nicht übermäßig durchgeführt werden. Das Axiom, dass ein Server immer Teil seines eigenen Cohort Sets ist [22, 18], muss ebenfalls explizit umgesetzt werden. In Wahlen wird dies wie zuvor beschrieben sichergestellt, beim Committen kann nach verschiedenen Ausfällen ein Server noch nicht durch einen erneuten erfolgreichen Heartbeat wieder aufgenommen sein. Vor allem der Start des Clusters muss um eine Prozedur ergänzt werden, die ein initiales Cohort Set generiert und dabei mindestens den Server selbst aufnimmt oder die Konfiguration in ein solches Set transformiert. Anfangs wurde nach jedem Heartbeat auch ein aktualisiertes Cohort Set an alle Follower übermittelt, was jedoch zu Ineffizienzen durch sehr viel Kommunikation und zu Speicherproblemen durch deutlich mehr Logs führt. Die Sets verwenden explizit keine Versionsnummern, da eine übermäßige Versionierung zu Problemen führen kann. In [18] betonen die Autoren unter anderem das Risiko eines (Integer) Overflows, sobald in Fehlersituationen Versionsnummern unerwartet schnell ansteigen. Wird die Versionsnummer des höchsten Voting-Sets plötzlich negativ, würde der aktuelle Wahlgang in einen Fehlerzustand übergehen.

3.1.3 Single-Node-Betrieb

Single-Node-Betriebe eines Clusters sollen grundsätzlich unterbunden werden. Auch Pirogue unterbindet Schreiboperationen mit einem einzigen Server [22], um Inkonsistenzen zu vermeiden. Wenn ein einzelner Server eine Menge an Schreiboperationen durchführt und dann ausfällt, kann es zu fatalen Datenverlusten kommen. Es muss allerdings eine gültige Konfiguration geben, die einen einzigen Server akzeptiert. Während das Cluster hochgefahren wird und womöglich nur ein Server verfügbar ist, allerdings bereits weitere konfiguriert werden, sollte dieser Server trotzdem zum Leader werden und die initiale Konfiguration in das Log schreiben können. An dieser Stelle ist eine Implementierung eines Wartemodus denkbar, bis alle konfigurierten Server verfügbar sind. Mit der Einführung des dynamischen Quorums aus Pirogue ergibt sich hier das Problem, dass der einzelne Leader nicht mehr zwischen der Situation einer vollständigen Isolierung (Netzwerkpartition) oder eines Betriebes als Single-Node unterscheiden kann. Falls der Leader alle weiteren Server (ob fälschlich oder korrekt) als ausgefallen identifiziert und daher aus seinem Cohort Set streicht, könnte er trotzdem eine Mehrheit bilden. Er sieht sich als

einzigen Teilnehmer des Quorums und führt einstimmig Operationen durch (nur durch seine eigene Stimme). Da dies in Testdurchläufen wie zu erwarten zu vielen Fehlern führte, wurde das statische Quorum aus Raft als Fallback eingeführt. Ein probeweise umgesetzter Wartemodus verlangsamte außerdem signifikant die Wahlprozedur, da auch hier oftmals fälschlicherweise Knoten den Leaderstatus erlangten, sich in den Wartezustand begaben und damit die Zeit, bis wieder eine Wahl stattfinden konnte, verzögerten. Das Datenbanksystem *rqlite* verfügt ebenfalls über eine Art Wartemodus, die für das Clustering in der späteren Durchführung zum Einsatz kommt (siehe 3.4).

3.1.4 Tie-Breaking

In Pirogue kann eine Konfiguration mit einer geraden Anzahl an Servern oder ein solches dynamisch entstandenes Quorum zu Pattsituationen führen. In diesen Fällen muss trotzdem weiterhin eine Leaderwahl und ein Fortschritt des Systems möglich sein. Gegebenenfalls muss die Stimme eines Servers stärker gewichtet werden können. Für dieses Privileg eines Servers muss ein immer anwendbares, geeignetes Kriterium bestehen.
Dynamic Voting, wie in [11] beschrieben, bietet die simple Erweiterung des *Dynamic Linear Voting* an, die auf eine Ordnung der Server setzt, wodurch eine Hierarchie erzeugt wird. Die Ordnung wird bei Pirogue durch die Hash-Werte der Server-IDs erzeugt. In einer Abstimmung hat die Ablehnung in einer Stimme allerdings einen logischen Grund. Beispielsweise veraltete Logs oder ein neuerer eigener Term im Vergleich zu dem des Candidates. In der Regel schlägt die Wahl zum neuen Leader an dieser Stelle fehl, ein anderer Server präsentiert sich als neuer Candidate und kann schließlich eine Mehrheit erhalten. Hierbei die Stimme des Candidate mit dem veralteten Term mehrfach zu gewichten, um eine direkte Entscheidung zu erhalten, erzeugt womöglich Inkonsistenzen oder Fehlerzustände. In einigen Testdurchläufen wurden daher bessere Ergebnisse erreicht, indem auf eine lineare Ordnung bei Leader-Wahlen verzichtet wurde. Im Fall von Commits wird im Problemfall der Index des Servers mit dem höchsten Hash doppelt gewichtet.

3.1.5 Witness

Witnesses können ein Bestandteil von einem Pirogue-Cluster sein [22]. Das Konzept wurde von den Autoren bereits vorher erarbeitet [21] und ebenso in [11] erwähnt. Im Vergleich zu einem normalen Knoten enthält er nur einen Speicher für die Logs, keinen Zustandsautomaten. Befehle werden somit nicht tatsächlich auf einen Witness angewendet, sondern

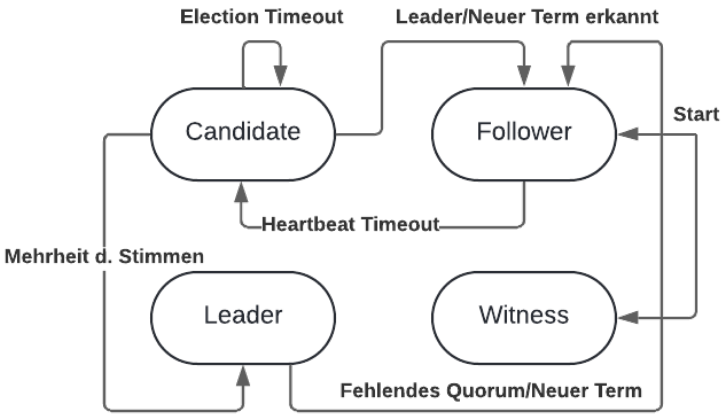


Abbildung 3.1: Zustände eines Pirogue-Knotens nach [20]

lediglich ohne Daten gespeichert. Der Witness bildet einen weiteren Zustand, neben dem Follower, Candidate und Leader. Er führt dann die normalen Operationen eines Followers durch, außer, dass er seinen Zustand nie selbst ändert und im Fall von einem fehlenden Heartbeat beispielsweise keine besondere Reaktion zeigt. Außerdem wurde auf die vollständige Umsetzung dieser Beschreibung eines Witness aus [22] verzichtet. Ein sehr kleiner Server mit nur einem leichtgewichtigen Prozess verbraucht offensichtlich weniger Ressourcen und ist weniger kostenintensiv, als ein Server mit erweiterter Hardwareausstattung, der ein Datenbankmanagementsystem ausführt. Für die Experimente wurde daher lediglich das Verhalten eines Witness implementiert, die Umgebung entspricht der gleichen, wie für alle anderen Knoten.

3.1.6 Konfigurationswechsel

Werden im Verlauf der Durchführung Konfigurationswechsel am Leader erzeugt, ist es notwendig, dass dieser sofort sein eigenes Cohort Set durch Hinzufügen oder Entfernen des entsprechenden Servers aktualisiert (direkte Schreiboperation, um Kommunikationsoverhead zu umgehen). Eine verzögerte Anpassung führte in Testdurchläufen zu einem frühzeitigen, ungewollten Rücktritt des Leaders. Kann der Leader nicht mehr das durch das Cohort Set definierte dynamische Quorum erreichen, tritt er zurück. Dies führt zu so vielen Rücktritten, dass das Erreichen eines stabilen Cluster-Zustandes stark erschwert wird. Die Wahl eines neuen oder hier ersten Leaders des Clusters scheitert dann daran, dass Follower `RequestVote`-Anfragen fast ausschließlich ablehnen, da sie den

Server nicht in ihr Cohort Set aufgenommen haben und ihn als nicht stimmberechtigt identifizieren. Außerdem führt der Abgleich der Sets in der Stimmauszählung zu einem resultierenden Set, das die Realität sehr wenig widerspiegelt. Beide Fälle lassen sich durch eine Abfrage der (statischen) Konfiguration zum Cluster-Start lösen. Die erwähnte Bootstrap-Prozedur ist unter anderem für eine formale Validierung der Konfiguration und einen initialen Commit dieser Konfiguration verantwortlich. Sie wird anfangs auf jedem stimmberechtigten Knoten ausgeführt. Das Hochfahren eines neuen Knotens oder das Wiederherstellen eines Snapshots benötigen allerdings kein hochaktuelles Cohort Set. Es wurde keinerlei Beeinträchtigung in Testdurchläufen festgestellt, bei der eine neue Instanz noch ein fast leeres Set enthält (fast leer meint das stets eigene Vorkommen im eigenen Set). Es reicht hier aus, wenn der Leader seine Auffassung kurz darauf an den Server repliziert.

3.2 Versuchsplan

Wie bereits in 1.2 erwähnt, wird nur eine Auswahl an kombinierten Faktoren durchgeführt. Im Vergleich zu einem traditionellen Vollfaktorplan testet dieser nicht alle Kombinationen der Faktoren [25]. Der nachfolgende Plan sieht fünf Faktoren mit jeweils zwei Stufen vor. Daraus würde sich ein Versuchsaufwand von bereits $n_r = 2^5 = 32$ Durchführungen ergeben [25]. Eine Alternative für einen möglichst hohen Informationsgehalt und reduzierten Aufwand [25] sind dann einige ausreichend aussagekräftige Kombinationen. Der folgende Plan prüft daher nicht alle Kombinationen der Faktoren und reduziert die Versuche auf 16. Ursprünglich sollte der Plan noch mehr Informationen liefern und daher über zehn Faktoren auf jeweils drei Stufen auswerten (CCD-Versuchsplan). Über 150 Experimente sind offensichtlich nicht praktikabel, daher erfolgte die Auswahl für die folgenden Faktoren und Stufen anhand einiger Vorabexperimente.

3.2.1 Faktoren

Pirogues Kernziele einer optimierten Verfügbarkeit sollen diese unter anderem validieren. Weitere Faktoren und deren Kombination überprüfen weiterführende Eigenschaften dieses moderneren Protokolls, um auch Aussagen über Fälle, die in [22] außer Acht gelassen werden, bewerten zu können. Die Bewertung der Auswirkungen erfolgt anhand verschiedener in 3.5 beschriebener Metriken, basierend auf den Statistiken aller Knoten.

Faktor	Stufe 1 (-)	Stufe 2 (+)
Ausfall (Crash)	10 %	20 %
Skalierung	5 Knoten	20 Knoten
Heartbeat Timeout	500 ms	2000 ms
Partitionen	1	2
Anteil Witness	0	0,5

Tabelle 3.1: Faktoren mit Stufen

Ausfall eines Knotens

In Pirogue sowie auch in Raft gehen Follower von einem Ausfall ihres Leaders aus, sobald kein Heartbeat mehr empfangen wird, wodurch die Transition zum Candidate mit einer neuen Wahl erfolgt. Nach der erfolgreichen Leaderwahl sollte der entsprechende Algorithmus wieder vollständig funktionsfähig sein und Anfragen für seinen Zustandsautomaten entgegennehmen können. Die Wahrscheinlichkeit der Ausfälle eines Followers sind ein Kernfaktor bei der Untersuchung von Protokollen, die eine erhöhte Fehlertoleranz bei Crash Failures mitbringen wollen. (siehe 2.3 sowie 2.8). Die gewählte Umgebung für den Algorithmus kann als sehr stabil angesehen werden, Ausfälle werden daher kontrolliert simuliert. Die Auswirkungen sollen sich im Vergleich daher in der Zeit, bis ein neuer Leader gewählt wurde, in der Zeit bis zur Konsensfindung und auch dem Durchsatz durch mehr Commits bzw. einen stärkeren Anstieg der committeten Indizes widerspiegeln.

Cluster-Skalierung

Auch wenn Pirogue besonders bei kleineren Clustern eine bessere Verfügbarkeit als Raft bieten möchte [22], soll die Aussage überprüft werden. Zudem wird ein Blick auf mittelgroße Cluster und die Performance gegenüber Raft in dieser Situation geworfen. Die initial geplanten großen Cluster mit 100 Knoten und mehr waren hingegen nicht praktikabel und sind kein Bestandteil der Experimente mehr. Ebenfalls werden Cluster dieser Größe in der Praxis, vermutlich aus Kostengründen und des entstehenden Kommunikationsoverheads, wenig eingesetzt. Auch die Autoren in [20] bezeichnen fünf Server als eine typische Größe. Die größere Ausprägung des Faktors der 20 Server erwies sich als durchführbar. Für Raft ist diese Konfiguration weniger geeignet, da initial keine eindeutige Mehrheit feststeht. Außerdem gilt sie als untypisch aufgrund der zuvor genannten Gründe, wurde dennoch gewählt, um feststellen zu können, ob diese Aussage auch bei Pirogue-Implementierungen zutrifft.

Heartbeat Timeout

Heartbeats laufen zunächst in Pirogue nicht anders ab als in Raft. In einem regelmäßigen, zeitlichen Intervall sendet der Leader diese an die anderen Knoten im Cluster. Die Probleme mit einem schlechten Wert sind bereits in 2.4 beschrieben und sollen nun genauer in der Praxis untersucht werden. Ein nahezu optimaler Heartbeat Timeout ist in der Theorie ausreichend lang, um möglichst weniger False Positives zu erzeugen, und ausreichend niedrig, um schnell auf Ausfälle reagieren zu können bzw. diese erst annähernd zu detektieren. In Pirogue führt eine Änderung der Reaktion auf einen Heartbeat durch einen Follower zwangsläufig zu einer Änderung des Cohort Sets, um das Quorum im Optimalfall direkt richtig abzubilden. Die hier gewählten Stufen wurden leicht unter, sowie über einem üblichen Wert von etwa einer Sekunde gewählt. Da es nicht möglich ist, einen Election- und einen Leader-Lease Timeout niedriger als den Heartbeat Timeout zu wählen, bedeutet diese Stufe stets, dass diese beiden Werte genauso konfiguriert werden.

Partitionierung

Netzwerkpartitionen sind nie vollständig auszuschließen. Die genutzten Konsensalgorithmen können dieses Problem unter vielen Umständen tolerieren und die Erreichbarkeit des Clusters prüfen oder bei dem erneuten Zusammenführen zu einem vollständigen Cluster Maßnahmen durchführen. Eine Hälfte der Server soll während der Durchführung zwischenzeitlich durch Netzwerkkonfiguration von der anderen Hälfe isoliert werden. Das Simulieren und wieder Zusammenführen der (Teil-)Cluster kommt besonders praktischen Szenarien nahe, bei dem Server einzelner lokaler Netze, beispielsweise verschiedener Unternehmensstandorte, über das Internet zu einem großen Cluster verbunden sind. Beide Protokolle prüfen ihre Quoren, um Inkonsistenzen durch dieses Problem zu reduzieren.

In der Durchführung starten die Cluster immer zunächst mit einer gemeinsamen Netzwerkkonfiguration, sodass ein Leader gewählt werden kann und Cohort Set sowie die Konfiguration repliziert sind. Zum Ende des Durchlaufes werden die Partitionen immer wieder zusammengeführt.

Anteil Witnesses

Ein Witness verwaltet Logs und Metadaten, jedoch keine tatsächlichen Daten. Pirogue behauptet, dass einige Knoten mit einem Witness ersetzbar sind und die Verfügbarkeit nicht eingeschränkt wird [22]. Witnesses tragen unter anderem weiterhin Cohort Sets zu Wahlen bei. Dieser Faktor setzt an einem Merkmal von Pirogue an, daher ist die entsprechende zweite Stufe nicht auf Raft anwendbar.

3.2.2 Teilfaktorieller Versuchsplan

In der Tabelle 3.1 steht ein + für die zweite Stufe, ein **-** für die erste.

Versuch	Ausfall	Skalierung	HB. Timeout	Partitionen	Witness
1	-	-	-	-	+
2	+	-	-	-	-
3	-	+	-	-	-
4	+	+	-	-	+
5	-	-	+	-	-
6	+	-	+	-	+
7	-	+	+	-	+
8	+	+	+	-	-
9	-	-	-	+	-
10	+	-	-	+	+
11	-	+	-	+	+
12	+	+	-	+	-
13	-	-	+	+	+
(14)	+	-	+	+	-
15	-	+	+	+	-
16	+	+	+	+	+

Tabelle 3.2: Teilfaktorieller Versuchsplan Auflösung V

3.2.3 Konstante Parameter

Die Hardwareressourcen sind in allen Versuchen konstant konfiguriert, es wird keine automatische Skalierung vorgenommen oder vergleichbare Techniken angewendet. Ein Knoten mit jeweils den beiden Prozessen läuft auf einer eigenständigen AWS EC2-Instanz. Die

Prozesse für rqlite mit Raft oder Pirogue basieren durch die Verwendung der Programmiersprache *golang* auf einem kompilierten Binary, somit kommt auch hier keine weitere Virtualisierung zum Einsatz. Auf die gleiche Art sind auch alle weiteren beschriebenen Prozesse, wie beispielsweise der Monitor, implementiert.
Die Testdaten (siehe 3.2.6) variieren nicht zwischen den Versuchen und deren Iterationen und werden an die Rechner des Clusters von der gleichen zentralen Instanz wie der Monitor gesendet. Es erfolgt ein konstantes Senden von Abfragen an die Knoten, die diese automatisch an den Leader weiterleiten. Ebenso findet ein konstantes Abfragen der Knoten für die Erhebung von Metriken statt. Verzögerungen zwischen einzelnen Abfragen ergeben sich durch die Netzwerklatenz, die daher stets mit erhoben wird.

Merkmal	**Konfiguration**
Prozessor	1 CPU Kern: Intel Xeon Haswell E5-2676 v3 oder Broadwell E5-2686 v4, bis zu 3,3 GHz Taktfrequenz
Arbeitsspeicher	1 GB
Festplattenspeicher	8 GB, SATA SSD
Betriebssystem	Canonical Ubuntu 24.04. LTS (x86)
Internetanbindung	5 Gbps (keine erweiterten Clusteringoptionen von AWS)
Region/Rechenzentrum	*eu-central-1* (Deutschland, Frankfurt)

Tabelle 3.3: Serverspezifikation nach [2]

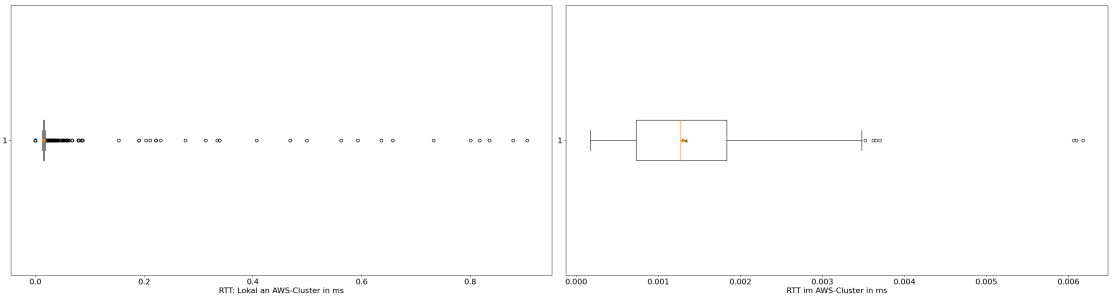
3.2.4 Störparameter

Durch den realitätsnahen Aufbau, mit einer Verteilung auf einzelne Server im Internet, wird eine höhere Latenz und ein verstärkter Nachrichtenverlust im Gegensatz zu einem lokalen Netzwerk erwartet. Eine starke Schwankung der Netzwerkbedingungen lässt sich nicht ausschließen. Da die Konsensalgorithmen TCP zur Kommunikation verwenden, ist kein vollständiger Verlust, jedoch eine längere Zeit bis hin zu einem konsistenten Zustand durch Retries der Netzwerkpakete erwartbar. Denkbar sind außerdem Performanceeinbußen gegen Abend, da Amazon zu dieser Tageszeit einen höheren Ressourcenbedarf für seine eigene Infrastruktur hat (beispielsweise mehr Amazon Prime Streaming Nutzer oder mehr Bestellungen auf der E-Commerce Plattform). Somit wäre eine schwankende Qualität der Ergebnisse, abhängig von der Zeit, denkbar. Diese Ungenauigkeiten haben sich jedoch in der Praxis nicht bestätigt. Eine testweise Durchführung von ICMP-Ping-Anfragen über einen Arbeitstag verteilt an die Instanzen eines Clusters mit drei Knoten

erzeugte sehr stabile Ergebnisse ohne die erwartbaren Schwankungen. Die Werte gelten für die Ausführung des Monitors von außerhalb des Clusters. Innerhalb des Clusters wurde ebenfalls eine vergleichbare Messung durchgeführt. Bei den extern durchgeführten Messungen können Verzögerungen toleriert werden, da die Knoten jeweils lokale Zeitstempel speichern (siehe 3.2.5). Kritisch für die Wiederholbarkeit der Experimente wäre eine hohe oder stark schwankenden Latenz innerhalb des Clusters. Doch auch hier konnte die Stabilität der verwendeten Infrastruktur bestätigt werden. In Abbildung 3.2a sind auf den ersten Blick viele Ausreißer zu erkennen, die sich jedoch mit Blick auf das Konfidenzintervall in 3.4 vernachlässigen lassen. Die für den Algorithmus relevanten Netzwerkbedingungen visualisiert der Boxplot aus Abbildung 3.2b. Die folgenden Werte sind dabei auf vier Nachkommastellen gerundet.

