TECHNISCHE UNIVERSITÄT DRESDEN

FAKULTÄT INFORMATIK
LEHRSTUHL SYSTEMS ENGINEERING
PROF. DR. CRISTOF FETZER

Großer Beleg

Simulation von verteilten Algorithmen im „Finite Average Response Time“-Modell

Josef Spillner
(Mat.-Nr.: 2763483)

Betreuer: Dipl.-Inf. Martin Süßkraut

Dresden, 9. Dezember 2005

Inhaltsverzeichnis

1 Einleitung

Für verteilte Systeme existieren etliche Protokolle, die die Zusammenarbeit der beteiligten Prozesse ermöglichen sollen und gleichzeitig einen negativen Einfluss von denjenigen Prozessen ausschließen müssen, die aufgrund eines bösartigen Verhaltens oder eines Systemabsturzes Schaden anrichten könnten. Diese Protokolle unterliegen gewissen Bedingungen, von denen das zugrundeliegende Kommunikationsmodell eine ist. Mit dem Modell der endlichen durchschnittlichen Antwortzeiten (FAR-Modell) ist es möglich, Abstimmungsverfahren in nahezu asynchronen Systemen zu implementieren, ohne dabei viele der sonst benötigten weiteren Einschränkungen beachten zu müssen.

Als so genannte Agreement-Protokolle wurden in dem Beleg sowohl das *Consensus-Protokoll* als auch das *Zwei-Phasen-Commit-Protokoll* sowie in Ansätzen die *Leader Election* implementiert. Als Voraussetzung und Möglichkeit der Vorführung und Überprüfung ist eine Simulationsumgebung entstanden, die den Namen *FARnodes* trägt, basierend auf einem Kernmodul *stubbornchannels*, welches sich von den *Stubborn Channels* ableitet, die im FAR-Modell verwendet werden.

Konventionen: Für spezielle Fachbegriffe aus dem FAR-Modell und dem Bereich der Verteilten Algorithmen generell wird eine *kursive Schreibweise* verwendet. Befehle für die Kommandozeile, Methodennamen und ähnliche software-bezogene Begriffe werden in `Schreibmaschinenschrift` dargestellt. Für Ausschnitte aus Quelltexten und Simulationsskripten werden spezielle zeilennummerierte Passagen eingefügt. Schlüsselwörter der Simulationsumgebung werden dabei rot, Kommentare blau eingefärbt, und die reservierten Namen der Programmiersprache erscheinen im Fettdruck.

2 Theoretische Grundlagen

2.1 Systemeigenschaften und Kommunikation

Zur Ausführung von Algorithmen in verteilten Systemen ist es notwendig, die Systemeigenschaften zu kennen und sie gegebenenfalls vorteilhaft für eine Implementierung zu nutzen. Reale Systeme werden dabei auf ein abstraktes Modell abgebildet, welches die signifikanten Merkmale derart ausdrücken kann, dass eine Einschätzung und ein Vergleich mit anderen Modellen damit möglich wird. Diese Modellvorstellungen basieren auf gewissen Annahmen über die Zustände und Zustandsänderungen der beteiligten Prozesse sowie der Verbindungen zwischen diesen.

2.1.1 Prozesse

Prozesse, als Begriff hier vereinfachend wie auch häufig in der diesbezüglichen Literator synonym mit Prozessoren verwendet, sind die grundlegenden Elemente zur Ausführung eines Algorithmus. Sie besitzen einen inneren Zustand, der häufig Informationen über andere Prozesse beinhaltet.

Prozesse senden Nachrichten aus, nehmen einkommende Nachrichten an und verarbeiten Informationen. Hierbei sind zwei Eigenschaften von Bedeutung: die Zuverlässigkeit während der Verarbeitung, und die Zeitdauer derselben.

Wenn ein Prozess die Verarbeitung wie geplant durchführt, also einen vorgegebenen Übergang von einem inneren Zustand zu einem weiteren vornimmt, wird er als *korrekt* bezeichnet. Andernfalls liegt ein Fehler vor, der vor, während oder nach der Verarbeitung einer Nachricht zum Ausfall des Prozesses führt, und der Prozess wird als *fehlerhaft* bezeichnet. In der Praxis dabei auftretende Störungen kann man in die Kategorien zufällig und bösartig einteilen. Zu den ersteren gehört *fail-stop*, der spontane Ausfall eines Prozesses, der temporär oder auch endgültig sein kann, sowie *omission*, eine fehlende Behandlung von Nachrichten beispielsweise durch Überlastung. Die zweite Kategorie, auch byzantinische Fehler genannt, erfordert den Einsatz von Authentifizierung zur effektiven Erkennung und soll nicht weiter Gegenstand dieser Belegarbeit sein. Auch temporäre Ausfälle von Prozessen mit Wiederherstellung durch eine *recovery*-Routine werden hier nicht betrachtet, es sollen ausschließlich endgültige Ausfälle (*crash*)

auftreten.

Die Verarbeitungsdauer bei korrekten Prozessen ist endlich. Sie ist im Falle der synchronen Prozesse nicht notwendigerweise bekannt, es existiert aber eine (bekannte) obere Grenze Φ. Im asynchronen Fall existiert diese nicht. Das Modell der partiellen Synchronität wurde aus diesem Grund eingeführt. Es erlaubt, eine Verarbeitungsdauer anzunehmen, deren obere Grenze existiert (aber nicht bekannt ist), oder aber die obere Grenze, egal ob bekannt oder unbekannt, erst nach einer Stabilisierungszeit *GST* eintritt, die selbst nicht bekannt ist. Da es in der Praxis schwer möglich ist, das Verhalten nach der *GST* zu garantieren, ist es üblich, den stabilen Zustand hinreichend lange garantieren zu können, beispielsweise bis ein in einem Prozess ablaufender Algorithmus terminiert.

2.1.2 Kanäle

Kanäle sind Verbindungen zwischen je zwei Prozessen mit bestimmten Parametern wie Bandbreite, Übertragungsgeschwindigkeit und Zuverlässigkeit. Dieser letzte Parameter ist hierbei von Bedeutung für diese Belegarbeit. Das Spektrum reicht von zuverlässigen Kanälen (*reliable channels*), bei denen jede gesendete Nachricht beim Empfänger ankommt, über eingeschränkt zuverlässige Kanäle (*CR-reliable channels*), bei denen bis zum Empfang die Funktionalität des Senders sichergestellt sein muss, bis hin zu so genannten *best-effort*-Kanälen, bei denen ein Übertragungsfehler bedeuten kann, dass die gesamte Kommunikation beendet wird. Oftmals wird dieser letzte Fall durch verbindungsorientierte Protokolle wie TCP/IP realisiert, dessen Modifikation TCP-R eine Wiederaufnahme der Verbindung im Fehlerfall vornimmt und damit ähnliche Qualitäten wie *CR-reliable* Kanäle vorweisen kann.

Am Ende des Spektrums stehen unzuverlässige Übertragungsdienste - *unreliable channels* genannt - welche sehr schwache Annahmen machen: Nachrichten, die empfangen werden, wurden auch gesendet; es wird keine Nachricht mehrfach empfangen; eine endliche Teilmenge der gesendeten Nachrichten wird empfangen. Dies ist in der Praxis auf das Protokoll UDP abbildbar, auch wenn dieses Doppelungen im Nachrichtenempfang nicht verhindert.

Für die Modellierung eines Systems ist es darüberhinaus wichtig zu wissen, ob Nachrichten verloren gehen können. Dies leitet sich direkt aus der Zuverlässigkeit ab. Für die Praxis hat sich zwischen dem Transport ohne Verlust von Nachrichten und dem mit möglichem Verlust der Begriff Fairness oder *fair lossy* herausgebildet, der besagt, dass auch bei möglichen verlorenen Nachrichten immer noch ein genügend großer Anteil aller gesendeten Nachrichten beim Empfänger ankommen wird.

Die Übertragungsdauer von Nachrichten in den Kanälen ist ähnlich der Verarbeitungsdauer von Prozessen bestimmend für die Synchronität der Kanäle. Im synchronen Fall ist eine obere Grenze Δ bekannt,

im asynchronen Fall ist sie nicht existent. Das dazwischen liegende partiell synchrone Verhalten erlaubt eine existierende unbekannte obere Grenze oder auch eine bekannte obere Grenze, die erst nach einer (vorher unbekannten) Stabilisierungszeit *GST* eintritt von da ab anhält.

2.1.3 Gesamtsystem: Synchronität und Fehlererkennung

Unter Nutzung der Eigenschaften von Prozessen und Kanälen hat sich ein Schema zur Beschreibung der Kommunikation im verteilten System etabliert. Die Synchronität eines Systems wird direkt durch die Synchronität der beteiligten Elemente bestimmt. So gibt es voll synchrone Systeme, die einfach in der Theorie sind, sich dafür aber in der Praxis schwer implementieren lassen. Die asynchronen Systeme hingegen, die keine Annahmen über die maximale Übertragungsdauer Δ auf einem Kanal oder die maximale Bearbeitungszeit einer Nachricht Φ in einem Prozess machen, sind nicht ausreichend für einige Algorithmen, sofern auch ein einziger Prozess fehlerhaft sein kann. Dieser bekannte Sachverhalt wurde als Unmöglichkeit der Implementierung eines Consensus-Protokolles in [FLP83] nachgewiesen, er betrifft aber auch weitere ähnliche Protokolle. Eine Vorstellung des generellen Consensus-Problems erfolgt in einem der nächsten Abschnitte.

Aus diesem Grund gibt es mehrere verschiedene Abstufungen von partiell synchronen Systemen, also Kombinationen von (partiell) synchronen (allerdings nicht asynchronen) Prozessen und Kanälen, die im wesentlichen auf einige wenige Grundtypen zurückzuführen sind. Ausschlaggebend für deren Nutzbarkeit für die Abarbeitung von den oben angesprochenen Protokollen ist die Fähigkeit eines Prozesses, langsame Prozesse oder lange Übertragungswege von ausgefallenen Prozessen unterscheiden zu können.

Daraus resultiert die Zahl der Prozesse, die fehlerhaft sein dürfen, ohne die Funktion eines Consensus-Protokolls zu beeinträchtigen. Diese Zahl wird als t-Resilienz bezeichnet, und liegt zwischen 0 und n, der Gesamtzahl der Prozesse. Laut [DLS88] existiert n im vollständig asynchronen System nicht, folglich ist keine Konstruktion eines t-resilienten Algorithmus möglich, im vollständig synchronen System ist n gleich t, und im partiell synchronen System ist n gleich $2t + 1$, folglich muss die Mehrzahl der Prozesse korrekt sein, sofern die Kanäle nicht vollständig synchron arbeiten.

Um die auf einem Modell aufbauenden Algorithmen zur Lösung eines Problems nicht dem Einfluss des zeitlichen Verhaltens aussetzen zu müssen, wurden für dieses ein Kapselungskonzept vorgesehen. Als Begriff wurde dabei der *Fehlerdetektor* (*failure detector*) eingeführt. Ein Fehlerdetektor ist ein gedankliches Modul als Zusatz zu einem Prozess, welches zu jedem Zeitpunkt eine Liste der verdächtigten anderen Prozesse liefern kann. Dabei ist diese Liste als unzuverlässig (*unreliable*) einzustufen.

Fehlerdetektoren können entsprechend ihrer Leistungsfähigkeit und sonstigen Eigenschaften in sehr vie-

len Ausprägungen auftreten, im hier betrachteten Fall liegen sie aber in einer von acht verschiedenen Basis-Klassen, wie sie durch [CT96] definiert wurden.

Die Klasse ergibt sich aus dem Potential der Erkennung aller nicht funktionierenden Prozesse, der Vollständigkeit (*completeness*), bei gleichzeitiger Vermeidung der fehlerhaften Einstufung korrekt funktionierender Prozesse, Genauigkeit (*accuracy*).

Es wird eine Unterscheidung vorgenommen in schwache und starke Fehlerdetektoren, die angeben, ob sich die ersten beiden Eigenschaften auf mindestens einen oder auf alle anderen Prozesse beziehen. (Es wurde allerdings in [CT96] bereits nachgewiesen, dass sich schwach-vollständige Fehlerdetektoren über ein Pseudoverfahren indirekt in stark-vollständige umwandeln lassen.) Unter den schwach-genauen Klassen werden schwach vollständige Detektoren der Klasse W als „schwach“ und stark-vollständige der Klasse S als „stark“ bezeichnet. Ist die Genauigkeit hingegen stark, so nennt man die Klasse Q für schwach vollständige und P für stark vollständige Fehlerdetektoren. Vertreter der letztgenannten Klasse werden auch als „perfekt“ bezeichnet.

Treten diese Eigenschaften erst nach einer bestimmten Laufzeit des Algorithmus ein, so wird dies durch den Zusatz *eventually* (letztendlich) gekennzeichnet, andernfalls als *perpetually* (fortwährend). Es entstehen somit zusätzlich die Fehlerdetektorklassen $\diamond W$, $\diamond S$, $\diamond Q$ und $\diamond P$.

In der folgenden Übersicht werden die Eigenschaften Genauigkeit und Vollständigkeit für einen Fehlerdetektor D definiert. Dabei ist Pc ein korrekter Prozess und Pm ein fehlerhafter, und $D_{Pc_i}(Pc_j)$ eine Verdächtigung des Prozesses Pc_j durch den Fehlerdetektor von Pc_i.

Genauigkeit: Ausschluss von Fehlern 1. Ordnung.

schwach: Ein korrekter Prozess wird von keinem anderen korrekten Prozess verdächtigt:

$\exists Pc_j D_{Pc_i}(Pc_j) = 0 \forall Pc_i$

Erfüllt durch $\diamond W$, $\diamond S$, W und S, sowie durch alle stark genauen Fehlerdetektoren.

stark: Kein korrekter Prozess wird von einem anderen korrekten Prozess verdächtigt:

$\forall Pc_j D_{Pc_i}(Pc_j) = 0 \forall Pc_i$

Erfüllt durch $\diamond Q$, $\diamond P$, Q und P.

Vollständigkeit: Ausschluss von Fehlern 2. Ordnung.

schwach: Ein korrekter Prozess verdächtigt alle fehlerhaften Prozesse:

$\exists Pc_j D_{Pc_j}(Pm_i) = 1 \forall Pm_i$

Erfüllt durch $\diamond W$, $\diamond Q$, W und Q, sowie durch alle stark vollständigen Fehlerdetektoren.

stark: Alle korrekten Prozesse verdächtigen alle fehlerhaften Prozesse:

$\forall Pc_j D_{Pc_j}(Pm_i) = 1 \forall Pm_i$

Erfüllt durch $\diamond S$, $\diamond P$, S und P.

