Fakultät II – Informatik, Wirtschafts- und
Rechtswissenschaften
Department für Informatik
Abteilung Systemsoftware und verteilte Systeme

– Seminar –

Konzepte und Methoden in verteilten Systemen

19.07.2011

Inhaltsverzeichnis

1 Einleitung und Berechnungsmodelle (Hoffmann) 1

2 Logische Uhren (Bartsch) 5

3 Clocks of Different Dimensions (Strudthoff) 9

4 Distributed Mutual Exclusion (Hofstee) 14

5 Voting in Distributed Systems (Schadek) 17

6 Globaler Zustand (Heinermann) 22

7 Beobachtung globaler Prädikate (Arjangui) 28

8 Message Ordering in Distributed Systems (Dinh) 35

9 Berechnung einer globalen Funktion (Schmidt) 41

10 Synchronisierer (Wolff) 47

11 Distributed Shared Memory (Gollücke) 53

12 Selbststabilisierung (Ittershagen) 59

13 Failure Detectors (Gandor) 65

Einleitung und Berechnungsmodelle

André Hoffmann
Carl von Ossietzky Universität Oldenburg Email: andre.hoffmann@uni-oldenburg.de

I. Einleitung

In dieser Arbeit geht es um die Abgrenzung verteilter und paralleler Systeme und es soll aufgezeigt werden, welche Eigenschaften ein verteiltes System hat und welche Berechnungsmodelle verteilten Systemen zugrunde liegen.
Diese Berechnungsmodelle sollen primär dem Zweck dienen, die allgemeinen Eigenschaften und die Struktur eines verteilten Systems zu beschreiben.

II. Abgrenzung Verteilte Systeme / Parallele Systeme

Zunächst soll eine Abgrenzung zwischen verteilten und parallelen Systemen gegeben werden. Des Weiteren soll aufgezeigt werden, welche Vorteile sich im Vergleich zu parallelen Systemen durch die Verwendung von verteilten Systeme ergeben. Um diese Abgrenzung durchführen zu können wird als erstes eine kurze Definition von verteilten und parallelen Systemen gegeben.

A. Was ist ein verteiltes System

Ein verteiltes System ist ein System, in dem sich Hardware- und Softwarekomponenten auf vernetzten Systemen befinden und miteinander über den Austausch von Nachrichten kommunizieren. Diesem liegt ein Berechnungsmodell zurgrunde, dieses legt fest, wie das jeweilige System modelliert werden kann.

B. Was ist ein paralleles System

Ein paralleles System ist ein System, welches aus mehreren Prozessoren besteht und über gemeinsamen Speicher miteinander kommuniziert.

C. Unterschiede zwischen verteilten und parallelen Systemen

In einem parallelen System findet man im Vergleich zum verteilten System eine gemeinsame Uhr, gemeinsamen Speicher und es gibt die Möglichkeit bei Problemen (zum Beispiel Prozessausfällen oder lange Antwortzeiten) eine klare Fehlerdiagnose durchzuführen.

D. Vorteile von verteilten Systemen

Verteilte Systeme besitzen den Vorteil, dass sie eine gute **Skalierbarkeit** bieten, da bei einem parallelen System mit zunehmenden Prozessoren ein Problem mit dem Speicher auftreten kann. Bei diesem Problem handelt es sich darum, dass zu wenig Speicher pro Prozessor zur Verfügung steht. Dieses Problem tritt bei verteilten Systemen nicht auf, da kein gemeinsamer Speicher verwendet wird.
Zudem bietet ein veteiltes System den Vorteil, dass eine bessere **Modularisierbarkeit** gegeben ist, da ein einzelner Prozess einfach hinzugefügt oder entfernt werden kann.
Ein weiterer Vorteil ist die **Datenverteilung**. Da verteilte Systeme ihre Daten in Datenbanken sammeln ist es bei einem solchen System einfacher die Daten mit mehreren Organisationen und somit der IT-Systemen zu teilen, da diese zentral gespeichert werden können. Bei einem parallelen System hingegen ist das Teilen von Daten komplizierter, da es sich um mehrere Speicherorte handelt, welche verteilt und freigegeben werden müssen.
Die gemeinsame **Ressourcenverwaltung**, welche bei verteilten System stattfindet, ist ebenfalls ein Vorteil gegenüber einem parallelen System. So können die Bestandteile des Systems bestimmte Systemressourcen (z.B. ein Bandlaufwerk) gemeinsam nutzen.
Ein verteiltes System gewährleistet zudem eine höhere **Verlässlichkeit** als ein paralleles System, da durch den Ausfall eines einzelnen Prozesses die Gesamtfunktionalität nicht beeinflusst wird. Selbst im schlimmsten Fall wird höchstens ein Teil des Gesamtsystems ausfallen.
Durch die hohe Verfügbarkeit von Netzwerken mit großen Bandbreiten und den niedrigen Preisen für Workstations werden verteilte Systeme auch aufgrund der **Wirtschaftlichkeit** den parallelen Systemen vorgezogen.

III. Eigenschaften verteilter Systeme

Es gibt einige Eigenschaften, welche für ein verteiltes System typisch sind und dieses charakterisieren. Auf diese Charakteristiken soll hier kurz eingegangen werden.

A. Keine gemeinsame Uhr

Bei einem verteilten System ist es unmöglich die Uhren zu synchronisieren, da es bei einer Synchronisierung zu Verzögerungen aufgrund der Datenübertragung kommen würde. Aus diesem Grund müssen die Prozesse durch

andere Lösungen synchronisiert werden.

B. Kein gemeinsamer Speicher

Bei einem verteilten System gibt es keinen gemeinsamen Speicher, was es kompliziert macht globale Variablen für das System zu pflegen bzw. anzupassen.

C. Keine klare Fehlerdiagnose

In einem verteilten System werden Aufrufe asynchron durchgeführt. Hierbei entsteht das Problem, dass nicht festgestellt werden kann, ob ein Prozess sehr langsam arbeitet oder ob er fehlgeschlagen ist. Dies kann bei der Entwicklung eines verteilten Systems zu Problemen führen und muss bei der Implementierung berücksichtigt werden.

IV. Berechnungsmodelle

Es soll hier auf einige Berechnungsmodelle von verteilten Systemen eingegangen werden. Hierbei wird zunächst auf Modelle eingegangen, welche auf Ereignissen basieren und abschließend auf Modelle, welche auf Zuständen basieren. Beginnen werde ich dem Grundmodell eines verteilten Systems, welches anderen Modellen als Grundlage dient.

A. Grundmodell eines verteilten Systems

Ein verteiltes System soll hier so verstanden werden, dass es ein System ist, welches Nachrichten verschickt um Prozesse anzustoßen. Hierbei ist zu beachten, dass die Systembestandteile, zwischen denen die Nachrichten verschickt werden keinen gemeinsamen Speicher und keine gemeinsame Uhr haben. Wenn eine Nachricht verschickt wird kann sich der Zustand des Prozesses wechseln. Dies möchte ich anhand von einem Beispiel zeigen, bei welchem zwei Prozesse (siehe Systemaufbau Abbildung 1) jeweils eine Nachricht verschicken und ihren Zustand wechseln. Zu Beginn befinden sich beide Prozesse in ihrem Anfangszustand.

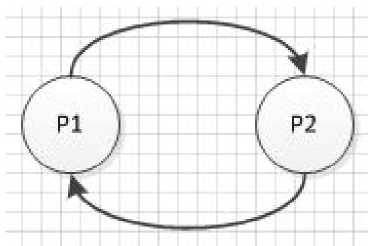


Figure 1. Systemaufbau

Hierbei ist zu beachten, dass *P1* zuerst eine Nachricht sendet und anschließend *P2* (siehe Abbildung 2). Beim Senden der jeweiligen Nachricht wechselt der Zustand des betroffenen Prozesses. Anhand der Abbildung, sieht man dass sich zunächst beide Prozesse *P1* und *P2* in ihrem Anfangszustand *S1* befinden. Hierbei wechselt der Prozess *P1* den Zustand nach dem Versenden der Nachricht in *S2* und wechselt beim Empfangen einer Nachricht wieder in den Zustand *S1*. Genau umgekehrt ist dieses beim Prozess *P2*. Dieser wartet zunächst auf eine Nachricht (Zustand *S1*) und verschickt anschließend eine Nachricht (Zustand *S2*).

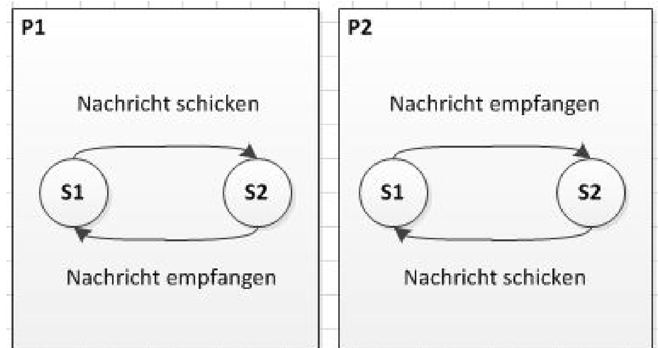


Figure 2. Zustandswechsel Prozess *P1* und *P2*

Auf diesem Berechnungsmodelle werden die folgenden Modelle (Interleaving, Happened Before, Potential Causality) basieren bzw. aufbauen.

B. Interleaving Model

In diesem Modell wird ein verteiltes System als eine globale Abfolge von Ereignissen gesehen, aus welchem sich ein globaler Zustand ableiten lässt. Der globale Zustand des gesamten Systemes kann als Kreuzprodukt der Zustände der beteiligten Prozesse abgeleitet werden. Dieser globale Zustand setzt sich demnetsprechend aus den Zuständen der unterschiedlichen Prozesse zusammen.
Wenn wir beispielsweise zwei Prozesse (in diesem Fall *S* und *T*) haben, welche beide noch kein Ereigniss angestoßen haben, das heißt beide Prozesse *S* und *T* befinden sich in ihrem Anfangszustand *1*, könnte der globale Zustand wie in Abbildung 3 gezeigt aussehen.

Sendet jetzt der Prozess *S* eine Nachricht, ändert sich sein aktueller Zustand in den Zustand *2* und somit ändert sich auch der globale Prozesszustand, da jetzt vom Prozess *S* zum Prozess *T* eine Nachricht geschickt wird. Der Zustand könnte jetzt wie in Abbildung 4 gezeigt aussehen.

Bei dem Interleaving Modell ändert sich dementsprechend für jede Zustandsänderung eines beteiligten Prozesses auch der globale Prozesszustand. Hierbei ist es jedoch aufwendig den globalen Prozesszustand zu ermitteln, da

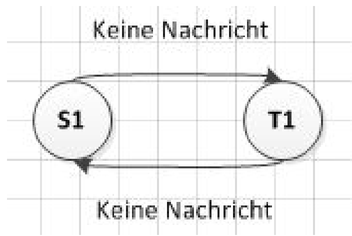


Figure 3. Globaler Zustand ohne Nachrichten

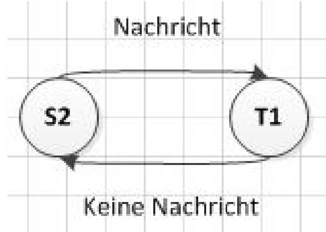


Figure 4. Globaler Zustand mit einer Nachricht

dieser nach jedem Ereignis welches in einem an dem Gesamtsystem beteiligten Prozesses ausgeführt wurde, neu ermittelt werden muss.

C. Happened Before Model

Bei dem Happened Before Modell geht es darum, dass jede Berechnung innerhalb dieses Modells als ein Tupel dargestellt wird. Dieses Tupel ist wie folgt aufgebaut *(E,→)*, wobei der Eintrag E für eine Menge von Ereignissen steht, welche aufgerufen werden können. Der Eintrag $\rightarrow$ hingegen legt fest, in welcher Reihenfolge die Ereignisse aufgerufen werden sollen.
Dieses Modell beschäftigt sich primär damit, in welcher Reihenfolge die Ereignisse aufgerufen werden, da es keine absolute Ordnung gibt. Das heißt nach dem ersten Ereignis kann ein weiteres Ereignis in diesem aber auch in einem anderen Prozess zur Durchführung einer Berechnung stattfinden.
Hierbei ist zudem wichtig, dass ein Ereignis nur der Nachfolger von einem anderen ist, wenn er wirklich direkt im Anschluss ausgeführt wird. Da die Prozesse nicht wirklich nacheinander laufen und es keine absolute Ordnung gibt werden sie auch als nebenläufig bezeichnet.
Um die Reihenfolge auf Grundlage des Happening Before Model zu bestimmen gibt es sogenannte Happening Before Diagramme(siehe Abbildung 5). Es kennzeichnet durch Verbindungen, welche Ereignisse andere Ereignisse aufrufen.

Ein Nachteil, welcher bei diesem Modell entsteht ist, dass es keinen Überblick über den globalen Prozesszustand gibt, da alle Prozesse parallel laufen können.

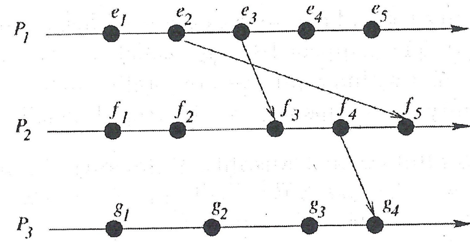


Figure 5. Beispiel für ein Happening Before Diagramm [1]

D. Potential Causality Model

Dieses Modell beschäftigt sich damit, welches Ereignis ein Ereignis verursacht hat und welche Auswirkungen das jeweilige Ereignis hat. Da die genaue Ermittlung bei einem komplexen System sehr aufwendig und somit auch kostenintensiv ist, wird bei diesem Model vor allem betrachtet, welche Ereignisse andere Ereignisse auslösen.
So muss ein Ereignis e welches ein Ereignis f verursacht, das verursachende Ereignis e auch potentiell die Möglichkeit haben das Ereignis f anzustoßen. Hieraus folgt jedoch nicht, dass das Ereignis f auch die Möglichkeit hat das Ereignis e anzustoßen. Aus diesen Erkenntnissen lässt sich erkennen, dass das Potential Causality Model ebenfalls ein valides Happened Before Model sein muss, denn wenn das Ereignis e das Ereignis f anstößt das Ereignis e der Vorgänger vom Ereignis f ist.
Dementsprechend ist ein Diagramm zu diesem Model ähnlich aufgebaut wie das Happening Before Diagramm (Abbildung 5). Ein Beispiel hierfür findet sich in Abbildung 6.