Art der Messung	**Mittelwert in ms**	**Median in ms**	**Konfidenzintervall 95 %**
Lokaler Rechner an Cluster	16,0071	15,2650	[15,8812; 16,1330]
Innerhalb Cluster	1,3405	1,2725	[1,3088; 1,3722]

Tabelle 3.4: RTT mit AWS EC2-Instanzen



(a) RTT: Lokal an AWS-Cluster (b) RTT innerhalb des AWS-Clusters

Abbildung 3.2: RTT Messungen bei AWS

3.2.5 Metriken

Die folgende Tabelle 3.5 listet alle Zielgrößen mit ihrer Einheit und der Art der Erfassung auf. Die Erfassungsarten sind in 3.3 verbildlicht. Knoten führen intern eine Art erweitertes Logging durch und speichern Zustandsinformationen zu unterschiedlichen Zeitpunkten persistent ab. Um ein präzises Bild zu erhalten, fragt der Monitor kontinuierlich die Knotenzustände ab.

Bezeichnung	Einheit
Durchsatz	Steigung FSM Index/Sekunde
Zeit bis neuer Leader	Sekunden
Zeit für Konsensfindung	Sekunden
Leaderwechsel	Anzahl/Minute
Terms	Steigung Term/Sekunde
Inkonsistente Zeiten	Anzahl/Sekunde
Inkonsistente Indizes	Anzahl/Sekunde
Verfügbarkeit	Anteil erreichbares Cluster/Minute

Tabelle 3.5: Zielgrößen für die Vergleichsexperimente

Die Metriken Durchsatz, Zeit bis Konsens/Inkonsistente Zeit und Zeit bis neuer Leader sollen die Performance des Systems darstellen können. Ein optimales System verarbeitet sehr viele Commits, nimmt nahezu keine Zeit für Leaderwahlen in Anspruch und ist nahezu nie inkonsistent (siehe 2.2). Die Anzahl der Leaderwechsel werden eher als Hilfsgrößen verstanden, um Auffälligkeiten der anderen Metriken zu erklären.

Berechnung der Metriken

Die genannten Metriken basieren auf Berechnungen der kontinuierlichen Status-Abfragen aller Knoten. Die entsprechende API antwortet dabei mit ihren eigenen Statistiken, die jeweils lokal erfasst wurden. Somit wird auch der Fehler bei Zeitmessungen sehr deutlich reduziert: Ein Knoten speichert Zeitstempel ab, wenn Ereignisse, wie die Wahl eines neuen Leaders, eintreten. Hierdurch entfällt eine komplexere Messung vom externen Monitor, bei der dieser ansonsten RTTs für die Abfrage des Status (in diesem Fall *Leader oder Follower?*) einberechnen müsste. Fällt der Knoten nur für eine sehr kurze Zeit aus, kann die Information mit dieser Methode in der nächsten Abfrage nachgeliefert werden. Der resultierende Durchsatz lässt sich über die Experimente hinweg vergleichen, da stets mit dem gleichen Durchsatz an Anfragen über Schreiboperationen gearbeitet wird. Ausschlaggebend ist für diese Zielgröße daher schlicht der größte Index, der zu dem letzten Zeitstempel der Iteration als committed gilt, abzüglich des kleinsten Indexes zum ersten Zeitpunkt der Durchführung. Analog dazu ergibt sich der Anstieg der Terms. Die Berechnung der Zeit, bis ein neuer Leader gewählt wurde, basiert auf dem Wert `leader_appended_time`, den jeder Knoten speichert, sobald er in seine Logs die Wahl

des neuen Leaders geschrieben hat und diesen somit als Leader des Clusters anerkennt. Da ein Knoten ebenfalls stets die Adresse seines Leaders speichert, kann für jede unterschiedliche Adresse aller Knoten die Zeitdifferenz berechnet werden. Auf eine ähnliche Art und Weise ergibt sich die Zielgröße der Dauer bis zur Konsensfindung, bei der die Zeitdifferenz, bis ein Index von der Mehrheit als committed gilt, betrachtet wird. Für jeden committeten Index für jeden Knoten wird die aktuelle Zeit dieses Knotens mit den anderen dieses Indexes verglichen.

Die Feststellung der allgemeinen Verfügbarkeit eines Clusters findet auf Basis der gesammelten Zeitstempel mit einem Intervall von einer Sekunde statt. Dies ist damit die Genauigkeit der Verfügbarkeitswerte. Spätere Auswertungen in 4.1 zeigen, dass diese Genauigkeit ausreicht, um Unterschiede zwischen den Protokollen beobachten zu können. Ein Cluster gilt hierbei als verfügbar, sofern die beiden folgenden Kriterien gleichzeitig erfüllt sind:

1. Es existiert mindestens ein Leader. Bei mehr als einem Leader muss es zwei Teil-Cluster aufgrund von Netzwerkproblemen geben. Verfügbar sind diese Teile somit jedoch weiterhin.

2. Es existiert ein abstimmungsfähiges Quorum. Das heißt, Raft muss mindestens die statisch konfigurierte Anzahl an aktiven Followern für eine Mehrheit aufweisen, Pirogue die dynamisch erforderliche (mindestens ein zusätzlicher Follower).

Inkonsistenzen sind etwas schwieriger auszuwerten, daher wird auf einen monotonen Anstieg der Indizes sowie inkonsistente Zeitspannen geachtet. Für den ersten Fall werden alle Indexwerte zeitlich sortiert. Steigen die Indizes dabei nicht monoton an, spricht dies dafür, dass einige Einträge aufgrund von Konflikten zurückgerollt werden mussten. Der zweite Fall wertet aus, über welche Zeitspannen (ebenfalls anhand der Zeitstempel der Knoten) Unterschiede bei den Indizes der Logs einzelner Knoten auftreten, so gesehen noch keine Überstimmung im Sinne des Konsenses erzielt wurde. Diese Berechnung ist daher eine Art Gegenprobe zur Zeitspanne, bis ein Konsens für einen Log-Eintrag erzielt wurde. Alle weiteren Werte, die sich auf einen Zeitabschnitt beziehen, wurden in Intervallen von einer Minute ausgewertet und dann im Durchschnitt pro Sekunde berechnet. Somit wurde von der Dauer der Durchführung über viele Stunden profitiert und eine möglichst hohe Vergleichbarkeit der Ergebnisse erzielt.

3.2.6 Testdaten

Die Art der Daten spielt für die später folgenden Auswertungen keine relevante Rolle. An der Datenbank, die das Konsensmodul des entsprechenden Algorithmus dann enthält, können beliebige Datenmodifizierungen durchgeführt und ausgewertet werden. Dazu zählen in diesem Fall SQL-Befehle, die Tabellen und Datensätze hinzufügen, aktualisieren oder entfernen. Für eine beliebige, aber sehr regelmäßige Ausführung von Leseoperationen ist der globale Monitor (siehe 3.3) verantwortlich, um ebenfalls Daten über die Konsistenz des Gesamtsystems zu erheben.

3.3 Deployment

Der Versuchsaufbau für jedes Experiment wird automatisiert hochgefahren. Das heißt, je erforderlicher Knoten im aktuellen Versuch wird eine EC2-Instanz erzeugt und konfiguriert. Auf diesen Knoten wird dann jeweils eine Instanz des rqlite-Systems mit Raft und eine mit Pirogue installiert und gestartet. Rqlite unterstützt automatisches Clustering, sodass alle Prozesse auf allen Rechnern ohne manuellen Eingriff ein Raft- oder Pirogue-Cluster bilden können. Dazu bekommt jeder Prozess die Anzahl der konfigurierten Server und die Adressen (IPv4 oder hier bei EC2 den öffentlichen DNS-Namen) inklusive des Ports übermittelt. Das automatische Clustering (https://rqlite.io/docs/clustering/automatic-clustering/) wartet, bis alle konfigurierten Server initialisiert sind, und führt dann eigenständig eine Leaderwahl durch.
Die Umsetzung dieses Deployment- und Installationsprozesses ist mit dem Tool *Terraform* für *Infrastructure as Code* (IaC) realisiert. Diese Lösung von HashiCorp erlaubt eine Beschreibung der benötigten Infrastruktur mit zugehörigen auszuführenden Skripten. Die Entscheidung für AWS fiel aufgrund der dort angebotenen Flexibilität. Es können beliebig viele Instanzen je nach aktuellem Versuch gestartet und beliebig über das Internet verteilt werden. Ein realistischer Versuchsaufbau ist so möglich. Für die Ausführung genügen EC2-Instanzen mit minimalen Anforderungen, sodass diese Flexibilität kostengünstig bis kostenlos ist. Besonders bei mehreren Versuchen, die eine häufige Durchführung und eine wechselnde Infrastruktur erfordern, muss mit Terraform keine aufwändige manuelle Konfiguration der Infrastruktur vorgenommen werden. Terraform bietet nativ sehr guten Support für AWS, daher wurde sich zugunsten dieses Tools entschieden. Sollte aus Kostengründen doch ein anderer Provider anstelle von AWS gewählt werden, kann dies umkonfiguriert werden. Die hauseigene Lösung CloudFormation ist hingegen nur für die

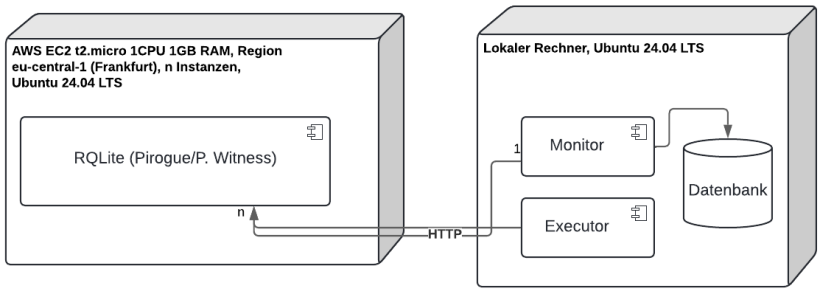


Abbildung 3.3: Verteilungssicht Versuchsaufbau

Konfiguration von AWS-Stacks entworfen und somit durch diese mangelnde Flexibilität wenig geeignet, auch wenn die Einstiegshürde bei Terraform aufgrund dessen neuer Konfigurationssprache *HCL* (HashiCorp Configuration Language) im Vergleich zu JSON oder YAML bei CloudFormation höher liegt. Abschließend erlaubt diese Technologiewahl eine einfache Reproduzierbarkeit der Experimente für einen beliebigen Informatiker. Auf den Instanzen selbst wird keine weitere Virtualisierung (z.B. durch Docker) eingesetzt, da ein Overhead an Netzwerkkonfigurationen eingeführt werden würde, der den Clustering-Prozess verlangsamen könnte. Während der automatisierten Einrichtung der Server werden diese mit einem Shell-Skript bespielt, das zeitgleich mit den Algorithmen gestartet wird. Es bekommt als Startparameter die Ausfallwahrscheinlichkeit für das jeweilige Experiment übergeben, um daraufhin zufällig sowie gleichzeitig Raft und Pirogue abzuwürgen. Nach einer kurzen Verzögerung startet das Skript beide Prozesse mit der gleichen Konfiguration erneut. Die Verzögerung bewegt sich dabei im Bereich weniger Minuten, da bei einem zu schnellen Start dieser Störkomponente das Clustering teilweise verhindert wurde. Während Timeouts, Witnesses oder die Cluster-Größe Teil der initialen Konfiguration sind, erweist sich die Simulation der Netzwerkkonfigurationen als weniger trivial. Die Durchführung erfolgt mittels AWS Security Groups des VPC-Service. Ebenfalls mit Terraform kann eine angepasste Netzwerkkonfiguration, die das Cluster in zwei Teilcluster unterteilt, zur Laufzeit eingespielt werden. Eine entsprechende Eigenschaft erlaubt nur noch Konfigurationen innerhalb der Partition, die den einzelnen Knoten anhand ihres Index zugeteilt ist. Ein Beispiel dafür ist in A.2 angehängt.

3.4 Ablauf

Jede Versuchskonfiguration nach 3.2.2 wird auf einer nur dafür angelegten Infrastruktur durchgeführt. Das zuvor beschriebene automatische Clustering sorgt stets dafür, dass alle relevanten Prozesse (nahezu) zeitgleich beginnen. Das Cluster führt initial automatisch eine Leaderwahl durch und repliziert Cohort Sets sowie die Cluster-Konfiguration.
Der Aufbau wird dabei zu unterschiedlichen Tageszeiten erzeugt (und nach Abschluss wieder heruntergefahren), um den Fehler durch unterschiedliche Netzwerkauslastungen zu minimieren. Die Durchführung erstreckt sich insgesamt über einen Zeitraum von 09:00 - 00:00 Uhr ME(S)Z. Neben dem Monitor-Prozess läuft auf dem gleichen Rechner ein Prozess, der für das Senden der Testdaten an die Knoten im Cluster verantwortlich ist. Er sendet über HTTP kontinuierlich Anfragen für Schreiboperationen an das Cluster (das Load Balancing von rqlite leitet diese an den Leader weiter). Die entsprechende API ist mit keinen Rate-Limits oder ähnlichen Sicherheitsmechanismen versehen. Dieser Server führt zudem in derselben Frequenz einen Ping an alle weiteren Server durch, um später den Fehler der Latenz in der Auswertung besser beachten zu können. Der Sende-Prozess wird als *Executor* bezeichnet. Der Monitor als Beobachter des Gesamtsystems fragt (ebenfalls über HTTP) kontinuierlich alle Server an. Für jedes Experiment, das heißt für jede zugehörige AWS-Infrastruktur, existiert auch ein separater Monitor (und Executor), die stets in eine eigene Tabelle einer SQLite-Datenbank auf dem gleichen Rechner schreiben. Die Schreibgeschwindigkeiten dieser Datenbank sind ausreichend schnell für die Abfragefrequenz und zudem nach eigenen Angaben bis zu 35 % schneller als Schreiboperationen über das Dateisystem auf die lokale Festplatte [1]. Die Abfragen der Knoten-APIs über HTTP sind über Threads mit der Messung des RTT über ICMP synchronisiert. Es kann daher davon ausgegangen werden, dass die Abfragen meist alle $\approx 15,9 - 16,1$ ms stattfinden (siehe Tabelle 3.4 und Abbildung 3.2a). Die berechneten Metriken unterliegen daher auch dieser Ungenauigkeit.

4 Auswertung

Dieses Kapitel formuliert für die Faktoren jeweils Erwartungen mit anschließender Beobachtung und Diskussion der Ergebnisse. Für das Aufbereiten der Daten wird Python 3 genutzt. Python bietet eine Vielzahl an Statistik-Bibliotheken an, die den Umgang mit Messdaten im Vergleich zur vorher verwendeten Golang vereinfachen, da über die Adressierung vom Speicher abstrahiert wird. Alle konkreten Hilfsmittel für statistische Berechnungen und Visualisierungen sind im Anhang A.1 aufgeführt. Da durch die in 3.2 geplanten Versuche sehr viele Daten erzeugt werden, befinden sich diese zum Großteil im Anhang A.4 und A.5.

4.1 Ausfall eines Knotens

Sobald es vermehrt zu zufälligen Ausfällen der Knoten zur Laufzeit der Experimente kommt, könnten Wahlen zum Leader in Pirogue schneller als in Raft erfolgen. Häufige Ausfälle sollten somit in Raft stärkere Auswirkungen auf die Leistung des Systems haben. Das System kann erst mit der Benennung eines Leaders Anfragen von Clients verarbeiten, sodass durch die Minimierung der Wahlen ein System mit der Nutzung von Pirogue schneller einsatzbereit sein dürfte, um somit auch mehr Anfragen über die gesamte Laufzeit zu verarbeiten. Auch dass Pirogue insgesamt auf die Zustimmung von weniger Servern setzt, kann ein Vorteil sein. Einige Knoten sind während der Wahl ausfallbedingt nicht im Cohort Set abgebildet, sodass der Candidate auf weniger Antworten von anderen Servern warten muss. Durch häufige Ausfälle und häufiges Neustarten der Prozesse kommt es zugleich zu vermehrten Änderungen des Cohort Sets. Dadurch ist es auf der anderen Seite denkbar, dass dies zu mehr Kommunikation der Pirogue-Prozesse führt, die daher langsamer werden könnten. Wenn sich die Ausfälle häufen, könnte das ständige Aktualisieren dieser Metadaten die Verarbeitung der eigentlichen Nutzdaten verlangsamen. Dies führt zu der Annahme einer höheren Initialisierungszeit bei Pirogue gegenüber Raft.

Sollte dies der Fall sein, würde Pirogue einen geringeren Anstieg der Commits aufweisen. Wie in Kapitel 3.1.2 erläutert, findet in jeder Abstimmung über einen neuen Leader auch zugleich eine Abstimmung über das aktuelle Cohort Set statt. Die entsprechende Prozedur könnte im Vergleich zu Raft jedoch ebenfalls die Wahlen verlangsamen. Auch wenn diese eine theoretisch effiziente Laufzeit aufweist, findet eine weitere Kommunikation über das Netzwerk statt. Raft führt hingegen keinen Kommunikations-Overhead durch laufende Ausfälle ein.

Eine hohe Ausfallwahrscheinlichkeit der Knoten könnte in Pirogue zu einer besseren Verfügbarkeit führen. In [22] analysieren die Autoren diese These bereits umfangreich. Dass ein Cluster trotz vieler Ausfälle länger Anfragen verarbeiten kann, indem die Definition für ein gültiges Quorum umformuliert wird, ist dabei einleuchtend (siehe 2.11). Immer kleiner werdende Cluster könnten wiederum das Risiko mitbringen, dass es bei einem vollständigen Ausfall des Systems zu Datenverlusten kommt. Auch wenn im weiteren Verlauf Knoten wieder neu gestartet, hergestellt und dem Cluster zugeführt wurden, können diese sich bereits in sehr stark abweichenden Zuständen befinden. Mindestens muss nach der Wiederherstellung eines Knotens der Leader seine aktuellen Logs an diesen replizieren – oder es muss noch erst ein neuer Leader bestimmt werden. Während Raft in solchen Szenarien viel Sicherheit mitbringt und ggf. keine Anfragen mehr verarbeiten würde, dürfte dieses womöglich riskante Verhalten Pirogues ebenso zu mehr Inkonsistenzen führen. Kann der Raft-Leader sein Quorum nicht mehr erreichen, tritt er zurück. Insgesamt muss später abgewägt werden, ob die bessere Verfügbarkeit mehr Vorteile bietet oder doch mehr Fehler verursacht.