In vollständig asynchronen Systemen sind Fehlerdetektoren nicht konstruierbar, da Consensus mit allen hier betrachteten Fehlerdetektoren lösbar ist, jedoch dies dem Resultat von [FLP83] widersprechen würde. In vollständig synchronen Systemen hingegen kann sogar ein perfekter Fehlerdetektor P trivialerweise über Timeouts realisiert werden.

Im partiell synchronen Modell wäre ein beliebiger Fehlerdetektor aus den oben vorgestellten acht Klassen ausreichend, es sind jedoch die *perpetual*-Fehlerdetektoren P, Q, S und W nicht implementierbar, so dass für die weiteren Abschnitte nur die letztendlichen Detektoren $\diamond P$, $\diamond Q$, $\diamond S$ und $\diamond W$ betrachtet werden sollen. Für einen nährere Erläuterung sei auf [LFA04] verwiesen.

Dabei ist $\diamond W$ der schwächste Fehlerdetektor, mit dem sich das Consensus-Protokoll implementieren lässt ([CHT92]). Als Annahme folgt, dass weniger als die Hälfte der teilnehmenden Prozesse fehlerhaft sein darf, während bei Nutzung des (nicht implementierbaren) Detektors W diese Einschränkung nicht existieren würde.

Fehlerdetektor-Klasse	Genauigkeit	Vollständigkeit	Bezeichnung
$\diamond W$	schwach nach $t \epsilon T$	schwach	letztendlich schwach
$\diamond S$	schwach nach $t \epsilon T$	stark	letztendlich stark
$\diamond Q$	stark nach $t \epsilon T$	schwach	-
$\diamond P$	stark nach $t \epsilon T$	stark	letztendlich perfekt

Eine offene Frage nach der Lösung des Consensus-Problems im partiell synchronen Modell blieb: Lässt sich das Problem auch ohne die Annahmen des Modells lösen? Gibt es also noch andere Erweiterungen des vollständig asynchronen Systems? Das FAR-Modell ist eine mögliche Antwort darauf.

2.2 Kanalmodell: Stubborn Channels

Bevor das FAR-Modell vorgestellt wird, ist es günstig, ein Kanalmodell zu betrachten, welches zur Lösung von verteilten Algorithmen in asynchronen Systemen (augmentiert mit Fehlerdetektoren) entworfen wurde, aber auch die Annahmen des FAR-Modells erfüllt. Es sind dies die so genannten *Stubborn Channels*, vorgestellt in [GOS97].

Stubborn Channels basieren nicht auf *best-effort*-Verbindungen wie TCP/IP, sondern im Gegensatz dazu auf unzuverlässigen wie UDP, wobei gewisse Eigenschaften garantiert werden können. Sie liegen damit

auf der Skala der Zuverlässigkeit zwischen den Verbindungstypen *CR-reliable* und *best-effort*. Genau genommen sind sie eine Abwandlung der *CR-reliable* Kanäle unter Wegnahme der Bedingung, dass der benötigte Pufferplatz unendlich sein müsse, was in der Praxis nicht realisierbar wäre.

Das grundlegende Prinzip ist wie folgt: Sendet eine Anwendung eine Nachricht m an einen Kommunikationspartner p, und anschließend keine Nachricht mehr bis zum Eintreffen der Bestätigung von p für m, so wird m nach unbestimmter Zeit auf alle Fälle ankommen und eine Bestätigung n von p losgesendet werden, die ebenfalls ankommt. Dies wird dadurch erreicht, dass zwar die Anwendung nur eine Nachricht sendet (also in den Sendepuffer schreibt), die Übertragungsschicht jedoch bis zur Bestätigung (bzw. unendlich lange im Fall der Bestätigung selbst) die Nachricht wiederholt sendet. Für jede Verbindung existiert pro Richtung genau ein Sendepuffer und ein Empfangspuffer der Größe eins, das heißt, genau eine Nachricht m kann jeweils darin aufgenommen werden. Damit wird eine simple Flusskontrolle erreicht: Egal wie viele Nachrichten ein Empfänger von einem Sender zugestellt bekommt, es befindet sich immer maximal eine einzige in seinem Empfangspuffer.

Zusammengefasst sind die beiden Eigenschaften von Stubborn Channels:

No-Creation: Wenn ein Prozess q eine Nachricht m erhält, dann hat ein Prozess p diese Nachricht an q gesendet.

Stubborn: Es seien p und q zwei korrekte Prozesse. Wenn p eine Nachricht m an q sendet, und danach unendlich lange keine weitere Nachricht mehr an q sendet, dann wird q irgendwann m erhalten.

Stubborn Channels können sowohl durch die Stubborn-Eigenschaft (durch Überschreiben) als auch durch die zugrunde liegende Kanalimplementierung Nachrichten verlieren. Eine Implementierung, die gewisse Fairness-Eigenschaften annimmt, kann den möglichen Verlust im zweiten Fall allerdings ignorieren, da durch die Stubborn-Eigenschaft jede gesendete Nachricht irgendwann ankommen wird.

2.3 Das FAR-Modell

Als Verhaltensmuster von Systemen unter einem hohen Kommunikationsaufkommen hat sich ergeben, dass Laufzeiten von Nachrichten oft nur zu bestimmten Belastungszeiten sehr lang sind, außerhalb dieser Zeiten jedoch wieder auf kleinere Werte zurückfallen. Diese Beobachtung hat zu einem neuen Modellansatz geführt, der unter dem Namen FAR-Modell (*Finite Average Response Times*) [FSS05] bekannt ist. Die Grundannahme ist dadurch begründet, dass lange Reaktionszeiten eines korrekt funktionierenden Prozesses zwar ab und zu aus Gründen der Überlastung auftreten können, dazwischen aber ebenso Nachrichten mit kurzen Reaktionszeiten übermittelt werden. Als zweite Annahme setzt das Modell eine

schwache Uhr voraus, also etwa einen inkrementierenden Zähler, dessen Frequenz nicht weiter spezifiziert ist und damit variabel sein kann.

Die drei sehr schwachen Annahmen sind also:

Endliche durchschnittliche Nachrichtenlaufzeit: Die Umlaufzeit Δt von Nachrichten m_i zwischen zwei Prozessen ist im Mittel endlich. Es existiert jedoch keine feste obere Grenze für diese Zeit.
$\frac{\sum_{i=1}^{n} \Delta t(m_i)}{n} < \infty$

Flusskontrolle: Die Kanäle müssen eine Flusskontrolle derart implementieren, dass langsame Prozesse nicht durch schnelle Prozesse überlastet werden können.

Schwache Uhr: Die Zeit, die für die Inkrementierung eines Integer-Wertes benötigt wird, ist nicht Null.
$\Delta t_{tick} = \Delta t_{inc(v)} >= G; G > 0; Gvariabel$

Es werden für Algorithmen im FAR-Modell Stubborn Channels mit einer festgelegten maximalen Nachrichtengröße (*fixed-size*) verwendet. Dadurch wird die Annahme der Flusskontrolle erfüllt. Durch Partitionierung lassen sich allerdings auch Nachrichten beliebiger Größe „stückweise“ versenden.

Desweiteren wird durch die Eigenschaften des Modells ein letztendlich-perfekter Fehlerdetektor (*eventually perfect failure detector*) der Klasse $\diamond P$ mit der speziellen Bezeichnung EA-FD ermöglicht. Als Anwendung im FAR-Modell wird er im nächsten Abschnitt genauer beschrieben.

2.4 Protokolle und Anwendungsfälle

Auf Basis des FAR-Modells lassen sich nun bekannte Protokolle aus dem Bereich der verteilten Algorithmen implementieren. Dabei sollten die Eigenschaften der Protokolle selbst nicht verändert werden, da nur der zugrunde liegende Transportmechanismus ausgetauscht wird und dieser den Ablauf der Protokolle genauso zusichern soll wie es von existierenden Mechanismen wie beispielsweise partiell synchroner Kommunikation getan wird.

2.4.1 Fehlererkennung

Ein auf FAR basierender Fehlererdetektor (mit den Eigenschaften eines *eventually perfect failure detector* $\diamond P$) basiert auf folgender Überlegung: Jedes Mal, wenn eine Bestätigung beim Sender eintrifft, wird

die Laufzeit der zugeordneten Nachricht gemessen, die sich als Summe aus Sendezeit, Verarbeitungszeit beim Empfänger und Sendezeit der Bestätigung ergibt.

Benötigt wird neben dem Grundsystem eine so genannte schwache Uhr, von der nur bekannt ist, dass sie zwar unendlich langsam gehen kann, aber nur endlich schnell. Die Genauigkeit der Uhr ist darüber hinaus nicht bekannt.

Die Messungen werden dann als Sequenz betrachtet. Es gibt einen Grenzwert für die Nachrichtenlaufzeit (*timeout*), bei dessen Überschreitung die Nachricht als langsam eingestuft wird, ansonsten als schnell. Die Zahl der langsamen Nachrichten (*slowmessages*) und die der schnellen zwischen zwei langsamen Nachrichten (*fastmessages*) wird vermerkt. Der Grenzwert wird dabei logarithmisch erhöht, wenn eine langsame Nachricht eintrifft, und linear, wenn eine schnelle Nachricht eintrifft. Bei Eintreffen einer langsamen Nachricht wird die Zahl der schnellen Nachrichten auf 0 gesetzt und damit fällt der Grenzwert wieder. Da der Timeout stets endlich ist, wird ein korrekter Prozess stets einen nicht mehr antwortenden Prozess erkennen können.

Die Berechnung dieses Grenzwertes erfolgt gemäß der Formel:

$$\textbf{timeout} = (1 + \textbf{fastmessage}) * (1 + \log(1 + \textbf{slowmessages}))$$

Dadurch wird erreicht, dass nach einer gewissen „Einschwingungszeit“ die Aussagen des Fehlerdetektors zur Korrektheit eines Prozesses stimmt und sich nicht weiter verändert.

2.4.2 Consensus-Problem

Das Consensus-Problem handelt von Teilnehmern, die sich gegenseitig hin und wieder Vorschläge unterbreiten und sich zum Schluss auf einen dieser Vorschläge einigen müssen. Das Consensus-Protokoll beschreibt nun, wie verschiedene vollständig untereinander vernetzte Prozesse über eine definierte Kommunikation diesen gemeinsamen Wert aushandeln. Der Einfachheit halber wird dabei ein asynchroner rundenbasierter Ablauf angenommen, es werden also pro Runde von jedem Teilnehmer Nachrichten verschickt und die Bestätigungen dafür entgegengenommen, wobei die Runden der einzelnen Teilnehmer anfänglich nicht notwendigerweise synchronisiert sind.

Es müssen nach [GOS97] die folgenden Bedingungen erfüllt sein.

Terminierung: Jeder korrekte Prozess wird sich letztendlich auf einen Wert festlegen.

Übereinkunft: Keine zwei korrekten Prozesse entscheiden sich unterschiedlich.

Gültigkeit: Wenn ein Prozess sich für einen Wert entscheidet, dann ist dieser Wert von einem anderen Prozess vorgeschlagen worden.

Nach [DLS88] sind Gültigkeit und Übereinkunft die Sicherheitsbedingungen, welche während der gesamten Laufzeit des Algorithmus gewahrt sein müssen, während die Terminierung erst nach dem letztendlichen Eintreffen der globalen Stabilisierungszeit (GST) gefordert ist.

Im FAR-Modell existiert keine GST. Gemäß [FSS05] ist Consensus dennoch im FAR-Modell lösbar. Eine Implementierung ist Teil dieser Belegarbeit.

2.4.3 Leader Election

Im Leader Election-Problem geht es darum, unter allen Teilnehmern eines verteilten Systems einen zu bestimmen, der als Anführer dient, und zwar mindestens so lange, bis der nächste Anführer gewählt wurde. Varianten der klassischen Leader Election dienen der Gewinnung einer einheitlichen Kenntnis der aktuellen Prozess- und Verbindungsstrukturen vor der Verkündung des Wahlergebnisses (konvergente Leader Election) sowie einer möglichst langen Verweildauer eines Leaders (Strong/Stable Leader Election).

Zumindest letztere ist unter den Annahmen des FAR-Modells nicht lösbar. Eine Hinzunahme lokaler Uhren führt zum Timed FAR-Modell, in welchem ein Lösungsalgorithmus existiert. Der Algorithmus ist ansatzweise als Simulationsskript für das Modell der partiell synchronen Systeme vorhanden, eine Umsetzung des für FAR angepassten Algorithmus ist aber im Rahmen dieser Belegarbeit aus Zeitgründen unterblieben.

3 Die Simulationsumgebung FARnodes

3.1 Vorstellung der Software

Für den experimentellen Nachweis und die Visualisierung der angesprochenen Protokolle ist eine Software erstellt worden, die, aufbauend auch auf den Annahmen des FAR-Modells, vielseitig verwendbar ist. Das verteilt einsetzbare Framework FARnodes ist in der Programmiersprache Python verfasst und besteht aus mehreren Programmen, die je nach Anwendung zum Einsatz kommen.

Es werden zwei prinzipielle Ausführungsmodi unterstützt: Zum einen die (lokal ablaufende) Simulation, bei der virtuelle Prozesse über Kanäle mit bestimmten simulierten Eigenschaften miteinander kommunizieren, zum anderen die lokale oder verteilte reale Ausführung, bei der Prozesse in einem Betriebssystem Nachrichten über die jeweils vorhandenen Netzwerkprotokolle niederer Ebene austauschen, solange dadurch die Modallannahmen nicht verletzt werden. Eine Simulation mit aus realen Durchführungen gewonnenen Werten (so genannten Traces) ist ebenfalls möglich. Dabei wird die Antwortzeit eines Prozesses um den Wert künstlich verzögert, der in der verteilten Ausführung für die Übertragung der Nachricht verwendet wurde, was abzüglich eines Scheduler-Overhead dem real gemessenen Verhalten sehr nahe kommt.