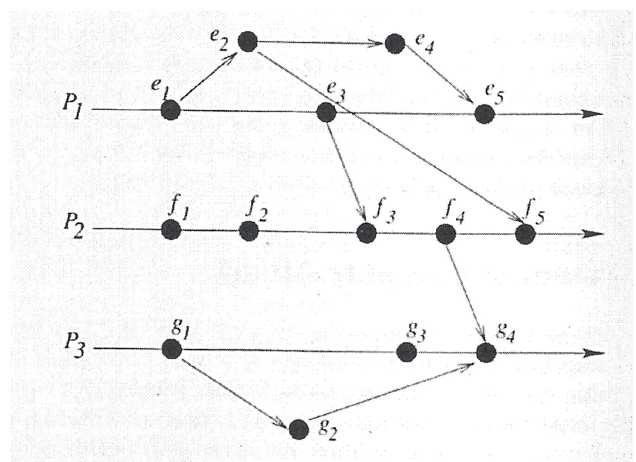


Figure 6. Beispiel für ein Potential Causality Diagramm [1]

Wie auch bei einem Happening Before Diagramm, kann

bei diesen Diagrammen eine Menge von Ereignissen E definiert sein, welche mit einer Reihenfolge, in diesem Fall die verursachten Aufrufe, $\rightarrow$ verbunden wird.

E. Modelle basierend auf Zuständen

Verteilte System können auch mit einem Modell aufbauend auf Zuständen, anstelle von Ereignissen arbeiten. Es lässt sich generell ein auf Ereignissen basierendes Modell in ein auf Zuständen basierendes Modell ableiten. Hierbei ist jedoch zu beachten, dass es keine Zyklen innerhalb des Systemes geben darf. Ansonsten dürfen in einem auf Zuständen basierenden Modell nur die Teile des Systemes dargestellt werden, welche keinen Zyklus bilden. Ein Zyklus in einem verteilten System wäre hierbei eine circulare Abhängigkeit unter den Prozessen des Systems.Ein Beispiel für die Ableitung eines Modells wäre eine simple Rechenoperation wie sie in der Abbildung 7 gezeigt wird.

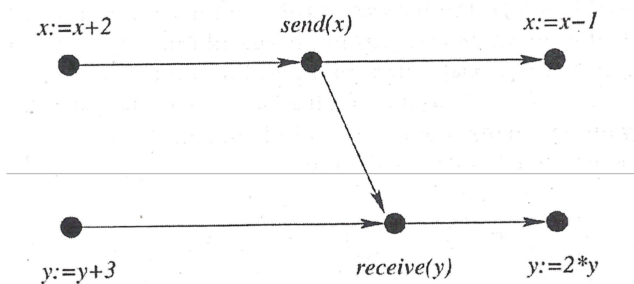


Figure 7. Beispiel für ein auf Ereignissen basierendes Modell [1]

Diese Abbildung kann auch in einem auf Zuständen basierenden Modell dargestellt werden, da es keine Zyklen beeinhaltet. Die abgeleitete Darstellung kann Abbildung 8 entnommen werden.

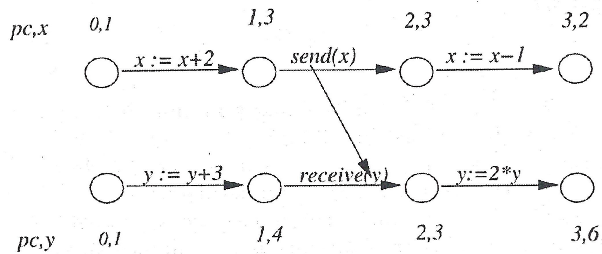


Figure 8. Beispiel für ein auf Zuständen basierendes Modell [1]

Um zu garantieren, dass sich die Ereignisse in einer gültigen Ordnung befinden, das heißt es gibt keine Zyklen, gibt es das so genannte **Deposet** Modell und das Modell der **Global Sequence of States**.

Ein **Deposet** ist ein Tupel, welches die Zustände beeinhaltet, welche durchlaufen werden. Hierbei wird sichergestellt, dass sich keine Zyklen bilden können, da innerhalb eines Schrittes (von Zustand zu Zustand) nicht zwei Operationen durchgeführt werden dürfen.

Beim **Global Sequence of States** geht es darum, dass nicht zwei Anweisungen gleichzeitig ausgeführt werden können. So kann in diesem Modell sichergestellt werden, dass keine Zyklen entstehen. Aus diesem Grund stellt es ein gültiges auf Zuständen basierendes Modell dar.

V. Zusammenfassung

Nachdem jetzt die speziellen Modelle betrachtet wurden sollte es die Möglichkeit geben festzulegen, welches Modell das Optimum für ein verteiltes System darstellt. Hierbei tritt jedoch das Problem auf, dass ein verteiltes System sein Modell immer auf Grundlage seines Anwendungsgebietes wählen sollte. Aus diesem Grund lässt sich keine klare Empfehlung geben. Hierzu soll auf einige Anwendungsfälle und das für diesen Fall sinnvoll anzuwendende Modell eingehen.

Als ersten könnte man sich ein System vorstellen, welches dazu dient ein verteiltes Programm zu überprüfen. Um dieses jedoch überprüfen zu können müssen wir den globalen Zustand des Programmes feststellen können. Dies ist jedoch nur mit dem **Interleaving Modell** möglich, aus welchem Grund die anderen Berechnungsmodelle für dieses Problemgebiet nicht optimal nutzbar sind.

Als zweitens könnte man sich ein System vorstellen, für welches wir feststellen wollen wann ein Zustand eintritt. Dies kann sehr gut mit dem **Happening Before Modell** geschehen, da nach der Erstellung eines solchen Diagramms das Programm sehr gut analysiert werden kann und festgelegt werden kann wann und in Folge von welchem Ereigniss ein bestimmter Zustand eintritt.

Einen ähnlichen Fall wie für das Happening Before Modell kann man sich auch für das **Potential Causality Modell** vorstellen. Der Unterschied hierbei liegt lediglich darin, dass wir nicht feststellen wollen ob ein Zustand eintritt sondern ob ein Zustand eintreten könnte und welche Voraussetzungen hierfür gegeben sein müssten.

Allein anhand dieser drei Anwendungsfälle sieht man, dass es abhängig vom Anwendungsfall ist, welches Modell für ein verteiltes System herangezogen werden sollte.

Quellen

[1] Vijay K. Garg, *Elements of Distributed Computing*, Wiley-IEEE Press, 2002

Ausarbeitung - Logische Uhren

Christian Bartsch
Universität Oldenburg
Seminar: Konzepte und Methoden in verteilten Systemen

***Zusammenfassung*—Diese Ausarbeitung beschäftigt sich mit logischen- und Vektoruhren in verteilten System. Dazu werden als Grundlage logische Uhren definiert und der Unterschied zu physischen Uhren hervorgehoben. Neben einem Algorithmus zur Implementierung einer logischen Uhr, wird auch deren Problem bei der Erkennung von nebenläufigen Ereignissen gezeigt. Zur Lösung dieses Problems, werden dann die Vektoruhren definiert und ein Algorithmus zur Implementierung dieser eingeführt.**

I. Einleitung

In einem verteilten System funktioniert die Kommunikation und Koordinierung von Aktionen, zwischen einzelnen Komponenten über den Austausch von Nachrichten. Um diese Nachrichten und die dazugehörenden Ereignisse in eine globale Ordnung bringen zu können, benötigt man ein gemeinsames Verständnis der „Zeit“ in allen beteiligten Knoten und Prozessen. Dazu gibt es zwei verschiedene Ansätze, die physikalische Uhren und die logische Uhren. In verteilten Systemen hat der Konzept der physikalischen Uhren den großen Nachteil, das es keine gemeinsame Uhr gibt, sondern jeder Prozess seine eigene physikalische Uhr und damit eine von anderen Prozessen abweichende Uhrzeit hat. Zur Lösung dieses Problems gibt es einige Algorithmen und Protokolle, welche von physikalischen Uhren in verteilten Systemen zur Synchronisation angewendet werden. Zu den bekannteren gehören der Algorithmus von Christian, der Berkeley-Algorithmus oder das Network Time Protocol. Ein dazu komplett anderer Ansatz, sind die logischen Uhren, die ohne Bezug zur Realzeit eine zeitliche Reihenfolge von Ereignissen feststellen können. Mit den Grundlagen von logischen Uhren, in Form der Definition, einen Algorithmus zur Implementierung und einem Beispiel zum Algorithmus beschäftigt sich das zweite Kapitel. In dritten Kapitel folgen Vektoruhren, eine Weiterentwicklung von logischen Uhren. Ergänzend zu der Definition von Vektoruhren wird es auch ein Algorithmus zur Implementierung von Vektoruhren und ein Beispiel zur Anwendung des Algorithmus vorgestellt.

II. Logische Uhren

Wie in der Einleitung schon angeführt, setzt das Konzept von logischen Uhren darauf, Ereignisse einen eindeutigen Zeitstempel zu geben, um damit die Beziehung zweier Ereignisse untereinander zu bestimmen. Dabei hat eine logische Uhr keinen proportionalen Bezug zur realen Zeit. Es ist einzig notwendig, dass Sie monoton steigende Werte abgibt, die als Zeitstempel dienen können. Um aus den Zeitstempeln einer logischen Uhr ein System von Abhängigkeiten ableiten zu können, muss eine logische Uhr die Happened-Before Beziehung erfüllen.

A. Die Happend-Before Beziehung

Die Happend-Before Beziehung „$\rightarrow$“ wurde 1978 von Leslie Lamport als Grundlage für logische Uhren formuliert. Seine Idee dabei war, eine logische Reihenfolge „$\rightarrow$“ (geschieht vor) von Ereignissen für verteilte Prozesse zu konstruieren.

Definition 1. *nach [Lamp].*
Für alle Ereignisse e, f und g gilt:

1) *Sind e und f Ereignisse im gleichen Prozess und e ereignet sich vor f, dann gilt:*
$$e \rightarrow f \text{ (e happened before f)}$$
2) *Für jede Nachricht gilt:*
$$e \rightarrow f$$
Wenn e das Sendeereignis im sendenden Prozess und f das Empfangsereignis im empfangenden Prozess darstellt
3) *Aus $e \rightarrow f$ und $f \rightarrow g$, folgt $e \rightarrow g$ (transitiv)*

Unabhängige Ereignisse, wenn weder $e \rightarrow f$ noch $f \rightarrow e$ gilt, heißen nebenläufig und werden als $e \parallel f$ geschrieben.

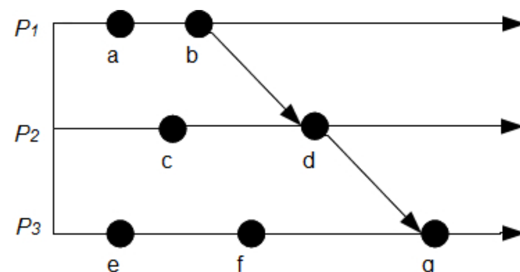


Abbildung 1.

In Abbildung 1. gelten $a \rightarrow b$, $c \rightarrow d$, $e \rightarrow f$, $f \rightarrow g$ und $e \rightarrow g$, weil diese sich Ereignisse jeweils im selben Prozess ereignen. Des Weiteren gilt $b \rightarrow d$, da b ein Sendeereignis und g das dazugehörende Empfangsereignis ist. Ebenfalls gilt $d \rightarrow g$, hierbei ist b das Sendeereignis und d das dazu passende Empfangsereignis. Die Beziehungen $b \rightarrow g$ und $c \rightarrow g$ gelten wegen der Transitivität. Ein Beispiel für die Nebenläufigkeit von zwei Ereignissen ist $a \parallel e$.

B. Definition Logische Uhr

Aus der Happend-Before Beziehung und der Vereinbarung das jeder Prozess einen Zähler C hat, der monoton steigende Werte abgibt, die als Zeitstempel dienen, lässt sich eine logische Uhr wie folgt definieren.

Definition 2. *Wenn Ereignis e Happened-before Ereignis f, dann ist der Zeitstempel C des Ereignis e kleiner als der Zeitstempel C des Ereignis f.*

$$e \rightarrow f \Rightarrow C(e) < C(f)$$

Die Umkehrung dieser Aussage gilt allerdings nicht, da sich aus den Zeitstempeln nicht eindeutig schließen lässt, ob zwei Ereignisse kausal voneinander abhängen oder nicht.

Anmerkung 1. *Aus $C(e) < C(f)$ folgt nicht, dass $e \rightarrow f$ gilt. Sondern aus $C(e) < C(f)$ folgt, dass $e \rightarrow f$ oder $e \parallel f$ gilt.*

C. Implementierung Logische Uhr

Dieser Algorithmus beschreibt die Implementierung einer logischen Uhr. Die Notation *(s, internal, t)* beschreibt einen internes Ereignis, welches den Prozess von Zustand s in den Zustand t überführt. Die Notation *(s, send, t)* beschreibt das Sendeereignis, welches den Prozess von Zustand s in den Zustand t überführt. Und die Funktion *(s, receive(u), t)* beschreibt das Empfangsereignis. Dabei wird eine Nachricht die in Zustand u von einen anderen Prozess gesendet wurde, in Zustand s empfangen und der daraus nachfolgende Zustand ist t.
Jeder Prozess hat seine eigene logische Uhr, welcher er seiner lokalen Variable c zuordnet. Die Notation s.c beschreibt den Wert des Zeitstempels der logischen Uhr im Zustand s. Beim Senden wird der aktuelle Zeitstempel um eins erhöht und dann mit der zusendenen Nachricht mitgesendet. Beim Empfangen einer Nachricht, wird aus dem eigenen und dem empfangenen Zeitstempel, der mit dem höheren Wert ausgewählt und dieser dann um eins erhöht. Bei einem internen Ereignis wird der Zeitstempel um eins erhöht.

