Fehlertoleranz in verteilten Systemen

Ausarbeitung im Rahmen des Seminars „Fehlertolerante Systeme“

Gona Fatah
Friedrich Alexander Universität Erlangen-Nürnberg
gona.fatah@fau.de

ABSTRACT

Verteilte Systeme (VS) sind heutzutage nicht mehr aus der Technik- und WWW-Welt weg zu denken. In unserem Alltag, der zum größten Teil durch die digitalisierte Welt wie die Nutzung von Clouds, Suchmaschinen, Onlinebestellungen, Onlinebanking usw. bestimmt und meist erleichtert wird, macht es unerlässlich, sich näher mit diesen Systemen auseinander zu setzen.

Im Rahmen dieser Arbeit sollen vorab die Bedeutung und der Aufbau von verteilten Systemen näher erläutert werden. Die Antworten auf Fragen wie: Was ist ein verteiltes System? Wie ist die Struktur? Was sind die wünschenswerten Eigenschaften in solch einem System? sollen als Einleitung in das Thema „Fehlertoleranz in verteilten Systemen“ dienen, welches die Basis dieser Arbeit bildet.

Um Fehlertoleranz in VS besser erläutern zu können, ist es enorm wichtig die Rahmenthemen kennen zu lernen. Dazu gehören das Fehlermodell, welches die einzelnen Fehlerarten beinhaltet, die in einem verteilten System auftreten können und die beiden Protokolle TCP- und Commit-Protokoll, die die Fehlerarten aus dem Fehlermodell aufspüren bzw. eliminieren können. Des Weiteren wird hier Leader Election (Koordinator Auswahlverfahren) vorgestellt, welches ein Prozess zur Auszeichnung eines einzelnen Prozesses, der als Koordinator (Leader) für andere Prozesse, die in verteilten Systemen sind und eine gemeinsame Gruppe angehören, dient.

Abschließend wird eine Testreihe von *Nicolas Schiper* und *Sam Toueg* vorgestellt, die sich mit der Implementierung und Evaluierung von Leader Election Services in verteilten Systemen auseinander gesetzt haben. Die Evaluierung soll u. a. die Robustheit und Erweiterbarkeit von verteilten Systemen, unter extrem schlechten Netzwerkkonditionen (Störungen und Ausfälle) aufzeigen. Dabei werden verschiedene Leader Election Services vorgestellt, die auf ihre Leader-Verfügbarkeit und Wiederherstellungszeit nach Ausfällen hin überprüft werden.

1. EINLEITUNG

„*Ein verteiltes System ist ein System, in dem sich Hardware- und Software-Komponenten auf vernetzten Computern befinden und miteinander über den Austausch von Nachrichten kommunizieren*“. [George Coulouris]

Diese Systeme, die räumlich (oder logisch) verteilte sind, werden aus den folgenden Gründen miteinander verbunden: **Kommunikations**-Verbund (Übertragung von Daten, insbesondere Nachrichten, an verschiedene, räumlich getrennte Stellen; z.B. E-Mail) I**nformations**-Verbund (Verbreiten von Information an interessierte Personen/Systeme; z.B. WWW) **Daten**-Verbund (Speicherung von Daten an verschiedenen Stellen: bessere Speicherauslastung, erhöhte Verfügbarkeit, erhöhte Sicherheit) **Last**-Verbund (Aufteilung stoßweise anfallender Lasten auf verschiedene Rechner: gleichmäßige Auslastung verschiedener Ressourcen) **Leistungs**-Verbund (Aufteilung einer Aufgabe in Teilaufgaben: Verringerte Antwortzeiten) **Wartungs**-Verbund (Zentrale Störungs-Erkennung und –Behebung: schnellere und billigere Wartung verschiedener Rechner) **Funktions**-Verbund (Verteilung spezieller Aufgaben auf spezielle Rechner; Bereitstellung verschiedener Funktionen an verschiedenen Orten) und **Kapazitäts**-Verbund (Ausnutzung sämtlicher zur Verfügung stehender Rechenkapazität).
Um die o. g. Verbunde optimal nutzen zu können, sollten folgende Eigenschaften in VS gegeben sein: **Gemeinsame Ressourcennutzung**: Hardware, Daten, Dienste etc. gemeinsam nutzen. **Offenheit**: Schlüsselschnittstellen (einheitlich) offen legen. **Nebenläufigkeit**: Mehrere gleichzeitig existierende Prozesse. **Skalierbarkeit**: auch mit vielen Komponenten gut funktionieren können **Sicherheit**: Verfügbarkeit, Vertraulichkeit, Integrität, Authentizität, etc. **Transparenz:** hier im Sinne, **etwas nicht sehen** bzw. durch etwas hindurch sehen können und **Fehlertoleranz**: Fehler erkennen, maskieren, tolerieren.

Quelle: [18], [19], [24]

2. Fehlertoleranz in verteilten Systemen

Ein verteiltes System wird als verlässlich definiert, wenn folgende Anforderungen erfüllt sind:
Verfügbarkeit (Wahrscheinlichkeit, dass das System zu einem bestimmten Zeitpunkt korrekt arbeitet und seine Dienste bereitstellt), **Zuverlässigkeit** (Mit hoher Wahrscheinlichkeit, lange Zeit ohne Unterbrechung arbeiten kann), **Funktionssicherheit** (bezeichnet den Umstand, dass es zu keiner Katastrophe kommt, wenn das System zeitweise unkorrekt arbeitet) und **Wartbarkeit** (Wie leicht kann ein ausgefallenes System repariert werden).
Neben den o. g. Anforderungen an verlässlichen Systemen, sind die Systemmodelle von enorme

Wichtigkeit. Sie beschreiben die allgemeinen Eigenschaften und das Design eines Systems. Das jeweilige Modell sollte die wichtigsten Komponenten, die Art der Interaktion und wie deren individuelles und kollektives Verhalten beeinflusst werden kann, abdecken. Zu diesen Systemmodellen gehören u. a. das **Architekturmodell** (Struktur, bestehend aus Komponenten (Funktionen) und deren Zusammenhang (Interaktionen, Abhängigkeiten)), **Interaktionsmodell** (Berechnungen erfolgen in Prozessen, die durch Nachrichtenaustausch interagieren, die Verzögerung durch Netzkommunikation in Betracht ziehen), **Sicherheitsmodell** (Die Sicherheit eines VS kann erreicht werden, indem die Prozesse und Kanäle gesichert und die Objekte, die von Prozessen verwaltet werden, gegen
unautorisierten Zugriff geschützt werden) und das **Fehlermodell**, dass die Klassifikation von Fehlermöglichkeit und die Abstraktion ihre Ursachen darstellt.