4.1.1 Beobachtung

In vielen Versuchen entspricht die Beobachtung sehr deutlich den Erwartungen. Im Vergleich zu denen bei Raft sind Pirogues Wahldauern mehrheitlich etwas oder sogar deutlich geringer. Die Streuung der Werte fällt dabei sehr unterschiedlich aus. Größtenteils kann Pirogue viel schnellere Leaderwahlen aufweisen, beispielsweise in einem kleinen Cluster mit wenig Ausfällen, Netzwerkpartitionen und Witnesses. Während Raft hier durchschnittlich 70 Sekunden benötigt, dauern die Wahlen im Mittel bei Pirogue nur rund 27 Sekunden. Der Median verweist ebenfalls auf eine höhere Differenz zugunsten Pirogues, der in etwa einer Sekunde einen Leader bestimmt, im Vergleich zu etwa 32 bei Raft. Die zugehörigen Konfidenzintervalle (Mittelwert, 95 %) drücken eine große Streuung aus, da diese bei beiden Protokollen minimal in den negativen Bereich und wiederum bis etwa

147 Sekunden (Raft) bzw. etwa 67 Sekunden (Pirogue) reichen. Die Abbildung 4.1 zeigt diesen Versuch. Hierbei muss allerdings beachtet werden, dass über die genauen Wahlen keine Metriken erfasst werden (können), sondern nur der Zeitpunkt, zu dem Knoten den Leader in ihre Konfiguration übernehmen, existiert (siehe 3.2.5).
Betrachtet man ein ebenso kleines Cluster (5 Knoten), mit zugleich einer Unterteilung in zwei Teilcluster durch die Netzwerkbedingungen und ohne weitere Einflüsse, ist Pirogue erneut deutlich schneller. Die Erwartung, dass Pirogue performanter sein könnte, dürfte sich hier bestätigen lassen. Auf der anderen Seite fällt bei Pirogue aber eine Vielzahl an Ausreißern auf, die bei Raft etwas geringer ausfallen. Diese Ausreißer lassen sich hier vernachlässigen, da 95 % der Werte von Pirogue zwischen rund 4-10 Sekunden liegen.
Um die schlechtere Performance des ersten Beispiels zu erklären, lässt sich ein Blick auf ein Cluster aus 5 Knoten mit nur Witnesses oder nur langsameren Heartbeat Timeouts betrachten. Letzteres zeigt eine größere Verteilung der Werte durch einen sehr großen Interquartilsabstand, der sich sogar über wenige Minuten erstreckt. Der Fall der Witnesses kann kein funktionsfähiges Pirogue-Cluster erzeugen und bringt sehr viele Ausreißer für Raft hervor, wobei weiterhin fast alle Werte zwischen 17 und 18 Sekunden liegen. Längere Zeitspannen, bis eine Mehrheit den Leader anerkannt hat, zeigen sich in Pirogue, sobald im kleinen Cluster neben vielen Ausfällen und langsamen Heartbeat Timeouts auch Witnesses eingesetzt werden. In diesem Versuch ist zudem die Streuung der Werte für beide Algorithmen relativ groß und reicht sogar bis rund 200 Sekunden. Das System ist also durch die starken Ausfälle teils wenige Minuten inoperabel. Wird in einem ähnlichen Versuch nun stattdessen eine Netzwerkpartition simuliert, liegt die Mehrheit der Werte nur noch im niedrigen Sekundenbereich.
Insgesamt sind vergleichbar schnelle Wahlen deutlicher zu erkennen, sobald 5 Knoten mit einer höheren Ausfallwahrscheinlichkeit konfiguriert ist. In diesen Fällen sind die Werte bei Pirogue jedoch zugleich stärker gestreut. Unabhängig von den Ausfällen und anderen intuitiv kritischen Faktoren sind in kleinen Clustern schlechtere Wahlergebnisse durch Witnesses erkennbar.
Eindeutig schnellere Abstimmungen zum Leader (bzw. das Akzeptieren der Follower eines neuen Leaders) sind bei großen Clustern für Pirogue auffällig. Beispielsweise liegen mit Netzwerkpartitionen und Witnesses die durchschnittlichen Dauern für Raft bei rund 42 Sekunden und für Pirogue nur bei 11 Sekunden. Auch in einem großen Cluster mit einem langsameren Heartbeat Timeout und einer höheren Ausfallwahrscheinlichkeit als einzig erhöhte Faktoren sind zum Beispiel rund 11 Sekunden im Fall von Pirogue und rund 69 Sekunden bei Raft erkennbar. In diesen Versuchen ist zudem die Streuung stets sehr gering, was auch das kleine Konfidenzintervall, in dem 95 % der Werte liegen, bezeugt.

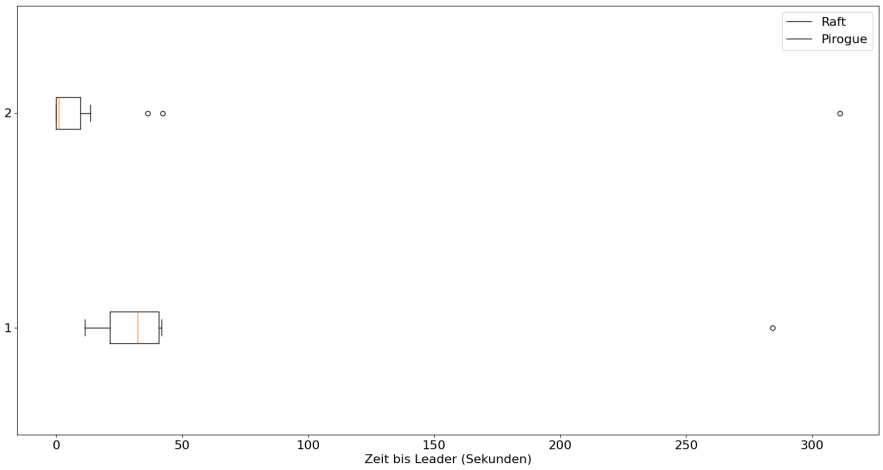


Abbildung 4.1: Leaderwahlen im zeitlichen Vergleich (5 Knoten, Partition, Witnesses)

Dass die Wahlen aber im Zusammenhang mit einer größeren Ausfallwahrscheinlichkeit noch schneller verlaufen, zeigt sich nur in Einzelfällen, wie beispielsweise bei dem zuvor erwähnten großen Cluster. Im Gesamtbild wird auch in Pirogue der Wahlvorgang stärker durch Ausfälle beeinträchtigt, sodass es vermehrt zu extremen Ausreißern mit einer längeren Zeit kommt. In großen Clustern zeigen sich zwar auch teils in Zusammenhang mit Witnesses viele extreme Ausreißer, diese können hier allerdings auch mit Heartbeat Timeouts in einen Zusammenhang gebracht werden, da sich beispielsweise 20 Knoten mit vielen Ausfällen und Witnesses (sowie einer Clusterteilung) diesbezüglich stabiler verhalten.

Da sich bisher nur auf die Dauer der Wahlen bezogen wurde, sollte zusätzlich kurz die Dichte der Wahlen betrachtet werden, um ein Gesamtbild zu erhalten. Insgesamt sticht durch die zusätzlichen Witnesses nun eine wesentlich geringere Streuung der Werte hervor. Zusätzlich ist die Häufigkeit der Leaderwechsel zu betonen, sobald Witnesses eingesetzt werden. Insgesamt kommt es zu weniger Leaderwechseln bei Raft, wobei hier längere Wahlen auch zu häufigeren Wahlen passen. In einem kleinen Cluster mit Witnesses treten Leaderwechsel in Pirogue etwas häufiger auf. Die zuvor auffälligen Versuche mit einer Netzwerkteilung zeigen bei Pirogue hier keine Auffälligkeiten. In einem großen Cluster mit Witnesses und einer 20%igen Ausfallwahrscheinlichkeit wechselt Pirogue nicht deutlich häufiger den Leader als Raft. 95 % der Werte bewegen dabei im Bereich von etwa

15,4-17,0 Sekunden bei Raft und 15,7-17,4 Sekunden bei Pirogue. Die Wahldauer bei Pirogue liegt nur etwa bei der Hälfte von der bei Raft.

Die erwartete Cluster-Verfügbarkeit mit der Verwendung von Pirogue zeigt sich besonders stark in nahezu allen Versuchen. Diese Beobachtung bestätigt die Theorie und die ersten Untersuchungen der Verfügbarkeit der Autoren aus [22]. Das erste Quartil der Verfügbarkeit von Pirogue setzt dabei teils erst an dem dritten Quartil von Rafts gemessener und errechneter Verfügbarkeit an. Dies zeigt zum Beispiel Abbildung 4.3a. Der Interquartilsabstand ist im Fall von Pirogue deutlich größer, mit der hohen Dichte an Ausfällen dieses Versuches, bleibt im Median jedoch bei über 60 %, Raft hingegen kann nur etwa die Hälfte bieten. Oftmals werden die Unterschiede sogar noch eindeutiger: Beispielsweise zeigt Abbildung 4.3b, dass Raft nur etwa 60 % Verfügbarkeit bieten kann, während sich Pirogue im Bereich über 75 % bewegt. In beiden Fällen gibt es jeweils einen Ausreißer nach oben und unten, jedoch liegt der niedrige dabei im Fall Raft nur noch bei 50 %, während Pirogue nicht unter 75 % sinkt. Die Beobachtung trifft daher auf kleine und große Cluster gleichermaßen zu. Die Cluster mit Pirogue können überwiegend (Median) eine vollständige Verfügbarkeit mit nur einer geringen Streuung nach unten bieten. Ein kleines Cluster mit vielen Ausfällen sticht besonders hervor, da sich die Werte hier über den kompletten Wertebereich von einem vollständig ausgefallenen Cluster bis zu einem einwandfreien bewegen. Während Rafts Interquartilsabstand allerdings dabei von 0 bis nur knapp zu einer 50 %igen Verfügbarkeit reicht, bewegt sich dieser bei Pirogue zwischen rund 50-80 %. Diese Beobachtung bekräftigt auch das Konfidenzintervall, bei dem 95 % der Werte von Raft zwischen rund 27-31 % liegen, wogegen dieses Intervall bei Pirogue von rund 61-64 % reicht. Die Verfügbarkeit ist somit überwiegend doppelt so hoch bei Pirogue.
Ein ähnlich aufschlussreiches Bild erzeugt der Versuch mit zusätzlicher Netzwerkpartition und Witnesses. Raft kann insgesamt kaum ein funktionsfähiges Cluster bieten, ausgenommen sind vereinzelte Ausreißer bis hin zu einer vollständigen Erreichbarkeit. Während der Interquartilsabstand nur bis knapp 20 % und das vierte Quartil bis knapp 50 %, erstrecken sich die Werte von Pirogue zwar erneut über die vollständige Skala, kennzeichnet dabei seinen Median aber bei über 60 %. Auch an dieser Stelle liegen 95 % der Raft-Werte nur zwischen rund 11 % und 14 %, während Pirogues zwischen rund 59-64 % zu verorten sind. Die Erwartung von einem Overhead durch viele Nachrichten bei Ausfällen in Pirogue lässt sich somit aus der Beobachtung nicht bestätigen.

Die Erwartungen, dass Pirogue mit häufigeren Ausfällen einen besseren Fortschritt erzielen könnte, zeigt sich in der Beobachtung wiederum nicht eindeutig. Die Steigung der

committeten Indizes verhält sich hier in beiden Algorithmen sehr ähnlich. In Einzelfällen kann Pirogue minimal bessere Durchsätze aufweisen, in anderen Versuchen schneidet Pirogue etwas schlechter ab. Etwas höher ist der Durchsatz für Pirogue in einem kleinen Cluster mit vielen Ausfällen, Teilung des Clusters und Witnesses sowie in einem großen Cluster mit vielen Ausfällen und einer Teilung. Ebenso verzeichnet Pirogue in einem großen Cluster mit vielen Ausfällen und Witnesses im Gesamtbild einen etwas besseren Durchsatz gegenüber Raft. Für gewöhnlich liegt die durchschnittliche Steigung jedoch bei wenigen Logs pro Sekunde. Beispielsweise liegt bei fünf Knoten mit 20 % Ausfallwahrscheinlichkeit der Großteil der Werte für beide Protokolle zwischen 1,7 (bzw. 1,6 bei Pirogue) und 2 Log-Einträgen pro Sekunde. Wie auch bei den Leaderwahlen sind die großen Cluster mit vielen Ausfällen Beispiele für eine leichte Verbesserung. Etwas geringere Anstiege der Commits sind nicht eindeutig einem Faktor zuzuordnen, da diese sowohl bei den Versuchen im kleinen Cluster mit zusätzlichen einem höheren Heartbeat Timeout als auch bei 20 Knoten mit einem höherem Heartbeat Timeout mit zwei Teilclustern erkennbar sind. Bei dieser Betrachtung sticht ein kleines Cluster mit vielen Ausfällen und Netzwerkpartitionen sowie langsamen Heartbeat Timeouts hervor: Offenbar konnte in keinem Durchlauf ein stabiles Cluster erzeugt werden, sodass keine Commits in diesen Versuchen zu erzeugen waren. Die Abbildung sowie eine zugehörige Tabelle mit den Messwerten fehlt dadurch für diesen Versuch auch im Anhang. Bezüglich der hier betrachteten Indizes lässt sich außerdem beobachten, dass die Interquartilsabstände sowie auch die ersten und vierten Quartile in beiden Protokollen sehr ähnlich sind – selbst die Ausreißer sind vergleichbar. Der Median der Steigungen ist dann in den meisten Versuchen minimal niedriger eingezeichnet gegenüber Raft. Darüber hinaus sind einige sehr große Ausreißer, insbesondere bei Experimenten mit 20 Knoten und zugleich zwei Teilclustern, bemerkenswert: In mehreren Fällen kann ein einzelner Wert sehr weit außerhalb der Quartile beobachtet werden, der beschreibt, dass offenbar vereinzelt ein schlagartiger Anstieg stattgefunden haben muss. Ein Beispiel zeigt die Abbildung 4.2, wo der entsprechende Wert über 200 liegt, während sich die Mehrheit der Werte im einstelligen Bereich aufhält. Alle Versuchsergebnisse zeigen bezüglich des Durchsatzes kleine Konfidenzintervalle um den Mittelwert sowie auch den Median, sodass die grafisch sichtbaren, einzelnen Ausreißer zu vernachlässigen sind. Zuletzt sind zwei Versuche in dieser Hinsicht auffällig: Mit fünf und auch mit 20 Knoten, langsamen Heartbeats, Netzwerkpartition und Pirogues Witnesses ist die Steigung im Mittel höher als in anderen Versuchen. Diese Konfigurationen stachen bereits bei den Leaderwahlen heraus.

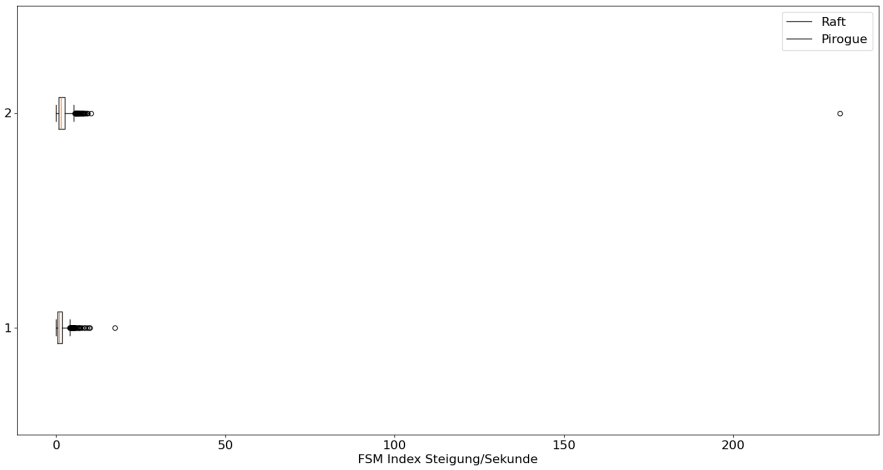
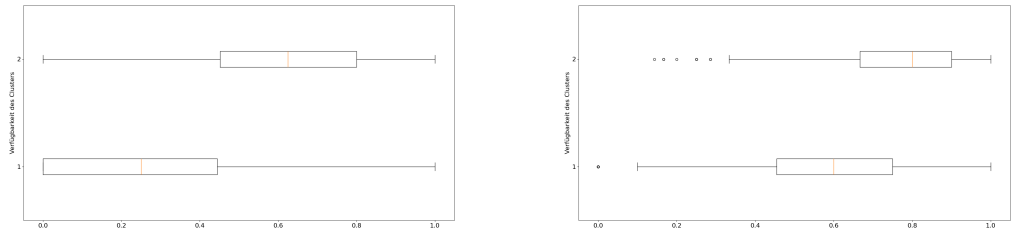


Abbildung 4.2: Extremer Ausreißer bei Netzwerkpartition und 20 Knoten



(a) Verfügbarkeit bei 5 Knoten (20 % Ausfallwahrscheinlichkeit)

(b) Verfügbarkeit bei 20 Knoten (10 % Ausfallwahrscheinlichkeit)

Abbildung 4.3: Verfügbarkeiten der Cluster

Ebenso wurde erwartet, dass die Beschlüsse mit einem teils wesentlich kleineren Quorum bei Pirogue zu mehr Inkonsistenzen führen könnten. Diese Behauptung zeigt sich nicht in den Daten, in einigen Versuchen ist sogar das Gegenteil der Fall. Ein Großteil der Versuchsergebnisse weist sehr ähnliche Werte für inkonsistente Zustände des Gesamtsystems auf. 5 Knoten mit einer niedrigen Frequenz der Heartbeats, Netzwerkpartition und Witnesses führen zum Beispiel zu etwas mehr Problemen bei Pirogue. Ohne Partitionierung und Witnesses oder nur mit Partitionierung, aber ohne Witnesses fallen diese Probleme wiederum bei Raft etwas stärker aus. Hier zeigt sich bereits, dass in dieser Hinsicht Witnesses einen Nachteil darstellen dürften.

Zuletzt soll in Hinblick auf die Instanzen noch kurz der Anstieg der Terms betrachtet werden, der genauere Schlüsse zu den Leaderwahlen zulassen kann. Hier bewahrheitet sich die Vermutung, dass Pirogue teilweise mehr Wahlen durchführt und sich häufiger fehlgeschlagene Wahlen ergeben. Teilweise steigen die Terms pro Sekunde häufiger an, als dass überhaupt eine vollständige Wahl stattfindet. Ein Blick auf ein großes Cluster mit 10 % Ausfallwahrscheinlichkeit zeigt für Pirogue einen Anstieg um rund 10 Terms pro Sekunde, für Raft mit rund 5 nur die Hälfte. In vielen Fällen steigen die Terms pro Zeiteinheit (hier Sekunden) in Pirogue etwas stärker. Auch wenn der Median nur um einige Terms pro Zeiteinheit über dem von Raft liegt, streckt sich besonders das obere Quartil des Boxplots im Fall von Pirogue deutlich weiter nach oben. Hiermit lässt sich der stärkere Anstieg der Terms in Pirogue und damit eine höhere Frequenz an Wahlen und somit auch Problemen mit dem aktuellen Leader-Knoten feststellen. Hier kann nur der stärkere Anstieg der Terms erkannt werden – wie zuvor beschrieben, wechseln die Protokolle dabei aber etwa gleich oft ihren Leader.