Algorithmus 1. *aus [Elem]*

Process$_i$

var:
Integer c := 0;

send event (s, send, t):
t.c wird als Teil der Nachricht mitgesendet
t.c := s.c + 1;

receive event (s, receive(u), t):
t.c := maximum(s.c, u.c) + 1;

internal event (s, internal, t):
t.c := s.c + 1;

D. Beispiel zum Algorithmus

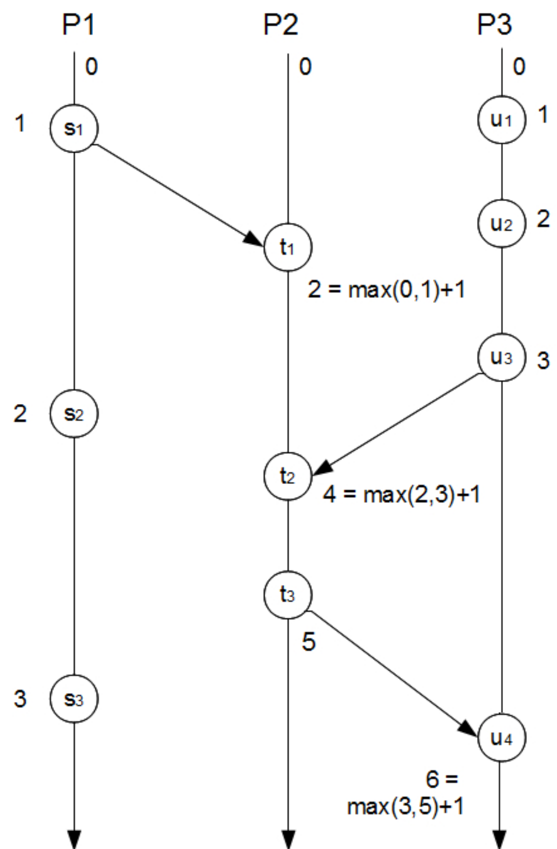


Abbildung 2.

Abbildung 2 zeigt die Anwendung des in Algorithmus 1 beschriebenen Algorithmus bei drei Prozessen und mehreren Ereignissen.

- Prozess P1 sendet in Zustand s_1 eine Nachricht an Prozess P2. Beim Empfang der Nachricht in Zustand t_1, bildet Prozess P2 das Maximum aus dem empfangenen Zeitstempel 1 und dem lokalen Zeitstempel 0. Das Ergebnis 1 wird dann um eins erhöht, so das der endgültige Zeitstempel 2 lautet.
- Im Zustand u_1 tritt ein internes Ereignis bei Prozess P3 auf. Daher wird der aktuelle lokale Zeitstempel von 0 auf 1 erhöht. Ein weiteres internes Ereignis in Prozess P3, tritt im Zustand u_2 auf. Dabei wird der aktuelle lokale Zeitstempel um eins von 1 auf 2 erhöht.
- Im Zustand u_3 sendet, Prozess P3 eine Nachricht an Prozess P2. Beim Empfang der Nachricht in Zustand t_2 wird wieder das Maximum des empfangenen und des lokalen Zeitstempels gebildet und das Ergebnis um eins erhöht. So das der endgültige Zeitstempel 4 ist.
- Im Zustand s_2 tritt ein internes Ereignis bei Prozess P1 auf. Daher wird der aktuelle lokale Zeitstempel von 1 auf 2 erhöht. Nachfolgend tritt im Zustand s_3 wieder ein internes Ereignis auf. Der aktuelle lokale Zeitstempel wird um eins von 2 auf 3 erhöht.
- Prozess P2 sendet im Zustand t_3 eine Nachricht an Prozess P3. Dort wird das Maximum aus dem aktuellen lokalen Zeitstempel 3 und dem empfangenen Zeitstempel 5 gebildet. Das Ergebnis um eins erhöht ergibt dann den endgültigen Zeitstempel 6 für den Empfang der Nachricht.

E. Verwendung und Grenzen der Logischen Uhr

Logische Uhren werden vor allem in Bereichen verwendet, in denen Kausalität und Verlässlichkeit besonders wichtig sind, wie z.B. in Transaktionssystemen oder zuverlässigen Netzwerkprotokollen.
Dabei bieten logische Uhren neben dem Vorteil der einfachen Implementierung und Synchronisation, nur einen störenden Nachteil. Es ist nicht möglich die Nebenläufigkeit von Ereignissen zu bestimmen. Zur Behebung dieses Nachteils wurden die Vektoruhren entwickelt.

III. Vektoruhren

Eine Vektoruhr ist eine Erweiterung von logischen Uhren, die es ermöglicht ein System von Abhängigkeiten ableiten zu können und sich dabei besonders dafür eignet, nebenläufige Ereignissen zu ermitteln. Ähnlich wie bei logischen Uhren führt jeder Prozess einen Zähler C, der bei jedem Ereignis erhöht wird und als Zeitstempel dient. Aber anders als bei der logischen Uhren besteht der Zähler C eines Prozesses nicht nur aus einer Variable, sondern aus einem Vektor von Variablen, die als Zeitstempel VC dienen. Für N Prozesse in einem verteilten System, hat der Vektor eine Größe von N. Der Vorteil eines Vektors ist, das jeder Prozess sich nicht nur seinen eigene Zählerstand, sondern auch die Zählerstände aller anderen Prozesse merken kann.

Definition 3. *Folgende Bedingungen müssen Vektoruhren erfüllen. Für alle Events e,f,g gilt:*

Wenn Ereignis e Happened-before Ereignis f, dann ist der Zeitstempel VC des Ereignis e kleiner als der Zeitstempel VC des Ereignis f.

$$e \to f \Rightarrow VC(e) < VC(f)$$

Wenn der Zeitstempel VC des Ereignis e kleiner ist, als der Zeitstempel VC des Ereignis f, dann ist Ereignis e Happened-before Ereignis f.

$$VC(e) < VC(f) \Rightarrow e \to f$$

Die beiden Bedingungen zusammengefasst:

$$VC(e) < VC(f) \Leftrightarrow e \to f$$

A. Vergleich von Vektoren

Um die Zeitstempel vergleichen zu können, gibt es die Relationen $=, \leq, <$ und $\|$.

Definition 4. *Für zwei Vektoren a und b der Größe N gilt:*

$$a \neq b \Leftrightarrow \exists i : a[i] \neq b[i]$$
$$a = b \Leftrightarrow \forall i : a[i] = b[i]$$
$$a \leq b \Leftrightarrow \forall i : a[i] \leq b[i]$$
$$a < b \Leftrightarrow a \leq b \wedge a \neq b$$

Für zwei Ereigniss e, f:

$$e \parallel f \Leftrightarrow (VC(e) \nleq VC(f)) \wedge (VC(f) \nleq VC(e))$$

B. Implementierung von Vektoruhren

Die Variable N bei der Definition des Vektors v ist die Anzahl der Prozesse im System. Dabei korrespondiert der i-te Prozess mit dem i-ten Eintrag einer Vektoruhr VC. Bei einem internen Ereignis wird der eigene Eintrag des Vektors um eins erhöht. Beim Senden einer Nachricht, wird der eigene Eintrag des Vektors um eins erhöht und dann eine Kopie des Vektors an die Nachricht angehangen. Beim Empfangen eine Nachricht, wird Komponentenweise das Maximum aus dem mitgesendeten Vektor und den lokalen Vektor gebildet. Im Anschluß wird der eigene Eintrag des Vektors um eins erhöht.

Algorithmus 2. *aus [Elem]*

Process$_j$

var:

Array of Integers[1 . . . N]v := 0;

send event*(s, send, t):*

t.v := s.v; t.v wird als Teil der Nachricht mitgesendet
t.v[j] := t.v[j] + 1;

receive event *(s, receive(u), t):*

for i := 1 to N *do*
t.v[i] := maximum(s.v[i], u.v[i]);
t.v[j] := t.v[j] + 1;

internal event *(s, internal, t):*

t.v := s.v;
t.v[j] := t.v[j] + 1;

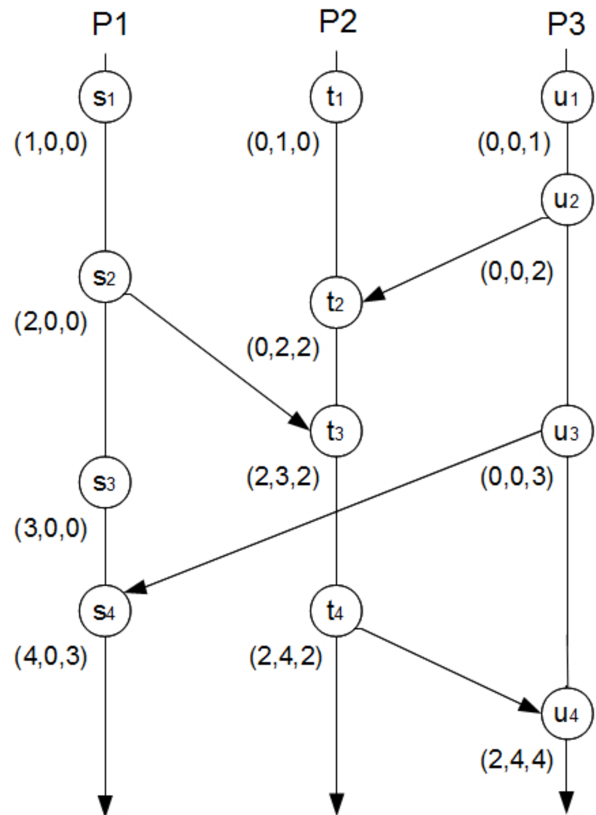


Abbildung 3.

C. Anwendung des Algoritmus für Vektoruhren

Abbildung 3 zeigt die Anwendung des in Abschnitt B beschriebenen Algorithmus für ein Beispiel mit drei Prozessen und verschiedenen auftretenen Ereignissen.

- Im Zustand s_1 tritt ein internes Ereignis beim Prozess P1 ein, daher wird die eigene Komponente des lokale Zeitstempel (0,0,0) um eins auf (1,0,0) erhöht.
- Prozess P3 sendet in Zustand u_2 eine Nachricht an Prozess P2. Dazu erhöht der Prozess P3 seine Komponente des lokalen Zeitstempel um eins auf (0,0,2) und verschickt den Zeitstempel mit der Nachricht. Beim Empfang durch Prozess P2, bildet dieser das Maximum (0,1,2) aus den aktuellen lokalen und empfangenen Zeitstempel und erhöht den eigenen Eintrag des Zeitstempels um eins auf (0,2,2).
- Prozess P1 sendet im Zustand s_2 eine Nachricht an Prozess P2. Dazu wird der lokale Zeitstempel (1,0,0) um eins auf (2,0,0) erhöht und als Kopie an die zusendende Nachricht gehängt. Beim Empfang der Nachricht bildet Prozess P2 dann das komponentenweise Maximum aus seinem aktuellen lokalen Zeitstempel (0,2,2) und dem mit der Nachricht empfangenen Zeitstempel (2,0,0). Im Ergebnis (2,2,2) wird dann der eigene Eintrag des Vektors aus dem Zeitstempel um eins auf (2,3,2) erhöht.
- Bei Prozess P1 tritt im Zustand s_3 ein internes Ereignis auf, so das Prozess P1 seine Komponente des lokalen Zeitstempels von (2,0,0) um eins auf (3,0,0)erhöht.
- Im Zustand u_3 wird durch Prozess P3 eine Nachricht an Prozess P1 gesendet. Prozess P1 bildet beim Empfang nun das Maximum aus dem aktuellen lokalen Zeitstempel (3,0,0) und dem vom Prozess drei empfangenen Zeitstempel (0,0,3). Im nächsten Schritt erhöht Prozess P1, die eigene Komponte des Ergebnisvektors (3,0,3) um eins auf den endgültigen Zeitstempel (4,0,3).
- Prozess P3 empfängt im Zustand u_4, eine Nachricht von Prozess P2. Es wird das Maximum gebildet aus (2,4,4) und (0,0,3). Dann wird der eigene Eintrag des Ergebnisses (2,4,3) durch Prozess P3 um eins auf (2,4,4) erhöht.

Literatur

[Lamp] Leslie Lamport, *Time, clocks, and the ordering of events in a distributed system.* New York, USA, 1978.

[Elem] Vijay K. Garg, *Elements of Distributed Computing*, 1rd ed. USA: John Wiley Sons, 2002.

[Vtgb] Günther Bengel, *Verteilte Systeme: Grundlagen und Praxis des Client-Server-Computing.* 3rd ed. Deutschland, 2004.

[rech] Martin Eisenhardt, Andreas Henrich, Stefanie Sieber, *Rechner- und Betriebssysteme, Kommunikationssysteme, Verteilte Systeme* http://www.uni-bamberg.de/minf/RBKVS-Buch, 10.06.2011.

Clocks of Different Dimensions

Marcel Strudthoff
Universität Oldenburg
Department für Informatik
Oldenburg, Germany
marcel.strudthoff@uni-oldenburg.de

***Abstract*—Multidimensional clocks allow asynchrounous operating systems to coordinate their actions. In this paper, there will be given two examples of clocks: Direct dependency clocks with a dimension of 1 and matrix clocks with a dimension higher than 1:. Direct dependency clocks work almost analog to vector clocks but having some updates to make the substitude of messages more effective. Matrix clocks are multidimensional clocks in which every row is analog to a vector clock. The function of this type of clocks lies in the case, that processes are allowed to know what other processes know, what means, that every process has knowledge of the global system state.**

***Keywords*-fault tolerantce; distributed systems; failure models; multidimensional clocks; matrix clocks; vector clocks**

I. Introduction

It is fact that different applications may need clocks with different dimensions. This paper shows two kinds of clocks, where the second is multidimensional, and their algorithms.

A. Basics

The main principles behind the design and use of clocks are given by logical and vector clocks. A vector clock has three functions:

***Send* event**
The procedures own logical clock is incremented by one. The adressed clock receives the current condition of all clocks given in the sended vector.

***Receive* event**
Its own logical clock is incremented by one. Secondary it updates its own vector by taking the maximum value from its own and the received vector.

***Internal* event**
Its own logical clock is incremented by one.

B. Notation

In this paper, the existing algorithm-pseudocodes and proofs for theorems have special notations to keep them more easier to understand. In the following tabular, these summarations are registered.

Summaration	**Meaning**
$x \stackrel{d}{\to} y$	x directly preceds y
(n, i)	The nth interval of process P_i
$\perp$	The null interval
$pred.u.i$	Predecessor of state u of process P_i
$succ.u.i$	Successor of state u of process P_i
M_k^n	The matrix on the nth interval of process P_k
$M[i, \cdot]$	The ith row of the Matrix M
$P_k.v[i]$	The ith entry in the vector of process P_k

In the following, there will be given two sections of clocks:

- Direct-dependency clocks (one-dimensional)
- Matrix clocks (multidimensional)

II. Direct-Dependency Clocks

As in vector clocks, direct-dependency clocks send only their local vector as their message. They also increment their own logical clock as vector clocks. The *internal* event is the same as in vector clocks.

A difference exists in the action of receiving a message. When a clock A receives a *send* event from clock B, it not only updates its own clock vector but updates the vector clock of B too. While updating the vector of clock B, clock A takes the maximum with the previous value.