Quelle: [18], [19], [22], [24], [25]

2.1 Das Fehlermodell in verteilten Systemen

In einem VS können sowohl in Prozessen als auch in Kommunikationskanälen Fehler auftreten. Ein *Fehler* wird als das Abweichen vom korrekten oder wünschenswerten Verhalten definiert. Dabei unterscheidet man folgende **Fehlerarten:**

• **Low-Level-Fehler:** Fehler auf der physikalischen Schicht:

- **Auslassungsfehler** (Der Kommunikationskanal lässt Nachrichten aus, da die Verbindung unterbrochen wird)

Abbildung 1

- **Verfälschte Nachrichten** (Eine gesendete Nachricht wurde auf dem Weg vom Sender zum Empfänger verändert bzw. verfälscht)

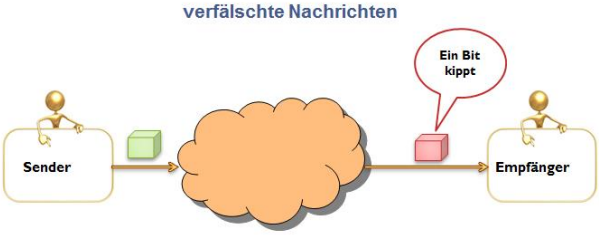

Abbildung 2

- **Verdoppelte oder in falsche Reihenfolge ankommende Nachrichten** (Ein Datenpaket kommt doppelt oder in der falschen Reihenfolge beim Empfänger an)

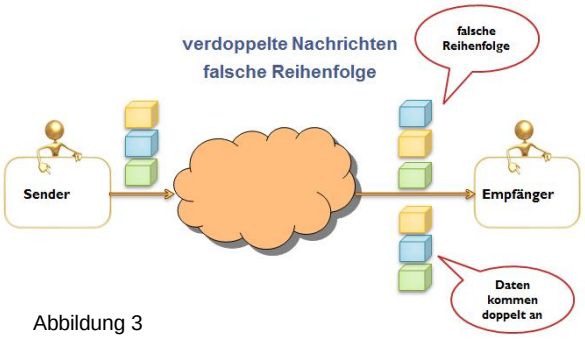

Abbildung 3

• **Anwendungsfehler:** Fehler auf der Anwendungsschicht:

- **Knotenausfall** (Ein Knoten im Netzwerk fällt aus)

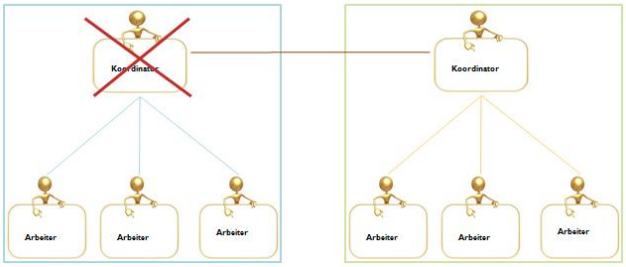

Abbildung 4

- **Zeitfehler** (Nachricht kommt irgendwann an)

Abbildung 5

• **Sonstige Fehler:** (wird im Rahmen diese Arbeit nicht weiter bearbeitet)

- **Byzantinische Fehler** (Ein Prozess lässt Nachrichten aus, sendet beliebige Nachrichten zu beliebigen Zeitpunkten, ggf. entgegen des Protokolls oder es versendet verfälschte oder bösartige Nachrichteninhalte).

Quelle: [19], [21], [22], [25]

2.2 TCP-Protokoll

Durch das TCP-Protokoll wird das Problem von verlorenen, verdoppelten oder verfälschten Nachrichten **(Low-Level**

Fehler) gelöst. Die Bewältigung diese Probleme wird im Folgenden beschrieben:

- Um Verluste von Nachrichten zu vermeiden, werden diese jeweils nach Ablauf einer gewissen Zeit (Timeout) immer wieder verschickt, bis die Ankunft der Nachricht vom Empfänger bestätigt wird (ACK).
- Um Nachrichten zu erkennen, die doppelt bei einem Empfänger angekommen sind, werden sie beim Datentransfer mit einer eindeutigen Sequenznummern versehen.
- Um verfälschte Nachrichten zu entlarven bzw. zu eliminieren, bekommen sie sog. Prüfsummen (checksum) anhand deren man die Richtigkeit einer Nachricht überprüfen kann.

Quelle: [21], [28]

2.3 Commit-Protokoll

Commit-Protokolle regeln die Festschreibung (Commit) von Daten, die durch eine (verteilte) Transaktion beispielsweise in einem Datenbankmanagementsystem verändert werden sollen. Zur Erfüllung ihrer jeweiligen Anforderungen beschreiben Commit-Protokolle, wie die an einer Transaktion teilnehmenden Prozesse (sog. „*agents*“) über einen Koordinator (sog. „Leader“) miteinander kommunizieren müssen, wie Informationen protokolliert (geloggt) werden und wie schließlich die betroffenen Daten festgeschrieben werden. Dabei werden verschiedene Fehlersituationen durch das Protokoll abgefangen, wie z. B. ein Absturz des Koordinators während einer Phase.

Ein Koordinator holt (meist der Prozess, der das Festschreibung (Commit) einleitet) in der ersten Phase des Protokolls die Zustimmung oder Ablehnung zur Festschreibung der Datenveränderungen aller beteiligten Prozesse ein (sog. „Abstimmungsphase“). Nur dann, wenn alle Teilnehmer zustimmen, entscheidet der Koordinator auf „Commit“, ansonsten lautet die Entscheidung „Rollback“ (Zurücksetzen). Ist die Entscheidung gefallen, unterrichtet der Koordinator in der zweiten Phase („Commit Phase“) des Protokolls die Teilnehmer über das Ergebnis. Gemäß diesem gemeinsamen Ergebnisses wird entweder die gesamte Transaktion zurückgesetzt, oder alle Teiltransaktionen werden zum erfolgreichen Ende geführt, indem die zwischenzeitlich gesperrten Ressourcen wieder freigegeben werden.

2.3.1 Algorithmus „2-Phasen Commit“

Während beider Phasen, werden die folgenden Nachrichten zwischen den Teilnehmern ausgetauscht:

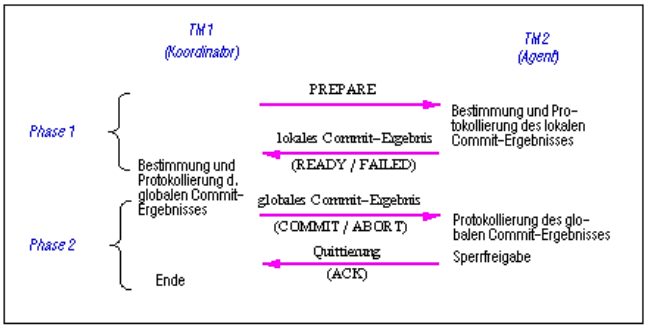

Abbildung 6

➢ **Commit-Request Phase (Phase 1):**

1. Der Koordinator sendet ein **prepare** an alle Teilnehmer und wartet auf Antworten aller Teilnehmer.

2. Die Teilnehmer verarbeiten die Transaktion bis zu dem Punkt, wo die Transaktion entweder mit commit oder rollback abgeschlossen wird. Dabei schreiben sie Einträge in ihr *undo log* und in ihr *redo log*.