4.1.2 Diskussion

Die Beobachtung von schnelleren Leaderwahlen in Pirogue im Vergleich zu Raft kann, aber muss kein Indikator für eine bessere Performance von Pirogue in einem kleinen sowie auch großen Cluster sein. Wie oben bereits erwähnt, ist der Algorithmus durch einen Leader einsatzbereit und kann Anfragen verarbeiten, Befehle für seinen Zustandsautomaten entgegennehmen, Logs replizieren oder auch Konfigurationen einschließlich des Cohort Sets verwalten. Um die Auffälligkeiten zwischen den kleinen partitionierten Clustern zu erklären, sollte jedoch der Anstieg der Terms hinzugenommen werden. Der viel schnellere Anstieg der Terms im Pirogue-Protokoll ruft Zweifel hervor, da sich der Term einer Wahl ebenfalls erhöht, sobald eine Wahl fehlgeschlagen ist. Auch in Raft steigen im Mittel die Terms jedoch stärker pro Sekunde gegenüber der Dauer der Leaderwahlen. Dies lässt auf eine Vielzahl an fehlgeschlagenen Wahlgängen in beiden Algorithmen, aber vermehrt in Pirogue schließen. Eine mögliche Erklärung lässt sich mit der Verwaltung der Cohort Sets und der Art der Entscheidung über das aktuelle Quorum für eine Wahl finden. Kürzlich wiederhergestellte Knoten können zugleich ein altes Cohort Set beisteuern. In der Wahl werden dann nicht korrekt die tatsächlich aktiven Knoten abgebildet. Somit kommt es vermehrt zu Fehlern bei Pirogue. Dass Raft jedoch Fehler in einer gleichen Größenordnung von etwa 5-15 wirft, lässt darauf schließen, dass diese nicht vollständig auszuschließend sind. Sind die gemessenen Zeiten für eine Leaderwahl nun also

sehr kurz, treten aber gemeinsam mit einem starken Anstieg der Terms auf, ist es denkbar, dass Leader zwar gewählt werden, jedoch kurz darauf wieder zurücktreten müssen. Dies kann besonders dann passieren, wenn eine kleine Partition entsteht, die dann auch noch überproportional viele Witness-Instanzen enthält. Der Candidate wird dann zum Leader, da durch die Witnesses keine weiteren Candidates zur Verfügung stehen, muss dann aber feststellen, dass kein Quorum erreichbar ist oder die Logs gegenüber denen der Witnesses veraltet sind. Da die Witnesses nur über Metadaten verfügen, kann keine Replikation ihrerseits stattfinden. Das Protokoll muss dann so lange warten, bis ein normaler Follower wiederhergestellt ist. Datenverluste sind dann nicht auszuschließen. Darüber hinaus ist es möglich, dass die in 4.1 gezeigte Beobachtung durch eine noch viel höhere Anzahl an Fehlern in Raft verfälscht ist. Wenn das kleine Raft-Cluster durch die Partitionierung einen Großteil der Zeit nicht betriebsfähig ist und keine Wahl stattfindet, basiert die Darstellung womöglich auf zu wenigen Messwerten. Insgesamt sind die Anzahl der Wechsel in Raft in beiden Fällen sehr ähnlich – diese Werte sollen auch nicht auffällig abweichen, da Raft keine Witnesses unterstützt und der Aufbau daher bei beiden Experimenten identisch ist. Für Pirogue lässt sich hingegen eine höhere Anzahl der Leaderwechsel beobachten, die auf den Verdacht der fehlerhaften Wahlen aufgrund der Schwierigkeit, überhaupt einen Leader zu bestimmen, zurückzuführen sind. Dies gilt auch mit Blick auf die Witness-Knoten, die nicht als Candidates für den Posten des Leaders zur Verfügung stehen.

Die Verfügbarkeit der Cluster durch die Verwendung der unterschiedlichen Algorithmen ist hier nur knapp zu diskutieren, da sich das Paper zu Pirogue vor allem um diese These dreht. Dass Pirogue besser erreichbar sein würde, ist dort ausführlich gezeigt. Dieses Merkmal lässt sich durch das dynamische Quorum einfach erklären, da der Definition nach ein Leader seltener zurücktreten muss und in der Gesamtheit eine geringere Anzahl an Knoten für eine Mehrheitsentscheidung notwendig ist.

Der auffällige schlagartige Anstieg der Commits aus der Grafik 4.2 kann mehr über das Verhalten der Algorithmen im Zusammenhang mit den Partitionen verraten. Eine denkbare Erklärung wäre die abschließende Zusammenführung der Teilcluster am Ende des Versuches. Der hohe Wert ist allerdings nur bei Pirogue deutlich erkennbar. Rafts einzelner höherer Ausreißer lässt sich nicht eindeutig demselben Szenario zuordnen, da dieser um ein Vielfaches kleiner ist. Pirogue könnte während der Teilung weiterhin zwei lauffähige kleinere Cluster betrieben haben, die nach dem Merge ihre gemeinsamen Commit-Indizes synchronisiert haben. Das könnte den schlagartigen Anstieg erklären. Raft dürfte währenddessen jedoch kaum Fortschritt erzielt haben, wodurch der Anstieg nicht auf-

tritt. Fraglich wäre dabei wiederum der trotzdem durchschnittlich vergleichbare Anstieg bei Raft, der sich somit nur auf die kurze Zeit des vollständigen Clusters beziehen kann. Zuletzt könnte hier die Vermutung aufgestellt werden, dass sich Raft aufgrund der sehr langen Problemsituation vollständig abgeschaltet hat. Somit wären keine ansteigenden Indizes beobachtbar gewesen. Unabhängig von der Abbildung zeigen die genauen Daten einen durchschnittlich doppelt so hohen Anstieg bei Pirogue – jedoch liegt dieser bei rund 2 (statt 1) pro Sekunde und ist damit nur sehr niedrig. Inkonsistenzen bewegen sich auf dem gleichen, niedrigen Niveau. Abschließend lässt sich festhalten, dass Pirogue im Zusammenhang mit der erweiterten Cluster-Verfügbarkeit schneller Entscheidungen treffen kann. Es kommt zwar oft zu einer stärkeren Streuung bei den Messdaten, jedoch nicht zu mehr Inkonsistenzen im Vergleich zu Raft.

4.2 Cluster-Skalierung

Bezüglich dieses Faktors wird angenommen, dass der Kommunikations-Overhead von Pirogue durch Cohort Sets bei einem größeren Cluster stärker ins Gewicht fällt. Ein größeres, global verteiltes Cluster, das dadurch mit höheren Latenzen beansprucht wird, dürfte häufiger inkonsistente Zustände aufweisen als Raft, da die Verwaltung des dynamischen Quorums den Fortschritt dieses größeren Systems aufhält. Die Größe des Clusters bildet eher die Basis, erkennbare Unterschiede zwischen den Protokollen werden daher umso mehr im Zusammenhang mit Netzwerkpartitionen erwartet (siehe 4.4).

4.2.1 Beobachtung

Die Erwartung, dass Pirogue besonders bei größeren Clustern häufiger Inkonsistenzen verzeichnen könnte, bestätigt sich nicht und schließt sich damit an die Beobachtungen der Inkonsistenzen in Zusammenhang mit der Ausfallwahrscheinlichkeit an. Auch bei 20 Knoten unterscheiden sich die Probleme nicht deutlich, die meisten Konfigurationen verhalten sich sehr ähnlich. Dennoch gibt es einige Ausnahmen, sobald die inkonsistenten Indizes betrachtet werden. Bei 20 Knoten mit vielen Ausfällen und teilweise zusätzlich langsamen Heartbeat Timeouts, Witnesses und Cluster-Teilungen sowie insgesamt bei großen Clustern mit Witnesses zeigen sich viele hohe Ausreißer bei Pirogue, die somit hier auch zu einer größeren Streuung führen. In kleinen Clustern gab es diese Auffälligkeit jedoch nicht, Pirogue verursachte oft weniger Fehler.

In ersten Auswertungen hat sich gezeigt, dass Pirogue nur im Zusammenhang mit Witnesses einen besseren Durchsatz erzielt, mit mehr Versuchsdurchläufen konnte das allerdings nicht mehr bestätigt werden. Die inkonsistenten Zeitabschnitte sind bei Pirogue nicht signifikant größer als bei Raft. In einigen Fällen sind diese sogar kürzer. Das Konfliktpotential als Trade-off mit Pirogues verbesserter Verfügbarkeit kann hier nicht eindeutig beobachtet werden. Mehrheitlich gibt es kaum Unterschiede bei den beiden Algorithmen hinsichtlich dieser Zielgröße.

Pirogue bezieht sich in [22] vor allem auf kleine Cluster in Hinblick auf einen schonenderen Umgang mit Ressourcen und dem Ziel von *Green Computing*. Sollte ein größeres Cluster unumgänglich für den Anwendungsfall sein, kann auch hier den Beobachtungen aus 4.1 zufolge von der Verwendung des Protokolls profitiert werden. Die Autoren beweisen dieses Merkmal zunächst theoretisch in [22], indem sie eine durchschnittliche Ausfall- sowie Wiederherstellungsrate eines Knotens annehmen. Die Ausfälle der Experimente überschneiden sich daher mit den Annahmen der Autoren. Eine Infrastruktur mit einer sehr hohen Verfügbarkeit, wie es hier bei AWS ohne die simulierten Störungen der Fall wäre, könnte somit auch noch bessere Ergebnisse erzielen.

4.2.2 Diskussion

Die Beobachtungen bezüglich dieses Faktors zeigen erneut, dass sich Annahmen über einen zu hohen Kommunikationsaufwand oder darüber hinaus eine Beeinträchtigung durch die Verwaltung des Cohort Sets nicht bestätigen lassen. Die abweichenden Ergebnisse für die Cluster-Skalierung der einzelnen Versuche sind daher durch andere Faktorkombinationen zu erklären. Auf die Einflüsse durch eine Netzwerkpartition soll in Abschnitt 4.4 noch genauer eingegangen werden.

Mehr Erklärungen erfordert hingegen der direkte Vergleich der inkonsistenten Zeiten, bei denen Pirogue nicht signifikant schlechter auffällt. Dies dürfte zum einen an den Testdaten liegen, die sich zwar durch die (pseudo-)zufällige Generierung unterscheiden, jedoch wenige direkt im Konflikt stehende Schreiboperationen beinhalten. Es wird beispielsweise keine Tabelle von einem Prozess erzeugt und von einem anderen daraufhin wieder gelöscht. Zuletzt ist es sehr wahrscheinlich, dass die bessere Verfügbarkeit Pirogues auch auf die Aufrechterhaltung der Konsistenz einzahlt. Da die wiederhergestellten Knoten zunächst die Commits des Leaders wieder übernehmen, ist das Konfliktpotential durch die reduzierten Follower verkraftbar. Die wenigen Knoten reichen aus, um sicher Commits

durchzuführen, ohne durch Ausfälle kritische Verluste zu erzeugen. Dies wäre besonders der Fall, wenn nur noch ein Follower zur Verfügung stünde, der dann auch noch ausfiele. Dieses Szenario trat wahrscheinlich so selten auf, dass die dadurch entstandenen Probleme im Gesamtbild keinen großen Unterschied erzeugen.

4.3 Heartbeat Timeout

Eine höhere Frequenz des Heartbeat vom Leader könnte in Pirogue mehr Probleme verursachen als in Raft. Mögliche Probleme für Pirogue bei einem hochfrequenten Heartbeat reihen sich in die Erwartungen bezüglich der Verlangsamungen durch einen erhöhten Bedarf an Kommunikation ein. Werden zu viele False Positives durch einen schnellen Heartbeat erzeugt, führt dies zu vielen weiteren Logs, in denen das Cohort Set aktualisiert wird. Auch dies würde sich in einem vergleichsweise geringerem Durchsatz widerspiegeln. Sollte Pirogue trotz (mutmaßlicher) Ausfälle eine bessere Verfügbarkeit des Clusters bieten können, ist es weiterhin denkbar, dass diese übermäßig viele Leaderwahlen und -wechsel verursachen, die den Systemfortschritt ausbremsen.

Auf der anderen Seite wäre auch ein deutlich besserer Umgang mit den vielen False Positives Pirogues denkbar. Sollten die Leaderwahlen sehr schnell ablaufen, könnte eine geringere Beeinträchtigung als bei Raft bestehen. Auch ist hier denkbar, dass es keinen erkennbaren Unterschied zwischen den Algorithmen gibt, da die beschriebenen Probleme sehr minimale Auswirkungen haben sollten.

4.3.1 Beobachtung

Vergleicht man die Zeiten, bis ein Algorithmus einen Eintrag committed hat und somit zu einem Konsens gekommen ist, bei allen Versuchen, die eine geringere Heartbeat-Frequenz (zwei Sekunden) verwenden, fallen einige Unterschiede auf. Nicht immer ist Pirogue deutlich schneller oder langsamer. In drei Versuchen gleichen sich die Zeiten sehr stark bei Raft und Pirogue. Die Streuung der Werte ist dabei ebenfalls sehr ähnlich, einzelne Ausreißer lassen sich etwa gleich häufig erkennen. Diese sehr ähnlichen Ergebnisse weisen eine Gemeinsamkeit auf: Die Pirogue-Cluster setzen alle Witnesses ein. Die anderen Faktoren sind dabei sehr unterschiedlich, auch kann dieser Zusammenhang nicht bei allen Versuchen mit Witnesses bei Pirogue identifiziert werden. Allgemein ist davon auszugehen, dass der Heartbeat keinen klaren Einfluss auf die Durchführung hat. Die Nebeneffekte sind

möglicherweise minimal und treten sehr unterschiedlich im Zusammenhang mit anderen Faktoren auf, sodass sich im Gesamtbild der Einfluss schwer bis gar nicht klar beobachten lässt. Auch in dieser Beobachtung verzeichnet Pirogue überwiegend größere Streuungen der Werte als Raft. Besonders, wenn noch eine Netzwerkpartition hinzukommt, bewegen sich sogar einige Werte im Sekundenbereich. Der Großteil bewegt sich hingegen insgesamt im Bereich von 0,02 bis 0,04 Sekunden. Die Abweichungen durch eine Veränderung des Heartbeat Timeouts in diesem Maße sind somit abschließend wenig herausragend. Auch die Erwartung, dass sich viele False Positives ergeben, die eine negative Auswirkung auf den Fortschritt des Systems haben, ist nicht deutlich geworden. Der Durchsatz mit Hinblick auf dieses Timeout ist sehr ähnlich, auch wenn es einige kleine Schwankungen gibt. Diese Ergebnisse können ebenfalls die These, dass sich der Kommunikationsaufwand in Pirogue zu sehr erhöht, widerlegen, so wie bereits in Abschnitt 4.2 und 4.1 beschrieben.

4.3.2 Diskussion

In der Beobachtung wurde bereits kurz angeschnitten, dass konkrete Zusammenhänge und auch Kipppunkte bei der Konfiguration des Heartbeat und in Kombination mit anderen Faktoren kaum identifizierbar sind. Eine Ursache könnte sein, dass der Heartbeat Timeout immer zusammen mit dem *Election Timeout* und dem *Leader Lease Timeout* konfiguriert wurde. Da alle diese Parameter stets den gleichen Wert aufweisen, um weiterhin eine gültige Konfiguration für die Protokolle zu erzeugen, ist es ebenso schwer, Einflüsse durch einen dieser Parameter zu identifizieren. Auch wurden die beiden Stufen dieses Faktors nur knapp um den Standardwert festgelegt. Um Parameter dieser Art besonders zu untersuchen, würde es sich anbieten, in Zukunft einen speziellen Versuchsplan mit verschiedenen Stufen für die angesprochenen Timeouts zu entwickeln und sich nur auf Versuche, die mit verschiedenen Timeouts arbeiten, zu konzentrieren. Dies könnte eine weiterführende Arbeit im Vergleich der Protokolle sein.

4.4 Partitionierung

Das dynamische Quorum könnte hier zum Nachteil werden, da eine Netzwerkpartition schlechter als eine solche identifiziert wird. Wie in 3.1.3 unter anderem erläutert, wurden Fallback-Mechanismen implementiert, die Alleingänge von Servern mit dem Risiko eines

hohen Datenverlustes unterbinden. Pirogue geht bei einem kleinen Teilgraphen von Ausfällen der anderen Cluster-Teilnehmer aus und bleibt zunächst funktionsfähig, während ein Raft-Cluster womöglich nicht mehr beschreibbar wäre. Der andere vollständige Teilgraph des Clusters könnte währenddessen auch weiter (Schreib-)Operationen durchführen. Ein erneutes Zusammenführen der Partitionen erzeugt bei Pirogue dann womöglich große Inkonsistenzen und Fehler. Hingegen wäre trotzdem ein besserer Fortschritt bzw. Durchsatz des Pirogue-Clusters erwartbar, da hingegen Raft gar keine Append-Entry-Anfragen mehr entgegennimmt. Pirogue geht so gesehen mit einer erhöhten Risikobereitschaft an das Problem. Erwartbar wäre zudem, dass dieses Verhalten bei einem kleineren Cluster zu einem besseren Durchsatz und weniger Inkonsistenzen bei Pirogue führt. Ist eine Minderheit isoliert, könnten die Datenverluste minimal ausfallen, da die kleinere Partition keinen nennenswerten Fortschritt erzielt. Bei der größeren Skalierungsstufe des Clusters ist ein größerer Verlust denkbar, der sich durch vermehrte Inkonsistenzen und zuletzt auch wieder einen geringeren Durchsatz zeigen würde. In dem Fall des kleinen Clusters mit Partitionierung könnte die ausgewertete Steigung der Terms im gesamten Experiment höher sein, jedoch keine Verbesserungen durch beispielsweise schnellere Wahlen und schneller verfügbare Cluster mitbringen.

4.4.1 Beobachtung

Einige Beobachtungen im Zusammenhang mit diesem Faktor sind bereits in Abschnitt 4.1 erläutert. In Einzelfällen bringt ein Cluster mit Pirogue, das netzwerkbedingt in zwei Teile unterteilt ist, in verschiedener Hinsicht bessere Ergebnisse hervor. Beispielsweise zeigt sich ein besserer Durchsatz bei fünf Knoten mit Partitionierung und einer geringen Heartbeat-Frequenz. In dieser Konfiguration ist außerdem die Zeit, bis ein Konsens erzielt wurde, geringer als bei Raft. Auch weniger Leaderwechsel und weniger Inkonsistenzen verzeichnen die Daten in diesem Versuch. Bei den Inkonsistenzen gibt es aber eine stärkere Streuung, wie auch zuvor schon bei einem Versuch mit Partitionierung erläutert. Auch ein kleines Cluster mit Witnesses, Aufteilung und vermehrten Ausfällen bringt bei Pirogue eine schnellere Konsensfindung, einen leicht erhöhten Durchsatz sowie etwas weniger Inkonsistenzen mit. In diesem Fall treten jedoch wieder mehr Leaderwechsel auf, was mit den Witnesses in Zusammenhang gebracht werden kann. Ein umgekehrtes Verhalten zeigt ein wiederum großes Cluster mit langsamen Heartbeats und der Partitionierung, genau wie ein großes Cluster mit vielen Ausfällen und zusätzlich dem Ersetzen von Knoten durch Witnesses bei Pirogue. Gerade dieser letzte Versuch erzeugt bei Pi-

rogue besonders viele Leaderwechsel im Zusammenhang mit den Ausfällen. Insgesamt bildet dieser Versuch aber auch eine Ausnahme, da große Cluster mit vielen Ausfällen mehrheitlich viele Fehler bei der Verwendung von Raft hervorrufen. Die inkonsistenten Indizes sind extrem gestreut und reichen teils bis in eine Größenordnung von 1000 pro Minute. Im Normalbetrieb liegt dieser Wert in vielen Ergebnissen unter eins, sodass hier von einer Ausnahmesituation durch ein komplett disfunktionales Raft-Cluster ausgegangen werden kann. Sobald Pirogue derart viele inkonsistente Indizes aufweist, sind die inkonsistenten Zeitabschnitte jedoch weiterhin nur sehr knapp gehalten. Werden nur die Zeiten bis zum Konsens im Vergleich betrachtet, fallen ähnliche oder sogar minimale Verbesserungen bei Pirogue auf. Das Gegenteil zeigt sich dann oftmals im Zusammenhang mit einer Partitionierung.