A. Algorithm

The following pseudocode explains the function of a direct-dependency clock.

Initialising a process (clock j)

```
(1)    v = [ ]
(2)    FOR i = 0 ... n , i = i + 1
(3)        IF i ≠ j
(4)            v[i] = 0
(5)        ELSE
(6)            v[i] = 1
```

Send event (s,*send*(u),t)

(1) $t.v[t.p] = s.v[t.p] + 1$

Receive event (s,*receive*(u),t)

(1) $t.v[t.p] = \max(s.v[t.p], u.v[u.p]) + 1$

(2) $t.v[u.p] = \max(s.v[t.p], s.v[u.p])$

Internal event (s,*internal*,t)

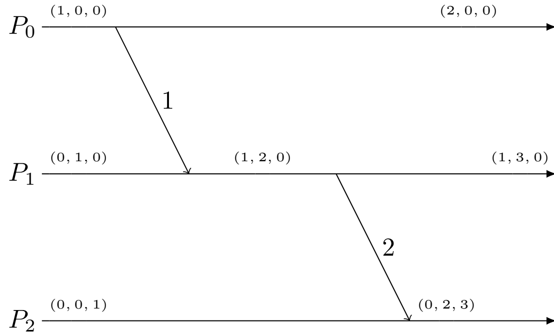
(1) $t.v[t.p] = s.v[t.p] + 1$

First, at the initialising process, the clock-vector for the process must be initialised. All timestamps are inizialised by *zero*, but the processes own timestamp is inizialised by 1.

At a send event, the process sends its own incremented timestamp $s.v[t.p]$ to another process in the system. By receiving a message, the other process maximizes two timestamps: The one from the sending process and its own. It maximizes its timestamps by comparing the sended timestamp $s.v[t.p]$ (respectively $t.v[u.p]$) with the one in its own vector $u.v[u.p]$ (respectively $s.v[u.p]$). Its own timestamp $s.v[t.p]$ also gets incremented. On an internal event, the process increments its own timestamp within its clock vector.

B. Example

The direct-dependency clock algorithm will now be explained via an example, which shows the characteristic features of it.



First, all processes $P_0 \dots P_2$ initialize their vectors. The vector of P_0 is initialised as $(1, 0, 0)$, the one of P_1 is initialised as $(0, 1, 0)$ aud finally the vector of P_2 is initialised as $(0, 0, 1)$.

Then P_0 sends a message to P_1. So a message from P_0 is received by P_1. P_0 sends its incremented value with this condition:

$$P_0.v[1] + 1 = 0 + 1 = 1$$

P_1 receives this message and maximises its vector:

$$\begin{aligned} P_1.v[0] &= \max(P_1.v[0], P_0.v[0]) &&= \max(0,1) \\ &= 1 \\ P_1.v[1] &= \max(P_1.v[1], P_0.v[0]) + 1 &&= \max(1,1) + 1 \\ &= 1 + 1 = 2 \end{aligned}$$

So, in result, P_1 gets its vector with this conditions:

$$P_{1.v} = (1, 2, 0)$$

The next step is a communication between P_1 and P_2. P_1 sends its actual vector. P_2 receives this message and maximises its vector:

$$\begin{aligned} P_{2.v}[1] &= \max(P_{2.v}[1], P_{1.v}[1]) &&= \max(0,2) \\ &= 2 \\ P_{2.v}[2] &= \max(P_{2.v}[2], P_{1.v}[1]) + 1 &&= \max(1,2) + 1 \\ &= 2 + 1 = 3 \end{aligned}$$

$$\Rightarrow P_{2.v} = (0, 2, 3)$$

C. Properties

There are some lemmas and definitions about direct-dependency clocks, which now will be proofed.

Lemma 1:

$$s \to t \Rightarrow s.v[s.p] < t.v[t.p]$$

This means that the value of every state s is lower than the one of a following state t. Note that s does not have to be a directly preceding state.

Proof by Induction: It is sufficent to show

$$\forall k > 0 \colon s \xrightarrow{k} t \Rightarrow s.v[s.p] < t.v[t.p]$$

We will perform on induction over k:

Base: $k = 1$

$s \xrightarrow{1} t$, so state t follows directly on s. In the algorithm, the preceding state s is sended by its process on a *send* event to another process with state t. By choosing the maximum of its own old value $t.v[t.p]$ and the sended value $s.v[s.p]$, it can be said that

$$t.v[t.p] = \max(s.v[s.p], t.v[t.p])$$

$$\Rightarrow t.v[t_{p+1}] \geq s.v[s.p]$$

Because of the incrementation at *send*, *receive* and *internal* events the new value of state t is higher than the old one of s. Thus

$$s.v[s.p] < t.v[t.p]$$

Induction: $k > 1$ $s \xrightarrow{k} t$, but $s \not\xrightarrow{1} t$. It is possible to go through the steps the processes did. Because of this there exists a state u such that

$$s \xrightarrow{k} t = s \xrightarrow{k-1} u \wedge u \xrightarrow{1} t$$

The induction hypothesis is

$$s \xrightarrow{k-1} u \Rightarrow s.v[s.p] < u.v[u.p]$$

From *Base* evidence it can be derivated:

$$u.v[u.p] < t.v[t.p]$$

Together we have:

$$s.v[s.p] < t.v[t.p]$$

Definition 2:

$$s \xrightarrow{d} t \overset{\text{def}}{s} \prec t \vee \exists q, r \colon s \preceq q \wedge q \to r \wedge r \preceq t$$

This means that a state s directly precedes a state t, if and only if there is a path from s to t using at most one message in the happened before diagram of the computation. It's a subset of the relation $\to$.

This definition is important for the next property, which makes direct-dependency clocks useful for many applications.

Lemma 3:

$$\forall s, t \colon s.p \neq t.p \colon (s \xrightarrow{d} t) \Leftrightarrow (s.v[s.p] \leq t.v[t.p])$$

This means that for all states s and t, in which the relations $s.p$ and $t.p$ are not equal, s directly precedes on t if and only if $s.v[s.p] \leq t.v[t.p]$

It can be proofed by induction, analog to the proof for $s \to t$. For $s \xrightarrow{d} t$, the attention will be restricted only to those chains that go only trough processes $s.p$ and $t.p$. From state q to r each link in this chain satisfies $q.v[q.p] \leq r.v[q.p]$

The formal proof is about using induction on the rank of t. The base case is for $k = 0$, using k as the rank of t. The whole proof can be found in [1].

III. Matrix clocks

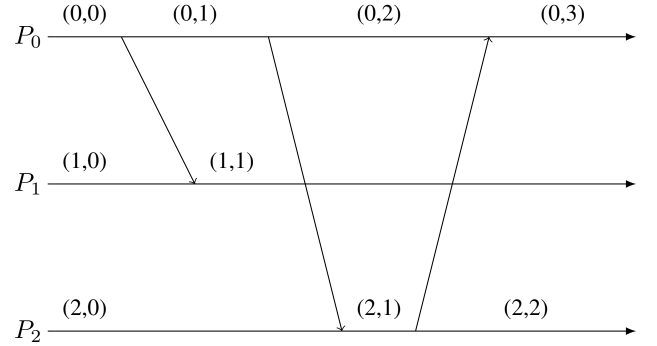
Matrix clocks are higher dimensioned than vector clocks and direct dependency clocks. They were generally buildt in $n \times n$ dimensions for n different processes. The benefit of matrix clocks is the possibility for a process i to get the knowledge of process j about process k, so there is a maximum of information in this matrix. In some applications, this enhanced information is necessary or could help to make algorithms more effective.

A. State Intervals

A state interval is defined as a sequence of states between two external events. An external event is either a *send* event or a *receive* event, where informations were added from outside the process.

The notation of a state intervall is denoted by (i, n), meaning the nth interval in process P_i. This is the subchain of $(S_i, \to)$ between the $(n-1)^{th}$ and the nth external event.

An overview of the functionality of state intervals is given in the following example:



The first intervals of all processes are subscripted by zero. After the first message is sended from P_0 and received from P_1, the indexes of both processes will be incremented to 1. So the state intervals are now

- $(0, 1)$ for P_0
- $(1, 1)$ for P_1 and
- $(2, 0)$ for P_2, because there were no external events arriving until now.

The next external event is a *send* event form P_0 received by P_2, so both processes P_0 and P_2 increment their indicies to $(0, 2)$ respectively $(2, 1)$. Finally, the last event in this example is a *send* event form P_2 back to P_0. P_0 and P_2 increment their index once again, so the last state intervals for each process is:

- $(0, 3)$ for P_0
- $(1, 1)$ for P_1
- $(2, 2)$ for P_2

B. Predecessor and Successor Functions

The predecessor is the latest state in a process that causally precedes this state. Let S_i be a state of process P_i, and $u \in S_i$, so the formal notation for the predecessor is:

$$pred.u.i = \max\{v \in S_i \colon v \to u\}$$

Let $v = pred.u.i$. In this case, v is the maximum element in $(S_i, \to)$ causally preceding u. $pred.u.i$ is $\perp$ if there is no element which causally precedes u.

The successor of a state (n, i) on S_i is just the previous state $(n-1, i)$. It is defined as followed:

$$succ.u.i = \min\{v \in S_i \colon u \to v\}$$

C. Pseudocode

The pseudocode of matrix clocks is similar to the one of direct dependency clocks, but there are a few different parts.

Inizialising a process (clock j)(clock j)

(1) $v = [\,][\,]$
(2) *FOR* $i = 0 \ldots n$, $i = i + 1$
(3) *FOR* $j = 0 \ldots n$, $j = j + 1$
(4) *IF* $i = k$ *AND* $j = k$
(5) $v[i][j] = 1$
(6) *ELSE*
(7) $v[i][j] = 0$

Send event

(1) Tag message with $M_k[.,.]$
(2) $M_k[k, k] = M_k[k, k] + 1$

Receive event $(W[.,.], w_p)$

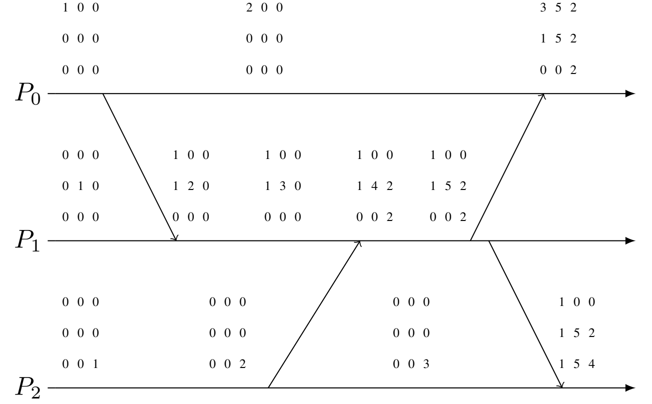
(1) *FOR* $i = 0 \ldots n$, $i \neq k$
(2) $M_k[i, .] = \max(M_k[i, .], W_k[i, .])$
(3) *FOR* $j = 0 \ldots n$
(4) $M_k[k, j] = \max(M_k[k, j], W_k[w_p, j])$
(5) $M_k[k, k] = M_k[k, k] + 1$

The initialisation of a process is similar to the process of vector and direct-dependency clocks. The difference is the initialisation of a $n \times n$ matrix and not only a vector. The matrix of every process is the same: all entries are initialised with zero but the entry $M_k[k, k]$ for process k will be initialised with 1.

At a send event, the process sends a tagged message with the processes matrix. The whole matrix will be sended. When a process receives such a *tagged message*, it will update its own matrix M_k by taking the maximum of the compared entries in its own and the received matrix W for every entry, except in the kth row, where the process updates its own knowledge by taking the knowledge of the sending process. Further it will increment its own entry in the matrix clock $M_k[k, k]$.

D. Example

The following shows an example for three processes P_0, P_1 and P_2 using matrix clocks.



In this example, all processes first initialize their matrix clock as shown in the pseudocode. So, P_0 initialises its matrix clock with zeros but $M[0, 0]$ with 1. P_1 and P_2 initialise analog.

The first message in this example is sended from P_0 to P_1, right after the initialising procedure. When P_1 receives the message from P_0, it updates its own matrix clock by taking the 0th vector from the matrix clock of P_0, updating the 0th entry in its own vector (the 1st) and increasing its own entry at $M[1, 1]$.

In the following, P_0 and P_1 are increasing their matrix clock due to internal events. It not only increases its matrix clock after that but sends its tag to process P_1. P_1 updates its matrix clock after receiving a ,message from P_2 once again by taking the 2nd vector from the matrix clock of P_2, updating the 2nd entry in its own vector. Finally, it increases its own entry.

Now, process P_1 has a farreaching knowledge:

- It knows, that P_0 only has knowledge of its own process but no knowledge of the processes P_1 and P_2 in internal
- It knows, that the last action of P_0 was at timestamp 1 (although thats not the exact state of P_0 at the moment)
- It knows, that the last action of P_2 was at timestamp 2
- It knows, that P_2 only has knowledge of its own process but no knowledge of the processes P_0 and P_1

In this example, P_1 changes this lack of knowledge. After the next action, in which it increments its own entry to step 5, it sends a message first to process P_0 and after that to P_2. P_0 updates its matrix clock by updating its 1st and 2nd vector

and additional the 1st and 2nd entrance of its own vector. After that, it increases its own entry to 3 at $M[0,0]$. From this message, P_0 not only got information of P_1 by updating its matrix clock but in addition the message includes some information of P_2.

P_2 works in the same way. The only difference is that between the sended message to P_1 and the received from P_1 back, some internal events occured, too. So the timestamp of process P_2 will now be increased to 4.

IV. Garbage Collection

For a matrix clock of dimension k it is necessary to compute k compositions of the $pred$ function. In the case that $k = 1$, the matrix clock is equivalent to a vector clock with one entry and the algorithm can be reduced. So, when $s.p \neq i$, $pred(s).i = s.v[i]$. For $k = 2$, a two-dimensional matrix, $pred(pred(s).i).j = s.M[i,j]$. In this way, the procedure can be extended to any dimension.

This treatment can be useful in many situations. One of this situations is the following theorem.

Theorem 4:
Let $s.p = r.p$. If

$$\forall i\colon r.M[r.p, r.p] \leq s.M[i, s.p])$$

then

$$\forall t\colon t \| s\colon r \to t$$

This means that when $s.p$ is equal to r.p and for all i the $s.p^{th}$ entry of the ith row in the matrix of process s is bigger or equal to the $r.p^{th}$ entry of the $r.p^{th}$ row in the matrix of process r, then for all t parallel to s hold that r happened before t.