3. Die Teilnehmer antworten mit **ready**, wenn die Transaktion erfolgreich war - oder sie antworten mit **failed**, wenn die Transaktion fehlgeschlagen ist.

➢ **Commit Phase (Phase 2):**

Wenn der Koordinator von *allen* Teilnehmern eine **ready** Meldung bekommen hat:

1. Der Koordinator sendet **commit** an alle Teilnehmer.

2. Die Teilnehmer schließen die Transaktion mit commit ab und geben alle Locks und Ressourcen frei.

3. Die Teilnehmer senden ein **acknowledgment** zurück.

4. Der Koordinator beendet die Transaktion, wenn er von allen Teilnehmern die Bestätigung erhalten hat.

Wenn *zumindest einer* der Teilnehmer ein **failed** schickt:

1. Der Koordinator sendet **abort** an alle Teilnehmer.

2. Die Teilnehmer schließen die Transaktion mit rollback ab (mittels des undo logs) und geben alle Locks und Ressourcen frei.

3. Die Teilnehmer senden ein **acknowledgment** zurück.

4. Der Koordinator beendet die Transaktion ebenso mit rollback, wenn er von allen Teilnehmern die Bestätigung erhalten hat.

2.3.2 Algorithmus „3-Phasen Commit“

Beim 2-Phasen-Commit besteht das grundsätzliche Problem, dass Teilnehmer zwischenzeitlich blockiert

werden. Das passiert, sobald ein Teilnehmer seine lokale „Commit"-Entscheidung dem Koordinator mitgeteilt hat. Danach wartet der Teilnehmer auf die globale (gemeinsame) Entscheidung. Das wird vor allem dann problematisch, wenn der Koordinator zwischenzeitlich ausgefallen ist. In dieser Situation kann der Teilnehmer weder die gesperrten Ressourcen freigeben, noch die lokale Transaktion zurücksetzen. Als Abhilfe wurden daher sogenannte "nicht-blockierende" Commit-Protokolle vorgeschlagen, die das Blockierungsproblem unter Inkaufnahme eines erhöhten Aufwandes abschwächen. Ein solches Verfahren ist das 3-Phasen-Commit Protokoll, das nachfolgend beschrieben wird.

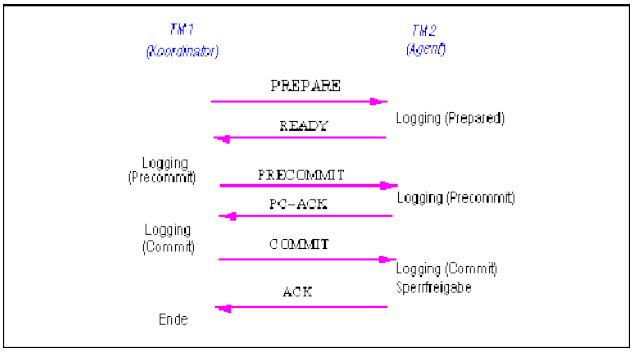

Abbildung 7

Wir beschreiben das 3PC-Protokoll für eine zentralisierte Kommunikationsstruktur zwischen Koordinator und Agenten. Die dabei anfallenden Nachrichten sind in Abb. 7 gezeigt. Man erkennt, dass die erste Phase mit der des 2PC-Protokolls übereinstimmt. Der (nicht gezeigte) Abort-Fall, wenn also wenigstens ein Agent mit FAILED stimmt, wird wie im 2PC-Protokoll behandelt. Eine zusätzliche Phase wird also nur notwendig, wenn alle Agenten mit READY stimmen. In dieser Situation nimmt der Koordinator jetzt zunächst einen Zwischenzustand (**Precommit**) ein, der mit einem entsprechenden Log-Satz protokolliert wird. Der Koordinator teilt das Precommit allen Agenten mit, die dieses Zwischenergebnis ebenfalls protokollieren und mit einer PC-ACK-Nachricht quittieren. Nachdem die PC-ACK-Nachrichten von K der N-1 Agenten eingetroffen sind, entscheidet der Koordinator auf Commit und schreibt den entsprechenden Log-Satz. Die letzte Phase (Mitteilung und Quittierung des Commit-Resultats) stimmt wiederum mit der des 2PC-Protokolls überein. Mit dem Precommit sichert der Koordinator zu, dass er von sich aus die Transaktion nicht mehr zurücksetzt. Allerdings ist es im Gegensatz zum Commit nach einem Precommit des Koordinators nach dessen Ausfall noch möglich, die Transaktion abzubrechen.

Wird während der Commit-Behandlung ein Koordinator-Ausfall erkannt, so erfolgt die Wahl eines neuen Koordinators. Damit der neue Koordinator die Commit-Behandlung fortführen kann, erfragt er zunächst von den überlebenden Rechnern den Commit-Zustand bezüglich der betroffenen Transaktion. Wenn einer der Agenten einen Commit-Satz oder Abort-Satz für die Transaktion protokolliert hat, wird das entsprechende Ergebnis global gültig gemacht. Wenn keiner der Agenten einen Abort- oder Commit-Satz, jedoch wenigstens einer einen Precommit-Zustand vorliegen hat, dann wird das 3PC-Protokoll mit dem Verschicken der PRECOMMIT-Nachrichten fortgesetzt. Es wird anderenfalls auf Abbruch der Transaktion entschieden.

Quelle: [20], [23], [27]

2.4 Leader Election Service

Leader Election Service (Koordinator-Auswahlalgorithmus) ist ein Prozess zur Auszeichnung eines einzelnen Prozesses, der als Organisator/Koordinator (Leader) für andere Prozesse, die in verteilten Systemen sind und eine gemeinsame Gruppe angehören, agiert.

2.4.1 Quality of Service (QoS)

Quality of Service (**QoS**) oder **Dienstgüte** beschreibt die Güte eines Kommunikationsdienstes aus der Sicht der Anwender. Formal ist QoS eine Menge von Qualitätsanforderungen an das gemeinsame Verhalten bzw. Zusammenspiel von mehreren Prozessen.