FARnodes umfasst etwa 3000 Zeilen teilweise komplexen Python-Code, wobei rund die Hälfte für eine minimale Simulation benötigt wird. Als Abhängigkeit ist dabei nur ein installierter Python-Interpreter, möglichst ab der Version 2.3, nötig. Für die vollständige Nutzung der Simulationsumgebung empfiehlt sich das Vorhandensein eines farbunterstützenden Terminals, eines SSH-Clients und der dazugehörigen Hilfsprogramme, sowie für die grafischen Oberflächen das Paket Pygame als Python-Anbindung an die Grafikbibliothek SDL und PyKDE als Anbindung an KDE.

Die folgenden Abschnitte behandeln vor allem die Konzepte und Hintergründe zur Umsetzung. Konkrete Nutzungsbeispiele und eine API-Dokumentation liegt der Software selbst bei, entsprechende Referenzen sind im Anhang dieser Belegarbeit zu finden.

3.1.1 Kernelemente

Als grundlegende Bibliothek fungiert das Python-Modul `stubbornchannels.py`, welches von den Programmen eingebunden wird. In ihr sind die Klassendefinitionen für Prozesse, Kanäle und einen implizit verwendeten Scheduler enthalten. Die Prozesse werden hierbei als Knoten (*nodes*) bezeichnet und enthalten als separate Komponenten einen Sender und einen Empfänger, sowie optional einen Fehlerdetektor.

3.1.1.1 Prozesse

Prozesse sind die aktiven Objekte innerhalb der Ausführungsumgebung. Sie können sich entweder auf lokale oder auf entfernte reale Prozesse beziehen und stellen somit im zweiten Fall nur symbolische Objekte (ähnlich einem Proxy) dar, die sämtliche Eingaben ignorieren.

Intern ist die Prozessklasse `node` aus einem Sendermodul, einem Empfängermodul sowie weiteren Attributen aufgebaut. Jedes Objekt dieser Klasse unterstützt zwei Callbacks: einerseits für nutzerspezifizierte Methoden auf Basis eines Zeitgebers (`callback()`), andererseits für wahrgenommene Änderungen an der Netzwerkstruktur, beispielsweise durch Fehlerdetektormodule (`fdcallback()`). Desweiteren besitzt jedes Objekt einen *idle*-Status, der als Indikator dafür dient, dass eine Simulation beendet werden kann. Dennoch ist ein derart markierter Prozess nicht stillgelegt, sondern nimmt weiter eingehende Nachrichten an. Anwendungsspezifisch kann dieses Verhalten durch Abfrage des Attributes `isidle` gesteuert werden.

Sowohl Sender- als auch Empfängermodul können eine Ausfallwahrscheinlichkeit sowie eine Mindestverzögerung bis zu deren erstmöglichem Eintreten aufweisen. Ohne diese Zuweisung wird in einer Simulation kein Prozess ausfallen.

Das Empfängermodul beinhaltet neben der Initialisierungsroutine eine Methode zur Generierung einer Bestätigung einer eingetroffenen Nachricht, deren Rückgabewert das Senden eben jener Bestätigung bestimmt, sowie einer dazugehörigen Post-Routine, die nur den Nachrichteninhalt ohne Header übertragen bekommt und keine Rückgabe liefert. Das Empfängermodul kann für Simulationszwecke eine bestimmte minimale Verarbeitungszeit aufweisen.

Das Sendermodul beinhaltet große Teile der Fehlerdetektormodule, da über dieses Modul die Bestätigungen für versandte Nachrichten behandelt werden. Eine Methode zur Generierung einer Nachricht inklusive Header ist ebenso vorhanden wie eine zum Einlesen der Bestätigung. In beiden Methoden werden Zeiten gemessen, verglichen oder abgespeichert, um dem jeweils aktivierten Fehlerdetektormodul Hinweise auf die Nachrichtenlaufzeit geben zu können.

Fehlerdetektoren lassen sich natürlich auch auf höherer Ebene direkt in den Simulationsskripten implementieren. Um dies nicht jedes Mal mit tun zu müssen, sind zwei davon bereits im Modul `stubborn-channels` vorhanden: Der Fehlerdetektor EA-FD für die Nutzung im FAR-Modell, und der PS-FD für Simulationen im partiell synchronen Modell. Die jeweils aktuelle „Vorhersage“ des Fehlerdetektors lässt sich unabhängig von dem Callback auch stets über das Vorhandensein in den Attributslisten `slownodes` und `fastnodes` eines jeden `node`-Objekts abfragen.

3.1.1.2 Kanäle

Kanäle sind passive Objekte, die nach ihrer Erzeugung mit jeweils einem Senderprozess und einem Empfängerprozess verknüpft werden müssen. Die Eigenschaften eines jeden Kanals umfasst die Wahrscheinlichkeit, eine Nachricht zu verlieren (im Normalfall 0), eine Mindestverzögerung bis zum erstmöglichen Eintreffen dieser Wahrscheinlichkeit sowie eine festgelegte maximale Nachrichtengröße, die jedoch wie auch im Normalfall durch die Angabe der Größe 0 deaktiviert ist. Der Grundtyp eines Kanals richtet sich nach der Verlässlichkeit - *reliable*, *stubborn*, *best-effort* oder *unreliable* - und muss mit dem Subtyp verträglich sein, der entweder *internal* für prozessinterne direkte Kommunikation über Python-Methoden oder *udp* beziehungsweise *tcp* für netzwerkbasierte Kommunikation sein kann und im Normalfall mit Ausnahme von TCP automatisch festgelegt wird.

Die folgende Tabelle stellt alle Kombinationen aus Kanaltyp und dem verwendeten Subtyp dar, wobei oft der Subtyp nur mit zusätzlichen auf ihm aufbauenden Erweiterungen in Frage kommt:

Zuverlässigkeit	Subtyp Simulation	Subtyp verteilte Ausführung
reliable	internal	-
cr-reliable	-	- (tcp-r möglich)
stubborn	internal	udp (tcp möglich)
best-effort	-	tcp
unreliable	internal (mit Parametern)	udp

Jede Kanalverbindung wird intern durch zwei Kanäle, je einen für jede Richtung, repräsentiert. Dieses Vorgehen entspricht der üblichen Architektur auch anderer Simulationsumgebungen.

Wenn wie im FAR-Modell eine Bestätigung für gesendete Nachrichten erwartet wird, so wird dafür ein separater Puffer verwendet, um das gegenseitige Überschreiben von Nachricht und Bestätigung zu verhindern.

Je nach Problemstellung könnte man desweiteren auf das für Stubborn Channels vorgesehene *Piggy-Backing* zurückgreifen, also eine Nachricht zusammen mit der Bestätigung für eine vorherige Nachricht der Gegenrichtung senden. Da es für derartige Nachrichten allerdings keine Empfangsgarantien gibt und

es auch noch nicht für Simulationen auf höherer Ebene benötigt wurde, ist *Piggy-Backing* momentan nicht implementiert.

Zu erwähnen ist an dieser Stelle noch der „Gentlemen-Timeout“. Dieser ist im Normalfall auf einen Wert größer Null gesetzt und verhindert, dass die ständig über einen Kanal gesendete Nachricht auf Netzwerk- oder Betriebssystemebene zu einer Überlastung führt. Er wird für sämtliche auf realen Netzwerkprotokollen basierenden Kanäle verwendet. Es werden dadurch keine theoretischen Annahmen verletzt, und die in der Praxis eventuell auftretenden Verzögerungen sind vernachlässigbar. Für spezielle Ausführungen wie beispielsweise im LAN kann dieser Timeout manuell auf Null zurückgesetzt werden.

3.1.1.3 Scheduler

Im Scheduler werden sämtliche Objekte eines Systems über die Methode `register()` registriert, unabhängig davon, ob es sich dabei um Prozesse oder Kanäle handelt. Auch die Zuordnung von Kanälen zu Prozessen wird über die Methode `connect()` bzw. `autoconnect()` im Scheduler realisiert.

Der Scheduler iteriert pro Aufruf der internen Methode `work()` über die Liste der Prozesse, um ausstehende Nachrichten zu senden respektive zu empfangen. Für die prozessinterne Zustellung werden die Nachrichten von dem Sendepuffer des sendenden Prozesses in den Empfangspuffer des Zielprozesses bewegt, natürlich unter Berücksichtigung der eventuell eingestellten virtuellen Fehlerrate des Kanals. Eine prozessexterne Zustellung erfordert die Methode `do_send()` des entsprechenden Kanals, in welcher die Zugriffe auf die Netzwerkschnittstelle gekapselt sind.

Die Methode `work()` wird nicht direkt von Simulationsskripten aufgerufen, sondern über die globale Methode `schedule()`, welche ihrerseits noch `schedule_internal()` als Zwischenschritt verwendet. Der Grund dieser Separierung sind potentielle temporäre Unterbrechungen eines einzelnen `schedule()`-Aufrufes durch den Nutzer, die damit nicht an mehreren Stellen des Aufrufes gleichzeitig abgefragt werden müssen.

Je nach Aufrufmodus von `schedule()` kann eine zeit- oder eine ereignisgesteuerte Arbeit vom Scheduler aufgenommen werden. Der erste Modus umfasst unendliche sowie zeitlich oder aus der Anzahl der internen Arbeitsschritte berechnete begrenzte Ausführungslängen, während die Ereignissteuerung ähnlich einer `select()`-Funktion eine Menge von Prozessen auf Aktivität hin beobachtet.

Eine weitere Unterstützung für Ereignisse, unabhängig von der obigen, ist die Änderung der Einstufung eines Prozesses durch einen Fehlerdetektor. Während die durch Ereignisse hervorgehenden Änderungen durch den jeweils beobachtenden Prozess selbst mitgeteilt werden, ist es die Aufgabe des Schedulers, die rein zeitlich bedingten Änderungen wie Überschreitung eines Timeout mitzuteilen. Jeder Fehlerdetektor

besteht damit in der Implementierung aus einem Prozessteil und einem Schedulerteil.

Auch statistische Zwischenwerte mit Statusinformationen über alle Objekte lassen sich über den Scheduler ausgeben, die Methode dafür lautet `information()`.

3.1.2 Kommandozeilen-Oberfläche

Das Programm `sc-commandline` ist die Hauptanwendung zur Durchführung einer lokalen Simulation. Es liest ein Simulationsskript ein und arbeitet es in einem Schritt ab. Die Interaktion mit dem Benutzer kann durch eine Unterbrechung (Tastenkombination Ctrl + C) erfolgen, welche eine Inspektions-Shell hervorruft. In dieser ist die Ausgabe von Statusinformationen möglich, aber auch die direkte Modifikation der internen Python-Objektstruktur durch den Aufruf einer integrierten Python-Umgebung. Nach dem Beenden der Inspektions-Shell kann die Simulation fortgesetzt werden, da ein virtueller Zeitgeber verwendet wird, welcher an die Systemzeit der Hardware-Uhr gebunden ist, jedoch die Startzeit sowie eventuelle Unterbrechungsdifferenzen herausrechnet.

Die Ausgabe des Programms erfolgt zeilenweise und mit farblichen Markierungen für den jeweiligen Typ der Meldung, wie Statusinformationen, Warnungen, Syntaxfehler oder Meldungen des Fehlerdetektors. Eine Auflistung und Beschreibung dieser Meldungen findet man in der Nutzerdokumentation von FARnodes.

3.1.3 Grafische Oberfläche

Mit Hilfe der grafischen Oberfläche `sc-simulation` ist es möglich, bereits erzeugte Simulationen sowohl schrittweise als auch komplett abzuarbeiten. Dabei werden die angelegten Elemente und Kanalverbindungen grafisch dargestellt und animiert, und desweiteren die aktuelle Position in der Simulation angezeigt. Als übersichtlichere Variante mit einer Draufsicht auf die Topologie ist noch `sc-simulation-topview` erstellt worden, welches von der Funktionalität her identisch mit `sc-simulation` ist.

Für die Realisierung wurde `pygame` verwendet, eine Python-Hülle um die Grafikbibliothek SDL. Damit sollte der Einsatz plattformübergreifend möglich sein. Es werden jedoch nur einfache lineare Simulationsskripte voll unterstützt. Für eine komplette analytische Ausführung mit Anzeige der aktuellen Position im Simulationsskript wäre eine Nutzung der Python-internen Informationen wie Stackframe, Sprungadressen und dergleichen notwendig, was den Rahmen des Programmes gesprengt hätte.

3.1.4 Verteilte Ausführung

Der FARnodes-Manager `fn-manager` ist ein Werkzeug, mit dem Simulationsskripte entsprechend der Ortsangaben der beteiligten Knoten verteilt und aufgerufen werden können. Diese Angaben können sowohl auf der Kommandozeile mitgegeben als auch direkt aus der Topologiebeschreibung der Simulation herausgelesen werden, wobei sich letzteres auf statische Topologien beschränkt.

Der Einsatz ist möglich, sofern voll qualifizierte Ortsangaben verwendet werden, also mindestens der Hostname, optional auch Benutzernamen und Partitionierungsangaben zur Unterscheidung von Ausführungsinstanzen unter ein und derselben Nutzerkennung auf dem selben Rechner. Das Schema lautet damit *[user@]host[/partition]*. Da die Knoten selbst zur Kommunikation über Rechnergrenzen hinweg die Angabe der Portnummer benötigen, und sich eventuell ein Gateway vor dem eigentlichen Ausführungshost befindet, komplettiert sich eine derartige Angabe zu *[proxyhost:proxyport!][user@]host:port[/partition]*.

Parameter	Beschreibung	Relevanz
host	Hostname des ausführenden Systems	Manager und Simulation
port	Portnummer des Knotens zum Nachrichtenempfang	Simulation
user	Nutzerkennung für die Ausführung	Manager (und Simulation zu Beginn)
partition	Schema zur Unterteilung einer Nutzerkennung	Manager (und Simulation zu Beginn)
proxyhost	Gateway für die SSH-Verbindung	Manager
proxyport	SSH-Port auf dem Gateway bzw. Ausführungsrechner	Manager

Der Aufruf von FARnode führt dann je nach Ortsangaben dazu, dass das Programm `sc-commandline` zusammen mit dem Simulationsskript entweder in ein lokales Verzeichnis oder über `scp` in ein entferntes Verzeichnis kopiert, und dann entweder über `exec()` oder `ssh` aufgerufen wird. Das Programm erkennt dann selbständig, welche der definierten Knoten ihm als „lokal“ zugeordnet werden. Zur Optimierung des Loginvorganges wird ein asymmetrisches Schlüsselpaar erzeugt, wonach der öffentliche Teil des Schlüssels beim ersten Login mit SSH auf den entfernten Rechner übertragen wird und somit die zukünftige Nutzung passwortlos erfolgen kann.