Proof:

$$\begin{array}{lrcl} & s & \| & t \\ \Rightarrow & pred.s.i & \to & t \\ & (\forall i\colon r.M[r.p, r.p] & \leq & s.M[i, s.p]) \\ \Rightarrow & r & \preceq & pred.(pred.s.i).(s.p) \\ \Rightarrow & r & \to & t \end{array}$$

V. Conclusion

Higher dimensional clocks are very important instruments to coordinate asynchron operating systems. They are able to give a process the information desk from another process or even from the whole system. In this paper, two higher dimensional clock types were presented. Direct dependency clocks do not need much space to work efficiently, but they can only give a process P_k the information desk from another process P_l. On the other hand, matrix clocks are better to get the desk of the whole system. Using matrixs clock, a process P_k can get the whole information from process P_l, either what P_l knows about another process P_m or (what can be very important, too) what P_l knows about P_k. The efficiency in matrix clocks lies in the case that P_k can get the information about P_l and P_m only asking one of the processes. That needn't be the right desk of the whole system (even when for example P_m and another process P_n operate together after P_m contacted P_l), but in huge systems with many processes, the number of messages can be reduced drastically.

References

[1] V. K. Garg, *Elements of Distributed Computing*, Wiley-Verlag, 2002, ISBN 0-471-03600-5

[2] V.K. Garg and A.I. Tomlinson *Using the causal domain to specify and verify distributed programs*, The University of Texas at Austin, october 1996.

Concepts and Methods in Distributed Systems Seminar
– Distributed Mutual Exclusion –

Pascal Hofstee
Carl von Ossietzky University
Oldenburg, Germany
pascal.hofstee@uni-oldenburg.de

***Abstract*—In a distributed computing environment, traditional approaches to ensuring mutual exclusion, e.g. semaphores, are not available, because there exists no system resource that can maintain identical state between all processes involved at any point in time. A way to deal with this limitation is to implement mutual exclusion by passing messages between processes in a structured manner.**

The algorithms used for this, can be classified into token-based, non-token-based and quorum-based approaches. Token-based algorithms function by passing around a unique token between involed processes, where the process currently holding the token is implicitely allowed to enter the critical section. Non-token-based algorithms function by an exchange of two or more rounds of messages between all processes to determine who can enter the critical section next. In quorum-based algorithms each process is assigned a subset of all available processes (called a quorum) with which to communicate. Each two of such quorums share at least one common element, which is responsible for ensuring only a single process is allowed entrance to the critical section.

***Keywords*-mutual exclusion algorithm, distributed system.**

I. Introduction

Just as with traditional single-site computing systems, distributed systems occasionally require mutual exclusion to handle critical sections (CS), e.g. to safely manage system resources. In a distributed environment however, traditional approaches, like semaphores, are not available, because there exists no system resource that can maintain identical state between all processes involved at any point in time. A way to deal with this limitation in a distributed computing environment, is to implement mutual exclusion by passing messages between processes in a structured manner.

The algorithms developed for this purpose, can be classified into three individual categories; token-based, non-token-based and quorum-based approaches.

This article explains the core methodology behind each of these, through describin algorithms representative of each category.

II. System model

Each algorithm presented, is written according to a specific set of assumptions regarding the used system model:

- there exist a total of n processes $p_1, p_2, \ldots, p_n$
- each process can be in any of three possible states: requesting CS, executing CS, idle (neither requesting nor executing CS).
- a process is blocked and cannot make further CS requests while in 'requesting CS' state, and is executing outside its CS when in 'idle' state.
- in token-based algorithms a process can be executing outside the CS while in possession of the token. This is called 'idle token state'.
- at any instance, a process may have several pending requests for CS from multiple processes, which are queued and processed sequentially.

These assumptions are only made to simplify the observed model and are not in itself limiting the general application of the presented algorithms.

III. Performance metrics

Because individual algorithms use different approaches to accomplish their goals, trying to compare their performance requires different metrics, a.o.:

- **Message complexity**: the number of messages required per CS execution by a process
- **Synchronisation delay**: the time between one process leaving the CS and another entering the CS, measured in units of T, where T is the average time required for a message to travel between two processes.

IV. Non-token based algorithms

The first class of algorithms presented are the non-token-based algorithms, for which Lamport's, and Ricart-Agrawala's algorithm are typical representatives. The basic principle behind this class of algorithms, is the passing of multiple rounds of messages between all processes involved, to determine which process is allowed to enter the critical section next.

These algorithms use Lamport-style logical clocks to prioritize CS requests; each request is tagged with a combination of process ID and timestamp (t_{p_i}, i). For this mechanism to work though, it is required that the communication channels used provide FIFO semantics [1].

A. Lamport's algorithm

Lamport's algorithm uses a simple data structure called a request queue RQ_i at each process p_i, in which that process stores requests for CS. sorted by increasing order of timestamps. It is based on the use of three different types of messages: REQUEST, REPLY, and RELEASE.

Requesting the critical section

When process p_i wants to enter the CS, it broadcasts a REQUEST(t_{p_i}, i) message to all other processes and places the request on its queue RQ_i.

When process p_k receives this REQUEST(t_{p_i}, i) message, it puts this request on its queue RQ_k, and sends a timestamped REPLY message back to p_i.

Executing the critical section

Once process p_i has received a message with timestamp greater than (t_{p_i}, i) from every other site, and p_i's request is at the top of queue RQ_i, it can safely enter the CS.

Releasing the critical section

Upon leaving the CS, process p_i removes its request from the top of its queue, and broadcasts a RELEASE message to all other processes.

When a process p_k receives a RELEASE message from p_i, it removes p_i's request from its request queue RQ_k. It is possible that this causes p_k's request to become top of the queue RQ_k, potentially enabling p_k to enter the CS.

For each execution of the critical section, Lamport's algorithm requires $(n-1)$ REQUEST messages, $(n-1)$ REPLY messages and $(n-1)$ RELEASE messages, giving it a message complexity of $3(n-1)$, with a synchronization delay of T.

B. Ricart-Agrawala algorithm

The Ricart-Agrawala algorithm is an optimization of Lamport's algorithm, reducing the message complexity by eliminating the need for RELEASE messages. Instead of the previous request queue RQ_i, each process maintains a Request-Deferred array RD_i. These arrays contain a boolean flag for each process to indicate whether or not that process has a REPLY message pending. Initially $\forall i \forall k : 1 \leq i, k \leq n :: RD_i[k] = 0$

Requesting the critical section

When process p_i wants to enter the CS, it broadcasts a REQUEST(t_{p_i}, i) message to all other processes.

When process p_k receives this REQUEST(t_{p_i}, i) message, it sends a REPLY message to process p_i, if p_k is neither requesting nor executing the CS, or if the timestamp in p_i's request is less than p_k's own request. Otherwise, the reply is deferred and p_k sets $RD_k[i] = 1$.

Executing the critical section

Process p_i enters the CS, once it has received a REPLY message from every process it sent a REQUEST message to.

Releasing the critical section

Upon leaving the CS, process p_i sends a REPLY message to each process marked with a deferred request in RD_i.

With the need for RELEASE messages eliminated, the Ricart-Agrawala algorithm has reduced the message complexity of the original Lamport algorithm down from $3(n-1)$ to $2(n-1)$, while maintaining a synchronization delay of T.

V. QUORUM-BASED ALGORITHMS

When compared to the non-token based algorithms presented earlier, quorum-based algorithms differ in two distinct ways:

1) Each process p_i is assigned a request set R_i containing a subset of all process IDs (a quorum). Requests for CS are not send to all other processes, but only to those processes referenced in its quorum.
   These request sets are always chosen such that each pair (R_i, R_k) always shares at least one member which mediates conflicts between that pair.
2) A process can send out only a single REPLY message at any time, which in turn can only be send after a RELEASE message has been received for the previous REPLY message.

A. Maekawa algorithm

Maekawa's algorithm was the first mutual exclusion algorithm to use a quorum-based approach. The request sets are constructed to satisfy following conditions:

- M1: $\forall i \forall j : i \neq j, 1 \leq i, j \leq n :: R_i \cap R_j \neq \emptyset$
- M2: $\forall i : 1 \leq i \leq n :: p_i \in R_i$
- M3: $\forall i : 1 \leq i \leq n :: |R_i| = K$
- M4: any process p_i is contained in K request sets.

Of these, only rules M1 and M2 are strictly required, whereas M3 and M4 provide the algorithm a fairness aspect in that each process has to do equal amount of work to invoke and enforce mutual exclusion. Using the theory of projective planes, Maekawa showed that $n = K(K-1) + 1$, resulting in $|R_i| = \sqrt{n}$ for each request set R_i.

For Maekawa's algorithm each process p_i, maintains a Request-Deferred queue RD_i in which it queues requests for CS from processes that it currently cannot grant.

Requesting the critical section

When process p_i wants to enter the CS, it sends a REQUEST(t_{p_i}, i) message to all processes in its request set R_i.

When process p_k receives this REQUEST(t_{p_i}, i) message, it sends a REPLY message to p_i, provided it has not sent a REPLY message to a process since it last received a RELEASE message. Otherwise, it puts p_i's request on its Request-Deferred queue RD_k

Executing the critical section

Process p_i enters the CS, once it has received a REPLY message from every process in R_i.

Releasing the critical section

Upon leaving the CS, process p_i sends a RELEASE message to each process in R_i.

When a process p_k receives a RELEASE message from p_i, it takes the top request from RD_k, and sends a REPLY message to that process. In case of an empty queue

it alters its state to reflect it has not sent out a REPLY message since the receipt of the last RELEASE message.

Since the size of a request set is $\sqrt{n}$, an execution of the critical section requires $\sqrt{n}$ REQUEST, REPLY, and RELEASE messages, giving a message complexity of $3\sqrt{n}$. Compared to the linear cost associated with the previously described non-token-based algorithms, this is a significant improvement. Synchronisation delay however has now increased to $2T$.

A problem with the algorithm as presented, is its sensitivity to deadlocks, where multiple quorums develop a circular dependency.

To resolve such deadlocks, the algorithm is extended by requiring a process to yield a lock if the timestamp of its request exceeds that of another request waiting for the same lock. This is managed by introducing three new messages:

- A FAILED message is sent from p_i to p_k, when p_i cannot grant p_k's request because it has granted permission to a process with a higher priority request. Higher priority here means a request with a lesser timestamp.
- An INQUIRE message is sent from p_i to p_k, when p_i wants to find out whether or not p_k has succeeded in locking all processes in its request set.
- A YIELD message is sent from p_i to p_k, when p_i returns permission back to p_k.

Handling Deadlocks

When a REQUEST(t_{p_i}, i) blocks at p_j because p_j has already granted permission to p_k, then p_j sends a FAILED(j) message to p_i, if p_i's request has lower priority, and queues p_i's request. Otherwise, p_j sends an INQUIRE(j) message to p_k.

In repsonse to an INQUIRE(j) message, p_k sends a YIELD(k) message to p_j, provided p_k either received a FAILED message from a process in its request set R_k or sent a YIELD(k) message to any process without having received a new REPLY message from it.

In response to a YIELD(k) message, p_j acts as if it received a RELEASE message from p_k, taking the top request from its queue, and sends a REPLY message to that process. Then it places p_k's request onto its queue appropriately.

This extension to the algorithm increases its message complexity up to $6\sqrt{n}$.

VI. Token based algorithms

The algorithms described up to this point are all assertion-based algorithms; each process asserts that it never acknowledges more than a single process simultaneously at any point in time. Token-based algorithms however, use a privilege-based approach, in which a single token is passed around, implicitely granting the process holding the token permission to enter the critical section.

A. Suzuki-Kasami algorithm

The Suzuki-Kasami algorithm uses process-specific sequence numbers instead of logical-clocks, which are used to identify outdated requests. Each process p_i maintains an array of sequence numbers RN_i, where it stores the highest observed sequence number for each process. Another distinction is that some of the state information is stored inside the token; an array $LN[1 \dots n]$ of most recently executed requests, and the token queue Q containing the queue of requesting processes. [2]

This is done so that each process holding the token always has all required information to ensure mutual exclusion despite potential message delays.

Requesting the critical section

When process p_i wants to enter the CS, and it does not currently hold the token, it increments its sequence number $RN_i[i]$, and sends out a REQUEST$(i, RN_i[i])$ message to all other processes.

When a process p_k receives this message, it sets $RN_k[i]$ to $\max(RN_k[i], RN_i[i])$. If p_k is in the idle token state, it sends the token to p_i if $RN_k[i] = LN_k[i] + 1$.

Executing the critical section

Process p_i enters the CS after it has received the token.

Releasing the critical section

Upon completing the CS, process p_i sets $LN[i]$ of the token array to $RN_i[i]$.

For every process p_k whose ID is not in the token queue yet, it appends that ID, if $RN_i[k] = LN[k] + 1$.

If the token queue is not empty after this update, p_i takes the top process ID from the token queue and sends the token to that process.

Since possession of the token implies permission to enter the critical section, no messages are needed if a process holds the token at the time of its request. Otherwise, the algorithm requires n messages to obtain the token. Synchronization delay in this algorithm is therefore either 0 in the former or T in the latter case.

VII. Outlook

The algorithms presented here are but a small subset of those available, and there exist many that combine aspects of multiple approaches, creating a more hybrid approach. It is safe to say though that token-based algorithms in general are more message-efficient than non-token-based algorithms, because a single token message replaces othwerwise multiple REPLY messages.

References

[1] M. Maekawa, A. E. Oldehoeft, and R. Oldehoeft, *Operating systems: Advanced concepts*. Benjamin/Cummings Pub. Co., 1987.

[2] I. Suzuki and T. Kasami, "A distributed mutual exclusion algorithm," *ACM Trans. Comput. Syst.*, vol. 3, pp. 344–349, November 1985.