4.4.2 Diskussion

An dieser Stelle ist die Aufmerksamkeit vor allem auf die Untersuchungen bei Ausfällen (siehe hierzu auch in Abschnitt 4.1) zu richten, mit der die Beobachtung hier besonders in Verbindung gebracht werden kann. Aus der Sichtweise der Prozesse gleicht die Teilung des Clusters einem Szenario von vielen Ausfällen. Ein Unterschied kann sich dann ergeben, wenn beide Teile weiter arbeiten und am Ende wieder zusammengeführt werden. Besonders hier fällt noch einmal auf, dass Pirogue kaum mehr Commits durchführen kann, was sich zudem im Anstieg der Commit-Indizes widerspiegeln würde. Hieraus kann geschlussfolgert werden, dass die Verfügbarkeit in einem solchen Szenario zwar mit Pirogue hochgehalten wird – was womöglich in einigen Einsatzszenarien Vorteile bieten kann. Jedoch ist kein klarer Vorteil erkennbar, da vermutlich durch das Zusammenführen der Cluster ein Großteil der Commits aus der Zeit der Aufteilung wieder zurückgesetzt werden muss. Umgekehrt wäre auch denkbar, dass der Ausreißer mit dem kurzen schlagartigen Anstieg hier für einen Vorteil von Pirogue spricht – vorausgesetzt, dass der Merge erfolgreich war und alle Zustandsautomaten die Commits wieder übernehmen konnten. Der durchschnittliche Anstieg bleibt natürlich gering, da es sich um einen kurzen Anstieg handelt, der in der Menge der Daten nicht mehr herausragt. Hierzu müssten einmal die absoluten Werte dieser Experimente betrachtet werden. Einige Stichproben dieser absoluten Werte aus der Datenbanken können diese Vermutung bekräftigen, wie die folgende Tabelle 4.1 verdeutlicht (die Differenz der Indizes bewegt sich in derselben Größenordnung, allerdings sind die Werte aufgrund anderer Startweite in der zweiten Zeile insgesamt höher). An dieser Stelle kann zuletzt ebenfalls der Versuch 14 mit fünf

Knoten, einer hohen Ausfallwahrscheinlichkeit von 20 % und der Partitionierung (sowie einem höheren Heartbeat Timeout) erwähnt werden, der in keiner Iteration und zu keiner Zeit ein lauffähiges Cluster erzeugen konnte. Die Ausfälle traten vermutlich immer so häufig auf, dass anfangs nie ein initialer Leader bestimmt werden konnte. Nach einiger Zeit kann davon ausgegangen werden, dass das Cluster dann abgestürzt ist.

Iteration	**Finaler FSM Index Raft**	**Finaler FSM Index Pirogue**	**Differenz**
1	25966	75376	49410
2	41937	102864	60927

Tabelle 4.1: Stichprobe: Finale Indizes

Ebenso bestätigt Abbildung 4.4 den Verdacht, dass Pirogue weiterhin Fortschritte machen kann, während das Raft-Cluster offensichtlich keine Schreiboperationen in dem dargestellten Zeitabschnitt mehr entgegennehmen konnte. Die Vermutung, dass Pirogue viele Datenfehler erzeugt, indem es während der Partitionierung weiter Anfragen verarbeitet, bestätigt sich nur in Einzelfällen. Pirogue kann trotz dieser Probleme mehr Anfragen als Raft unter schweren Netzwerkbedingungen und so gesehen auch viele Ausfälle verarbeiten. Hierbei sollte außerdem weiter untersucht werden, inwieweit sich diese Probleme bei sehr stark im Konflikt zueinander stehenden Anfragen entwickeln. Der Versuch mit vielen Ausfällen und der Partitionierung zeigt, dass es in Hinblick auf große Cluster einen Kipppunkt geben könnte, ab dem Pirogue besser abschneidet.

Eine bessere Konsensfindung und eine Handhabung dieser Netzwerkprobleme führt bei Pirogue zu weniger Problemen, als in Raft. Ein bessere Durchsatz konnte ebenfalls gezeigt werden. In Verbindung mit Witnesses konnten teilweise Probleme festgestellt werden, auch vereinzelte deutliche Ausreißer der sonst besseren Konsensfindung wurden deutlich.

4.5 Anteil Witness

Witnesses als leichtgewichtige Knoten sollen genauso zu der Verfügbarkeit beitragen wie normale Knoten, die nicht nur Metadaten mitschreiben [22, 21]. Diese These kann sich bestätigen, sie wirkt jedoch zunächst kontraintuitiv: Sobald ein höherer Anteil an Witness-Knoten konfiguriert ist, stehen zugleich weniger Instanzen als Candidates bzw. als Leader

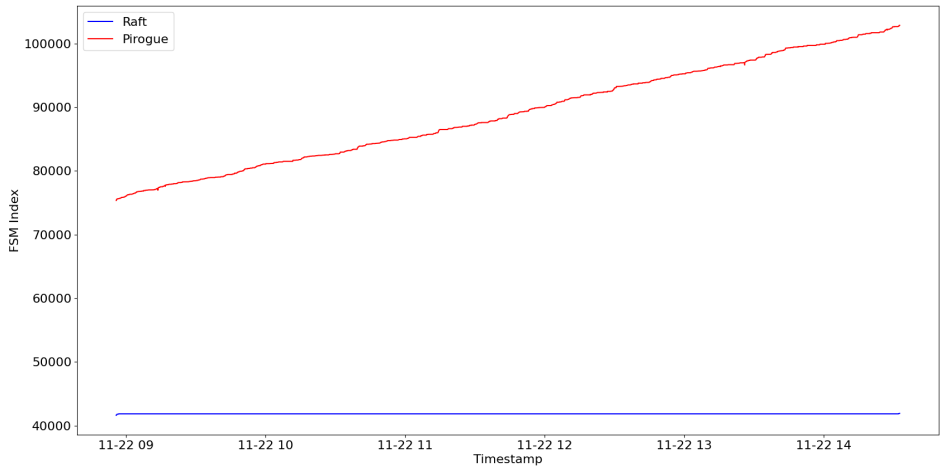


Abbildung 4.4: Auszug: Vergleich der comitteten Indizes im zeitlichen Verlauf (9.00 - 14 Uhr MEZ)

zur Verfügung. Das Quorum ist in diesem Fall zwar gleich groß, Ausfälle könnten das Cluster jedoch stärker einschränken, da die Witnesses immer in ihrem Zustand bleiben. Auch mit einer Netzwerkpartitionierung könnte dies zum Problem werden, sobald mehrere Witnesses in einer Partition vertreten sind – eventuell sogar mehr Witnesses als normale Follower (und der Leader). Dieser Teil des Clusters wäre dann durch den fehlenden Leader handlungsunfähig. Ein Vorteil wird somit nur erwartet, sobald in einem großen Cluster mit einem Anteil von Witness-Knoten gearbeitet wird. Kleine Cluster dürften sich bei weniger Ausfällen genau wie Raft, bei vermehrten Ausfällen sogar schlechter als Raft verhalten. Durch das zeitweise Fehlen des Leaders bleibt ein Fortschritt ganz aus und der gemessene Durchsatz am Ende des Experiments wäre abermals geringer. Auch kann dieser Leader-lose Zustand die Anzahl der fehlgeschlagenen Wahlen stark erhöhen, was wiederum die Steigung der Terms ebenso stark erhöhen würde. Die gemessenen Inkonsistenzen dürften in diesem Fall geringer ausfallen, was jedoch dann täuschen würde, da insgesamt nur deutlich weniger Logs übertragen wurden.

4.5.1 Beobachtung

Zusammenhänge zwischen der Konsenszeit und dem Einsatz von Witnesses zeigen sich ebenfalls teilweise. Besonders auffällig ist, dass sich bei einer Mehrheit der Versuche der

Wertebereich weit in den Bereich von Sekunden, teils sogar über den Bereich einer Minute, erstreckt. In den Untersuchungen der anderen Faktoren haben sich bereits häufiger Effekte in Zusammenhang mit Witnesses ergeben, sodass auf diese hier nur noch kurz eingegangen werden soll. Bezüglich des Durchsatzes fällt auf, dass besonders große Cluster mit 20 Knoten eine bessere Leistung mit Pirogue erbringen können. Eine Ausnahme stellt die zusätzliche Verwendung von Witnesses und die Störung durch eine Partition dar. Ein großes Cluster mit einer Ausfallwahrscheinlichkeit von 10 % weist kaum erkennbare Unterschiede auf, ebenso wenig wie ein kleines Cluster mit vielen Ausfällen. Hingegen konnte ein kleines Cluster mit vielen Ausfällen und einer Partitionierung einen besseren Durchsatz, aber auch kürzere Zeiten zur Konsensfindung vorweisen. Auch mit weniger Ausfällen ergaben sich hier leichte Verbesserungen bei Pirogue. Anfangs wurde der Einsatz von Witnesses deutlich kritischer betrachtet, da dieser bei der Betrachtung von Leaderwahlen im Verdacht stand, größere Abweichungen nach oben oder sogar mehr Probleme zu verursachen.

4.5.2 Diskussion

Grundsätzlich sollten Witnesses, da sie in den Versuchen normale Knoten ersetzen, keinen positiven Effekt erzielen. Die Erwartung sollte sich eher darauf stützen, ob mit den leichtgewichtigen Witnesses die Performance von Raft gehalten werden kann. Dies zeigt sich durch die Beobachtungen teilweise. Die Einschränkung ist nicht besonders groß oder kaum vorhanden. Trotzdem ist bei kleinen Clustern eine Beeinträchtigung durch die weniger als Leader zur Verfügung stehenden Knoten deutlich. Der erwartete stärkere Anstieg der Terms durch mehr Probleme bei Wahlen im Zusammenhang mit Witnesses ist erkennbar, auch wenn dieses Verhalten nicht eindeutig mit den Witnesses zu erklären ist. Wie bereits in 4.1 ausgeführt, lässt sich dieses Verhalten bei Pirogue allgemein beobachten. Insgesamt lässt sich aus den Effekten im Zusammenhang mit Witnesses ein gemischtes Fazit ziehen: Zur Vermeidung des Risikos bei der Aufteilung eines Clusters durch das Netzwerk können Witnesses helfen, da die Verwaltung des aktuellen Cohort Sets verbessert werden kann. Die Wahlen eines Leaders sind womöglich etwas eingeschränkter.

5 Fazit

Je nach konkretem Kontext der Anwendung und den spezifischen Anforderungen kann eine Abwägung zwischen den beiden Algorithmen getroffen werden. Die Vielzahl der Versuche zeigt die Probleme beider Protokolle sowie auch deren Vorteile. Ist ein hochverfügbares System gefordert, kann die Implementierung von Pirogue empfohlen werden. Diese Empfehlung kann aber nicht allgemein ausgesprochen werden, sondern muss sich ebenso nach den Netzwerkgegebenheiten und der Art der Verteilung richten. Durch die Versuche konnte gezeigt werden, dass einige von Pirogues Optimierungen, wie die Witnesses, zwar keinen starken Effekt mitbringen, aber zur Stabilität des Systems beitragen können. Dies bezieht sich vermutlich nur auf instabile Netzwerke, bei denen Partitionen häufiger auftreten. Darüber hinaus sind Witnesses kein so eindeutiger Vorteil, wie es zunächst in [22] scheint. Zumindest ein gravierender Nachteil konnte durch die Reduzierung des eigentlichen Quorums mit Witnesses nicht bestätigt werden, was insofern für die Thesen und die Forschung der Autoren von [22, 21] spricht. Ein Nachteil durch Witnesses konnte teilweise beobachtet werden, da es zu vermehrten Problemen mit den Leaderwahlen in kleinen Clustern (5 Knoten) kam. In diesen Szenarien bleibt Raft ein empfehlenswertes Protokoll, da es Stabilität und Sicherheit, erkennbar an einer geringeren Streuung der Werte mitbringt. Dass Pirogue zu viel Kommunikation über das Netzwerk einführt und deutlich langsamer abschneidet, konnte nicht bestätigt werden. Die dynamische Gestaltung dieses moderneren Protokolls führt in den meisten Fällen zu einer Beschleunigung von Prozeduren, die das Quorum erfordern. Eine hohe Dichte an Problemen bezüglich inkonsistenter Zeiten des Gesamtsystems ergeben sich dabei nicht. Des Weiteren konnte die Arbeit zeigen, dass sich Pirogue gleichermaßen für große Cluster (hier 20 Knoten) eignen könnte. Die Mehrheit der Versuche zeigte schnellere Konsenszeiten und schnellere Wahlen eines Leaders, was ebenfalls zur hohen Verfügbarkeit von Pirogue beitragen dürfte. Die Ergebnisse bezüglich Netzwerkpartitionen sind besonders überraschend, da hier zunächst größere Probleme bei dynamischen Quoren denkbar waren. Trotz des besseren Durchsatzes und keinen auffälligen Inkonsistenzen bei Pirogue in diesen Versuchen, sollten die möglichen Fehler weiter untersucht werden, da kein hohes Konfliktpotential

der Anfragen bestand. Zuletzt hat sich eine Ausnahme bei vielen Ausfällen und der Aufteilung des Clusters gezeigt, bei der kein lauffähiges Cluster möglich war. Unter diesen Umständen kann auch Pirogue keine Abhilfe mehr schaffen.

5.1 Ausblick und anschließende Forschung

Neben einer vertiefenden Bewertung der Erkenntnisse, sind technologische Optimierungen denkbar, um die Ergebnisse dieser Arbeit zu erweitern. Dazu zählen umfangreichere Metriken, die sich mithilfe einer Timeseries Datenbank (wie beispielsweise *InfluxDB*) realisieren ließen. Die zusätzliche Performance und die Fähigkeit, die Zeit des Beobachters in die Auswertung einfließen zu lassen, ermöglicht weitere Analysen. Wie bereits in der Beobachtung zu Inkonsistenzen erwähnt, sollten Probleme dieser Art bei Pirogue noch genauer untersucht werden. Dazu könnte sich ein Versuch anbieten, der noch stärker Konflikte provoziert. Auch die spezifische Konfiguration von Timeout-Parametern, auf denen die Protokolle sehr stark basieren, sollte noch weiter untersucht werden, da sich dort eine Vielzahl an Möglichkeiten für beide Algorithmen ergibt. Es hat sich auch gezeigt, dass die Auswahl von vielen Metriken und einer Vielzahl an Faktoren mit sogar mehr als zwei Stufen nicht automatisch zu einem sehr umfassenden Gesamtbild führen, aus dem sich dann eine Wahrheit ablesen lässt. Schon die Komplexität der Experimente in dieser Arbeit nimmt sehr schnell zu und die Skalierung von Faktoren und Metriken führt nicht unbedingt zu einem klareren Ergebnis. Die hohe Komplexität erschwert vielmehr das Auswerten, sodass eine Betrachtung von stets kleinen Ausschnitten viel mehr das Ziel sein sollte.

Literaturverzeichnis

[1] o.A.: *35 % Faster Than The Filesystem*. – URL https://sqlite.org/fasterthanfs.html. – letzter Zugriff am 25.10.2024

[2] o.A.: *Amazon EC2 T2 Instances*. – URL https://aws.amazon.com/ec2/instance-types. – letzter Zugriff am 23.10.2024

[3] Bengel, Günther: *Grundkurs Verteilte Systeme*. Springer Vieweg, 2014. – 366 S. – ISBN 978-3-8348-2150-8

[4] Berghoff, Dr. C. ; Gebhardt, Dr. U. ; et.all., Dr. Manfred L.: *Blockchain sicher gestalten - Konzepte, Anforderungen, Bewertungen*. 2019. – URL https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Krypto/Blockchain_Analyse.pdf%3F__blob=publicationFile%26v=5

[5] Castro, Miguel ; Liskov, Barbara: Practical Byzantine fault tolerance. In: *Proceedings of the Third Symposium on Operating Systems Design and Implementation*. USA : USENIX Association, 1999 (OSDI '99), S. 173–186. – ISBN 1880446391

[6] Eckert, Claudia: *IT-Sicherheit - Konzepte - Verfahren - Protokolle*. Berlin, Boston : De Gruyter Oldenbourg, 2018. – 7–8 S. – URL https://doi.org/10.1515/9783110563900. – ISBN 9783110563900

[7] Fokkink, Wan: *Distributed Algorithms, Second Edition : An Intuitive Approach*. MIT Press, 2018. – URL https://ebookcentral.proquest.com/lib/hawhamburg-ebooks/detail.action?docID=6246589.

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

[9] Herlihy, Maurice P. ; Wing, Jeannette M.: Linearizability: a correctness condition for concurrent objects. In: *ACM Trans. Program. Lang. Syst.* 12 (1990), jul, Nr. 3, S. 463–492. – URL https://doi.org/10.1145/78969.78972. – ISSN 0164-0925

[10] Hofmann, Martin ; Lange, Martin: *Automatentheorie und Logik*. Springer Berlin, Heidelberg, 2011. – 238 S. – ISBN 978-3-642-18089-7

[11] Jajodia, S. ; Mutchler, David: Dynamic voting algorithms for maintaining the consistency of a replicated database. In: *ACM Trans. Database Syst.* 15 (1990), jun, Nr. 2, S. 230–280. – URL https://doi.org/10.1145/78922.78926. – ISSN 0362-5915

[12] Kalis, Rosco ; Belloum, Adam: Validating Data Integrity with Blockchain. In: *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2018, S. 272–277

[13] Lamport, Leslie: Time, Clocks and the Ordering of Events in a Distributed System. In: *Communications of the ACM 21, 7 (July 1978)* (1978), S. 558–565. – URL https://lamport.azurewebsites.net/pubs/pubs.html#time-clocks

[14] Lamport, Leslie: The Part-Time Parliament. In: *ACM Transactions on Computer Systems 16, 2 (May 1998), 133-169. Also appeared as SRC Research Report 49. This paper was first submitted in 1990, setting a personal record for publication delay that has since been broken by [60].* (1998), May. – URL https://www.microsoft.com/en-us/research/publication/part-time-parliament/. – ACM SIGOPS Hall of Fame Award in 2012

[15] Lamport, Leslie: Paxos Made Simple. In: *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (2001), December, S. 51–58. – URL https://www.microsoft.com/en-us/research/publication/paxos-made-simple/

[16] Lemire, D. ; Ssi-Yan-Kai, G. ; Kaser, O.: *Consistently faster and smaller compressed bitmaps with Roaring*. 2018. – URL https://arxiv.org/pdf/1603.06549

[17] León, Fernando P. ; Kiencke, Uwe: *9. Zeit in verteilten Systemen*. S. 453–496. In: *Ereignisdiskrete Systeme*. München : Oldenbourg Wissenschaftsverlag, 2013. – URL https://doi.org/10.1524/9783486769715.453. – ISBN 9783486769715

[18] Long, D.D.E. ; Paris, J.-F.: Voting without version numbers. In: *1997 IEEE International Performance, Computing and Communications Conference*, 1997, S. 139–145

[19] Lynch, Nancy A.: *Distributed algorithms*. 8. [print.]. Morgan Kaufmann, 2008 (The Morgan Kaufmann series in data management systems). – URL http://www.gbv.de/dms/bowker/toc/9781558603486.pdf

[20] Ongaro, Diego ; Ousterhout, John K.: In Search of an Understandable Consensus Algorithm (Extended Version). In: *USENIX Annual Technical Conference*, USENIX Annual Technical Conference, 2014. – URL https://api.semanticscholar.org/CorpusID:14689258

[21] Paris, Jehan-Francois: *Voting with Witnesses: A Constistency Scheme for Replicated Files*. 01 1986. – URL https://www.researchgate.net/publication/221459447_Voting_with_Witnesses_A_Constistency_Scheme_for_Replicated_Files

[22] Pâris, Jehan-François ; Long, Darrell D. E.: Pirogue, a lighter dynamic version of the Raft distributed consensus algorithm. In: *2015 IEEE 34th International Performance Computing and Communications Conference (IPCCC)*, IEEE, 2015, S. 1–8. – URL https://ieeexplore.ieee.org/servlet/opac?punumber=7399586

[23] Rahm, Erhard ; Saake, Gunter ; Sattler, Kai-Uwe: *Konsistenz in Cloud-Datenbanken*. S. 353–369. In: *Verteiltes und Paralleles Datenmanagement: Von verteilten Datenbanken zu Big Data und Cloud*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2015. – URL https://doi.org/10.1007/978-3-642-45242-0_15. – ISBN 978-3-642-45242-0

[24] Schill, Alexander ; Springer, Thomas: *Verteilte Systeme*. Springer Vieweg, 2011. – 440 S. – ISBN 978-3-642-25796-4

[25] Siebertz, Karl ; Bebber, David van ; Hochkirchen, Thomas: *Statistische Versuchsplanung*. Springer Berlin, Heidelberg, 2010. – ISBN 978-3-642-05493-8

[26] Tanenbaum, Andrew S. ; Steen, Maarten van: *Verteilte Systeme Prinzipien und Paradigmen*. Pearson Deutschland, 2007. – 725 S. – URL https://elibrary.pearson.de/book/99.150005/9783863266684. – ISBN 9783827372932

[27] Wijnhoven, Fons ; Spil, Ton ; Stegwee, Robert ; Fa, Rachel Tjang A.: Post-merger IT integration strategies: An IT alignment perspective. In: *The Journal of Strategic Information Systems* 15 (2006), Nr. 1, S. 5–28. – URL https://www.sciencedirect.com/science/article/pii/S0963868705000387. – ISSN 0963-8687

[28] Wiling, Barry: Scientific Study of CAP Theorem and Understanding its Different Implementation Methods. In: *Mathematical Statistician and Engineering Applications* (2022). – URL https://api.semanticscholar.org/CorpusID:246393831

[29] Yaga, Dylan ; Mell, Peter ; Roby, Nik ; Scarfone, Karen: Blockchain Technology Overview. In: *ArXiv* abs/1906.11078 (2018), S. 18. – URL https://api.semanticscholar.org/CorpusID:69842399

[30] Zimmermann, Karl-Heinz: *Berechenbarkeit - Berechnungsmodelle und Unentscheidbarkeit*. Springer Spektrum Wiesbaden, 2020. – URL https://doi.org/10.1007/978-3-658-31739-3. – ISBN 978-3-658-31739-3

A Anhang

A.1 Verwendete Hilfsmittel

In der Tabelle A.1 sind die im Rahmen der Bearbeitung des Themas der verwendeten Werkzeuge und Hilfsmittel aufgelistet.