Als alternatives Aufrufmodell unterstützt der FARnodes-Manager eine Überprüfung aller in Frage kommenden Rechner auf ihre Unterstützung der Simulationssoftware hin. Dabei wird das Resultat in Tabellenform und mit Signalfarben gekennzeichnet ausgegeben.

3.1.5 Performance

Trotz der Implementierung in einer bytecode-interpretierenden Sprache sind keine nennenswerte Geschwindigkeitsnachteile erkennbar. Im realen Betrieb ist ohnehin die Netzwerkanbindung ausschlagge-

bend für das Gesamtverhalten des verteilten Systems. Eine Ausnahme scheint die Erzeugung von Python-Objekten im Größenbereich von mehreren tausend zu sein, wie es bei der kombinatorisch hohen Zahl von Kanälen auftreten kann.

Eine möglicherweise performantere Reimplementierung der Kernklassen in C++ wurde während der Entwicklungszeit der Software in Form einer Bibliothek `libfarmodel.so` erstellt. Es hat sich herausgestellt, dass dies für die Praxis nicht notwendig war, der Vollständigkeit halber soll es aber hier mit erwähnt werden. Mit Hilfe des Interface-Generators `SWIG` wird für diese eine Einbindung in Python ermöglicht. Speziell wäre es auch möglich, innerhalb von Python von diesen Klassen abzuleiten. Somit hätte man zwar nicht die volle Flexibilität, da auf das Innere der Klassen kein Zugriff besteht, bekäme im Gegensatz dazu aber eine schnellere und auch auf Systemen ohne Python-Interpreter einsetzbare Simulationsbibliothek.

3.2 Aufbau und Durchführung einer Simulation

Die Simulation wird pro teilnehmender Programminstanz durch ein Ablaufskript realisiert. In dieser beispielsweise `test.scsim` genannten Datei befinden sich einfache Python-Methodenaufrufe zur Erzeugung von Sendern, Empfängern und Kanälen, sowie zur Verbindung und Nutzung der Kanäle mit Nachrichten. Ebenfalls möglich sind erweiterte Anweisungen, sogar die Erstellung eigener Klassen ist möglich, die von den ursprünglichen abgeleitet sind und dazu dienen können, Nachrichten abzufangen oder gewisse Eigenschaften hinzuzufügen.

Ein minimales Skript sieht wie folgt aus:

```
1  # Erzeugen der Komponenten
2  s1 = node("Sender")
3  r1 = node("Empfaenger")
4  c1 = channel(type = 'stubborn')
5
6  # Verbindungsstrukturen
7  connect(s1, r1, c1)
8
9  # Ablaufsteuerung
10 send(s1, r1, "hello world")
11 schedule(3)
```

Sowohl Senderteil als auch Empfängerteil eines Knotens sowie der Knoten selbst sollten einen Namen zugewiesen bekommen, wobei die Nichtzuweisung eine automatische Namensgebung im Schema „sender#???“, „receiver#???“ bzw. „(node#???)“ erzwingt. Desweiteren können optionale benannte Parameter, d.h. Schlüssel-Wert-Paare, in beliebiger Reihenfolge übergeben werden. Für den Empfänger kann man so beispielsweise den Hostnamen als Parameter `url` angeben. Fehlt die Angabe, handelt es sich um einen lokalen Empfänger, der nur interne Nachrichten entgegennimmt. Ist sie hingegen in Form einer

IP-Adresse oder eines Hostnamens vorhanden, so wartet der erzeugte Knoten, sofern diese Angabe mit der Umgebung seiner Ausführungsinstanz übereinstimmt, von Beginn an auf Nachrichten. Handelt es sich um einen „entfernt“ befindlichen Empfänger (dazu zählen auch Instanzen unter anderen Nutzerkennungen und mit unterschiedlichen Partitionen), dann dient die lokale Instanz nur zur Symbolisierung, die nicht auf Eingaben reagiert.

Mit diesen Empfängern wird stets per UDP bzw. TCP kommuniziert (`remote locality`), während lokale Empfänger die Nachrichten, die von lokalen Sendern kommen, zum Zweck der Simulation direkt als Python-Objekte zugestellt bekommen (`local locality`). Eine Hybris ist ein Empfänger mit der IP-Adresse 127.0.0.1 bzw. einer äquivalenten Angabe, der sich zwar im lokalen Adressraum befindet (und damit beispielsweise der grafischen Oberfläche den Zugriff auf den internen Zustand ermöglicht), aber dennoch selbst mit lokalen Sendern über UDP bzw. TCP kommuniziert (`local-net locality`).

Jeder Kanal kann mit genau einem Sender und Empfänger verbunden werden, jedoch ist es durchaus zulässig, mehrere Kanäle pro Sender oder Empfänger angeschlossen zu haben. Der Versand von Nachrichten wäre in dem Fall gewissermaßen parallelisierbar. Auch ist es möglich, die Verbindungsstrukturen zur Laufzeit zu ändern, Objekte zu löschen oder neue hinzuzufügen.

Nachrichtenobjekte können völlig beliebig an die `send()`-Methode übergeben werden - von Zahlen und Zeichenketten hin bis zu kompletten Objektstrukturen. Die Daten werden zu einer Zeichenkette serialisiert und im Tupel zusammen mit einem eindeutigen, stets inkrementierenden Identifikator versendet, die die Zuordnung der Bestätigung zur ursprünglichen Nachricht ermöglicht. Auf Empfängerseite wird eine Deserialisierung vorgenommen, und je nach Kanalmodell eine Bestätigung entsprechend mit dem selben Identifikator zurückgesendet.

Die Methode `schedule()` übernimmt dabei die wichtige Aufgabe der Abfrage aller eingehenden und ausgehenden Nachrichten und der dafür vorgesehenen Ereignisbehandlungen. Für eine interne Nachrichtenzustellungen muss sie drei mal, ansonsten aber wesentlich öfter aufgerufen werden. Aus diesem Grund gibt es verschiedene Aufrufvarianten, die in dem folgenden Ausschnitt erläutert werden.

```
1  n1 = node()
2
3  # Zeitsteuerung: 100 mal die interne Methode work() aufrufen
4  schedule(100)
5  # Zeitsteuerung: eine Sekunde lang arbeiten
6  schedule(timeout = 1.0)
7  # Ereignissteuerung: so lange arbeiten, bis n1 Aktivitaet zeigt
8  schedule(selectset = [n1])
9  # Unendlich lange arbeiten
10 # (Abbruchbedingungen koennen durch Reimplementierung jeglicher Klassen
11 # gegeben sein, dafuer existiert scheduler.cancel())
12 schedule()
```

Die Reimplementierung einer Klasse soll hier nur kurz skizziert werden. Ein oft benötigter Fall ist dabei die Nutzung der Methode `generate_ack_postroutine()` des Empfängerteils eines Knotens, die es ermöglicht, individuell auf eingegangene Nachrichten zu reagieren.

```
1  # Derivat der Empfaengerklasse
2  class MyReceiver(Receiver):
3          # Initialisierung der Elternklasse sowie eigener benoetigter Attribute
4          def __init__(self, n = None, url = None, port = None):
5                  Receiver.__init__(n, url, port)
6                  self.my_value = 42
7
8          # Virtuelle Methode, aufgerufen bei Eintreffen einer Nachricht
9          # Ein ventuelles ACK (z.B. bei Stubborn Channels) wird vorher versandt
10         def generate_ack_postroutine(self, sender, message):
11                 print "OK, we got", str(message), "from", sender.name
12
13 # Erzeugung eines Prozesses mit der spezialisierten Empfaengerklasse
14 r2 = MyReceiver(url = "localhost", port = 12345)
15 n2 = node("MyNode", receiver = r2)
```

3.3 Auswertung einer verteilten Ausführung

Es ist sehr sinnvoll, eine grafische Analyse eines durchgeführten Experiments vorzunehmen, da durch die automatisierte Zusammenführung von Informationen schnell und präzise Aussagen über das Laufzeitverhalten eines Algorithmus möglich sind. FARnodes unterstützt die Analyse durch eine Sammlung von Werkzeugen, die alle unterschiedliche Verwendungszwecke und Anforderungen an die Eingabedaten haben.

Für eine Untersuchung des Laufzeitverhaltens des Fehlerdetektors EA-FD existiert `showstats.py`. Die grafische Anzeige gibt basierend auf der Log-Ausgabe von `sc-commandline` für jede Nachricht zu einem Prozess an, wie groß der Timeout des Eintreffens der Bestätigung bis zur Verdächtigung des Prozesses durch den Fehlerdetektor zu dem Zeitpunkt war. Der Timeout wird durch ein dunkelblaues Quadrat über dessen Kantenlänge symbolisiert, während darauf in Dreiecksform die tatsächliche Umlaufzeit angegeben wird. Solange diese noch unterhalb des Timeout liegt, erscheint das Dreieck grün, ansonsten geht die Färbung in rot über. Ein Beispiel dazu findet sich in der Abbildung 4.1 im Kapitel über die Auswertung der Experimente.

Die Übernahme von real gewonnenen Verzögerungswerten, so genannten Traces, in eine Simulation zur deterministischen Wiederholung eines Szenarios ist noch nicht voll automatisch möglich, wird jedoch durch das Werkzeug `makestats.py` unterstützt. Es erwartet als Eingabedatei jeweils die von einem Prozess erzeugte Logdatei. Eine Verwendung der (bereits zusammengefassten) Ausgabe des FARnodes-Managers für alle beteiligten Prozesse ist noch nicht möglich. Die von dem Programm für jeden verwendeten Kanal erzeugten Wertelisten können als Trace-Parameter an `sc-commandline` übergeben

werden. Sie führen zu einer künstlichen Verzögerung beim Senden der Bestätigungen gemäß der Werte in diesen Listen.

Das Programm *FARnodes Analyser*, unter `tools/analyser` zu finden, ist eine auf KDE-Basis entstandene Desktop-Anwendung, die genau diese Aufgabe übernimmt. Als Eingabe dient die textuelle Ausgabe von `fn-manager`. Deren einzelnen Abschnitte werden den entsprechenden physikalischen Ausführungsinstanzen zugeordnet und anhand kausaler Merkmale wie Sende- und Empfangsvorgängen von Nachrichten in eine auch zeitlich übereinstimmende Reihenfolge gebracht. Dieses Verfahren umgeht die Problematik der nicht notwendigerweise synchronisierten Uhren auf der Ausführungshardware.

Neben einer Echtzeit-Ansicht der Topologie (Abbildung 3.1), die von `sc-simulation-topview` übernommen wurde, werden Listen aller verwendeten Prozesse, Kanäle und Nachrichten angelegt. Die Überprüfung der aktuellen Tätigkeit der Komponenten sowie aggregierter Merkmale wie der mittleren Antwortzeit von Nachrichten ist damit auf einfache Art und Weise möglich. Über eine Eingabemaske für ein Schlüsselwort können sehr einfach diejenigen Nachrichten herausgefiltert werden, die von Interesse sind, um so einfacher deren Status überprüfen zu können (Abbildung 3.2).

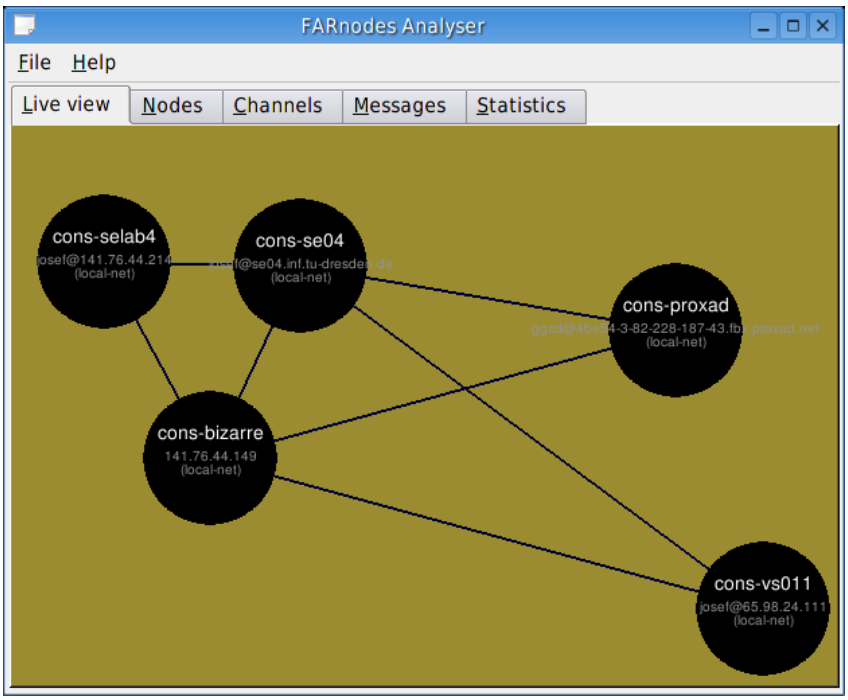
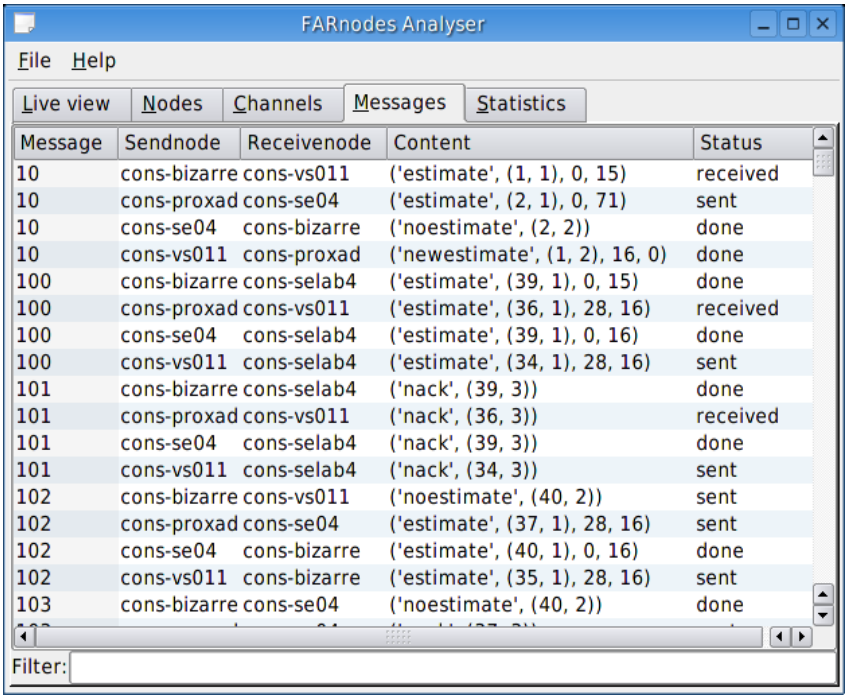