Voting in distributed Systems

Robert Schadek
University Oldenburg
Department Systemsoftware und verteilte Systeme
Oldenburg Germany
robert.schadek@informatik.uni-oldenburg.de

***Abstract*—This paper will give you an overview on voting in distributed systems. First , an introduction is given why voting is necessary and error pron. To give a base for argumentation the assumptions of Hector Garcia-Molina will be presented. In this paper voting algorithm for synchronous as well as asynchronous networks are presented. For both groups an algorithm will be discussed. The bully algorithm will be discussed in depth for the synchronous networks. For the asynchronous networks the algorithm by Afek and Gafni will be discussed. To conclude voting on topology will be discussed. Finally a conclusion is given.**

***Keywords*-distributed system; voting; leader election; fault tolerance;**

I. Introduction

Electing a single entity to fulfill some task may seem counterproductive in a distributed system but it has its purpose nevertheless. The reason for electing a leader is to make functions easier. For instance, mutual exclusion with a single instance managing the exclusion is easier and faster than algorithms doing exclusion in a distributed manner. Another case where having a leader makes things easier is when it comes to give instances names or to log in or out of a network. However the problem with having a single instance in charge is, what happens when this instance cannot be reached for whatever purpose? Then, the distributed system must vote for a new leader. In case more than one node starts an election it must be avoided that two or more leader rise to power. Another way to think of the election process is defined by Hector Garcia-Molina: "The election of a coordinator is basically a synchronization of parallel processes problem". Synchronization is a well known problem in distributed systems [3] [4].

II. Assumptions

Before discussing voting algorithms, some assumptions must be made. These assumptions are made to define a common base for reasoning. For example, analysing a voting system where no error model is defined makes it rather hard to argue, about the algorithms. Most voting algorithm use the assumptions defined by Hector Garcia-Molina implicit or explicit. Garcia-Molina defined nine assumptions. `Assumption 1`: All nodes use the same election algorithm. `Assumption 2`: No software ever fails. `Assumption 3`: For every received message a message was sent earlier. `Assumption 4`: All nodes have stable storage. This storage is used for storing the state of the election. `Assumption 5`: If a node fails, it halts immediately (fault stop). After repair the node is reset to a fixed state and resumes execution. `Assumption 6`: No transmission errors occur. If a message is received, it is the message sent. `Assumption 7`: The messages sent from node i to j are processed in the order they were send. Assumption 6 and 7 do not imply that no messages are lost. `Assumption 8`: There is a time limit in which messages are delivered. If no acknowledgment is received in this time limit, the receiver node has failed. `Assumption 9`: Sending a acknowledgment on a received message is the task with the highest priority for every node [3].

III. The bully algorithm

The bully algorithm was presented by Hector Garcia-Molina in 1982 in the paper "Election in a Distributed Computing System" [3]. The name is chosen because the node with the highest ID in the networks forces/bullies its way to the leadership. The idea behind the bully algorithm is, that whenever some node of the network suspects the leader to have died or it thinks that his ID is the highest, a new election is started. The new leader will be the node with the highest ID. All nine assumptions presented earlier are expected to hold. The stable storage defined in Assumption 4 is for instance used to save the state of the node. The time complexity of the bully algorithm is $O(n^2)$. Where n is the number of nodes [7].

A. Prerequisites

Every node has one of four possible states. These four states are *Normal*, *Election*, *Reorganization* and *Down*. A node is in the state *Down* if was repaired after a crash or if it suspects the leader to have crashed. A node enters the state *Election* if it takes part in an election. The state *Reorganization* is entered if a node receives a message defining a new leader. The fourth state, *Normal*, is occupied in any other case.

B. Messages

Nodes can send four kinds of messages. The first one is *Election*. This messages indicates that an election has

started. The message *Coordinator* announces a new leader. The ID of the leader is part of the message. To check the status of other nodes a *Status_Poll* message can be send. To give state information for recovery a *New_State* message will be send. Replies to these messages are tagged as such and contain the ID as well as an appropriate response. As mentioned earlier a node stores information about itself in the stable storage defined in Assumption 4. This information are: Its state, its ID, the ID of the coordinator, the ID of the coordinator to be, a set of IDs of currently active nodes and a set of IDs of nodes who have answered to a *Coordinator Self* message. A message *Coordinator Self* is a message send by the new leader containing its ID. The coordinator to be is the ID of a possible new leader.

C. The voting process

An electing is initiated whenever a new node has entered the system or the leader has crashed. If a new node gets the highest ID in the system, this node needs to become the leader. If the leader has crashed a new leader has to be found.

1) The election procedure: The election can be partitioned into four phases. A node in *Phase 1* checks if nodes with a higher id are active. Should this be the case the node terminates its election process. After a node passes *Phase 1 Phase 2* is entered. In *Phase 2* a message is send to all nodes with a lower id, informing them of the election. It also informs them that the node sending the message might be the new coordinator. In return the node gets the IDs of all active nodes. *Phase 3* checks if all active nodes agree that the initiating node is new leader. If not, the election is restarted. *Phase 4* is used for housekeeping. New recovery information is distributed to all active nodes. Should a node have changed its state since *Phase 2*, the election is restarted. Listing 1 shows the election process as pseudocode. The only thing about the pseudocode that has not been defined earlier is the method *Wait with timeout for reply*.

2) The election loop: To react to received messages every node of the network runs a particular function. Pseudocode for this function can be seen in listing 2. This function works by waiting for a message, checks what kind the message is of and than reacts to it. The pseudocode shows the responses to the four kind of messages defined earlier. III-B

3) Heartbeat protocol: One task of the coordinator is to monitor *his* network. For this purpose he will run the socalled *Heartbeat protocol* in regular intervals. If the coordinator finds a node to be inactive or not in the *Normal* state he starts a new election. Listing 3 show this protocol in pseudocode for better understanding. [1]

4) Recovery from failure: Should node recover from failure it has to start an election. As discussed in III-C, it cannot be predicted if the recovered node is a potential leader. Therefore, on every recovery, an election follows to

```
Elect:
  //Phase 1
  For each node j > SELF.ID
    Send STATUS_POLL message to j;
    Wait with timeout for reply;
    If there is a reply
       return; // SELF cannot be the new
            coordinator
  // Phase 2
  Set ACTIVE to the empty set;
  For each node j < SELF
    Send ELECTION message to j
    Wait with timeout for reply;
    If there is a reply
       add j to ACTIVE;
  // Phase 3
  Set ANSWERS to the empty set;
  For each node j < SELF
    Send COORDINATOR SELF message to j;
    Wait with timeout for reply;
    If there is a reply
       add j to ANSWERS;
  If ANSWERS != ACTIVE
      Goto Elect; // We restart things since the
           network has changed
  // Phase 4
  Set ANSWERS to empty;
  For each node j in ACTIVE
    Send NEW_STATE message and recovery
        information to j;
    If there is a reply
       add j to ANSWERS;
  If ANSWERS != ACTIVE
     Goto Elect;
```

Listing 1. The bully voting algorithm

```
while (1) {
  Wait for a message m;
  switch (m.messageType) {
     case STATUS_POLL:
      Send SELF, STATUS;
      break;
     case ELECTION(sender):
      Set STATUS to ELECTION;
      Let CANDIDATE be a new variable and set it
          to sender;
      Stop processing on this node;
      Stop any other election that may be in
          progress at this time;
      Send SELF;
      break;
     case COORDINATOR(sender):
      If Status == ELECTION and sender ==
          CANDIDATE
        Set STATUS to REORGANIZATION;
        Set COORDINATOR to sender;
        Send SELF;
      break;
     case NEW_STATE(sender,RecoveryInfo):
      If STATUS == REORGANIZATION and sender ==
          COORDINATOR
        Set STATUS to NORMAL;
        Use the Recovery Information;
  }
}
```

Listing 2. The bully voting loop

```
If SELF == COORDINATOR and STATUS == NORMAL
For each node j
    Send STATUS-POLL message to j;
    Wait with timeout for reply(sender, status);
    If j is not in ACTIVE or returned status is
        not NORMAL
        Call Elect procedure;
```

Listing 3. Heartbeat protocol

```
Set STATUS to DOWN;
Call Elect procedure;
```

Listing 4. "Recovery"

reintegrate the node into the network. Listing 4 shows the process of reintegration. [1]

IV. Afek and Gafni's Algorithm

Afek and Gafni prove in their paper "Time and Message bounds for Election in Synchronous and Asynchronous Complete Networks" that the time complexity for electing a leader in a complete connected network $nlog(n)$. In case every node in a network of n nodes tries to be leader at the exact same time one comes to the conclusion that n^2 messages are necessary. In the same paper the author present three asynchronous algorithms, called A, B and C and one synchron algorithm. Asynchron in this context means that messages occur at arbitrary time with no global clock generating something like a round. The asynchronous algorithm derive from the synchronous algorithm. The first two algorithm can be considered trade-offs between time and message complexity. The message complexity of algorithm A is $2 \cdot n \cdot log(n)$ and a time complexity of $n \cdot log(n)$. The time complexity of B is n where it's message complexity is $2,773 \cdot n \cdot log(n)$. The third algorithm C combines A and B to a message complexity of $2n \cdot log(n)$ and a time complexity of n. Afek and Gafni are using the assumptions defined by Garcia-Molina, though not mentioned explicit [1].

A. The algorithm assumptions

On top of the assumptions by Garcia-Molina some more assumptions about the network and its nodes are made. Every election is initiated by a subset of all given nodes. All member of the subset know that the leader has crashed. Every node of this subset tries to capture all nodes in the network and is therefore a candidate for being coordinator. Every node carries a variable by the name of level. This level is used to estimate the number of nodes the node has captured. To capture a node means different things for every algorithm. Further this level is used to contest two nodes.

[1]The Elect procedure defined in Listing 1.

```
1  Candidate program:
2    untraversed := E /* E is the set of all Node
         of the network */
3    level := -1 ;
4    Each round do:
5      level := level + 1 ;
6      if level is even:
7        if untraversed is empty
8          ELECTED, STOP; /* At this point the
               election is won */
9        else:
10         /* This branch send the level and the
11         ID of the candidate to K number of
12         nodes. The Minimum function makes
13         sure that no more message are send
14         then there are nodes in untraversed */
15         K := Minimum(pow(2,level/2) ,|
               untraversed|);
16         Send(level, id) to K Nodes from
               untraversed, and
17         remove these Nodes from untraversed;
18     else:
19       Receive all acknowledgment type messages
20       if received less than K acknowledgments:
21         STOP; /* Not a candidate any more,
               stop the candidate programm */
```

Listing 5. Candidate program

B. The synchronous algorithm

This election algorithm for synchronous complete networks has a message complexity of $3n \cdot log(n)$ messages and a round complexity of $2 \cdot log(n)$ rounds. [2] Listing 5 and 6 describes the algorithm. Every candidate runs the candidate program of Listing 5. The first if-else construct is used to evaluate whether it is time to send messages or wait for them. Should it be time to send messages the expression in line 10 of the listing creates a set of links over which messages, are sent. Every candidate at level i tries to capture 2^i ordinary nodes every round. To capture a node in the synchronous algorithm means to have sent the lexicographically largest level, ID pair to a node. This can be also been seen in line 10. If the candidate program is in an odd level and does not receive as much messages as it has sent one round earlier, it hast lost the election and is removed from the candidate subset. The nodes who have lost the election run the ordinary program from that point on. Compare with line 19 through 21 of Listing 5. A election is won if the set of untraversed links is empty. The ordinary program checks every round if it has received a message from a node whose level is higher than the level of the current owner of the node. Should this be the case, the sender of this message is made the new owner. To give the candidate feedback that it has captured the node, line 6 an acknowledgment is sent every round. Every regular node stores the level of it's owner. Therefore every node has to increment this level by itself, because the level value is only transmitted on the capture but increments

[2]Round complexity means how many rounds are needed to elect a coordinator.

```
1  Ordinary program:
2    l* := nil;
3    level := −1;
4    owner_id := id;
5    Each round do:
6      Send an acknowledgment to l*;
7      level := level + 1;
8      Receive all candidate messages (level,id)
          of nodes l;
9      Let (level*; id*) be the lexicographically
          largest
10     (level; id) candidate message, and
11     l* the node from which it arrived;
12     if (level*; id*) > (level; owner id):
13       (level; owner id ) := (level*; id*);
14     else:
15       l* := nil;
```

Listing 6. Ordinary program

every round.

C. The asynchronous algorithms

To construct the first asynchronous algorithm the synchronous algorithm is modified. Not modifying the algorithm can lead to problems. One problem can be described as followed. Candidates C_1C_2 both try to capture the nodes u, v. The message of C_1 has arrived first at u. The message of C_2 has arrived first a v. Taking the synchronous algorithm in consideration both C_1C_2 would stop there election program. Using the synchronous algorithm for a asynchronous communication mode is not deadlock free.

1) Algorithm A: Algorithm A differs from the synchronous algorithm in following points. The level variable now defines the number of candidates a candidate has captured. To capture a node, the level of the candidate must be strictly larger than the level of the concurrent candidate node. When a candidate tries to capture a node some rules must be followed: If the level of the candidate is lower than that of the ordinary node the candidate is killed. If a node gets captured, it obtains the level of the candidate. Is the level of the node is equal to the level of the candidate, the candidate is sent to the current owner of the node. When this case happens, the following rules apply. P is the candidate sent to the candidate Q. If $(Level(P), id(P)) < (Level(Q), id(Q))$, P is killed. If $(Level(P), id(P)) > (Level(Q), id(Q))$, Q gets killed and P captures the node. Additionally every node gets two pointers called father and potential_father. The father points to the node that has captured the node. The potential_father pointer points to the node that tries to capture the node. Listing 7 shows the algorithm in pseudocode.

```
1  Initially:
2    level := size := 0; untraversed := E; killed
        := true; owner_id := potential id := 0;
3    father := potential father := nil; 0; /*
        size used in algorithm C only, E is
4    the set of all Nodes*/
5
6  Candidate(id):
7  while ( untraversed not empty):
8    l := any( untraversed ) ; send(level; id) on
         l;
9  R:  receive(level*; id*) over l*;
10   if(id* = id)AND not killed:
11   level := level*; untraversed := untraversed
         − l;
12   elif(level*,id* < level, id) OR killed:
13     Discard the message, goto R;
14   else:
15     send(level*,id*) over l*; killed := true;
           goto R;
16
17 Ordinary:
18   level := −1
19   while(not terminated):
20     receive(level*, id*) over l*;
21     case level* of:
22       (1) level* < level: Discard message;
23       (2) level* > level: //replace father
24         father := l*; level := level*;
              owner_id := id*;
25         potential_id := 0; potential_father :=
              nil;
26         send(level*,id*) to father;
27       (3) level* = level:
28         if(id* < owner_id):
29           Discard message;
30         elif(id* = potential_id):
31           father := potential_father;
32           level* := level* + 1;
33           owner_id := id*; potential_id := 0;
34           potential_father := nil;
35           send(level*,id*) to father ;
36         elif there is alread a
              potential_father:
37           Discard message;
38         else: /* there is no potential_father
              */
39           potential_id := id*;
                potential_father := e*;
40           send(level*,id*) to father;
```

Listing 7. Algorithm A

2) Algorithm B and Algorithm C: Algorithm B builds on Algorithm A by changing the way the level is computed and by changing the capturing and elimination rules. In continuation of this approach algorithm C derives itself from Algorithm B. Giving a detailed description of these algorithm would exceed the scope of the paper and the reader is therefore directed to the publication of Afek and Gafni [1].