Aus der Sicht von Leader Election Service werden die QoS-Anforderungen zur Bestimmung der Qualität von Kommunikationsverhalten zwischen Prozessen genutzt und zur Bewertung der Performance des LE-Services. Die drei wichtigsten **QoS-Metriken** werden nachfolgend vorgestellt:

1) **Geschwindigkeitsmetrik:** Sie erfasst die Geschwindigkeit des Leader Election Services. Dabei wird die Zeit gemessen, die der Service braucht, um sich von einem Ausfall (Crash) des aktuellen Leaders zu **erholen**.

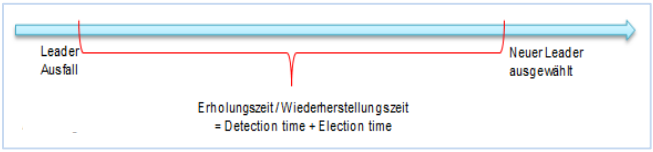

Abbildung 8

Erholen bedeutet: Die Zeit zwischen „Gruppe verliert ihren aktuellen Leader aufgrund eines Ausfalls" und „ es wird ein neuer Gruppen-Leader vom Service bestimmt.

Die **Erholungszeit/Wiederherstellungszeit** wird aus der Zeit zweier Perioden berechnet:

1.1) **Detection Time** = ist die Zeit, die der Service benötigt, um herauszufinden, dass der aktueller Leader ausgefallen ist.

1.2) **Election Time** = ist die Zeit um sicherzustellen, dass alle „alive"-Prozesse in der Gruppe mit einem neuen Leader einverstanden sind.

Beachte, dass die Gruppe während dieser Zeit tatsächlich keinen Leader hat.

2) **Durchschnittliche Fehlerrate:** Erfasst die Präzision & Stabilität von Leader Election Services. Mit anderen Worten, es ist der Anteil an Fehlern, die ein Service macht, in dem es einen vollfunktionsfähigen Leader zurückstuft.

Dieser Art von Fehler kann aus zwei Gründen stattfinden:

2.1) **Fehlerdetektions-Mechanismus** des LE-Services kann fehlerhaft sein. Das bedeutet: Irrtümlich vermuten, dass der Gruppen-Leader ausgefallen ist und dadurch der Trigger ausgelöst wird, den aktuellen Leader zu ersetzen.

2.2) Der **Service ist unstabil**. Es kann ein wohl-funktionierender Leader aus falschen Gründen abgelöst werden. Das heißt, ein Leader könnte zurückgestuft und ersetzt werden, nur weil ein Prozess der Gruppe neu beitritt, der einem kleineren Prozess-Identifier als den des aktuellen Leaders hat.

Quelle: [26], [29]

3) **Verfügbarkeit eines Gruppen-Leaders:** Entspricht der Wahrscheinlichkeit, dass zu einer zufälligen Zeit eine Gruppe einen gemeinsamen, von allen Prozessen ausgewählten und funktionsfähigen Leader hat. Diese Metrik ist besonders interessant für Applikationen, denn sollte zu einer zufälligen Zeit eine Applikation einen leaderbasierten Service nutzen wollen, so entspricht die o. g. Metrik, der Wahrscheinlichkeit, dass genau zu dieser Zeit ein Leader verfügbar ist.

2.4.2 Fehlerdetektor (FD) –Modul mit QoS

Im Kern jedes Leader Election Services existieren sog. Fehlerdetektoren (FD). Sie helfen dabei

- aufzuspüren, ob ein aktueller Leader (Koordinator) fehlgeschlagen bzw. ersetzt werden muss
- festzustellen, welche von den Kandidaten, die den fehlgeschlagenen Leader ersetzen könnten, aktuell betriebs-/einsatzbereit sind.

Für die Experimente in Kapitel 2.6 wurden ein Fehlerdetektor-Modul, zur Entdeckung von Leader-Ausfällen, ein Link Quality Estimator-Modul, zur Abschätzung der Verbindungsqualität im Netzwerk und ein Scheduler-Modul, der die Versendung der *I'm alive*-Nachrichten von beteiligten Prozessen steuert, zu einem gemeinsamen LE-Service-Modul vereinigt.

Wie die genaue Struktur und die Aufgaben der Teilmodule im Einzelnen sind, wird im Folgenden beschrieben:

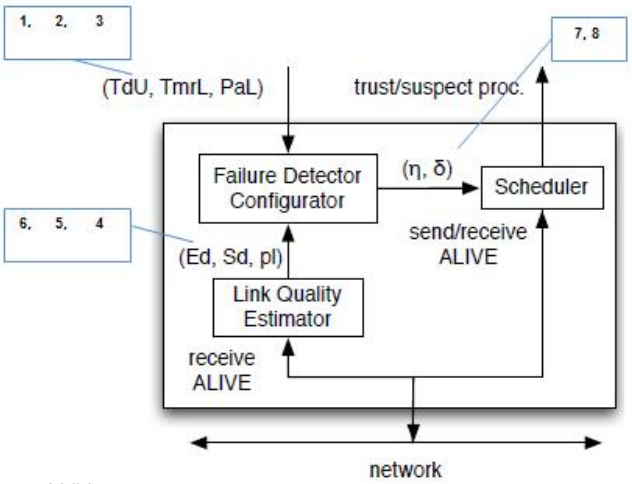

Abbildung 9

a. Stochastic failure detector Algo. von Chen [5]
b. Link Quality Estimator- Module
c. Scheduler-Module

Die o. g. Module stellen einige QoS zur Fehlerdetektion bereit unter den jeweils gegebenen Netzwerkkonditionen.

Wenn ein Prozess p, den Status von einem anderen Prozess q beobachtet, übergibt er die QoS-Anforderungen aus seiner Überwachung in Form von 3 Parametern:

1) TdU = Ist die max. Zeit, die der Fehlerdetektor braucht, um einen Ausfall von Prozess q zu entdecken.

2) TmrL= Ist die min. geschätzte Zeit zwischen zwei Missverständnissen des FD' s, das bedeutet FD macht einen Fehler, wenn es an Prozess p berichtet, dass Prozess q ausgefallen ist, obwohl dieser noch intakt ist.

3) PaL = Ist die min. Zeit für die Wahrscheinlichkeit, dass zu einer zufälligen Zeit, der FD korrekt liegt bzgl. seines Berichtes über die Betriebsbereitschaft von Prozess q.

Die o. g. QoS-Parameter sind abhängig von:

i. Der **Qualität der Kommunikationsverbindung** zwischen Prozess p und q
ii. **Der Häufigkeit:** Wie oft Prozess q die Nachricht *„I'm alive"* an Prozess p sendet.
iii. Vom **Timeout**, dass Prozess p auf dieser *„I'm alive"* - Nachricht anwendet.

Die in Abbildung 9 genannten Module „Link Quality Estimator", „Failure Detector Configurator" und „Scheduler" stehen in Beziehung zu den o. g. QoS-Anforderungen (i, ii, iii). Welcher Modul welche Anforderung zugeordnet ist, wird hier im Weiteren erläutert:

Link Quality Estimator-Module (zu i.)