Abbildung 3.1: Darstellung der Systemtopologie im *Analyser*



Message	Sendnode	Receivenode	Content	Status
10	cons-bizarre	cons-vs011	('estimate', (1, 1), 0, 15)	received
10	cons-proxad	cons-se04	('estimate', (2, 1), 0, 71)	sent
10	cons-se04	cons-bizarre	('noestimate', (2, 2))	done
10	cons-vs011	cons-proxad	('newestimate', (1, 2), 16, 0)	done
100	cons-bizarre	cons-selab4	('estimate', (39, 1), 0, 15)	done
100	cons-proxad	cons-vs011	('estimate', (36, 1), 28, 16)	received
100	cons-se04	cons-selab4	('estimate', (39, 1), 0, 16)	done
100	cons-vs011	cons-selab4	('estimate', (34, 1), 28, 16)	sent
101	cons-bizarre	cons-selab4	('nack', (39, 3))	done
101	cons-proxad	cons-vs011	('nack', (36, 3))	received
101	cons-se04	cons-selab4	('nack', (39, 3))	done
101	cons-vs011	cons-selab4	('nack', (34, 3))	sent
102	cons-bizarre	cons-vs011	('noestimate', (40, 2))	sent
102	cons-proxad	cons-se04	('estimate', (37, 1), 28, 16)	sent
102	cons-se04	cons-bizarre	('estimate', (40, 1), 0, 16)	done
102	cons-vs011	cons-bizarre	('estimate', (35, 1), 28, 16)	sent
103	cons-bizarre	cons-se04	('noestimate', (40, 2))	done

Abbildung 3.2: Darstellung aller gesendeten Nachrichten

3.4 Vorhandene Simulationsskripte

Die Nutzbarkeit der Software erfordert meist speziell entwickelte Ablaufskripte, es stehen jedoch für bestimmte Aufgaben fertige Beispiele bereit. Zu beachten ist, dass einige davon für lokale Simulationen ausgelegt sind, andere hingegen real existierende Hostnamen eingetragen haben, was auf eine erfolgte verteilte Ausführung schließen lässt.

Die Beispielskripte sind eingeteilt in verteilte Algorithmen (im Verzeichnis `distributed`), Testaufrufe für verschiedene Topologieszenarien (`locality`) sowie weitere, nicht konkret eingeordnete Skripte direkt im Hauptverzeichnis `examples`.

Verteilte Algorithmen: So lässt sich beispielsweise innerhalb der verteilten Algorithmen das Consensus-Problem über die Datei `consensus.scsim` nachprüfen. Da das Protokoll eine hohe Symmetrie aufweist, wird hierbei nur ein Prozessobjekt erzeugt und dieses dann entsprechend oft geklont. Es sollte deshalb beispielsweise über `sc-commandline -i 5 consensus.scsim` aufgerufen werden, um 5 Instanzen des Prozesses einen gemeinsamen Wert finden zu lassen. Trivialerweise funktioniert es aber natürlich auch mit nur einem Teilnehmer.

Die Leader Election wurde sukzessive implementiert, wobei die einzelnen Schritte als separate Skripte zur Verfügung stehen. Ausgehend von sehr starken Systemannahmen ohne jegliche Ausfälle in `leader-`

`gamma.scsim` wurde der Ausfall von Knoten in `leader-omega.scsim` zugelassen.

Das Zwei-Phasen-Commit-Protokoll, unter `commit.scsim` zu finden, ist durch die starke asymmetrische Ausprägung je nach Prozessrolle mit unterschiedlichen Abläufen ausgestattet. Es ähnelt von der Struktur her stark dem Consensus-Protokoll.

Topologieszenarien: Die unter dem Begriff *Locality* zusammengefassten Skripte demonstrieren noch einmal die möglichen prozesslokalen sowie verteilt arbeitenden Verbindungsarten. Prozesslokale interne Verbindungen sind in `nodes.scsim` in Gebrauch, während UDP-Verbindungen innerhalb eines Prozesses in `localnetnodes.scsim` verwendet werden. Über Prozessgrenzen hinweg ist die Kommunikation auf UDP-Basis sowohl in Ausführungsinstanzen unter der selben Nutzerkennung möglich (`partitions.scsim`), als auch über physikalische Systemgrenzen hinweg, z.B. über Ethernet in `remote.scsim`. Schließlich gibt es auch Mischformen, wobei die Prozesse nicht notwendigerweise mit einander kommunizieren können. Entsprechende Warnungen bei Fehlen von voll qualifizierten Verbindungsinformationen werden durch `manynodes.scsim` provoziert.

Sonstige Skripte: Zur Überprüfung der Eigenschaften des Fehlerdetektors EA-FD existiert `ea-fd.scsim`. Der Sendeprozess in dieser Simulation passt sich in seiner Wartezeit an das reale Verhalten des Empfängerprozesses an, welches durch eine zufällig bestimmte künstliche Verzögerung der Beantwortung von Nachrichten von bis zu 10 Sekunden gekennzeichnet ist.

Verschiedene Varianten des Scheduling werden in `dynamic.scsim` verwendet. Konkret wird das Skript nach dem Senden der ersten Nachricht erst dann weiter ausgeführt, wenn dazu die Bestätigung eingetroffen ist.

4 Erzielte Ergebnisse

4.1 Simulation

Einige der vorgestellten Simulationsskripte sind auf ihr Zeitverhalten hin untersucht worden. Die Ergebnisse dieser Experimente sollen hier näher betrachtet werden.

4.1.1 Fehlerdetektor

Das Consensus-Protokoll sollte im Idealfall mit einer Nachricht pro Phase und Empfänger lösbar sein. Eine Simulation hat gezeigt, dass dies im FAR-Modell auch tatsächlich der Fall ist.

Für die reale Ausführung hingegen und ebenso für Simulationen mit vielen Teilnehmern wäre dieser Idealzustand sehr unwahrscheinlich. Aufgrund der bereits vorgestellten Parameter zur zufallsbestimmten Erzwingung von Übertragungs- und Verarbeitungsfehlern kann man auch in einer lokalen Simulation bestimmte Verhaltensmuster betrachten.

Durch eine Parametrisierung der Simulation wird erkennbar, wie der Fehlerdetektor EA-FD in Einzelfällen reagiert. Die nachfolgende Grafik 4.1 ist durch das Werkzeug `showstats.py` erzeugt worden. (Siehe Abschnitt 3.3 zu den Auswertungs-Werkzeugen.) Als Eingabe diente dabei das ebenfalls bereits vorgestellte Skript `ea-fd.scsim`. Sehr deutlich lässt sich erkennen, wie zu Beginn auch Nachrichten mit relativ kurzer Laufzeit vom EA-FD als langsam eingestuft wurden, während nach der adaptiven Erhöhung des Timeout auch Nachrichten mit längerer Laufzeit im tolerierten Rahmen blieben und somit der Prozess trotz nichtdeterministischer Antwortzeiten nicht mehr verdächtigt wurde.

4.1.2 Consensus mit vielen Teilnehmern

Die Umsetzung des Consensus-Problems basiert auf dem Algorithmus von [GOS97]. Das Protokoll beginnt damit, dass alle Teilnehmer an einen ihnen bekannten Koordinator die Nachricht mit dem Vorschlag senden. Sind mehr als die Hälfte von ihnen, inklusive dem Koordinator selbst, dabei erfolgreich, so kann der Koordinator einen dieser Werte für angenommen erklären und ihn wiederum an alle Teilnehmer senden. Diese müssen nun, ähnlich der ersten Runde, ihre Zustimmung erklären, wobei der Koordinator die

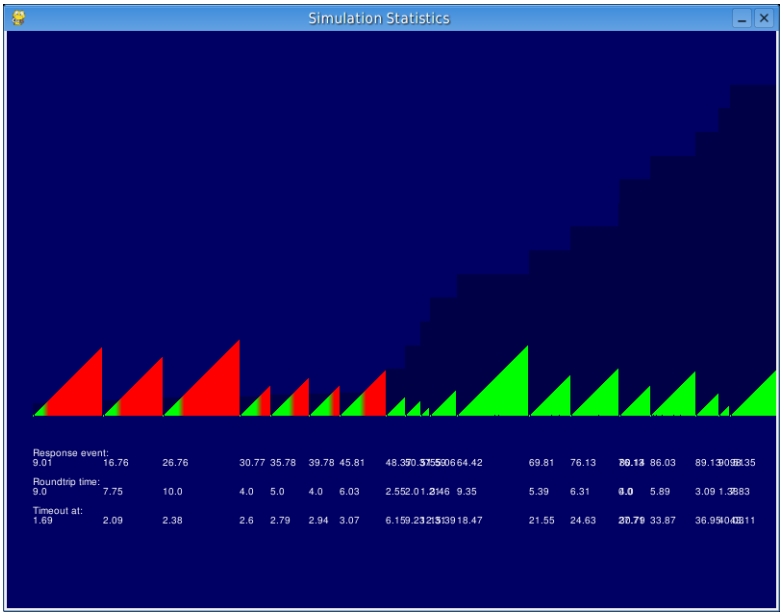


Abbildung 4.1: Verdächtigung eines Prozesses durch den EA-FD

Wahl erst dann für endgültig erklärt, wenn auch hierbei mehr als die Hälfte aller Teilnehmer erfolgreich war.

Der Algorithmus teil jede Runde in fünf Phasen ein, von denen der Koordinator die Phasen eins bis vier, die anderen Teilnehmer hingegen die Phasen eins, drei und fünf durchlaufen. Der Koordinator selbst rotiert, je nach aktueller Runde des jeweiligen Prozesses. Aufgrund der asynchronen Runden kann es also vorkommen, dass mehrere oder gar keine Teilnehmer mit Koordinatorrolle gleichzeitig aktiv sind, was das Finden eines gemeinsamen Wertes erschwert.

Ein Vergleich zwischen dem FAR-Modell und dem partiell synchronen Modell ist insofern schwierig, als dass jeglicher Parameter eine Auswirkung auf das Resultat haben kann. Um einen deterministischen Vergleich zu ermöglichen, wurde das Consensus-Protokoll mit 100 Teilnehmern lokal simuliert. Durch die Ausgabe der Statusmeldungen auf die Textkonsole wurde eine künstliche Verzögerung erreicht. Da die Teilnehmer alle miteinander verbunden sein sollten, wurden automatisch durch die Simulationsumgebung 4950 Kanäle erschaffen.

Sowohl bei Einsatz des Fehlerdetektors EA-FD als auch bei der Annahme eines partiell synchronen Modells und dem Einsatz des Fehlerdetektors PS-FD wurden Stubborn Channels als Kanalimplementierung genutzt. Da sich der letzte Fall manchmal nur auf die Prozesse bezieht und die Kanäle als synchron übertragend angenommen werden, wurde zum Vergleich ein auf zuverlässigen Kanälen basierender Ablauf hinzugenommen.

Die Erfolgsrate eines jeden Prozesses, in der Koordinatorrolle hinreichend viele Nachrichten des gewünsch-

ten Typs zu empfangen, ist auf den nun folgenden drei Grafiken 4.2 bis 4.4 visualisiert worden. Dabei wird auf eine zeitliche Komponente in der Darstellung verzichtet - wesentlich ist nur, wie viele Nachrichten ein Koordinator erhalten konnte, und weniger, wie lange er dafür benötigt hat, um dann entweder einen Erfolg verkünden zu können, oder aber aufzugeben. Es sind pro Bild jeweils 100 Runden dargestellt, welches aufgrund der Wiederholung der Koordinatorrollen als Makrorunde bezeichnet werden soll. Links sind die von den Koordinatoren empfangenen *estimate*-Nachrichten eingezeichnet, rechts davon die *ack*-Nachrichten, die dann von den Teilnehmern versendet werden, wenn ein Koordinator genügend *estimate*s sammeln konnte und anschließend ein *newestimate* gesendet hat. Die Nummerierung der Koordinatoren läuft entgegen dem Uhrzeigersinn, und beginnt an der Kante zwischen dem dritten und dem vierten Quadranten. Die Länge der Linien entspricht der Zahl an empfangenen Nachrichten durch den jeweiligen Koordinator.

Die lokale Simulation mit dem Fehlerdetektor PS-FD konnte nach 77 Runden einen Konsens erzielen (Abbildung 4.2). Interessant ist hierbei, dass die Erfolgsrate sich nicht kontinuierlich verbesserte, sondern starken Schwankungen unterlag. Als festes Intervall zum Senden der *ALIVE*-Benachrichtigungen wurden 20 Sekunden gewählt, und die Erhöhung des Timeouts im Falle der Ankunft einer solchen Nachricht von einem bereits verdächtigten Prozess auf 5 Sekunden festgesetzt, da ein vorheriger Versuch mit einem geringerem Anstieg in der doppelten Zeit maximal 20 *ack*-Nachrichten pro Koordinator ermöglichte.

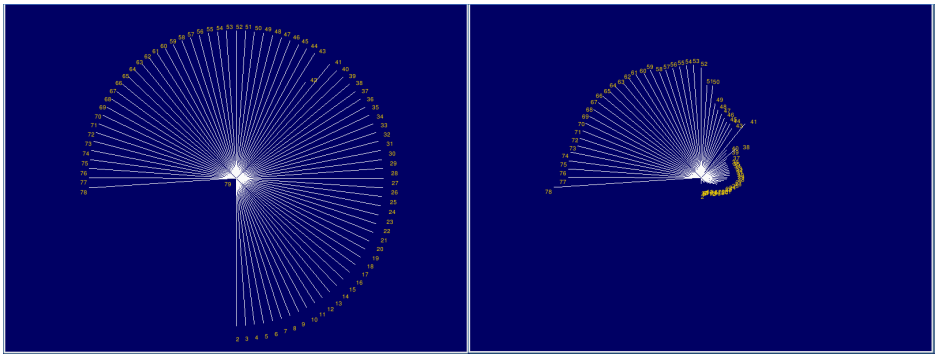


Abbildung 4.2: Erste Makrorunde mit PS-FD

Bei Einsatz des Fehlerdetektors EA-FD wurden in der ersten Makrorunde nahezu alle *estimate*-Nachrichten, jedoch keine *ack*-Nachrichten empfangen. Auf die Darstellung wird hier aus Platzgründen verzichtet, hingegen sind die Makrorunden 2 und 3 von Interesse für das Finden des Konsens, der nach insgesamt 293 Runden gefunden werden konnte (Abbildungen 4.3, 4.4).