Tabelle A.1: Verwendete Hilfsmittel und Werkzeuge

Tool	**Verwendung**
LaTeX	Erstellung dieses Dokuments
Jupyter	Python-Umgebung
NumPy 2.1.3	Berechnung der Statistiken
Pandas 2.2.3	Datenverwaltung während Auswertung
Matplotlib Pyplot 3.10	Erzeugung von Diagrammen
Terraform 1.9	IaC-Lösung für Versuchsaufbau
DataTab Statistikrechner	https://datatab.de/statistik-rechner (Versuchsplan)
JetBrains DataSpell 2024.2.2	IDE für Jupyter-Notebooks
JetBrains GoLand 2024.2.2	IDE für Golang

A.2 Auszug aus Konfiguration des Versuchsaufbaus mit Terraform

Listing A.1: EC2-Instanz mit Provisioner

```
1 resource "aws_instance" "rqlite" {
2   count         = var.instance_count
3   ami = "ami-0084a47cc718c111a" # Ubuntu 24.04 LTS canonical
4   instance_type = "t2.${var.instance_type}"
5   tags = {
6     Name    = "rqlite-${count.index}"
7     project = var.aws_resource_tag
```

```
8   }
9   key_name = var.ssh_key_name
10  vpc_security_group_ids =
11    [aws_security_group.rqlite-security-group[count.index % var.partitions].id]
12  provisioner "file" {
13    source      = var.chaos_monkey_script
14    destination = "/home/ubuntu/chaos.sh"
15    connection {
16      type    = "ssh"
17      user    = "ubuntu"
18      private_key = file(var.private_key)
19      host    = self.public_dns
20      timeout = "1m"
21    }
22  }
23  connection {
24    host = self.public_dns # AWS mainly uses the public DNS for access
25    user = "ubuntu" # Default user for Ubuntu AMI
26    type    = "ssh"
27    private_key = file(var.private_key)
28    # Sometimes the instance is considered as active but the setup is not finished yet
29    timeout = "5m"
30  }
31  provisioner "remote-exec" {
32    inline = [
33      # Download rqlite/rqlite release
34      "curl -L https://github.com/rqlite/rqlite/releases/download/v8.32.3/\
35      rqlite-v8.32.3-linux-amd64.tar.gz \
36      -o rqlite-v8.32.3-linux-amd64.tar.gz",
37      # Download rqlite-pirogue release (from private repo)
38      "curl -vLJO -H 'Authorization: Bearer ${var.gh_token}' -H 'Accept: \
39      application/octet-stream' \
40      https://api.github.com/repos/johann-lr/rqlite-pirogue/releases/assets/204260052\
41      -o rqlite-pirogue.tar.xz",
42      "tar xvfz rqlite-v8.32.3-linux-amd64.tar.gz",
43      "tar xvfJ rqlite-pirogue.tar.xz",
44    ]
45  }}
46 resource "terraform_data" "start_rqlite" {
47  count = length(aws_instance.rqlite)
```

```
48   triggers_replace = aws_instance.rqlite[*].id
49   connection {
50     host = element(aws_instance.rqlite[*].public_dns, count.index)
51     type     = "ssh"
52     user     = "ubuntu"
53     private_key = file(var.private_key)
54     timeout = "2m"
55   }
56   provisioner "remote-exec" {
57     inline = [
58       "cd rqlite-v8.32.3-linux-amd64",
59       # Start rqlite with automated clustering (port 4001 for HTTP and 4002 for raft)
60       # nohup keeps the background process running while output is directed to rqlite.log
61       "nohup ./rqlited -http-addr ${aws_instance.rqlite[count.index].public_dns}:4001 \
62       -raft-addr ${aws_instance.rqlite[count.index].public_dns}:4002\
63       -raft-timeout=${var.raft_timeout} \
64       -raft-election-timeout=${var.raft_election_timeout}\
65       -raft-apply-timeout=${var.raft_apply_timeout}\
66       -raft-snap-int=${var.raft_snap_int}\
67       -raft-leader-lease-timeout=${var.raft_leader_lease_timeout}\
68       -bootstrap-expect ${var.instance_count}\
69       -join ${join(",", formatlist("%s:4002", aws_instance.rqlite[*].public_dns))} \
70       data > rqlite.log 2>&1 < /dev/null &",
71       "sleep 5"
72       ... # rqlite-pirogue equivalent
73     ]
74   }
75 }
```

Listing A.2: Beispielkonfiguration für ein Experiment

```
1  module "nodes20_crash010_hb2s_witness2" {
2    source                = "./common"
3    gh_token              = var.gh_token
4    crash_prob            = "0.1"
5    instance_count        = 20
6    local_ip              = var.current_ip
7    experiment_name       = "nodes20_crash010_hb2s_witness2"
8    raft_timeout          = "2s"
9    raft_election_timeout = "2s"
10   raft_leader_lease_timeout = "2s"
```

```
  witness_ratio            = 2
}
output "nodes20_crash010_hb2s_witness2" {
  value = module.nodes20_crash010_hb2s_witness2.rqlite_ips
}
```

Listing A.3: Beispiel Auswertungsfunktion aus Jupyter Notebook

```
def time_until_new_leader(df: pandas.DataFrame, majority=True) -> list[int]:
    """
    Calculates the time until the cluster has a new leader (all nodes appended it)
    :param majority: if all nodes or the majority should be checked
    :param df: dataframe containing the results of the experiment
    :return: time deltas until the cluster has a leader
    """
    results = []
    # grp is a database marker for different iterations so there might be time gaps
    for grp in sorted(df.grp.unique()):
        term_set = df[df.grp == grp]
        for i in sorted(term_set.raft_term.unique()):
            subset = df[(df.raft_term == i) & (df.grp == grp)]
            required_unique_peers = subset.peer_address.nunique()
            if majority:
                required_unique_peers = int(required_unique_peers / 2) + 1
            if subset.peer_address.nunique() >= required_unique_peers:
                min_time = subset.leader_appended_time.min()
                max_time = subset.leader_appended_time.max()
                duration = (max_time - min_time).total_seconds()
                results.append(duration)
    return results
```

A.3 Auszug der erfassten Daten

Zur besseren Übersichtlichkeit sind Netzwerkadressen, Zeitstempel und die Bezeichnungen der Spalten verkürzt dargestellt.

peer_address	rtt	node_current_time	node_uptime
ec2-18-185-1-24	14263	2024-11-11T17:24:19.841Z	2m58.164021517s
ec2-18-185-1-24	14263	2024-11-11T17:24:19.843Z	2m53.166781866s
ec2-3-74-149-234	13472	2024-11-11T17:24:19.846Z	3m8.669647033s
ec2-3-74-149-234	13472	2024-11-11T17:24:19.846Z	3m3.66672901s
ec2-35-157-116-213	14237	2024-11-11T17:24:19.845Z	3m20.308625243s
ec2-35-157-116-213	14237	2024-11-11T17:24:19.849Z	3m15.313354534s
ec2-35-159-118-250	13957	2024-11-11T17:24:19.849Z	2m58.355464673s
ec2-35-159-118-250	13957	2024-11-11T17:24:19.8452Z	2m53.350779136s
ec2-35-158-197-221	14017	2024-11-11T17:24:19.8447Z	3m24.075935542s
ec2-35-158-197-221	14017	2024-11-11T17:24:19.8465Z	3m19.078452865s

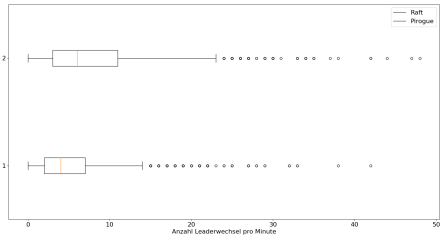
index	term	leader	leader_appended	commit
0	0	ec2-3-74-149-234	0001-01-01T00:00:00Z	2
0	0	ec2-3-74-149-234	0001-01-01T00:00:00Z	2
3	2	ec2-3-74-149-234	2024-11-11T17:24:19.809Z	3
0	0	ec2-3-74-149-234	2024-11-11T17:24:19.809Z	2
0	0	ec2-3-74-149-234	2024-11-11T17:24:19.803Z	2
0	0	ec2-3-74-149-234	2024-11-11T17:24:19.893Z	2
3	2	ec2-3-74-149-234	2024-11-11T17:24:19.809Z	3
0	0	ec2-3-74-149-234	2024-11-11T17:24:19.803Z	2
3	2	ec2-3-74-149-234	2024-11-11T17:24:19.909Z	3
2	2	ec2-3-74-149-234	2024-11-11T17:24:19.863Z	2

last_contact	last_log_index	last_term	num_peers	state
20.932547ms	3	2	4	Follower
38.582771ms	2	2	4	Follower
0	3	2	4	Leader
0	2	2	4	Leader

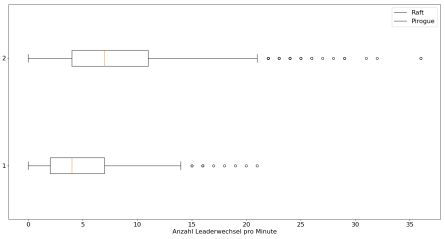
last_contact	last_log_index	last_term	num_peers	state
23.679424ms	3	2	4	Follower
66.380762ms	2	2	4	Follower
26.232957ms	3	2	4	Follower
61.919584ms	2	2	4	Follower
19.407001ms	3	2	4	Follower
5.804553ms	2	2	4	Follower

Tabelle A.2: Beispiel Messdaten

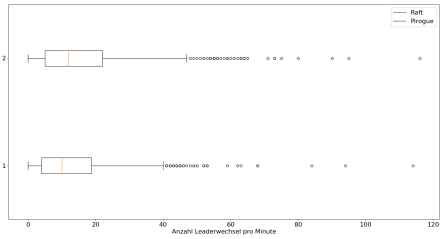
A.4 Weitere Abbildungen der Versuchsauswertungen