Mit dem *Link Quality Estimator-Module* wird in regelmäßigen Abständen die Qualität der Verbindung

zwischen Prozess q und p gemessen. Das Messen der „Qualität“ wird anhand von 3- Größen erreicht:

4) **PL** = Wahrscheinlichkeit, dass Nachrichten verloren gehen
5) **Ed** = Erwartungswert der Nachrichtenverzögerung (geschätzte Verzögerungszeit unter den aktuellen Netzwerkkonditionen)
6) **Sd** = Standardabweichung der Nachrichtenverzögerungen („normale“ Verzögerungszeit unter den gegebenen Netzwerkkonditionen)

➔ Die Einschätzung der Qualität der Kommunikationsverbindung erfolgt nach der Meldung „I’m alive“, die Prozess p von Prozess q empfängt.

Failure Detector Configurator- Module (zu ii.)

Der *Failure Detector Configurator-Module* berechnet die Fehlerdetektor-Parameter, die die erforderlichen QoS unter den Annahmen/Voraussetzungen des Netzwerkverhaltens sicherstellen. Genauer dieses Modul berechnet dynamisch:

7) Die **Häufigkeit (η)** in dem Prozess q „I’m alive“-Nachrichten an Prozess p versenden muss
8) Die **Timeouts (δ)**, die Prozess p nutzt, um den Status von Prozess q zu bestimmen.

Die o. g. Größen **Häufigkeit (η)** und **Timeout (δ)** werden anhand von zwei Inputs in diesem Modul berechnet:

a) Die erforderliche QoS aus der Überwachung von Prozess q, das heißt die Werte TdU, TmrL und PaL, die von Prozess p hinterlegt werden.
b) Die geschätzte Qualität der Kommunikations-verbindung zwischen q und p, das heißt der letzter Wert von PL, Ed und Sd, dass vom Link Quality Estimator- Module berechnet wurde.

Scheduler-Module (zu iii.)

Das Modul *Scheduler* nutzt die Werte (**Häufigkeit (η)** und **Timeout (δ)**) des *Failure Detector Configurator- Module.* Es plant die Versendung der *I’m alive*-Nachrichten von Prozess q anhand der Häufigkeit (η) und es nutzt die aktuelle Timeout (δ) und die Zeit in dem Prozess p seine letzte *I’m alive*-Meldung von q empfängt, um die trust/suspect notification (vertrauenswürdige/verdächtige Meldung) an Prozess p zu errechnen.

Häufigkeit (η) + Timeout (δ) + T_letzte Nachricht
=
Anzahl der vertrauenswürdigen/verdächtigen **Meldungen** (trust/suspect Proc.) an Prozess p

Quelle: [5], [26]

2.5 Leader Election Service-Architektur

Die hier beschriebene Leader Election-Architektur ist basierend auf den Fehlerdetektor von Deianov et al. in [6] und auf die Implementierung von Ivan et al. in [11]. Diese Architektur reduziert die Belastung auf das Gesamtnetz und den CPU-Steuerungsaufwand, in dem einige Tasks, die entstanden sind, verteilt werden. Das bedeutet, Abschätzen der Qualität der Kommunikationsverbindung oder die Ermittlung, ob eine Workstation betriebsbereit ist. Damit wird der Gesamtaufwand von diesen Tasks auf verschiedene Applikationen verteilen, die zur selben Zeit den gleichen Service nutzen.

Applikationsprozesse sind mit eine gemeinsamen Bibliothek verbunden, die den Service API (Application Programming Interface) beinhalten. Die Haupt API-Funktion erlaubt den Prozessen sich in dem Service an- und abzumelden (registrieren/abmelden) und einer Gruppe beizutreten bzw. zu verlassen. Diese Bibliothek kommuniziert mit dem „Command Handler Module“, um die Anfragen von Applikationsprozessen zu bedienen.

Für die Nutzung von Leader Election Service, muss ein Prozess p sich erst registrieren im Service und zwar mit einer einmaligen und eindeutigen Prozess-Identifier. Ist das erledigt, kann Prozess p jederzeit jede Gruppe beitreten bzw. verlassen.

Um einer Gruppe g beizutreten, muss Prozess p die folgenden 4-Parameter angeben:

1) Gruppe g’s Identifier
2) Ob Prozess p ein Kandidat für g’s–Führung (Leadership) ist oder nicht

3) Den Weg, dass sich Prozess p wünscht, um den aktuellen Leader von g zu finden.

3.1) Durch ein „interrupt“ von Service, falls sich der Leader von g geändert hat

3.2) Durch „querying“ vom Service, ob Prozess p es so haben möchte.

4. Die QoS (TdU, TmrL, PaL) aus dem bereits beschriebenen Fehlerdetektor, der vom Service zum Auswählen des Gruppen-Leaders benutzt wird.

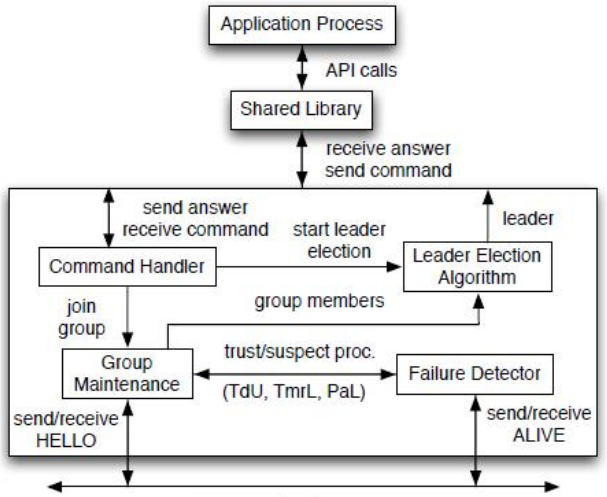

Abbildung 10

Die Kernfunktionalität des Services befindet sich in der „*Group-Maintenance*“, „*Failure Detector*“ und im *„Leader Election Algorithm- Module“*. Für jede Gruppe g baut und verwaltet der „Group Maintenance“-Modul:

a) Eine Menge von Prozessen, die aktuell in g sind
b) Eine Teilmenge an Prozessen von Gruppe g, die aktuell *aktiv* in g sind (Ein Prozess gilt als aktiv, wenn es aktuell „*alive“* ist und um die Gruppenführung von g konkurriert). Abhängig vom Leader Election-Algorithmus, sind entweder alle Prozesse in g *aktiv* oder es bleibt ausschließlich der Leader von g *aktiv*).