V. Election on topologies

A. Chang and Roberts algorithm

Ernest Chang and Rosemary Roberts [2] [6] proposed a voting algorithm that works on unidirectional rings. Like most other presented voting algorithms the node with the highest ID becomes leader. An election is started whenever some node suspects the leader to have crashed. Is this the case, the node generates a message with its ID and sends it to its left neighbour. A node receiving a message compares the given ID with its own. If the id is lower than its own the message is not propagated. Should the id be bigger, the

message will be sent to the left. If the ID is the same, as the ID of the node receiving the message, the node which has received the message is the new leader. On top of his ID, every node stores a boolean value indicating if the node is participating in an election. Should a message arrive and the boolean value indicates false, the node generates its own voting message, should the received message hold a lower ID. After the sending of a message, the node sets the value to true, indicating that it is participating in the election.

1) Performance Analysis: The time complexity can be devised from the knowledge that, should every node send a message at the same time, a time of n is required because the message of the node with the highest id must be sent n times to have passed every node. The worst case would be if the node with the lowest id starts the election. Should the node with the highest id be right of that node, it would take $n-1$ messages to reach the node with the highest ID. This node then has to send a message which would take n to pass every other node, brining it to a total of $2n-1$ messages. The message complexity can be broken into three case *best*, *worst* and *average*. Best would be if all nodes were ordered clockwise by there id. This would take a total number of $2n-1$ messages. The worst complexity appear if all nodes where ordert counterclockwise by their ids. The complexity for such a situation would be $n(n+1)/2$. The average case arrives at a complexity of $n \cdot log(n)$.

B. HS algorithm

The HS algorithm by Hirschberg and Sinclair is another election algorithm that works on rings. The difference to the algorithm proposed by Chang and Roberts is, that it works bidirectional rings. A node suspecting the leader to have failed will sent a message with its id in both directions. Every message has a phase assigned to it. This phase defines how many nodes the message will pass before it is send back the way it came to the sender. Every node receiving a message will check the received id against its own. Should the id be larger the message will be propagated, otherwise the message will not be send in any direction whatsoever. Should the original sender receive both messages back it increases the massage phase by 2^{Phase_Number}. Phase_Number is a simple counter for the phases, the initial value is 0. If the node receives both its messages from both neighbours, it knows that it has won the election. Time complexity for this algorithm is n and the message complexity is $n \cdot log(n)$. Figure 1 shows three phases exemplary. It is to note that the number of nodes per phase is not correct, this is not relevant to grasp the algorithm. The colored lines represent the phases.

C. Election and Mutual Exclusion

Voting and mutual exclusion can be considered quite similar. Being a coordinator is like holding an exclusiv resource [3]. The problem of mutual exclusion has been discussed in many publication. For instance Lamport's distributed mutual exclusion algorithm can be used to elect leader [5]. Instead of entering a critical region, a node becomes leader. Should a node suspect the leader to have failed, it sends a message to all nodes dequeuing the leader from their queue.

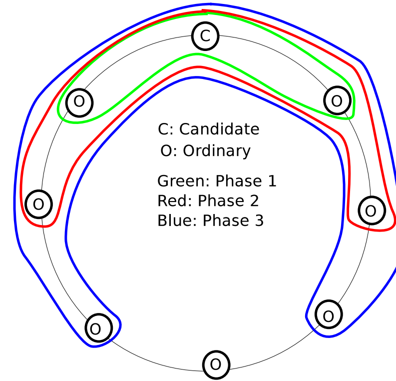


Figure 1. HS Phases

VI. Conclusion

This paper shows that voting in distributed systems synchron as well as asynchron is a well understood problem. The bully algorithm was discussed in detail to give an overview on how election algorithm work. The election algorithm by Afek and Gafni is presented to show how synchronous algorithm can be transfered into a asynchronous algorithm. To conclude voting algorithm on topologies are present.

References

[1] Yehuda Afek and Eli Gafni. Time and message bounds for election in synchronous and asynchronous complete networks. In *Proceedings of the fourth annual ACM symposium on Principles of distributed computing*, PODC '85, pages 186–195, New York, NY, USA, 1985. ACM.

[2] Ernest Chang and Rosemary Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Commun. ACM*, 22:281–283, May 1979.

[3] H. Garcia-Molina. Elections in a distributed computing system. *IEEE Trans. Comput.*, 31:48–59, January 1982.

[4] Vijay K. Garg, Ph.D. *Elements of distributed computing*. John Wiley & Sons, Inc., New York, NY, USA, 2002.

[5] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 5:1–11, January 1987.

[6] Valia Mitsou. Distributed leader election algorithms in synchronous networks. 2007.

[7] jholliday Steve Armstrong. Dc6: Chapter 12 coordination. 2001.

Globaler Zustand

Justin Philipp Heinermann
Carl-von-Ossietzky-Universität
Department für Informatik
Email: justin.heinermann@informatik.uni-oldenburg.de

***Zusammenfassung*—In verteilten Systemen besteht die Schwierigkeit, dass Prozesse während einer verteilten Berechnung nicht auf den globalen Zustand zugreifen können. Mit einem Schnappschussalgorithmus wird eine konsistente und nützliche Annäherung an einen tatsächlichen globalen Zustand berechnet. Diese Arbeit beschäftigt sich mit der Problemstellung der Berechnung eines konsistenten Schnappschusses und stellt den Schnappschuss-Algorithmus von Chandy und Lamport sowie dessen Anwendungen vor.**

***Keywords*-globaler Zustand, Schnappschuss, Konsistenter Schnitt, Schnappschussalgorithmus, stabile Prädikate, Chandy-Lamport-Algorithmus, observing global predicates, FIFO, happened before Relation, globales Debugging, verteiltes System**

I. EINLEITUNG

Eine der Schwierigkeiten in verteilten Systemen ist die Tatsache, dass kein Prozess ohne Weiteres auf den globalen Zustand des Systemes zugreifen kann. Die einfachste Möglichkeit, einen globalen Zustand zu gewinnen wäre es, die verteilte Berechnung anzuhalten. Es ist aber wünschenswert, in einer laufenden Berechnung auf den globalen Zustand zuzugreifen. Für viele Anwendungen ist es ausreichend, statt eines gegenwärtigen Zustandes einen Zustand aus der Vergangenheit zu gewinnen. Damit kann bspw. ein System nach einem Ausfall von einem solchen globalen Zustand als Checkpoint ausgehend die Berechnung weiterführen. Andere Beispiele für die Aussagekraft eines solchen vergangenen Zustandes sind die Erkennung von globalen Verklemmungen oder globalen Terminierungen [1]. Wenn diese sog. stabilen Prädikate einmal den Wahrheitswert $true$ angenommen haben, so behalten sie diesen. Daher ist in diesem Fall ein vergangener Zustand ausreichend. Eine solche Momentaufnahme eines vergangenen globalen Zustandes wird auch *Schnappschuss* (engl. „global snapshot") genannt [1]. Einen verteilten Algorithmus, der einen solchen Schnappschuss berechnet, nennt man *globalen Schnappschussalgorithmus*.

Die Schwierigkeit beim Erstellen eines Schnappschusses ist, dass ein solcher konsistent sein muss. Diese Problematik lässt sich mit einem Beispiel illustrieren, bei dem ein Foto des gesamten Himmels aufgenommen werden soll, um die Anzahl der Vögel am Himmel zu bestimmen. Der Himmel sei zu groß um ihn mit einem einzigen Bild darzustellen; daher werden mehrere Fotografien gemacht und zusammengeklebt. Analog zu einem verteilten System ohne eine perfekt synchronisierte Zeit können die Einzelbilder nicht zur exakt gleichen Zeit aufgenommen werden. Man nehme den ungünstigen Fall an, ein Vogel werde von der Kamera erfasst, bewege sich aber anschließend in ein anderes Bildsegement und werde erneut erfasst. In diesem Fall würde die ermittelte Anzahl verfälscht werden und das zusammengesetzte Gesamtbild die Aussagekraft verlieren [1].

Diese Arbeit beschäftigt sich mit globalen Schnappschussalgorithmen. Zunächst werden in Kapitel II die benötigten Grundlagen und Definitionen gegeben, mit denen die Problemstellung formell spezifiziert werden kann. Es wird eine logische, ereignisbasierte Halbordnung der Zustände verwendet, um konsistente Schnitte zu modellieren. Der erste korrekte Schnappschussalgorithmus wurde 1985 von Chandy und Lamport für FIFO-Nachrichtenkanäle angegeben. Dieser ursprüngliche Algorithmus ist Gegenstand von Kapitel III. Da der Algorithmus nur für begrenzte Anwendungsgebiete geeignet ist, wurde er auf vielfältige Weise erweitert. In Kapitel IV werden daher kurz weitere Herangehensweisen an das Schnappschussproblem betrachtet, die andere Annahmen für verschiedene Einsatzgebiete und Beschaffenheiten der Nachrichtenkanäle treffen als der ursprüngliche Schnappschussalgorithmus. Ein wichtiges Einsatzgebiet für Schnappschussalgorithmen ist das Erkennen stabiler Prädikate, welches in Kapitel V betrachtet wird. Im Gegensatz zu stabilen Prädikaten gibt es noch weitere globale Eigenschaften, die man in verteilten Systemen untersuchen möchte. Hier wird in Kapitel VI beispielhaft das verteilte Debugging vorgestellt. Abschließend wird in Kapitel VI eine Zusammenfassung gegeben.

II. GLOBALE ZUSTÄNDE UND KONSISTENZ

Es stellt sich die Frage, wie man den Begriff des gesuchten globalen Zustandes definieren soll. Grundsätzlich gibt es hierfür zwei Möglichkeiten: Die erste Möglichkeit ist es, ein auf der physischen Zeit basierendes Modell zu verwenden. Dies gestaltet sich in verteilten Systemen jedoch als schwierig, da keine perfekt synchronisierte Zeit zur Verfügung steht. Die andere Möglichkeit ist das

von Chandy und Lamport vorgestellte, ereignisbasierte „happenend before“-Modell [2], welches eine Halbordnung der Ereignisse darstellt. In diesem Modell lässt sich ein globaler Zustand als eine Menge von lokalen Zuständen definieren, die zueinander nebenläufig sind. Dies bedeutet, dass keine zwei lokalen Zustände der Menge eine „happened before“-Beziehung haben, also dass sie gleichzeitig auftreten. Ein Grund für die Verwendung diese Modells ist das Fehlen einer perfekt synchronisierten Zeit in verteilten Systemen. Außerdem bietet dieses abstraktere Modell den Vorteil, dass viele der interessanten Eigenschaften einfacher formuliert werden können als in einem zeitbasierten Modell [1]. Dies ist insbesondere deshalb nützlich, weil die Berechnungen in verteilten Systemen parallel und bzgl. der Aufführungsgeschwindigkeit und -reihenfolge nichtdeterministisch ablaufen [3]. Eine im „happened before“-Modell formulierte Eigenschaft trifft auf unterschiedliche tatsächliche zeitliche Reihenfolgen zu. In [1] und im Folgenden wird daher das happened before-Modell verwendet, welches sich für die Betrachtung von globalen Zuständen als gebräuchlich erwiesen hat.

Ein globaler Zustand besteht nicht nur aus der Menge der lokalen Zustände der Prozesse, sondern muss auch alle Nachrichten, die sich zwischen zwei Prozessen ausgeliefert werden, umfassen. Ein konkretes, vereinfachtes Beispiel (siehe [1]) für die Wichtigkeit der Berücksichtigung der Zustände der Nachrichtenkanäle ist die Verwaltung von zwei Bankkonten A und B, wie es in Abb. 1 dargestellt ist. Seien die anfänglichen Werte für die jeweiligen Guthaben der Konten \$500 und \$300. Wenn A eine Überweisung mit dem Betrag \$200 an B sendet, und anschließend ein Beobachter jeweils die lokalen Zustände von A und B anfordert, würde A den mit Kontostand mit \$300 angeben. Wenn die Überweisung zu dem Zeitpunkt der Zustandsanfrage noch nicht eingetroffen ist, würde B den Kontostand ebenfalls mit \$300 angeben. Der Beobachter würde also eine falsche Gesamtsumme von \$600 errechnen, obwohl insgesamt \$800 im Umlauf sind. Daher muss für die Richtigkeit des globalen Zustandes die auf dem Weg befindliche Überweisungsnachricht berücksichtigt werden.

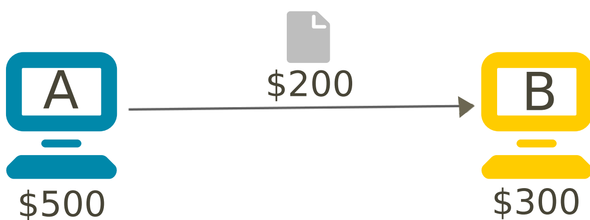


Abbildung 1. Beispiel für die Wichtigkeit des Zustandes der Nachrichtenkanäle für den globalen Zustand

Das Ziel eines Schnappschussalgorithmus ist es, einen konsistenten Schnappschuss zu berechnen. Um dieses Problem zu formalisieren, nutzt Garg in der formellen Beschreibung von Schnappschussalgorithmen [1] neben der verwendeten Halbordnung der lokalen Zustände den zentralen Begriff *konsistenter Schnitt*, der im Folgenden erläutert wird.

Die verwendete Ordnung der Ereignisse ordnet zwar die Ereignisse eines einzelnen Prozesses eindeutig. Aus globaler Sicht liegt aber nur eine Halbordnung (*poset*) vor, da die Reihenfolge der (evtl. parallelen) Ereignisse nicht eindeutig ist [1]. Dadurch ergibt sich das Problem, dass nicht alle möglichen Ereignisreihenfolgen einer validen verteilten Berechnung entsprechen: Es sind Zyklen zwischen Ereignissen möglich und es kann dann keine eindeutige Ereignisreihenfolge bestimmt werden. Daher verwendet Garg [1] ein *deposet* (decomposed partially ordered set): Die Halbordnung der Ereignisse wird insofern eingeschränkt, als zwischen zwei aufeinanderfolgenden Zuständen nur ein *send*- oder *receive*-Ereignis auftreten kann.