Resultat: Obwohl das Protokoll im partiell synchronen Modell weniger Runden benötigte, so lief es doch im FAR-Modell schneller ab. Diese Diskrepanz ist eindeutig auf das mehrmalige Senden von 9900

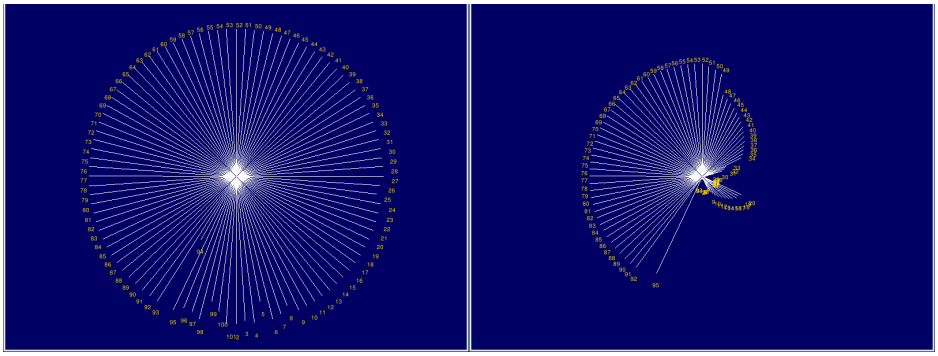


Abbildung 4.3: Zweite Makrorunde mit EA-FD

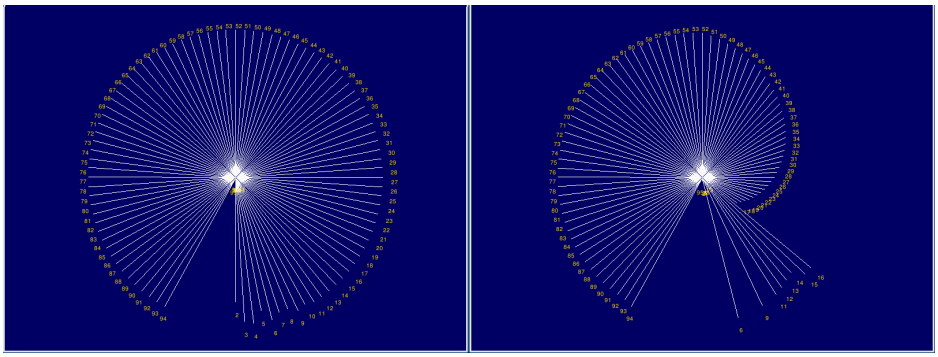


Abbildung 4.4: Dritte Makrorunde mit EA-FD

ALIVE-Benachrichtigungen durch den PS-FD zurückzuführen. Die Laufzeit betrug 1927 Sekunden bei Einsatz des EA-FD und 3129 für den PS-FD. Es wurden etwa 119000 Nachrichten mit EA-FD und 389000 mit PS-FD versendet.

Zu Beginn der Simulation wurden jeweils etwa 20300 Zeilen Informationen durch die Simulationsumgebung ausgegeben, welches allein schon zu einer Verzögerung von etwa 640 Sekunden vor dem eigentlichen Start führte. Diese Werte sollten mit beachtet werden.

Gerundete Werte:

	PS-FD/reliable	PS-FD/stubborn	EA-FD/stubborn
Gesamtlaufzeit	6202 Sekunden	3129 Sekunden	1927 Sekunden
Nachrichten	509000	389000	118000
Laufzeit bis Consensus	6091 Sekunden	3003 Sekunden	1807 Sekunden
Nachrichten bis Consensus	490000	375000	108000
Runden bis Consensus	85	77	293
Maximaler Timeout	121,0 Sekunden	76,0 Sekunden	12,79 Sekunden
Logzeilen	2,51 Mio	3,17 Mio	1,18 Mio
Loggröße	178 MB	242 MB	84 MB

Eine halbautomatisierte Auswertung der Verhaltensweise des Fehlerdetektors EA-FD zeigte, dass der überwiegende Teil der Nachrichten als langsam eingestuft wurde. Die Grafik 4.5 zeigt die kumulativen Werte für alle 100 Detektoren, und zwar dargestellt entsprechend der Funktionsweise von EA-FD. Auf der horizontalen Achse befindet sich die Abfolge der langsamen Nachrichten, auf der vertikalen Achse die jeweils im Zustand einer bestimmten Zahl von diesen die Abfolge der schnellen Nachrichten. Die Mengenangaben an den Übergängen geben an, wieviele Nachrichten jeweils den Übergang von einer langsamen Nachricht zu einer schnellen oder einer anderen langsamen Nachricht verursacht haben. Übergänge von schnellen zu langsamen Nachrichten sind implizit durch Rückfall auf den nächsten Wert der horizontalen Achse gegeben, wie ja auch in der Fehlerdetektor-Formel zur Bestimmung des Timeout die Anzahl der schnellen Nachrichten beim Eintreffen einer langsamen Nachricht zurückgesetzt wird. Zu beachten ist dabei noch, dass nicht zu allen gesendeten Nachrichten überhaupt eine Bestätigung eingetroffen ist: die etwa 118000 Nachrichten haben zu nur etwa 95000 Bestätigungen geführt.

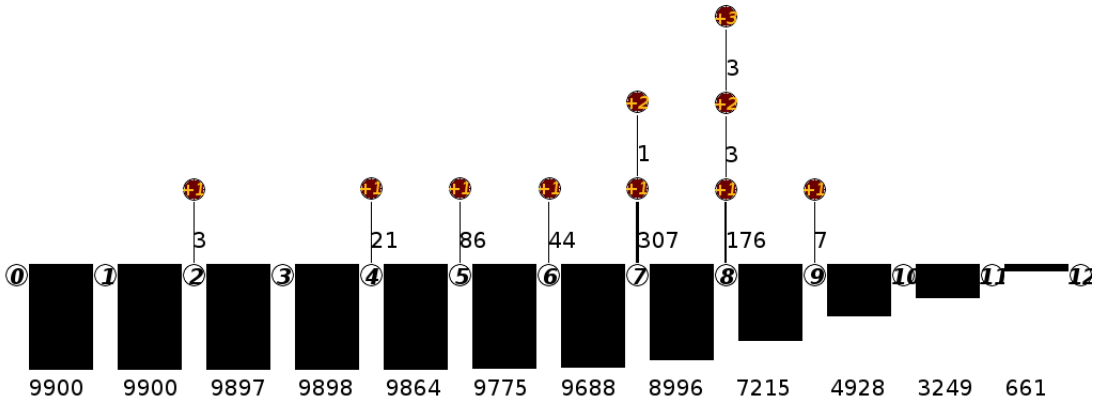


Abbildung 4.5: Schnelle und langsame Nachrichten, eingestuft durch EA-FD

4.2 Verteilte Ausführung

Neben der lokalen Durchführung sind Algorithmen unter FARnodes auch im realen Einsatz getestet worden. Dabei wurde insbesondere auf die Bestätigung der Annahmen im FAR-Modell geachtet. Als Einstieg soll jedoch ein Experiment dienen, bei denen zwar eben jene Annahmen gemacht werden können, allerdings die Funktionsweise von Stubborn Channels zunächst ohne Fehlerdetektor untersucht wurde, da Verdächtigungen in dem Experiment ohnehin nicht eingeplant sind.

4.2.1 Zwei-Phasen-Commit-Protokoll

Die erste vor- und nachbereitete Ausführung eines verteilt arbeitenden Algorithmus betraf das 2PC-Protokoll. Drei durch Netzwerkverbindungen physikalisch voneinander getrennte Systeme waren dafür im Einsatz. Als Koordinator diente ein Laptop vom Typ Apple iBook G3, welches über 100-MBit-Ethernet mit einem Desktop-PC mit AMD Athlon-Prozessor der Klasse 3400+ verbunden war. Beide Rechner befinden sich in Dresden und sind mit dem Betriebssystem Linux ausgestattet. Der dritte Teilnehmer ist über eine über 18 Router laufende Internet-Verbindung erreichbar gewesen, und lief auf einer User-Mode-Linux-Instanz im US-Bundesstaat New Jersey.

Auf allen drei Systemen wurde eine Datenbank vom Typ PostgreSQL in verschiedenen Versionen (7.4 bis 8.1) eingerichtet. Ziel war es, durch den Koordinator 1000 INSERT-Befehle absetzen zu lassen, und danach die Datenbanken auf Vollständigkeit und Konsistenz zu überprüfen. Besonders schwierige Bedingungen wurden dadurch geschaffen, dass ab etwa der Mitte des Ablaufes künstliche Datenverluste durch Portsperrung hervorgerufen werden sollten.

Für die Protokollimplementierung wurden die Skripte `commit.scsim` (Protokoll), `commit-pg.scsim` (Datenbankanbindung), `commit-bizarre.scsim` (Koordinator) und `commit-se01-vs011.scsim` (übrige Teilnehmer) entwickelt. Als Basis diente dabei das bereits implementierte Consensus-Protokoll, welches über weite Abschnitte eine hohe Ähnlichkeit damit aufweist, sowie die Protokollbeschreibung von Singh [Sin94].

Ergebnisse: Die Gesamtlaufzeit, ausgehend von einer Maximalfrequenz von zwei Nachrichten selben Inhalts pro Sekunde durch den bereits in der Beschreibung der Kanalimplementierung erwähnten „Gentlemen-Timeout“, lag bei exakt 4000 Sekunden. Die Hälfte der Nachrichten wurde bereits nach etwa 28,4% dieser Zeit ausgetauscht.

Die Datenbankinhalte waren vollständig und bis auf Duplikate in einer Datenbank konsistent. Die Duplikate entstanden durch Annahme und Bearbeitung von bei Verwendung der Stubborn Channels oft mehrfach hintereinander eintreffender Nachrichten, während gleichzeitig immer noch auf die Bestätigung auf eine selbst versandte Nachricht gewartet wurde. Die Implementierung wurde daraufhin derart abgeändert, dass in einem solchen Szenario keine Nachrichten mehr außer der letztlich eintreffenden Bestätigung angenommen werden. Dies entspricht mehr dem Idealfall einer nicht-blockierenden Umsetzung der *Stubborn*-Eigenschaft der Stubborn Channels [GOS97]. Ein echtes Blockieren wäre im Falle einer Ausführung mit mehr als einem Knoten pro Simulationsinstanz fatal, dennoch sollte Autoren von Simulationsskripten die Möglichkeit gegeben werden, eine entsprechende Primitive zu verwenden.

Es zeigt sich, dass die Annahme des FAR-Modells, dass die Nachrichtenlaufzeiten im Durchschnitt endlich sind, bestätigt wird. Darüber hinaus ist die dadurch implizierte Annahme, dass nur wenige Nachrichten für den überaus größten Teil der Verzögerungen verantwortlich sind, deutlich geworden. In der folgenden Tabelle sind die Durchschnitte der Roundtrip-Zeiten von Nachrichten des Koordinators an beide Teilnehmer aufgeführt, und zwar berechnet über jeweils ein Zehntel der wertmäßig geordneten Messergebnisse zur Reduktion der Daten (Cluster-Prinzip).

Bereich der Messwerte	Durchschnittliche Antwortzeit in Sekunden
1/10	0.002
2/10	0.003
3/10	0.012
4/10	0.026
5/10	0.085
6/10	0.117
7/10	0.129
8/10	0.145
9/10	0.197
10/10	5.006

Eine derartige Einteilung der Zeitachse bringt hingegen ein sehr inhomogenes Bild zum Vorschein. Diese Werte sind aber ebenfalls durch die (zeitlich verstreut) auftretenden „Ausreißer“ bedingt, sowohl vor als auch nach der künstlichen Störung durch Sperrung des entsprechenden Ports auf einem der Teilnehmersysteme, aber natürlich dadurch besonders verstärkt. Die zweite Tabelle verdeutlicht den Sachverhalt.

Zeitabschnitt	Durchschnittliche Antwortzeit in Sekunden
1/10	0.477
2/10	0.247
3/10	0.084
4/10	0.903
5/10	0.615
6/10	0.814
7/10	0.494
8/10	2.76
9/10	0.916
10/10	1.476

4.2.2 Consensus-Protokoll

Eine größer angelegte verteilte Ausführung stellte das Consensus-Experiment unter den Annahmen des FAR-Modells dar. Fünf Prozesse haben dabei gleichzeitig versucht, sich auf einen gemeinsamen Wert zu einigen. Neben den im voherigen Abschnitt bereits vorgestellten Trägersystemen kamen hierfür noch die Hosts `selab4` und `proxad` hinzu. Das letztgenannte System ist ein in Lorraine, Frankreich stehender Server.

Ein kommentierter Ausschnitt aus dem Algorithmus soll das grundlegende Verständnis für den Ablauf erhöhen:

```
1  # Erklaerungen zum Ablauf in der ersten Phase des Consensus-Protokolls
2
3  # Sowohl der Koordinator als auch alle anderen Teilnehmer senden die
4  # Nachricht 'estimate' an den Koordinator (dieser demnach an sich selbst)
5  msg = ("estimate", self.steptuple(), self.cp_adopted, self.cp_estimate)
6  send(self.parentnode, self.cp_coord, msg)
7
8  # Je nach Rolle wird dann unterschiedlich weiter verfahren
9  if not self.iscoordinator():
10         # Die regulaeren Teilnehmer ueberspringen die zweite Phase...
11         self.cp_phase = 3
12         # ...und gehen in die dritte ueber, in der sie so lange warten,
13         # bis der Koordinator ihnen eine Nachricht gesendet hat, oder
14         # bis der Fehlerdetektor den Koordinator verdaechtigen wuerde
15         coordtimeout = self.parentnode.timeout(self.cp_coord)
16         self.parentnode.callback(coordtimeout, self.callback_handler)
17 else:
18         # Der Koordinator hingegen wartet in der zweiten Phase auf alle
19         # einkommenden 'estimate'-Nachrichten...
20         self.cp_messages = {}
21         self.cp_phase = 2
22         # ... und zwar ohne jegliche zeitliche Beschraenkung
```

Ergebnisse: Der Prozess auf `cons-selab4` hat nach 24 Runden mit der Zustimmung der beiden im selben LAN befindlichen Prozesse auf `cons-bizarre` und `cons-se04` das Ergebnis mit dem Wert 43 bestimmt und verbreitet.