Um die o. g. Mengen herausfinden zu können, muss der *Group Maintenan*ce-Module den Prozess-Status jeder Gruppe bestimmen. Die Bestimmung erfolgt anhand des *Failure Detector*-Modul, welches in Kap. 2.4.2 beschrieben ist.

Abschließend der „*Leader election Algorithmus*“- Module, welches ein Leader in jeder Gruppe g in dem der Algorithmus ausgeführt wird verwaltet.

Quelle: [2], [4], [5], [6], [11], [26]

2.6 Leader Election Service-Evaluation

In diesem abschließenden Kapitel, wird eine Experimentenreihe von *Nicolas Schiper* und *Sam Toung* vorgestellt. In den Experimenten werden drei verschiedene Versionen des Leader Election Services (S1, S2, S3) sowie den Aufwand für CPU und Netzwerkbandbreitenkapazität evaluiert und verglichen.

Rahmenbedingung für die Experimente:

Element	Beschreibung
Service-Architektur	Wie in 2.5 beschrieben
Leader Election Algorithmus- Module	3 verschiedene Algorithmen, variiert je nach Service
	(Erklärung folgt)
QoS-Metriken	Wie in 2.4.1 beschrieben
Systemparameter	**Beschreibung**
Netzwerk	5 Netzwerke, 12 Workstations
Workstation	- P4 3.2 GHz mit 512 MB RAM - Betriebssystem: Suse Linux 9.2
Anzahl der Applikationen	12 Prozesse, einer pro Workstation
Dauer der Experimenten	1-5 Tage
Nachrichtenverzögerung	0,025 ms

Beschreibung der Systemparameter:

I. **Workstation-Verhalten:**

Diese Werte sind zwar pessimistisch gewählt im Vergleich zu einer realen Umgebung, sie sollen aber genügend Stichproben zur Bewertung der Service-Belastbarkeit und die durchschnittliche Wiederherstellungszeit in angemessene Zeit unter ungünstigen Umständen liefern.

i) Im Durchschnitt fällt jedes Prozess alle 10 min aus

ii) Erholungszeit einer Workstation nach einem Leader-Ausfall beträgt im Durchschnitt 5 Sekunden.

II. **Kommunikationsverbindung:**

Für die Experimente werden folgende Kommunikationsverbindungen zur Simulation genutzt:

i) verlustbehaftete Verbindung:

Kommunikationsverbindungen mit zufälligen Nachrichtenverluste bzw. Verzögerungen

ii) Ausfallanfällige Verbindung:

Kommunikationsverbindungen, die Gegenstand von zufälligen Ausfällen & Wiederherstellung sind.

III. Fehlerdetektor:

In den Experimenten wurde der Fehlerdetektor (Kap. 2.4.2) eingesetzt mit den folgenden QoS-Parametern:

Für jedes Prozess p, dass ein Prozess q überwacht gilt:

- Prozess p braucht max. 1 Sekunde, um Prozess q’s Ausfall zu entdecken
- Im Durchschnitt macht p höchstens einen Fehler alle 100 Tage bzgl. Status von Prozess q.
- Die Wahrscheinlichkeit, dass Prozess p korrekt q’s Funktionsstatus schätzt, zu einer zufälligen Zeit, liegt bei mind. 99% (0,999999988).

2.6.1 Service S1

Leader Election Algorithmus- kleinster **Prozess-Identifier**

Beschreibung:

- Der **Gruppen-Leader** ist der jeweilige Prozess **mit dem kleinsten Prozess-Identifier** unter allen Prozessen der Gruppe, die als betriebsbereit (*alive*) gelten.
- Um die Betriebsbereitschaft der Prozesse abschätzen zu können, **senden alle Prozesse** in regelmäßigen Abständen (alle η Sekunden) sog. ***I’m alive*-Nachrichten**

an alle Prozesse der Gruppe. Des Weiteren senden sie Timeouts δ, damit Prozess p mitteilen kann, dass Prozess q fehlgeschlagen ist, da er keine *I'm alive*-Nachrichten von q erhalten hat.

Auswertung von:

- Durchschn. Wiederherstellungszeit eines Leaders
- Durchschn. Fehlerrate

Ergebnisse:

Fehlerrate: Service S1 ist **nicht stabil**, es macht über 6 Fehler/Stunde. in allen 5 Netzwerken. wir erinnern uns, ein Fehler tritt auf, wenn der Service einen Leader „unberechtigt" zurückstuft, obwohl dieser voll funktionsfähig ist. Im Falle von S1, wird dieser Fehler getriggert, da ein Prozess der Gruppe beitritt, der einen kleineren Identifier als den des aktuellen Leader hat.

Wiederherstellungszeit: ca. 1 sec. in allen 5 Netzwerken.

Die Leader-Unstabilität aus S1 ist von Nachteil für die Robustheit eines Systems, daher wird im nächsten Schritt ein Leader Election Service (S2) ausgewertet, welches als stabiler gilt.

2.6.2 Service S2

Leader Election Algorithmus- kleinste **Beschuldigungszeit**

Beschreibung:

- Jedes Prozess p behält das letzte Mal (Zeitpunkt) im Auge in dem er verdächtigt wurde ausgefallen zu sein (sog. Accusation time (**Beschuldigungszeit**)) und
- **Prozess p wählt seinen Leader** unter eine Menge von Prozessen aus, die wie folgt aufgebaut sind (2-Phasen):

  Phase 1: P selektiert seinen **lokalen Leader** als den Prozess mit der frühesten Beschuldigungszeit unter den Prozessen, wo Prozess p der Meinung ist, dass diese betriebsbereit (*alive*) sind (Das bedeutet unter den Prozessen, die vor kurzem eine *I'm alive*-Nachricht versendet haben bzw. Prozess p eine *I'm alive*-Nachricht empfangen hat).

  Phase 2: P selektiert seinen **globalen Leader** aus der Menge aller lokalen Leader mit der frühsten Beschuldigungszeit, die nach Prozess p's Meinung betriebsbereit sind.

Auswertung von:

- Durchschn. Wiederherstellungszeit eines Leaders
- Durchschn. Fehlerrate

Ergebnisse:

Fehlerrate: Hier wurden der Service S2 und S1 miteinander unter den gleichen Voraussetzungen verglichen. Schnell war es deutlich, dass im Gegensatz zu S1, S2 **sehr stabil** ist, denn in jedem der 5 Netzwerke wurde beobachtet, dass es **keine *unberechtigten* Zurückstufungen** eines Leader gab.

Wiederherstellungszeit: Die durchschnittliche Leader-Wiederherstellungszeit von S2, hingegen ist etwas größer als die von S1 (aufgrund des Leader-Ermittlungsmechanismus von S2, welches minimal die Zurückstufung von ausgefallenen Leadern verzögert).
Nichts desto trotz, dank der exzellenten Stabilität von S2, hat dieser Service eine bessere Leader-Verfügbarkeit als S1 in allen 5 Netzwerken. Sie stellt zu 99,82% der Zeit einen Leader zur Verfügung.