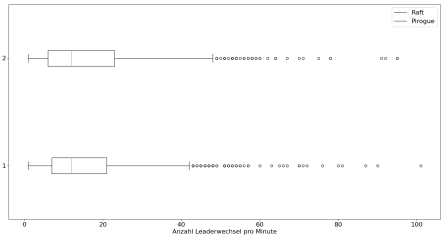
(a) Versuch 1



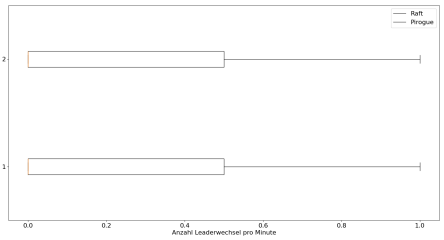
(b) Versuch 2



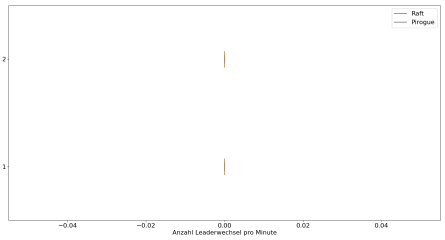
(c) Versuch 3



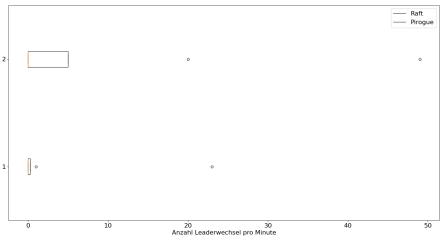
(d) Versuch 4



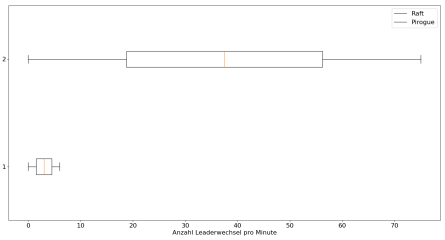
(e) Versuch 5



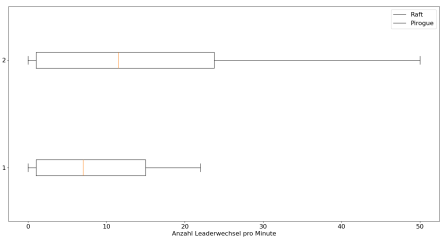
(f) Versuch 6



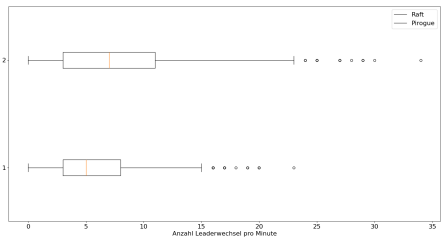
(g) Versuch 7



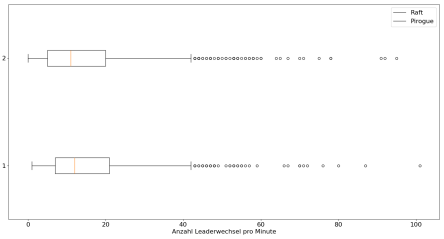
(h) Versuch 8



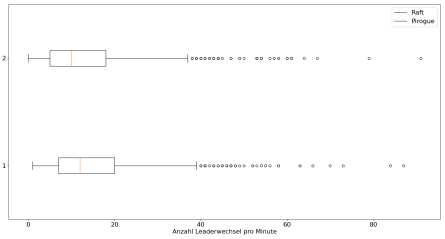
(i) Versuch 9



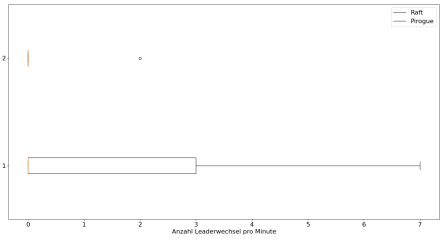
(j) Versuch 10



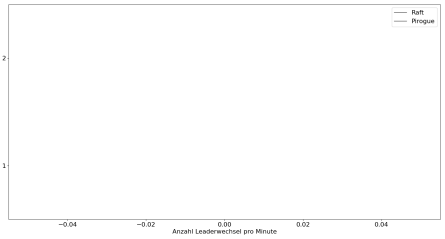
(k) Versuch 11



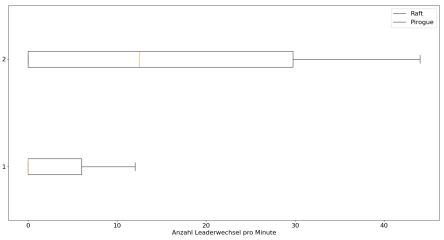
(l) Versuch 12



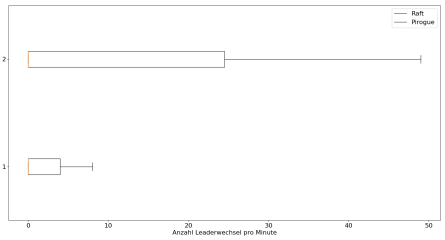
(m) Versuch 13



(n) Versuch 14

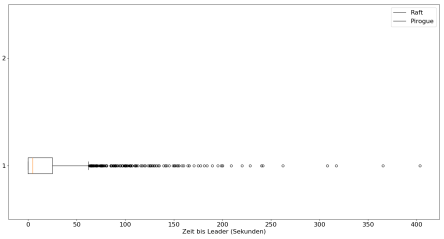


(o) Versuch 15

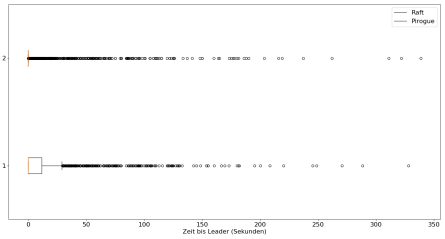


(p) Versuch 16

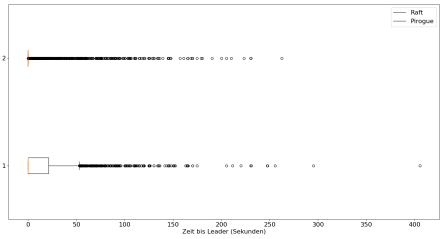
Abbildung A.1: Häufigkeit Leaderwechsel



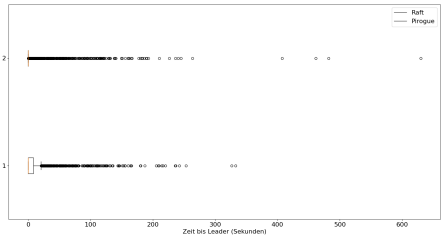
(a) Versuch 1



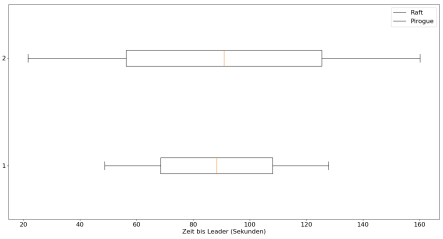
(b) Versuch 2



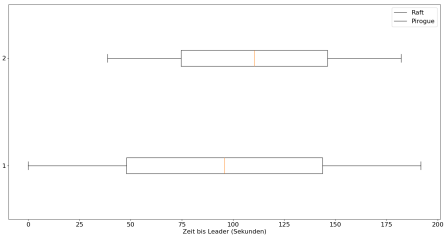
(c) Versuch 3



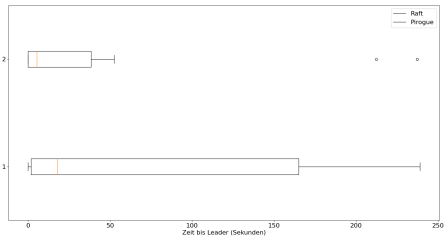
(d) Versuch 4



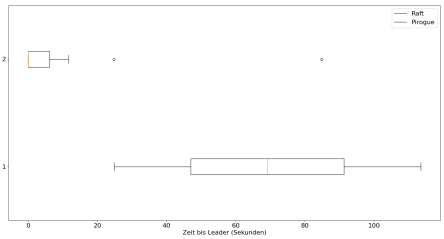
(e) Versuch 5



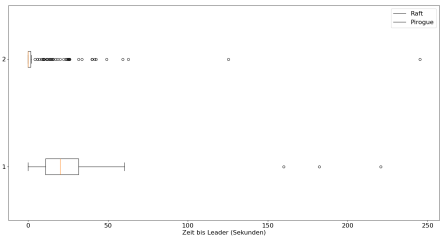
(f) Versuch 6



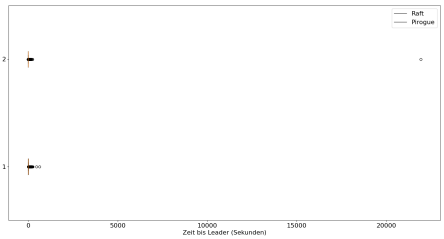
(g) Versuch 7



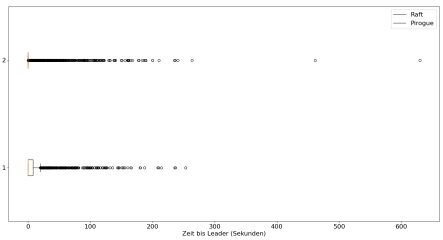
(h) Versuch 8



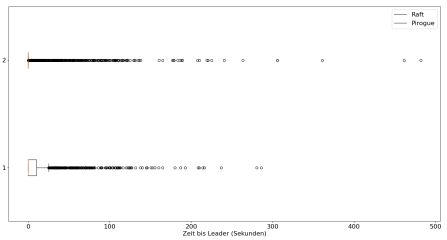
(i) Versuch 9



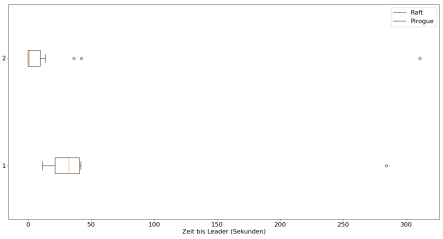
(j) Versuch 10



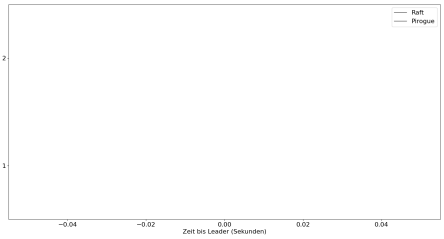
(k) Versuch 11



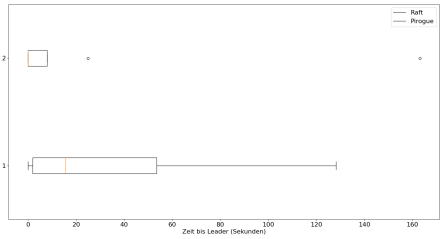
(l) Versuch 12



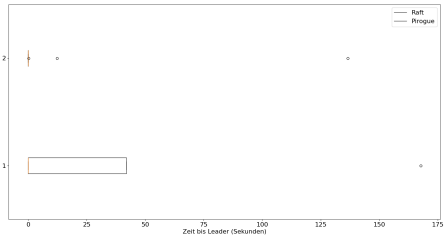
(m) Versuch 13



(n) Versuch 14

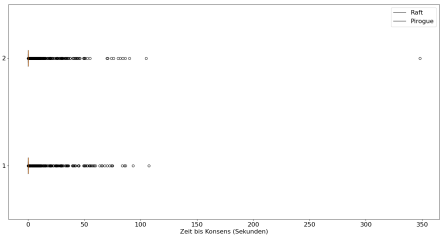


(o) Versuch 15

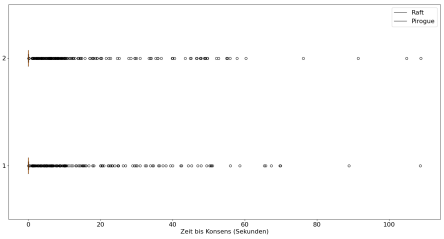


(p) Versuch 16

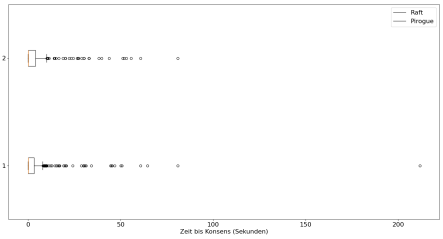
Abbildung A.2: Dauer Leaderwechsel



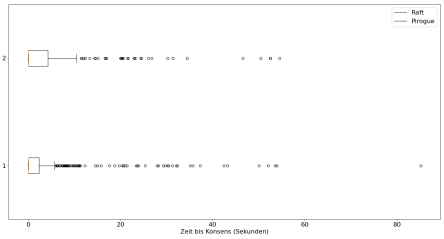
(a) Versuch 1



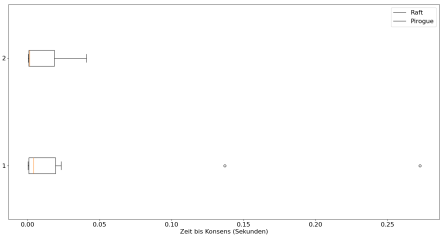
(b) Versuch 2



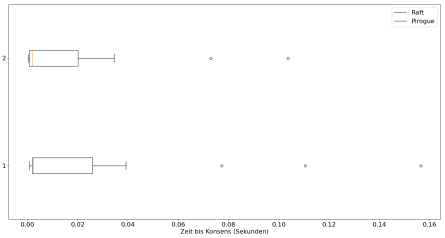
(c) Versuch 3



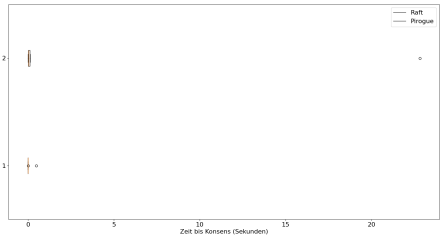
(d) Versuch 4



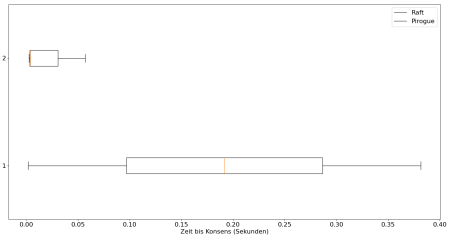
(a) Versuch 5



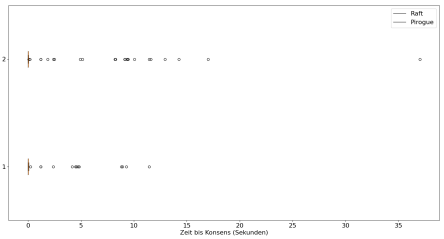
(b) Versuch 6



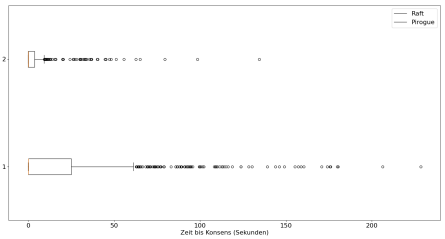
(c) Versuch 7



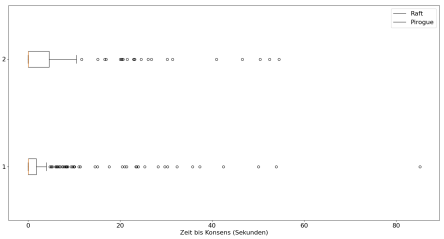
(d) Versuch 8



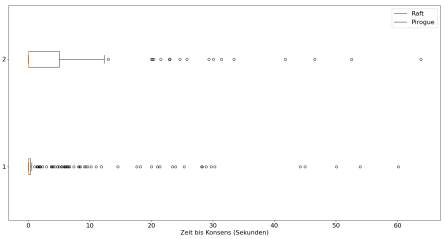
(e) Versuch 9



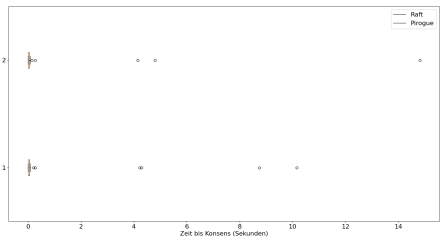
(f) Versuch 10



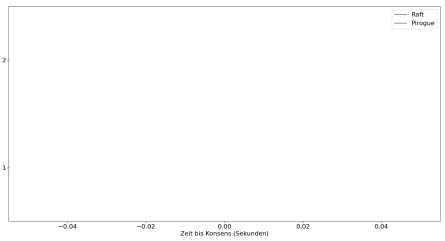
(g) Versuch 11



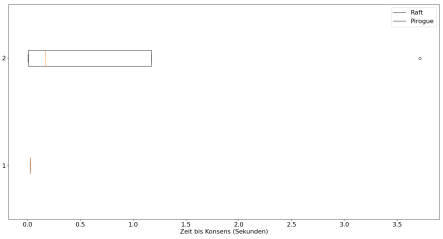
(h) Versuch 12



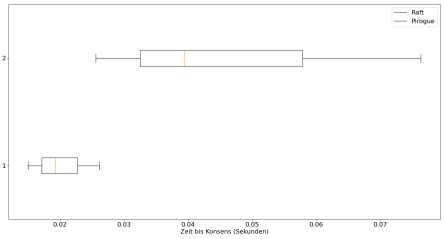
(i) Versuch 13



(j) Versuch 14

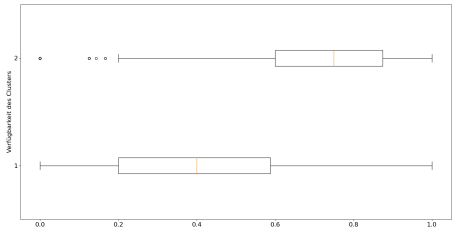


(k) Versuch 15

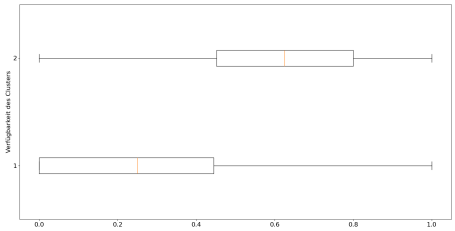


(l) Versuch 16

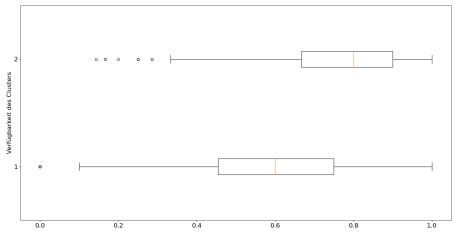
Abbildung A.4: Dauer Konsensbildung



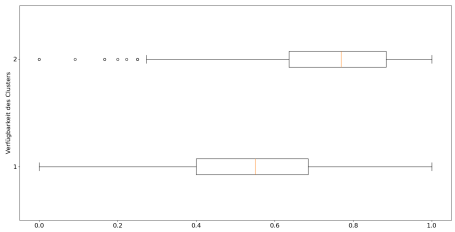
(a) Versuch 1



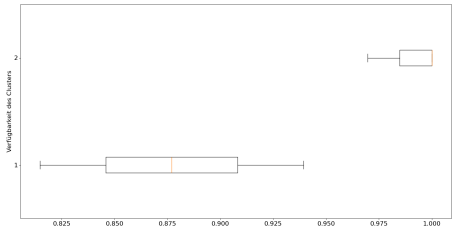
(b) Versuch 2



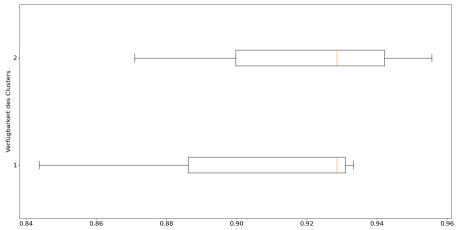
(c) Versuch 3



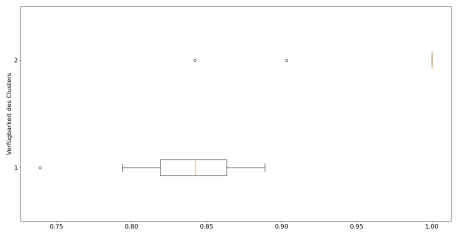
(d) Versuch 4



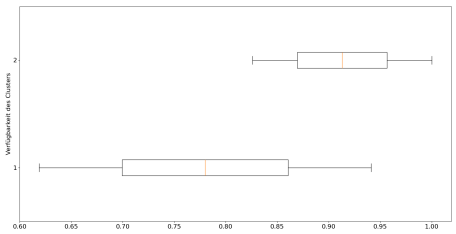
(a) Versuch 5



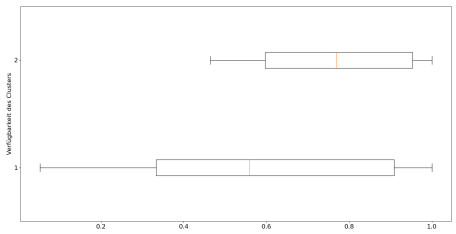
(b) Versuch 6



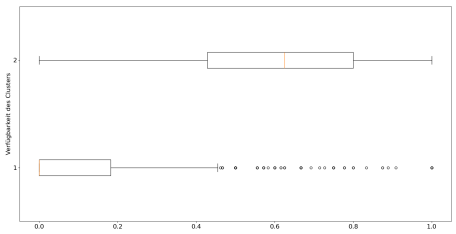
(a) Versuch 7



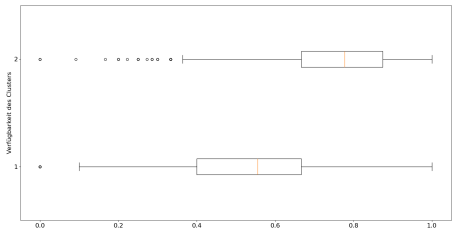
(b) Versuch 8



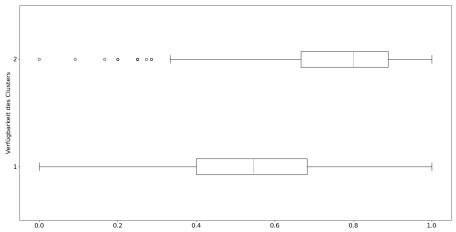
(c) Versuch 9



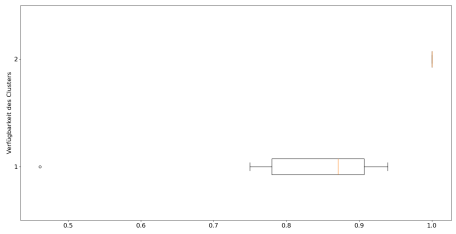
(d) Versuch 10



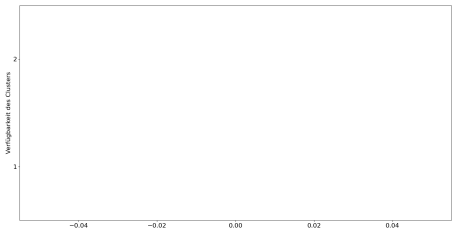
(e) Versuch 11



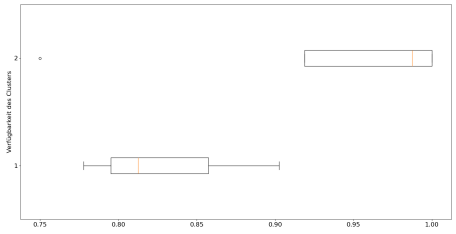
(f) Versuch 12



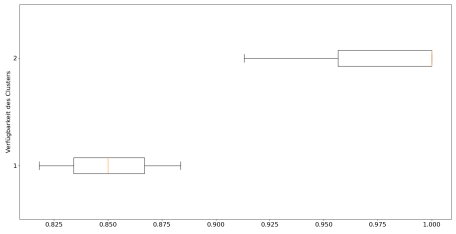
(g) Versuch 13



(h) Versuch 14

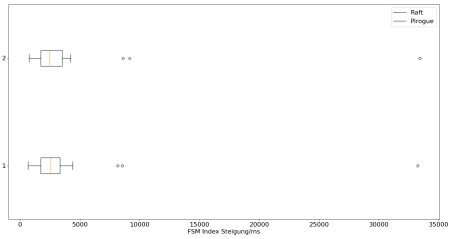


(i) Versuch 15

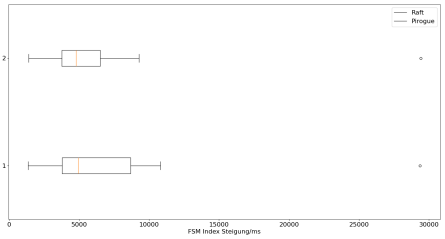


(j) Versuch 16

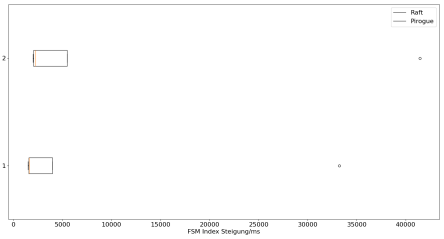
Abbildung A.7: Verfügbarkeiten



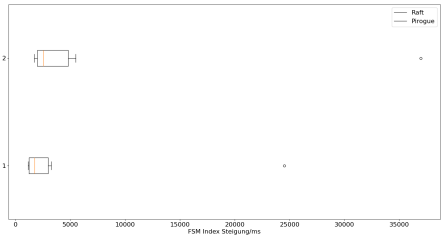
(a) Versuch 1



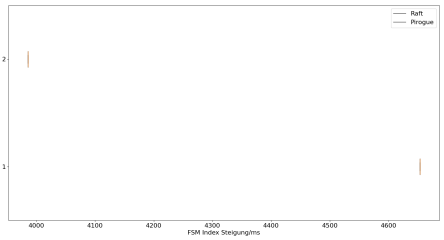
(b) Versuch 2



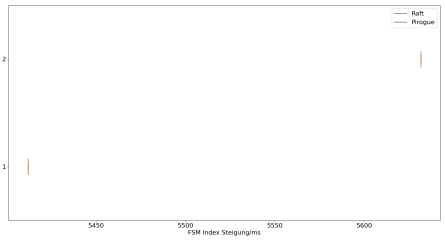
(c) Versuch 3



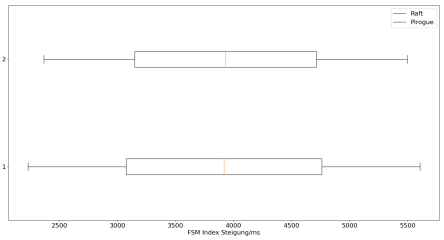
(d) Versuch 4



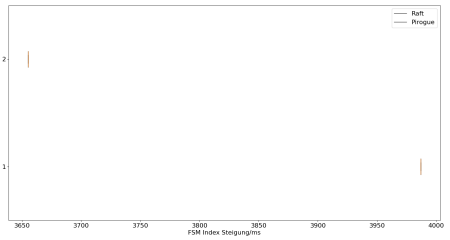
(e) Versuch 5



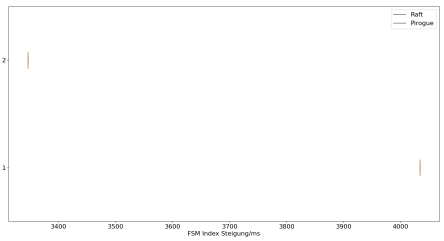
(f) Versuch 6



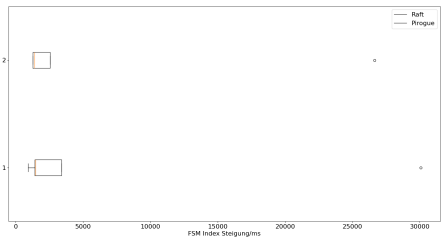
(g) Versuch 7



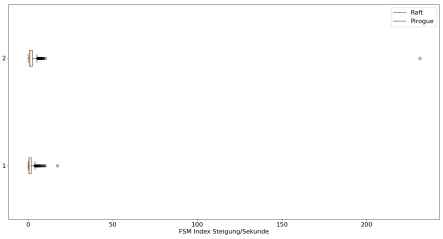
(h) Versuch 8



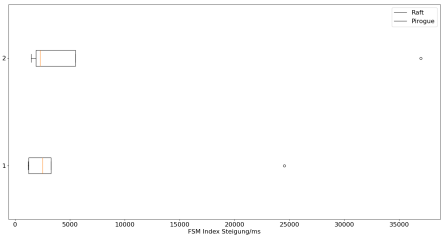
(i) Versuch 9



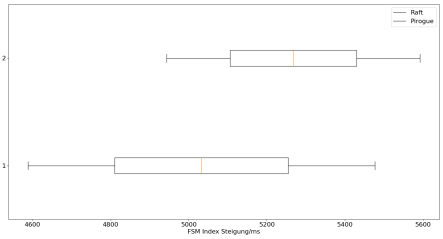
(j) Versuch 10



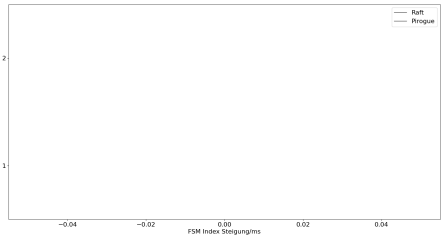
(a) Versuch 11



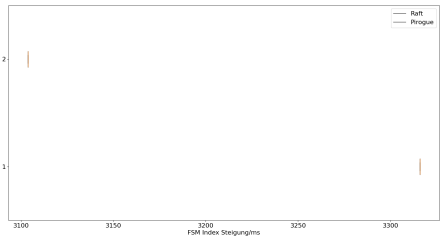
(b) Versuch 12



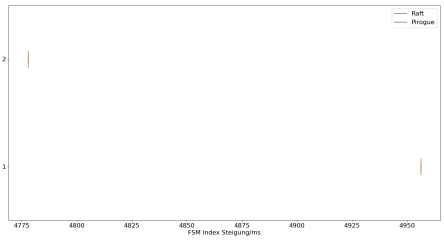
(c) Versuch 13



(d) Versuch 14



(e) Versuch 15



(f) Versuch 16

Abbildung A.9: Durchsatz

A.5 Tabellen mit allen Statistiken

Der Algorithmus mit der Nummer 0 ist dabei jeweils Raft, Nummer 1 kennzeichnet Pirogue. Die Abkürzung *CI* steht für das Konfidenzintervall, in dem 95 % der Werte verortet sind.