Während der Ausführungszeit von etwa siebeneinhalb Minuten wurden von jedem Prozess aus über 800 Nachrichten versandt. Dabei zeigte sich, dass der Großteil der UDP-Pakete, welche über das Internet übertragen wurden, verloren ging, während die LAN-Verbindung prinzipiell die Zustellung sämtlicher Pakete übernommen hat.

Dies konnte jedoch nicht erklären, warum nicht eine einzige Nachricht der anderen beiden Prozesse eingetroffen war, obwohl anderweitig problemlos von den drei am Ergebnis beteiligten Prozessen eine Kommunikation mit ihnen möglich war. Von den eingetroffenen Nachrichten wurden etwa 4/5 verworfen, da sie aus einer vorherigen Runde stammten oder aber falsche Phaseninformationen angenommen hat-

ten. Die Abbildung 4.6 verdeutlicht das Ergebnis aus der Sicht des Prozesses „spuk“. Jedes Kästchen repräsentiert eine von dem der Spalte zugeordneten Prozess empfangene Nachrichten, wobei natürlich auch an sich selbst gesendete Nachrichten (linke Spalte) empfangen wurden. Dabei stehen blaue Kästchen für in dieser Runde und Phase akzeptierte Nachrichten, weiße Kästchen hingegen für verworfene Nachrichten. Die grau hinterlegten Zeilen stehen für Runden, in denen der Prozess „spuk“ die Rolle des Koordinators inne hatte.

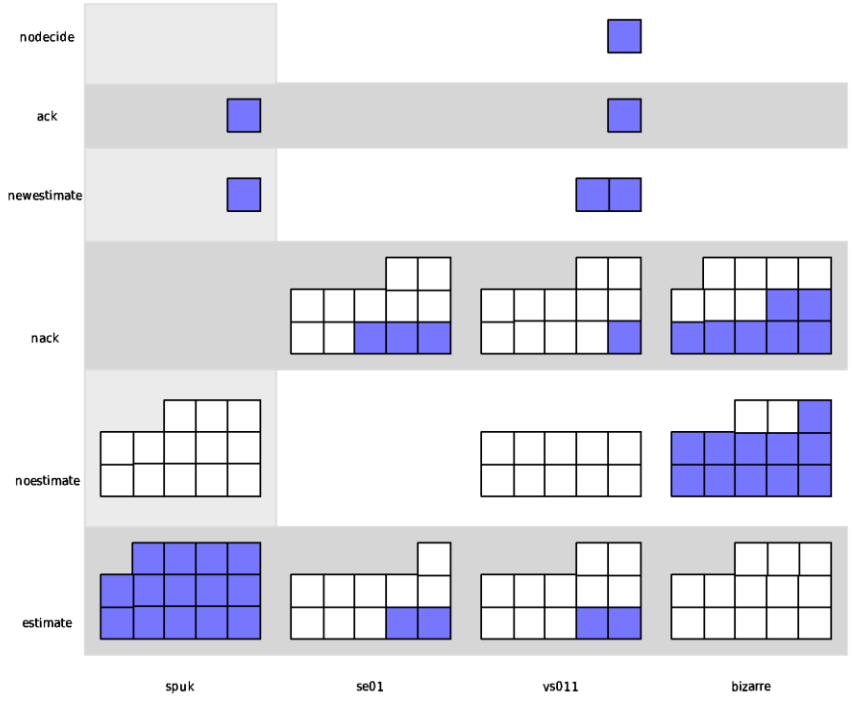


Abbildung 4.6: Empfangene Nachrichten im Prozess „spuk“

Als Fehlergrund stellte sich heraus, dass die Implementierung des Algorithmus auch für den Koordinator einen Timeout vorsah, also die Möglichkeit, bei all zu langer Wartezeit eine neue Runde zu beginnen. Dies kann im Extremfall zu einer ständigen diametralen Phasenposition jeweils zweier Koordinatoren führen. Gemäß der Spezifikation wurde dieser Timeout dann entfernt und ein neuer Durchlauf gestartet. Das Resultat in Abbildung 4.7 wies eine wesentlich geringere Zahl an benötigten Nachrichten auf, um zum Consensus zu kommen.

4.2.3 Consensus-Protokoll in Vergleichsszenarien

Das selbe Consensus-Simulationsskript wurde für die Gewinnung von Informationen über die Ablaufzeit sowohl mit EA-FD als auch mit PS-FD im LAN- sowie im Internet-Einsatz mit jeweils drei beteiligten Systemen getestet. Jedes System verwendete die Partitionierung von FARnodes, um zwei Prozesse zu

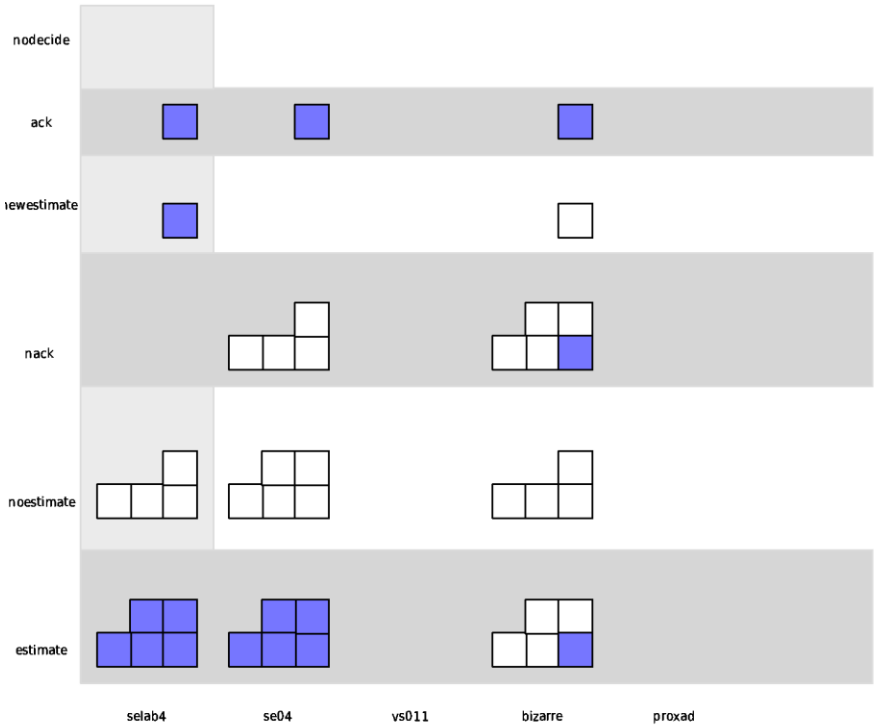


Abbildung 4.7: Empfangene Nachrichten im Prozess „selab4“

erzeugen. Es wurden wiederum wie schon bei der lokalen Simulation mit 100 Teilnehmern Stubborn Channels für beide Detektoren sowie zusätzlich unzuverlässige UDP-Verbindungen für den PS-FD eingesetzt.

Ergebnisse: Da die unzuverlässigen Verbindungen keine Zustellung der Nachrichten garantieren, sind sie eigentlich nur der Vollständigkeit halber aufgenommen worden. Sie zeigen jedoch ein interessantes Phänomen auf: Aufgrund der verschiedenen Startzeiten sind etliche *nack*-Nachrichten nicht bei den (noch nicht gestarteten) Koordinatoren angekommen. Dies führte zu einer Deadlock-Situation, da die maximale Zahl von *nack*s selbst im besten Fall um eins verfehlt wurde. Die nachstehende Tabelle rekonstruiert den Verlauf der Abarbeitung und macht deutlich, dass ein derartiger Nachrichtenverlust nicht mit dem partiell synchronen Modell vereinbar ist.

Prozess	(lokale) Runde	(lokaler) Koordinator	Anzahl an *nack*s
selab4/a	0	bizarre/a	0
selab4/b	0	bizarre/a	0
selab4/a	1	bizarre/b	0
selab4/b	1	bizarre/b	0
selab4/a	2	se04/a	0
se04/b	0	bizarre/a	0
se04/a	0	bizarre/a	0
se04/a	1	bizarre/b	0
selab4/b	2	se04/a	0
selab4/a	3	se04/b	0
se04/a	2	se04/a (selbst)	1
se04/b	1	bizarre/b	1
se04/b	2	se04/a	1
selab4/b	3	se04/b	0
selab4/a	4	selab4/a (selbst)	0
se04/b	3	se04/b (selbst)	1
selab4/b	4	selab4/a	0
selab4/b	5	selab4/b (selbst)	0
bizarre/b	0	bizarre/a	0
bizarre/a	0	bizarre/a (selbst)	0
bizarre/b	1	bizarre/b (selbst)	0
bizarre/a	0	(selbst)	1
bizarre/b	1	(selbst)	0
se04/a	2	(selbst)	3
se04/b	3	(selbst)	2
selab4/a	4	(selbst)	1
selab4/b	5	(selbst)	0

In der auf Stubborn Channels basierenden Ausführung hingegen konnte sowohl im LAN als auch im WAN stets ein gemeinsamer Wert gefunden werden. Die Zeiten dieses Ereignisses liegen dabei im LAN bei 5,35 Sekunden (in der 6. Runde des erfolgreichen Prozesses) für EA-FD und bei 7,6 Sekunden (in der 5. Runde) für PS-FD. Die durchschnittliche Antwortzeit von etwa 0,74 Sekunden bei PS-FD im Gegensatz zu nur 0,465 Sekunden bei Einsatz des EA-FD lassen sich wiederum auf die erhöhte Zahl an ALIVE-Nachrichten zurückführen. So wurden im ersten Fall 112, im zweiten Fall nur 91 Nachrichten insgesamt versandt.

Im WAN sind längere Laufzeiten, aber auch eine höhere Distanz der Rundenzahl zwischen den Prozessen unabdingbar gewesen. So wurde der gemeinsame Wert erst nach etwa 10,3 Sekunden Laufzeit eines

Prozesses in der 2. Runde gefunden, während die Gesamtlaufzeit für einen anderen Prozess nur 3,57 Sekunden betrug, im Falle des EA-FD. Der Wert wurde bei Einsatz des PS-FD hingegen erst in der 5. Runde eines Prozesses nach 4,54 Sekunden lokaler Zeit gefunden.

4.2.4 Einschränkungen

Die in den Experimenten gemessenen Zeitwerte unterliegen nicht nur geringen Schwankungen, sondern oft erheblichen temporären Ausfällen der Datenübertragung, vor allem bei der Einbindung von Systemen über das Internet, die ihren Zugang über Leitungen mit niedriger Übertragungsqualität realisieren. Falls erforderlich, ließe sich eine weniger gestörte und dennoch reale Bedingungen anbietende Konstellation von Rechnern und Verbindungen über Forschungsnetzwerke wie PlanetLab nutzen.

Eine inhärente Eigenschaft verteilter Ausführungen ist es, dass die Startzeitpunkte für die Ausführung des Skripts teilweise recht große relative Differenzen aufweisen. Die Rückverfolgung von gesendeten Nachrichten ist demnach sehr schwierig von Hand durchzuführen. Abhilfe könnte die Garantie einer zeitsynchronisierten Ausführungsumgebung schaffen, die in der Praxis aber nur mit Aufwand erreichbar wird.

4.3 Simulation mit experimentell gewonnenen Parametern

Wie bereits in der Beschreibung der Software geschildert, unterstützt FARnodes die wiederholte Durchführung einer verteilten Ausführung im Rahmen einer Simulation. Dieser hybride Modus konnte mit den Werten des 2PC-Algorithmus erfolgreich durchgeführt werden; eine Wiedergabe der Daten ist deshalb an dieser Stelle nicht notwendig. Es sei allerdings darauf hingewiesen, dass in Grenzfällen eine mangelnde Genauigkeit der Zahlenwerte zu einem anderen Kommunikationsverhalten eines Prozesses und damit auch im Gesamtsystem führen kann, woraufhin sämtliche folgenden Traces ihre Grundlage verlieren und das Ergebnis extrem verfälschen können.

5 Diskussion

5.1 Systemmodelle

Die im FAR-Modell erzielten Ergebnisse sollen nun mit den zu erwartenden Resultaten in anderen Systemmodellen verglichen werden. Dabei werden vollständig synchrone und asynchrone Systeme nur der Vollständigkeit halber erwähnt, der Schwerpunkt soll auf den vergleichbaren Modellen der partiellen Synchronität und des Einsatzes von Fehlerdetektoren liegen.

5.1.1 Synchrone Systeme

Sämtliche Agreement-Protokolle lassen sich in synchronen Systemen lösen. Es gibt in diesen definierte obere Grenzen für die Übertragungsdauer einer Nachricht und deren Verarbeitungsdauer in einem Prozess. Dennoch muss, wie [GOS97] erwähnten, auf gewisse Konsequenzen geachtet werden, so beispielsweise eventuelle Verstöße gegen die Eigenschaft *uniform agreement* im Consensus-Problem.

5.1.2 Asynchrone Systeme

Im Gegensatz zu synchronen Systemen existieren hier keine derartigen Grenzen. Es ist erwiesen, dass das Consensus-Problem, und damit auch andere, in diesem Modell nicht lösbar ist, sofern auch nur ein einziger Prozess fehlerhaft ist. (In den Betrachtungen dieser Belegarbeit trifft dies auch auf die Kanäle zu.) Aus diesem Grund kommt nur eine um weitere Annahmen erweiterte Variante dieses Modells in Frage, die in den nächsten drei Abschnitten auf jeweils unterschiedliche Art und Weise gebildet wird.

5.1.3 Partiell synchrone Systeme

Partiell synchrone Kommunikation ist über viele Jahre hinweg weiträumig untersucht und kategorisiert worden. Die Unterschiede zum FAR-Modell zeigen sich einerseits in der Stärke der Annahmen, andererseits auch in der Auswirkung der Implementierungen vor allem des Fehlerdetektors.