Bei beiden Services S1 und S2, ist jedoch die Anzahl der *I'm alive*-Nachrichten, die zwischen den Prozessen ausgetauscht werden in der Regel sehr hoch, daher betrachten wir nun ein Leader Election Service (S3) mit einer kleineren Anzahl an Nachrichten („Message-Overhead"), die ausgetauscht werden.

2.6.2 Service S3

Leader Election Algorithmus - **Kommunikationseffizient**

Beschreibung: Dieser Service S3 basiert auf ein Leader Election Algorithmus, welches kommunikations-effizient ist. Das bedeutet nur der ausgewählte Leader einer Gruppe sendet *I'm alive*-Nachrichten an den anderen Prozessen einer Gruppe. Dabei selektiert ein Prozess (wie in S2 beschrieben), seinen Leader anhand der kleinsten Beschuldigungszeit aus eine Menge von Prozessen, die für das „Amt" des Gruppen-Leaders kandidieren.
Kommunikationseffizienz wird erreicht, in dem die Menge aller kandidierenden Prozesse reduziert wird. Die Reduzierung funktioniert wie folgt beschrieben:

1) Ein Prozess p berücksichtigt ein Prozess q als Kandidaten, nur wenn Prozess p direkt *I'm alive*-Nachrichten von Prozess q empfängt.
2) Wenn Prozess p herausfindet, dass ein kandidierender Prozess q existiert, der eine kleinere Beschuldigungszeit hat, folglich besser als Leader-Kandidat geeignet wäre, tritt in diesem Fall Prozess p freiwillig als Kandidat zurück, in dem er keine *I'm alive*-Nachrichten mehr versendet.
   Bedenke aber, wenn Prozess p das Versenden der Nachrichten einstellt, könnten andere Prozesse annehmen, dass p ausgefallen ist, auch wenn das nicht der Fall ist. Dieses LE-Algorithmus sieht allerdings vor, dass diese falsche Anschuldigung Prozess p's-Beschuldigungszeit nicht erhöht.

Auswertung von:

- Durchschn. Wiederherstellungszeit eines Leaders

- Durchschn. Fehlerrate

Ergebnisse:

Bei dem Vergleich von S2 und S3 unter den gleichen Voraussetzung/Einstellungen, hat sich herausgestellt, dass S3 kommunikations-effizienter als S2 ist.

Fehlerrate: S3 ist außergewöhnlich stabil. Wie bereits unter S2 festgestellt, ist der Leader Election Algorithmus (kleinste **Beschuldigungszeit)** sehr robust, da ein funktionierender Leader nie zurückgestuft wird.

Wiederherstellungszeit: Die durchschnittliche Leader-Wiederherstellungszeit von S3 entspricht beinahe der Zeit von S2 (max. 1 Sek.). Daher stellt dieser Service ebenfalls zu 99,82% der Zeit einen funktionierenden Leader zur Verfügung.

2.6.2 Erkenntnisse aus der Auswertung S1, S2 und S3

- **CPU-Auslastung:**
  S2 und S3 sind unmaßgeblich für die CPU-Auslastung. Der schlechteste Wert betrug 0,04% CPU-Last.

- **Netzwerkbandbreite:**
  Die schlechteste Messung betrug 6,48KB/Sekunde Nachrichtenverkehr pro Workstation.

- **Leader Election:**
  Leader-Verfügbarkeit ist in S2 und S3 sehr hoch.

- **Wiederherstellungszeit:**
  Sehr kurze Wiederherstellungszeit in S1, S2 und S3.

Quelle: [26]

3. SCHLUSSWORT

Verteilte Systeme sind heutzutage nicht mehr aus der Technik- und WWW-Welt weg zu denken. In unserem Alltag, der zum größten Teil durch die digitalisierte Welt wie die Nutzung von Clouds, Suchmaschinen, Onlinebestellungen, Onlinebanking usw. bestimmt und meistens erleichtert wird, macht es unerlässlich, sich näher mit diesen Systemen auseinander zu setzen.

Prof. George F. Coulouris (University of London) definiert ein verteiltes System wie folgt:

„*Ein verteiltes System ist ein System, in dem sich Hardware- und Software-Komponenten auf vernetzten Computern befinden und miteinander über den Austausch von Nachrichten kommunizieren*“.

Im Rahmen dieser Ausarbeitung wurden vorab die Bedeutung und der Aufbau von verteilten Systemen näher erläutert. Dabei wurde verdeutlicht, dass verteilte Systeme eine Gruppe unabhängiger Rechner sind, die einem Anwender als ein einzelnes System erscheint. Eigenschaften wie: Gemeinsame Ressourcennutzung, Nebenläufigkeit, Skalierbarkeit, Sicherheit, Transparenz, Offenheit und Fehlertoleranz kennzeichnen die tragenden Säulen einer zufriedenstellenden Systemaufbaus. Speziell ist die Eigenschaft FEHLERTOLERANZ von großer Wichtigkeit, um die Funktionen und die Rolle von Leader Election besser zu verstehen, welches Kern dieser Arbeit bildet.

Was Fehlertoleranz in VS bedeutet, wird erst deutlich, in dem das Fehlermodell vorgestellt wird. Hier werden die Fehlerarten, die in solch ein System auftreten können näher gezeigt. Das bedeutet Fehler auf der physikalischen Schicht (Low-Level Fehler) wie verlorene, verdoppelte und verfälschte Nachrichten, sowie Fehler die auf der Anwendungsschicht erst auftreten, wie Knotenausfall oder „Nachrichten, die irgendwann in der Zukunft ankommen“. Diese Fehler können mit Hilfe von zwei Protokollen entdeckt und eliminiert werden. Zum einen das TCP-Protokoll, das sich um die Low-Level Fehler kümmert. Die Ankunft einer Nachricht wird dadurch gewährleistet, in dem die Nachricht wieder versendet wird, falls keine Empfangsbestätigung (ACK) beim Sender vom Empfänger ankommt. Nachrichten die doppelt beim Empfänger ankommen, können anhand von Sequenznummern erkannt und ggf. verworfen werden. Das Problem der verfälschten Nachrichten, wird dadurch beseitigt, dass gesendete Nachrichten sog. Prüfsummen zugeordnet werden, die dabei helfen sollen, Manipulationen an den Nachrichtenpaketen zu erkennen.