Algo.	Leaderwechsel/min			FSM Index Steigung/s		
	Mittel	Median	CI	Mittel	Median	CI
0	5.576	4.0	(5.3328, 5.8183)	2.137	1.539	(2.0325, 2.2425)

Algo.	Leaderwechsel/min			FSM Index Steigung/s		
	Mittel	Median	CI	Mittel	Median	CI
1	7.984	6.0	(7.6482, 8.3198)	2.103	1.476	(2.0009, 2.2057)

Algo.	Inkonsistenz Zeiten/s			Inkonsistenz Index/s		
	Mittel	Median	CI	Mittel	Median	CI
0	12.587	11.284	(8.2854, 16.8886)	54.688	0.284	(8.29, 16.89)
1	11.949	10.268	(7.8266, 16.0722)	1237.375	0.262	(7.83, 16.07)

Algo.	Konsenszeit (s)			Dauer Leaderwahl (s)		
	Mittel	Median	CI	Mittel	Median	CI
0	2.590	0.022	(2.26, 2.92)	17.992	4.749	(17.0152, 18.9694)
1	2.517	0.021	(2.13, 2.91)	-	-	-

Algo.	Verfügbarkeit		
	Mittel	Median	CI
0	0.399	0.400	(0.386, 0.412)
1	0.728	0.750	(0.718, 0.738)

Tabelle A.3: Versuch 1

Algo.	Leaderwechsel/min			FSM Index Steigung/s		
	Mittel	Median	CI	Mittel	Median	CI
0	4.897	4.0	(4.6732, 5.12)	1.845	1.177	(1.704, 1.9868)
1	8.249	7.0	(7.8232, 8.6744)	1.782	1.171	(1.6036, 1.9601)

Algo.	Inkonsistenz Zeiten/s			Inkonsistenz Index/s		
	Mittel	Median	CI	Mittel	Median	CI
0	11.552	7.491	(4.0356, 19.0682)	3566.722	0.342	(-795.58, 7929.02)
1	10.200	5.572	(3.096, 17.3036)	2092.368	0.133	(-1574.48, 5759.22)

Algo.	Konsenszeit (s)			Dauer Leaderwahl (s)		
	Mittel	Median	CI	Mittel	Median	CI
0	2.912	0.022	(2.33, 3.49)	12.597	0.0	(11.3482, 13.8464)
1	3.314	0.023	(2.67, 3.96)	5.047	0.0	(4.417, 5.6763)

Algo.	Verfügbarkeit		
	Mittel	Median	CI
0	0.287	0.25	(0.2693, 0.3056)
1	0.627	0.625	(0.6107, 0.6443)

Tabelle A.4: Versuch 2

Algo.	Leaderwechsel/min			FSM Index Steigung/s		
	Mittel	Median	CI	Mittel	Median	CI
0	13.432	10.0	(12.7193, 14.1441)	1.693	1.040	(1.5701, 1.8168)
1	15.906	12.0	(15.0666, 16.7461)	2.204	1.588	(1.8815, 2.5261)

Algo.	Inkonsistenz Zeiten/s			Inkonsistenz Index/s		
	Mittel	Median	CI	Mittel	Median	CI
0	3.926	4.955	(2.6217, 5.2298)	324.697	0.145	(-63.7, 713.09)
1	3.750	4.747	(2.5717, 4.9288)	39.456	0.199	(-9.28, 88.19)

Algo.	Konsenszeit (s)			Dauer Leaderwahl (s)		
	Mittel	Median	CI	Mittel	Median	CI
0	4.934	0.045	(3.25, 6.62)	16.415	0.0	(15.2769, 17.5539)
1	4.375	0.046	(3.18, 5.57)	7.114	0.0	(6.5346, 7.6925)

Algo.	Verfügbarkeit		
	Mittel	Median	CI
0	0.594	0.6	(0.5813, 0.6061)
1	0.781	0.8	(0.7713, 0.7916)

Tabelle A.5: Versuch 3

Algo.	Leaderwechsel/min			FSM Index Steigung/s		
	Mittel	Median	CI	Mittel	Median	CI
0	16.205	12.0	(15.3942, 17.0167)	1.423	0.904	(1.3143, 1.5323)
1	16.566	12.0	(15.7331, 17.3989)	2.300	1.536	(1.7074, 2.8935)

Algo.	Inkonsistenz Zeiten/s			Inkonsistenz Index/s		
	Mittel	Median	CI	Mittel	Median	CI
0	4.510	4.082	(2.1967, 6.8239)	909.285	0.193	(-696.03, 2514.6)
1	4.622	3.930	(2.2063, 7.0384)	2133.747	0.167	(-569.32, 4836.81)

Algo.	Konsenszeit (s)			Dauer Leaderwahl (s)		
	Mittel	Median	CI	Mittel	Median	CI
0	4.145	0.045	(3.09, 5.2)	10.915	0.0	(10.0033, 11.8266)
1	3.894	0.045	(2.94, 4.84)	5.603	0.0	(5.0088, 6.198)

Algo.	Verfügbarkeit		
	Mittel	Median	CI
0	0.538	0.551	(0.5255, 0.5508)
1	0.753	0.769	(0.7427, 0.7642)

Tabelle A.6: Versuch 4

Algo.	Leaderwechsel/min			FSM Index Steigung/s		
	Mittel	Median	CI	Mittel	Median	CI
0	0.333	0.0	(-0.2001, 0.8668)	4.687	3.653	(0.98, 8.3935)
1	0.333	0.0	(-0.2001, 0.8668)	3.890	1.284	(-0.2849, 8.0643)

Algo.	Inkonsistenz Zeiten/s			Inkonsistenz Index/s		
	Mittel	Median	CI	Mittel	Median	CI
0	36.347	36.347	(36.3471, 36.3471)	64.266	64.266	(-24.57, 153.1)
1	28.369	28.369	(28.3688, 28.3688)	0.216	0.216	(0.22, 0.22)

Algo.	Konsenszeit (s)			Dauer Leaderwahl (s)		
	Mittel	Median	CI	Mittel	Median	CI
0	0.045	0.004	(-0.01, 0.1)	88.263	88.263	(33.4726, 143.0533)
1	0.009	0.001	(0.0, 0.02)	90.866	90.866	(-5.0994, 186.8311)

Algo.	Verfügbarkeit		
	Mittel	Median	CI
0	0.877	0.877	(0.7908, 0.9634)
1	0.990	1.000	(0.9737, 1.0061)

Tabelle A.7: Versuch 5

Algo.	Leaderwechsel/min			FSM Index Steigung/s		
	Mittel	Median	CI	Mittel	Median	CI
0	0.0	0.0	(0.0, 0.0)	6.244	5.730	(3.631, 8.8566)
1	0.0	0.0	(0.0, 0.0)	6.117	5.552	(3.4255, 8.8085)

Algo.	Inkonsistenz Zeiten/s			Inkonsistenz Index/s		
	Mittel	Median	CI	Mittel	Median	CI
0	56.437	56.437	(56.4369, 56.4369)	10364.778	10364.778	(-3999.43, 24728.98)
1	57.256	57.256	(57.2564, 57.2564)	260.668	260.668	(-100.03, 621.37)

Algo.	Konsenszeit (s)			Dauer Leaderwahl (s)		
	Mittel	Median	CI	Mittel	Median	CI
0	0.028	0.002	(0.0, 0.05)	95.829	95.829	(-36.9833, 228.642)
1	0.018	0.002	(0.0, 0.03)	110.382	110.382	(11.1312, 209.6335)

Algo.	Verfügbarkeit		
	Mittel	Median	CI
0	0.902	0.929	(0.8553, 0.9485)
1	0.918	0.929	(0.8784, 0.9583)

Tabelle A.8: Versuch 6

Algo.	Leaderwechsel/min			FSM Index Steigung/s		
	Mittel	Median	CI	Mittel	Median	CI
0	3.0	0.0	(-2.2432, 8.2432)	3.824	4.041	(2.1938, 5.4543)
1	8.222	0.0	(-2.0331, 18.4776)	3.751	3.588	(2.2865, 5.2152)

Algo.	Inkonsistenz Zeiten/s			Inkonsistenz Index/s		
	Mittel	Median	CI	Mittel	Median	CI
0	32.587	32.587	(18.4543, 46.7194)	3610.091	74.449	(-2433.39, 9653.57)
1	30.970	30.970	(19.613, 42.3276)	2908.515	95.311	(-1920.8, 7737.83)

Algo.	Konsenszeit (s)			Dauer Leaderwahl (s)		
	Mittel	Median	CI	Mittel	Median	CI
0	0.055	0.002	(-0.04, 0.15)	80.890	17.939	(-1.1178, 162.8971)
1	2.583	0.043	(-2.09, 7.26)	48.874	5.331	(-1.0898, 98.8369)

Algo.	Verfügbarkeit		
	Mittel	Median	CI
0	0.834	0.842	(0.8026, 0.8659)
1	0.972	1.0	(0.9359, 1.0075)

Tabelle A.9: Versuch 7

Algo.	Leaderwechsel/min			FSM Index Steigung/s		
	Mittel	Median	CI	Mittel	Median	CI
0	3.0	3.0	(-1.1578, 7.1578)	4.789	4.789	(-0.8922, 10.4702)
1	37.5	37.5	(-14.4723, 89.4723)	4.458	4.458	(-1.0805, 9.997)

Algo.	Inkonsistenz Zeiten/s			Inkonsistenz Index/s		
	Mittel	Median	CI	Mittel	Median	CI
0	29.952	29.952	(29.9522, 29.9522)	46.864	46.864	(-17.13, 110.86)
1	22.994	22.994	(22.9936, 22.9936)	4.988	4.988	(-0.89, 10.87)

Algo.	Konsenszeit (s)			Dauer Leaderwahl (s)		
	Mittel	Median	CI	Mittel	Median	CI
0	0.192	0.192	(-0.07, 0.45)	69.197	69.197	(7.772, 130.622)
1	0.021	0.004	(-0.01, 0.05)	11.089	0.0	(-3.3925, 25.5697)

Algo.	Verfügbarkeit		
	Mittel	Median	CI
0	0.780	0.780	(0.5569, 1.0033)
1	0.913	0.913	(0.7925, 1.0336)

Tabelle A.10: Versuch 8

Algo.	Leaderwechsel/min			FSM Index Steigung/s		
	Mittel	Median	CI	Mittel	Median	CI
0	8.138	7.0	(5.5296, 10.7463)	4.272	3.526	(2.9688, 5.575)
1	15.107	11.5	(9.4416, 20.7727)	3.647	2.329	(2.6816, 4.612)

Algo.	Inkonsistenz Zeiten/s			Inkonsistenz Index/s		
	Mittel	Median	CI	Mittel	Median	CI
0	31.329	31.329	(31.3285, 31.3285)	90.494	90.494	(-34.02, 215.01)
1	27.574	27.574	(27.5744, 27.5744)	47887.602	47887.602	(-18480.46, 114255.66)

Algo.	Konsenszeit (s)			Dauer Leaderwahl (s)		
	Mittel	Median	CI	Mittel	Median	CI
0	0.576	0.003	(0.22, 0.93)	29.688	20.136	(19.1639, 40.2121)
1	1.533	0.004	(0.74, 2.33)	6.967	0.0	(3.7797, 10.1541)

Algo.	Verfügbarkeit		
	Mittel	Median	CI
0	0.590	0.559	(0.4845, 0.6961)
1	0.766	0.769	(0.6973, 0.834)

Tabelle A.11: Versuch 9

Algo.	Leaderwechsel/min			FSM Index Steigung/s		
	Mittel	Median	CI	Mittel	Median	CI
0	5.626	5.0	(5.3796, 5.8727)	1.312	0.838	(1.2122, 1.4123)
1	8.222	7.0	(7.6294, 8.8155)	1.792	1.015	(1.2814, 2.3032)

Algo.	Inkonsistenz Zeiten/s			Inkonsistenz Index/s		
	Mittel	Median	CI	Mittel	Median	CI
0	3.014	2.822	(2.2223, 3.8063)	125.424	0.079	(-90.57, 341.42)
1	1.832	1.565	(0.7968, 2.8679)	62.973	0.064	(-17.65, 143.6)

Algo.	Konsenszeit (s)			Dauer Leaderwahl (s)		
	Mittel	Median	CI	Mittel	Median	CI
0	19.634	0.038	(17.17, 22.1)	10.319	0.0	(9.1905, 11.4469)
1	5.020	0.025	(3.78, 6.26)	14.408	0.0	(-4.4722, 33.2886)

Algo.	Verfügbarkeit		
	Mittel	Median	CI
0	0.124	0.0	(0.11, 0.1374)
1	0.615	0.625	(0.5909, 0.639)

Tabelle A.12: Versuch 10

Algo.	Leaderwechsel/min			FSM Index Steigung/s		
	Mittel	Median	CI	Mittel	Median	CI
0	16.182	12.0	(15.2161, 17.1475)	1.390	0.924	(1.2617, 1.519)
1	14.851	11.0	(14.0445, 15.6566)	2.195	1.461	(1.6787, 2.7119)

Algo.	Inkonsistenz Zeiten/s			Inkonsistenz Index/s		
	Mittel	Median	CI	Mittel	Median	CI
0	3.520	4.136	(2.2879, 4.7517)	57.367	0.193	(-9.71, 124.44)
1	3.414	4.000	(2.3301, 4.4989)	2207.547	0.161	(-1489.8, 5904.89)

Algo.	Konsenszeit (s)			Dauer Leaderwahl (s)		
	Mittel	Median	CI	Mittel	Median	CI
0	3.863	0.046	(2.57, 5.16)	10.838	0.0	(9.7925, 11.8843)
1	4.226	0.048	(2.97, 5.49)	5.746	0.0	(5.1571, 6.3351)

Algo.	Verfügbarkeit		
	Mittel	Median	CI
0	0.541	0.556	(0.5265, 0.5564)
1	0.753	0.778	(0.742, 0.764)

Tabelle A.13: Versuch 11

Algo.	Leaderwechsel/min			FSM Index Steigung/s		
	Mittel	Median	CI	Mittel	Median	CI
0	15.568	12.0	(14.6876, 16.4489)	1.488	1.010	(1.3525, 1.6225)
1	13.419	10.0	(12.7177, 14.1206)	2.159	1.474	(1.6742, 2.6443)

Algo.	Inkonsistenz Zeiten/s			Inkonsistenz Index/s		
	Mittel	Median	CI	Mittel	Median	CI
0	3.449	4.210	(2.3272, 4.5699)	17.176	0.090	(-13.74, 48.1)
1	3.239	3.246	(2.3899, 4.0887)	163.807	0.108	(-80.6, 408.21)

Algo.	Konsenszeit (s)			Dauer Leaderwahl (s)		
	Mittel	Median	CI	Mittel	Median	CI
0	3.403	0.041	(2.23, 4.57)	11.287	0.0	(10.2103, 12.3637)
1	4.044	0.048	(2.88, 5.21)	5.703	0.0	(5.1125, 6.2935)

Algo.	Verfügbarkeit		
	Mittel	Median	CI
0	0.541	0.545	(0.5265, 0.556)
1	0.767	0.800	(0.7563, 0.7773)

Tabelle A.14: Versuch 12

Algo.	Leaderwechsel/min			FSM Index Steigung/s		
	Mittel	Median	CI	Mittel	Median	CI
0	1.833	0.0	(-0.354, 4.0206)	5.181	5.837	(3.5516, 6.8109)
1	0.333	0.0	(-0.2631, 0.9297)	5.471	5.860	(3.7124, 7.2296)

Algo.	Inkonsistenz Zeiten/s			Inkonsistenz Index/s		
	Mittel	Median	CI	Mittel	Median	CI
0	32.307	32.307	(23.6033, 41.0113)	250.598	0.416	(23.6, 41.01)
1	34.346	34.346	(24.3142, 44.3785)	278.226	0.423	(24.31, 44.38)

Algo.	Konsenszeit (s)			Dauer Leaderwahl (s)		
	Mittel	Median	CI	Mittel	Median	CI
0	0.622	0.004	(0.02, 1.22)	70.251	32.373	(-6.8066, 147.3079)
1	0.414	0.001	(-0.1, 0.93)	27.967	1.016	(-10.8721, 66.806)

Algo.	Verfügbarkeit		
	Mittel	Median	CI
0	0.802	0.870	(0.671, 0.9329)
1	1.0	1.0	(1.0, 1.0)

Tabelle A.15: Versuch 13

Algo.	Leaderwechsel/min			FSM Index Steigung/s		
	Mittel	Median	CI	Mittel	Median	CI
0	4.0	0.0	(-2.4013, 10.4013)	4.045	1.977	(0.1538, 7.9357)
1	17.25	12.5	(-0.8916, 35.3916)	3.234	1.432	(0.0387, 6.4299)

Algo.	Inkonsistenz Zeiten/s			Inkonsistenz Index/s		
	Mittel	Median	CI	Mittel	Median	CI
0	25.842	25.842	(25.8418, 25.8418)	0.799	0.799	(0.8, 0.8)
1	24.553	24.553	(24.5533, 24.5533)	0.683	0.683	(0.68, 0.68)

Algo.	Konsenszeit (s)			Dauer Leaderwahl (s)		
	Mittel	Median	CI	Mittel	Median	CI
0	0.026	0.026	(0.03, 0.03)	39.850	15.596	(-11.3414, 91.0413)
1	1.014	0.167	(-0.52, 2.55)	22.075	0.0	(-10.8776, 55.0274)

Algo.	Verfügbarkeit		
	Mittel	Median	CI
0	0.831	0.813	(0.7715, 0.8903)
1	0.931	0.988	(0.8282, 1.0343)

Tabelle A.16: Versuch 15

Algo.	Leaderwechsel/min			FSM Index Steigung/s		
	Mittel	Median	CI	Mittel	Median	CI
0	2.667	0.0	(-1.6009, 6.9342)	5.415	6.416	(2.271, 8.5593)
1	16.333	0.0	(-9.8054, 42.4721)	5.527	5.601	(1.7989, 9.2548)

Algo.	Inkonsistenz Zeiten/s			Inkonsistenz Index/s		
	Mittel	Median	CI	Mittel	Median	CI
0	43.582	43.582	(43.5825, 43.5825)	8441.325	8441.325	(-3256.61, 20139.26)
1	41.693	41.693	(41.6932, 41.6932)	1973.979	1973.979	(-760.34, 4708.3)

Algo.	Konsenszeit (s)			Dauer Leaderwahl (s)		
	Mittel	Median	CI	Mittel	Median	CI
0	0.020	0.019	(0.01, 0.03)	41.935	0.0	(-29.246, 113.1163)
1	0.047	0.039	(0.02, 0.07)	10.651	0.0	(-7.723, 29.0243)

Algo.	Verfügbarkeit		
	Mittel	Median	CI
0	0.851	0.850	(0.8204, 0.8809)
1	0.971	1.000	(0.9246, 1.0174)

Tabelle A.16: Versuch 16

Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original