Die hier verwendete Methode zum Senden von *ALIVE*-Nachrichten von allen Prozessen an alle anderen

ist die ursprünglichste Form. Verbesserungen in der Effizienz, bedingt vor allem durch eine Reduktion der zu sendenden *ALIVE*-Nachrichten, sind in verschiedener Form untersucht worden, so beispielsweise durch die Einführung eines Ringmodells für die Kommunikation ([LFA04]). Auch gibt es spezielle Fehlerdetektor-Implementierungen der Klassen $\diamond W$ bis $\diamond P$, die genau die gewünschten und keine weiteren Annahmen erfüllen.

Weiterhin gibt es Fehlerdetektoren, die nur die Anzahl der empfangenen *ALIVE*-Nachrichten ausgeben, jedoch keine Verdächtigungen selbst vornehmen. Je nach Protokoll kann dies zu einer Verbesserung des Laufzeitverhaltens führen.

Natürlich setzen diese Erweiterungen meist zusätzliche Annahmen voraus, die ihre Verwendung einschränken, so dass der reine Geschwindigkeitsgewinn nicht das einzige Vergleichskriterium sein kann.

Eine Untersuchung sämtlicher dieser Erweiterungen im Vergleich mit dem FAR-Modell wäre sicherlich sehr interessant, ist aber nicht Teil dieser Belegarbeit geworden. Als Erkenntnis hat sich herausgestellt, dass ein direkter Vergleich von Protokollen beispielsweise anhand der Rundenzahl bereits sehr schwierig sein kann, durch die Hinzunahme von externen Einflüssen wie unterschiedlichen Startzeiten jedoch noch weitere zu beachtende Punkte hinzukommen.

5.1.4 Asynchrone Systeme mit zeitlichen Annahmen

Im so genannten *timed asynchrony model* wechseln stabile und weniger stabile Phasen einander ab. Die Motivation dahinter ist das Verhalten realer Systeme, in denen Störungen zwar häufig vorkommen mögen, jedoch nicht immer.

Die Protokolle sind nicht weiter in diesem Modell untersucht worden. Eine Simulation wäre aber durch entsprechende Ereignisskripte durchaus denkbar. Sowohl in der Spalte der Zeitpunkte für die Aktionen als auch der für deren Werte könnten durch Nutzung der Methode `math.rand()` entsprechende nichtdeterministische Grundlagen geschaffen werden.

5.1.5 Fehlerdetektoren

Fehlerdetektoren als Kapselung jeglicher zeitlicher Annahmen sind sowohl im partiell synchronen Modell als auch (als Realisierung) im FAR-Modell als vorteilhaft anzusehen, da sie eine Vergleichsebene schaffen.

Eine eventuelle Verbesserung der Stabilisierungszeit des EA-FD könnten entsprechende Testszenarien für FARnodes ermöglichen. Die Möglichkeit, den Fehlerdetektor mit Hilfe eines anderen zu kalibrieren,

wurde zwar nicht direkt umgesetzt, erscheint aber ob der doch sehr langsamen Einstellung auf das reale Zeitverhalten von Nachrichten in den Kanälen als ein Weg zur Verbesserung. Ein weiterer Weg wäre, protokollspezifisch die Werte eines EA-FD als Voreinstellung für den nächsten Durchlauf zu verwenden, um zumindest tendentiell eine schnellere Anpassung zu ermöglichen. Der Ereignis-Mechanismus von FARnodes bietet dafür eine existierende Grundlage an.

5.1.6 FAR und Timed FAR

Der Unmöglichkeitsbeweis zur Lösung des Strong Leader Election-Problems im FAR-Modell und die daraus resultierende Gewinnung des Timed FAR-Modells zeigen sehr deutlich die Annährung an das vollständig asynchrone Modell. Im FAR-Modell werden eben keine expliziten zeitlichen Annahmen mehr über einzelne Nachrichten getroffen, sondern nur noch über deren Durchschnitt.

Dennoch kommt das Timed FAR-Modell ohne Annahmen zur partiellen Synchronität aus. Schon aus diesem Grund wäre es sinnvoll, eine Implementierung dieses Protokolls für FARnodes zu erstellen, und auch für andere Algorithmen die Überprüfung auf hinreichend starke Annahmen durchzuführen. Die Methode `timeout()` eines jeden `node`-Objekts sollte die erforderliche Schnittstelle für die im abstrakten Algorithmus verwendeten *lease times* bereitstellen, und die Methode `timer()` des Schedulers kann bereits lokale Hardware-Uhren verwenden.

5.2 Simulationsumgebungen

An dieser Stelle soll ein kurzer Vergleich von FARnodes mit ähnlichen Simulations- und/oder Ausführungsumgebungen gezogen werden.

5.2.1 OMNeT++

OMNeT++ ist ein sowohl im Quelltext (`http://www.omnetpp.org/`) als auch kommerziell unter dem Namen OMNEST verfügbares Simulationssystem für Rechnernetze, Nachrichten und Protokolle. Es ist in C++ implementiert, und ermöglicht die Schaffung von Modellen auf Basis des Simulationskerns. Diese den erweiterten Prozessen und Kanälen in den Simulationsskripten von FARnodes entsprechenden Modelle werden ebenfalls in C++ geschrieben, während die Topologie in so genannten NED-Dateien (*Network DEscription*) festgelegt wird. Als grafische Oberfläche kommt Tcl/Tk zum Einsatz, und weitere im Unix-Umfeld gebräuchliche Bibliotheken werden für die Darstellung von Graphen, das Einlesen von XML-Dateien und weiteren Aufgaben verwendet.

Der Fokus der OMNeT++-Nutzer liegt dabei auf der Simulation von realen Netzwerken, insbesondere der Internet-Protokollfamilien. Das Angebot an Modellsammlungen wie *INET*, diversen IPv6-Modellen und Kommunikation in den unteren Netzwerkschichten wie *MAC* oder *Ethernet* ist in der Hinsicht eindeutig. Netzwerkrouten und Übergänge zwischen verschiedenen Kanaltypen werden ebenso unterstützt wie die grafische Editierung von Topologien. Aber auch Untersuchungen zur Consensus-Problematik und insbesondere zum Zwei-Phasen-Commit-Protokoll im OMNeT++-Simulator sind von [KHW04] durchgeführt worden. Dort lag der Fokus dann allerdings wiederum auf quantitativen Erkenntnissen einer speziellen Kommunikationsstruktur.

Im Vergleich zu FARnodes stellt OMNeT++ ein ausgereiftes Framework dar, welches aufgrund seiner Anwendungskreise vor allem auch die grafische Ausgabe betont, die von den einzelnen Simulationen aus individuell erweitert werden kann. Es unterstützt jedoch keine reale verteilte Ausführung eines Algorithmus, wie dies durch FARnodes geleistet wird. (Es gibt sehr wohl einen in Java geschriebenen *Remote OMNeT++-Manager*, der aber eher die Aufgabe übernimmt, lokale Simulationsskripte auf anderen Hosts auszuführen und die Ergebnisse zentral zu speichern.) Auch gibt es bis jetzt keine unterstützenden Elemente zur Modellierung bestimmter Synchronitäten, so dass ein nicht unbedeutender Teil der in FARnodes investierten Arbeit nicht als duplizierend, sondern eher als vorbereitend anzusehen ist, wenn denn diese Simulationen auch in OMNeT++ ausgeführt werden sollten.

5.2.2 ADAM/EDEN

Im Rahmen des INRIA-Forschungsprojektes ADEPT ist eine Sammlung von „Generischen Agreement-Komponenten“ entstanden. Mit Hilfe dieser Komponenten lassen sich Algorithmen implementieren, die sich auf Consensus-Protokolle zurückführen lassen, wie beispielsweise das theoretische Problem der Gruppenmitgliedschaft. Diese Komponentensammlung heißt ADAM und ist in Java implementiert. Als Basis dient EVA, eine ereignisgesteuerte Softwarearchitektur. ADAM wird einer fehlertoleranten CORBA-Implementierung namens EDEN verwendet.

Das Design der Komponenten ist so gestaltet worden, dass Limitierungen vorheriger Projekte wie HORUS überkommen werden konnten.

ADAM ist nicht direkt mit FARnodes vergleichbar, da es mehr dem Baukasten-Prinzip gleicht. Die Software, seit 2003 in Entwicklung, ist leider nicht öffentlich verfügbar, so dass ein funktionaler Vergleich an dieser Stelle ausbleiben muss. Die Idee, Agreement-Komponenten einzusetzen, scheint aber sehr vernünftig zu sein - schon die Ableitung des 2PC-Algorithmus in FARnodes vom Consensus-Algorithmus zeigte ein hohes Maß an ähnlichen Strukturen.

5.2.3 NS2

Der Network Simulator 2 (`http://www.isi.edu/nsnam/ns/`) ist ähnlich OMNeT++ in C++ mit Unterstützung für Tcl/Tk-Ausgaben geschrieben worden. Dabei wird Tcl auch als API-Sprache für Simulationsskripte verwendet, worin wiederum C++-Module referenziert werden können.

Die Simulationsumgebung zielt primär auf TCP/IP-Kommunikation in allen Facetten ab. Es werden aber auch UDP und darauf basierende Protokolle sowie Routing-Simulation unterstützt. Auf höherer Ebene steht ähnlich wie in FARnodes auch Unterstützung für diskrete Ereignisse sowie Trace-Dateien zur Verfügung.

Eine Visualisierung erfolgt über das Programm Network Animator (`nam`). Zusätzliche Änderungen zur Laufzeit sind durch Randomisierungs-Variablen möglich. Es steht außerdem eine Sammlung an Hilfsskripten zur visuellen und textuellen Auswertung von Trace-Dateien in Verbindung mit externen Statistikprogrammen bereit.

Eine Entsprechung zur verteilten Ausführung von FARnodes ist die Emulation in NS2. Das modifizierte Programm `nse` kann Daten aus der Simulation in das Netzwerk einspeisen und ebenso Daten von dort einlesen. Die Emulation ist der Dokumentation nach als experimentelles Merkmal des Simulators einzustufen.

6 Ausblick

Durch die erstellte Software und die durchgeführten Experimente konnten die Eigenschaften des FAR-Modells und anderer Systemmodelle geprüft und analysiert werden. Die Idee zur Implementierung des Consensus-Protokolls hat sich als durchführbar erwiesen. Weitere verteilte Algorithmen können nun auf dieser Basis getestet werden. In den folgenden beiden Abschnitten sollen abschließend noch Vorschläge zu möglichen zukünftigen Arbeiten gegeben werden.

FARnodes: Mit der Fertigstellung dieser Belegarbeit wurde die Software FARnodes in der Version 1.0 freigegeben. Mögliche Verbesserungen sind unter anderem:

- Nutzung der Sandbox-Technik für jedes im Scheduler registrierte Objekt zur Abschottung der Objekte gegeneinander
- Schnittstelle zur Einbindung zusätzlicher Fehlerdetektoren
- Verbesserungen des Zeitverhaltens und der interaktiven Eingriffsmöglichkeiten
- Robusterer Einsatz von Trace-Dateien
- Reportgenerator auf Basis der vorhandenen Statistikskripte
- Sprachkonstrukte auf höherer Ebene und Einsatz von Komponenten zur Erleichterung der Skripterstellung

Sinnvoll erscheinen beispielsweise Konstrukte wie `wait_until(message, callback_success, callback_failure)` und `majority_wait(message, number, callback_success, callback_failure)`.

Auch dynamische Rollenzuweisungen mit automatischer Generierung von Annahmeregeln für Nachrichten könnten den Aufwand zur Erstellung eines Simulationsskripts senken.

Im Bereich der Implementierung wäre aus Gründen der Vollständigkeit eine Unterstützung für TCP-R als Umsetzung für *CR-reliable*-Kanäle in synchronen Systemen wünschenswert.

Im Bereich der Protokolle wären Algorithmen zur Leader Election und zu 3PC nützlich.

Physikalische Umsetzung: Es soll an dieser Stelle auch ein nicht rein informatik-bestimmter Ansatz für Kommunikation in verteilten Systemen einen Platz finden. Seit einiger Zeit wird an der Université de Genève an einer auf quantentheoretischen Prinzipien basierenden physikalischen Umsetzung des *Reliable Broadcast*-Protokolls geforscht [FGM01]. Dazu müssen Quanten in einen verschränkten Qutrit-Zustand eintreten, der garantiert, dass zwei Quanten den selben Wert aus der Menge $\{0, 1, 2\}$ annehmen, und bei Messung eine parallele Änderung auf einen gleichen anderen Wert durchführen. Aufgrund der inherenten *tamper proof*-Eigenschaft der Quanten soll es damit sogar möglich sein, das Problem der *Byzantinischen Generäle* zu lösen, also als Fehlermodell auch bösartige Prozesse einbeziehen zu können. Diese Arbeiten könnten in der Praxis einige der hier behandelten Schwierigkeiten weiter auflösen.

Literaturverzeichnis

[CHT92] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. In Maurice Herlihy, editor, *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing (PODC'92)*, pages 147–158, Vancouver, BC, Canada, 1992. ACM Press.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM, 43(2), pp:225–267*, 1996.

[DLS88] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[FGM01] Matthias Fitzi, Nicolas Gisin, and Ueli Maurer. Quantum solution to the byzantine agreement problem. *Phys. Rev. Lett.*, 87, 2001.

[FLP83] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty processor. In *Proceedings of the 2nd ACM Symposium on Principles of Database Systems (SIGACT-SIGMOD'83)*, pages 1–7, 1983.

[FSS05] Christof Fetzer, Ulrich Schmid, and Martin Suesskraut. On the possibility of consensus in asynchronous systems with finite average response times. In *Proceeding of the IEEE International Conference on Distributed Computing Systems (ICDCS 2005)*, 2005.

[GOS97] Rachid Guerraoui, Rui Oliveira, and André Schiper. Stubborn communication channels. Technical Report Research Report, Ecole Polytechnique Fédérale de Lausanne, 1997.

[KHW04] A. Köpke, H.Karl, and A. Wolisz. Consensus using aggregation - a wireless sensor network specific solution. Technical Report TKN-04-004, Telecommunication Networks Group, Technische Universität Berlin, April 2004.

[LFA04] Mikel Larrea, Antonio Fernández, and Sergio Arévalo. On the implementation of unreliable failure detectors in partially synchronous systems. *IEEE Transactions on Computers, Vol. 53, No. 7, July 2004*, 2004.

[Sin94] Harvinder Singh. Distributed database management. In Raman Khanna, editor, *Distributed Computing: Implementation and Management Strategies*. Prentice Hall, 1994.