Mit Hilfe des Commit-Protokolls wird das Problem des Knotenausfalls behandelt. Hier wird die Festschreibung (Commit) von Daten, die durch eine (verteilte) Transaktion beispielsweise in einem Datenbankmanagementsystem verändert werden sollen geregelt. Zur Erfüllung ihrer jeweiligen Anforderungen beschreiben Commit-Protokolle, wie die an einer Transaktion teilnehmenden Prozesse („*agents*“) über einen Koordinator („Leader“) miteinander kommunizieren müssen, wie Informationen protokolliert (geloggt) werden und wie schließlich die betroffenen Daten festgeschrieben werden. Dabei werden verschiedene Fehlersituationen durch das Protokoll abgefangen, wie z. B. ein Absturz des Koordinators (Knoten) während einer Phase.
Im Weiteren wird Leader Election vorgestellt, welches ein Prozess zur Auszeichnung eines einzelnen Prozesses, der als Koordinator (Leader) für andere Prozesse, die in verteilten Systemen sind und eine gemeinsame Gruppe angehören, agiert. Abschließend wird eine Testreihe von Nicolas Schiper und Sam Toueg vorgestellt, die sich mit der Implementierung und Evaluierung von Leader Election Services in verteilten Systemen auseinander gesetzt haben. Die Evaluierung zeigt die Robustheit, Erweiterbarkeit von verteilten Systemen, unter extrem schlechten Netzwerkkonditionen (Störungen und Ausfälle). Dabei werden Fehlerdetektoren und Leader Election Services vorgestellt, die für die Erfüllung der QoS-Metriken sorgen. Die dabei entstandenen Testergebnisse sind wie folgt:

- **CPU- Auslastung:**

Die Services (S2 und S3), die den LE-Algorithmus verwenden, der seine Leader anhand von kleinster Beschuldigungszeit auswählen, sind unmaßgeblich für die CPU-Auslastung. (Der schlechteste Wert betrug 0,04% CPU-Last.)

- **Netzwerkbandbreite:**

Die schlechteste Messung betrug 6,48KB/Sekunde Nachrichtenverkehr pro Workstation.

- **Leader Election:**

- Leader-Verfügbarkeit ist in S2 und S3 sehr hoch.
  - Der Algorithmus (kleinster Identifier) wie in S1 ist nicht stabil, da 6 Fehler/Std. (Fehler bedeutet: unberechtigter Ablösung eines funktionsfähigen Leaders).

- **Wiederherstellungszeit:**

Sehr kurze Wiederherstellungszeit in S1, S2 und S3.

4. REFERENCES

[1] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Stable leader election. In proceedings of DISC'01, pages 108–122. Springer-Verlag, 2001.

[2] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. On implementing Omega with weak reliability and synchrony assumptions. In proceedings of PODC'03, pages 306–314. ACM Press, 2003.

[3] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Communication-efficient leader election and consensus with limited link synchrony. In proceedings of PODC'04, pages 328–337. ACM Press, 2004.

[4] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. On implementing Omega with weak reliability and synchrony assumptions. Technical Report HAL-00259018, CNRS - France, November 2007.

[5] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. IEEE Transactions on Computers, 51(5):561–580, May 2002.

[6] B. Deianov and S. Toueg. Failure detector service for dependable computing. In proceedings of FTCS'00, pages B14–B15. IEEE computer society press, 2000.

[7] A. Fernandez, E. Jimenez, and M. Raynal. Eventual leader election with weak assumptions on initial knowledge, communication reliability, and synchrony. In proceedings of DSN'06, pages 166–178. IEEE Computer Society, 2006.

[8] C. Fetzer and F. Cristian. A highly available local leader election service. IEEE Transactions on Software Engineering, 25(5):603–618, 1999.

[9] R. Guerraoui and P. Dutta. Fast indulgent consensus with zero degradation. In proceedings of EDCC'02, pages 191–208. Springer-Verlag, 2002.

[10] I. Gupta, R. van Renesse, and K. P. Birman. A probabilistically correct leader election protocol for large groups. In proceedings of DISC'00, pages 89–103. Springer-Verlag, 2000.

[11] D. Ivan and S. Toueg. An implementation of a shared failure detector service with QoS. 2001. Private Communication.

[12] L. Lamport. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 21(7):558–565, July 1978.

[13] L. Lamport. The Part-Time parliament. ACM Transactions on Computer Systems, 16(2):133–169, May 1998.

[14] M. Larrea, A. Fernandez, and S. Arevalo. Optimal implementation of the weakest failure detector for solving consensus (brief announcement). In proceedings of PODC'00, page 334. ACM Press, 2000.

[15] D. Malkhi, F. Oprea, and L. Zhou. Omega meets Paxos: Leader election and stability without eventual timely links. In proceedings of DISC '05, pages 199–213. Springer, 2005.

[16] A. Most´efaoui and M. Raynal. Leader-based consensus. Parallel Processing Letters, 11(1):95–107, 2001.

[17] R. D. Prisco, B. Lampson, and N. Lynch. Revisiting the Paxos algorithm. Theoretical Computer Science, 243(1–2):35–91, 2000.

[18] Verteilte Systeme. Dozent: Clemens Düpmeier http://www.iai.kit.edu/~clemens.duepmeier/vs-vorlesung.html

[19] Verteilte Systeme und Fehlermodell Uni Hagen 2014. http://www.fernuni-hagen.de/FACHSCHINF/1678/1678_Zusammenfassung.pdf

[20] Commit Protokoll, Uni Leipzig 2014,

http://dbs.uni-leipzig.de/buecher/mrdbs/mrdbs-66.html

[21] TCP-Protokoll. Dr. Volkmar Sieh, Universität Erlangen/Nürnberg 2014. Private Communication.

[22] Fehlermodell, Fehlertoleranz TU Chemnitz 2014, http://osg.informatik.tu-chemnitz.de/lehre/old/ws0708/disos/bsvs10.pdf

[23] Commit Protokoll TU Dresden, https://wwwdb.inf.tu-dresden.de/misc/SS12/WSDM/04%20-%20VerteilteTransaktionsverarbeitung.pdf

[24] verteilte Systeme Universität Paderborn, http://www2.cs.uni-paderborn.de/cs/ag-kao/de/teaching/ws05/vs1/script/vs05_kap1_2seiten.pdf

[25] Fehlertoleranz Universität zu Lübeck, http://media.itm.uni-luebeck.de/teaching/ws2013/vs/PDF/VS-09-Fehlertoleranz.pdf

[26] Paper, Evaluation. FAU Erlangen http://www.cs.toronto.edu/~samvas/teaching/2221/handouts/LeaderElectionImplementations.pdf

[27] 2-Phasen Commit. Wikipedia, http://de.wikipedia.org/wiki/Commit-Protokoll

[28] TCP Protokoll, TU Berlin

https://www2.informatik.hu-berlin.de/~jzapotoc/docs/02studTCP_pr.pdf

[29] QoS. 4whatitis 2014. http://www.4whatitis.com/index.php?page=1645