Sei S die oben beschriebene Halbordnung der Ereignisse und S_i die Ereignissequenz für einen Prozess P_i. Die Begriffe *Schnitt* und *konsistenter Schnitt* sind wie folgt definiert [1].

Definition (Schnitt) Ein *Schnitt* ist eine Untermenge von S, die von jeder Ereignissequenz S_i zu jedem Prozess P_i genau einen Zustand enthält.

Definition (Konsistenter Schnitt) Eine Untermenge $G \subset S$ ist ein *konsistenter Schnitt*, genau dann wenn $\forall s, t \in G : s \parallel t \wedge |G| = N$.

Die Schreibweise $s \parallel t$ drückt dabei aus, dass die Zustände s und t nebenläufig sind, $|G| = N$ bedeutet, dass ein lokaler Zustand von jedem Prozess im Schnitt enthalten sein muss. Im Berechnungsmodell liegt ein Schnitt G „vor“ einem Schnitt H, wenn für jeden Prozess der jeweilige Zustand in G zeitlich vor dem entsprechenden Zustand in H liegt. In Abb. 2 ist ein Zeitdiagramm mit einem Beispiel für einen Schnitt dargestellt, der die Ereignisse in Vergangenheit und Zukunft teilt.

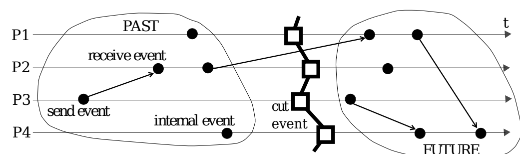


Abbildung 2. Beispiel für einen Schnitt [3]

Ein konsistener Schnitt kann auch als *konsistenter globaler Zustand* bezeichnet werden. In Abb. 3 sind Beispiele für einen konsistenten und einen inkonsistenten Schnitt gegeben: Während C nur nebenläufige Zustände enthält und damit konsistent ist, hat C' zwei nicht nebenläufige Zustände. Diese erkennt man an der Nachricht, die nach dem Schnitt versendet wird, aber vor dem Schnitt ankommt. Eine solche Sequenz führt zu einer Inkonsistenz.

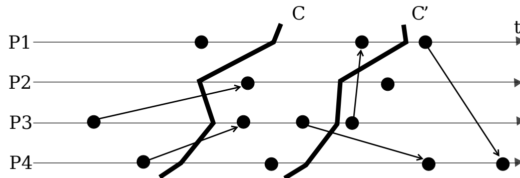


Abbildung 3. Konsistenter Schnitt C (links)und nichtkonsistenter Schnitt C' (rechts) [3]

Um einen globalen Zustand zu gewinnen, wird ein verteilter Algorithmus benötigt, der das oben beschriebene Konsistenz-Kriterium erfüllt, sodass nur nebenläufige Zustände und auf dem Weg befindliche Nachrichten enthalten sind.

III. Der globale Schnappschussalgorithmus von Chandy und Lamport

In [1] wird ein Algorithmus beschrieben, um einen globalen Schnappschuss eines verteilten Systems aufzunehmen. Dieser beruht auf dem 1985 von Chandy und Lamport [4] veröffentlichten Schnappschussalgorithmus, der die erste korrekte Lösung dieses Problems darstellt [3]. Der Algorithmus wurde von Dijkstra abgewandelt, indem die im Folgenden verwendete, leicht verständliche Beschreibung mit Farben genutzt wird [5]. Die Berechnung eines konsistenten Schnittes kann von einem oder mehreren Prozessen initiiert werden. Voraussetzung ist hierbei, dass alle Nachrichtenkanäle unidirektional sind und die FIFO[1]-Eigenschaft erfüllen. Durch diese Eigenschaft ist es einfacher, eine Lösung zu finden: Die Reihenfolge der Nachrichten ist eindeutig bestimmt.

Die Idee des Algorithmus, der in Listing 1 als Pseudocode dargestellt ist, ist es, dass jeder Prozess als Eigenschaft eine *Farbe* hat, die entweder weiß oder rot ist. Anfangs sind alle Prozesse weiß; wenn ein Prozess für die Berechnung des Schnappschusses seinen lokalen Zustand aufgenommen hat, nimmt er die Farbe rot an. Der globale Zustand, der durch einen globalen Schnappschuss dargestellt wird, entspricht den lokalen Zuständen, bevor diese rot werden sowie den Zuständen der Nachrichtenkanäle.

```
Pi::
var
  color: {white, red} initially white;
  //assume k incoming channels
  chan: array[1..k] of queues of messages initially
       null;
  closed: array[1..k] of boolean initially false;

turn_red() enabled if (color=white)
  save_local_state;
  color := red;
  send (marker) to all neighbors;

Upon receive(marker) on incoming channel j:
  if(color=white) then
    turn_red();
  closed[j]:=true;

Upon receive(program_message) on incoming channel j:
  if(color=red) ∧ ¬closed[j] then
    //append the message
    chan[j] := append(chan[j], program_message);
```

Listing 1. Chandy-Lamport-Algorithmus aus [1]

Die Regeln für die Farbänderung müssen zwei Eigenschaften erfüllen:

1) Die gespeicherten Zustände müssen nebenläufig sein.
2) Es wird ein Mechanismus benötigt um den Zustand der Nachrichtenkanäle aufzunehmen

Im Algorithmus von Chandy und Lamport kommt hierfür eine spezielle *marker*-Nachricht zum Einsatz [1]. Wenn ein Prozess die Farbe rot annimmt, muss er über alle seine ausgehenden Nachrichtenkanäle eine *marker*-Nachricht senden. Wenn ein Prozess eine *marker*-Nachricht empfängt, so nimmt er die Farbe rot an. Dadurch, dass alle Nachrichtenkanäle nach dem FIFO-Prinzip arbeiten, ist gewährleistet, dass kein weißer Prozess eine Nachricht von einem roten Prozess erhalten kann. Somit sind alle aufgenommenen lokalen Zustände nebenläufig.

Für den Zustand der Kanäle müssen vier Typen von Nachrichten berücksichtigt werden:

1) Eine Nachricht von einem weißen Prozess an einen anderen weißen Prozess wurde vor dem Schnappschuss gesendet und empfangen.
2) Eine Nachricht von einem roten Prozess an einen anderen roten Prozess wurde nach dem Schnappschuss gesendet und empfangen.
3) Eine Nachricht von einem roten an einen weißen Prozess macht den Schnappschuss inkonsistent und sollte verhindert werden.
4) Eine Nachricht von einem weißen an einen roten Prozess wird zum Zustand des Nachrichtenkanals während des Schnappschusses gerechnet.

Um den Zustand eines Nachrichtenkanals aufzunehmen, speichert ein Prozess P_j alle Nachrichten, die er von einem

[1] First In, First Out

Prozess P_i erhält, nachdem P_j rot wurde. Da P_i, wenn er rot wird, einen *marker* an P_j sendet, können nach diesem marker keine weißen Nachrichten von P_i an P_j mehr gesendet werden.

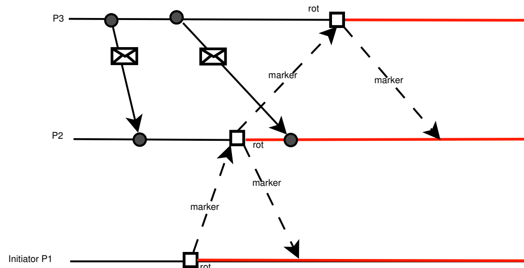


Abbildung 4. Beispiel für die Ausführung des Schnappschuss-Algorithmus

Der Algorithmus terminiert, wenn jeder Prozess eine *marker*-Nachricht auf allen eingehenden Kanälen empfangen hat. Anschließend gibt es verschiedene Möglichkeiten, den globalen Zustand aus den lokal gespeicherten Zuständen zu errechnen [6]: Eine Möglichkeit ist es, dass jeder Prozess seinen lokalen Schnappschuss an den Initiator des Algorithmus sendet. Eine Alternative ist das Versenden der gespeicherten lokalen Schnappschüsse über alle ausgehenden Nachrichtenkanäle. Jeder Prozess, der diese Informationen zum ersten Mal erhält, sendet diese über die ausgehenden Kanäle weiter an seine Nachbarn. So erhalten alle Prozesse Zugriff auf die aufgenommenen Zustände und können den globalen Zustand ermitteln [6].

Es stellt sich die Frage, ob der Algorithmus korrekt ist, d.h. ob er einen korrekten, konsistenten globalen Zustand liefert. Um die Korrektheit des Algorithmus zu zeigen, werden bei Kshemkalyani [6] zwei Eigenschaften bewiesen:

C1 Jede Nachricht m_{ij}, die von einem Prozess P_i als gesendet in den lokalen Zustand gespeichert wird, muss entweder im Zustand des Nachrichtenkanals von P_i zum Empfänger P_j gespeichert sein oder im lokalen Zustand von P_j selbst.

C2 Für jedes Ereignis im Schnappschuss muss auch die Ursache im Schnappschuss liegen. Eine Nachricht, die nicht als gesendet gespeichert wurde, darf auch nicht als empfangen oder unterwegs verzeichnet sein.

Wie im Folgenden gezeigt wird, erfüllt der oben beschriebene Algorithmus beide Bedingungen:

Eigenschaft C1: Wenn ein Prozess P_j eine Nachricht m_{ij} empfängt, die auf dem Nachrichtenkanal C_{ij} der *marker* vorangeht, reagiert er folgendermaßen: Falls er seinen lokalen Zustand noch nicht abgespeichert hat, schließt er m_{ij} mit in diesen ein. Hat P_j bereits auf einem anderen Kanal einen *marker* erhalten, speichert er die Nachricht m_{ij} in den Zustand von C_{ij} ab.

Eigenschaft C2: Da ein Prozess seinen lokalen Zustand in dem Moment abspeichert, wenn er die erste *marker*-Nachricht auf einem eingehenden Nachrichtenkanal erhält, werden keine dem *marker* nachfolgenden Nachrichten mehr gespeichert. Der Zustand des Nachrichtenkanales wird beim Eintreffen eines *markers* auf diesem Kanal gespeichert. Wegen der FIFO-Eigenschaft der Kanäle folgt, dass keine Nachricht nach dem *marker* mehr in den Zustand des Nachrichtenkanals gespeichert wird.

Die Nachrichtenkomplexität des Algorithmus hängt von der Anzahl E der unidirektionalen Kommunikationskanäle ab und liegt somit in der Ordnung $O(E)$. Die Betrachtung des Algorithmus zeigt, dass durch die Annahme der FIFO-Eigenschaft für die Kanäle die Lösung des Problems relativ einfach ist.

IV. Andere Herangehensweisen

Der in Kapitel III vorgestellte Chandy-Lamport-Algorithmus hat einige Einschränkungen: Er funktioniert nur mit FIFO-Nachrichtenkanälen und liefert keine optimale Effizienz. Des Weiteren kann der Algorithmus für den Fall, dass mehrere Prozesse gleichzeitig einen Schnappschuss anfordern, verbessert werden. Es wurden viele Varianten und Verbesserungen des ursprünglichen Algorithmus entwickelt, die diesen für bestimmte Eigenschaften optimieren oder für andere Annahmen abändern. Von diesen sollen im Folgenden einige wichtige kurz beschrieben werden. Eine Übersicht findet sich in [6].

Spezialetti und Kearns [7] haben den ursprünglichen Chandy-Lamport-Algorithmus für die mehrfache Initiierung optimiert: Wenn mehrere Prozesse den Schnappschussalgorithmus initiieren, so werden die einzelnen Schnappschüsse zu einem einzigen vereint und effizienter an die Initiatoren zurückgesendet. Die *marker*-Nachricht wird hier jeweils um eine Identifizierung des Initiiators erweitert. Es werden bidirektionale FIFO-Nachrichtenkanäle vorausgesetzt.

Für viele Anwendungen ist es sinnvoll, wiederholt einen Schnappschuss des verteilten Systems zu berechnen. Dazu kann der Chandy-Lamport-Algoritmus wiederholt initiiert werden, was aber einen hohen Nachrichtenoverhead zur Folge hat. Beispielsweise bei der Erstellung von Checkpoints für die verteilte Berechnung ist das Verfahren von Venkatesan [8] sinnvoller: Schnappschüsse werden inkrementell aktualisiert und nur Unterschiede übertragen. Dadurch müssen weniger Nachrichten versendet.

Diesen drei Ansätzen ist gemein, dass Sie nur auf FIFO-Kanälen arbeiten. Dies vereinfacht die Problemstellung, da

die Reihenfolge der Nachrichten eindeutig bestimmt ist. Lai und Yang [9] geben aufbauend auf dem Chandy-Lamport-Algorithmus eine Variante für Nicht-FIFO-Kanäle an. Dabei werden auch die Nachrichten der verteilten Berechnung weiß bzw. rot eingefärbt. Eine rot eingefärbte Nachricht dient somit als impliziter Ersatz für die *marker*-Nachricht. Für die Berechnung der Zustände der Nachrichtenkanäle wird eine Nachrichtenhistorie benötigt. Ein weiterer Ansatz für Nicht-FIFO-Kanäle wurde von Mattern [3] vorgestellt. Dieser ähnelt dem Ansatz von Lai und Yang, benötigt aber keine Nachrichtenhistorien, sondern einen Zähler für jeden Nachrichtenkanal. Beim Vergleich der Algorithmen fällt auf, dass das Verfahren von Lai und Yang wegen der Nachrichtenhistorien mehr Speicher benötigt, das von Mattern dafür aber länger benötigt bis es terminiert [6].

Aus diesen Beispielen wird ersichtlich, dass ein Schnappschussalgorithmus immer für die jeweiligen Anwendungsbedingungen abgestimmt sein muss.

V. Erkennen stabiler Prädikate

Eine wichtige Anwendung für die Berechnung eines globalen Schnappschusses ist das Erkennen von *stabilen Prädikaten* [1]. Ein stabiles Prädikat ist eine Aussage über den Zustand des Systemes, welche als monotone boolsche Funktionen angegeben werden kann; nimmt diese den Wahrheitswert $wahr$ an, bleibt sie in jedem folgenden Zeitpunkt ebenfalls wahr [3]. Stabile Prädikate, die sich mit dem Schnappschussalgortihmus erkennen lassen, sind beispielsweise [10]:

- **Terminierung:** Sei γ eine terminierte Konfiguration der verteilten Berechnung und δ eine nachfolgende Konfiguration. Dann ist $\delta = \gamma$ ebenfalls eine terminierte Konfiguration. Zur Erkennung der Terminierung kann ein globaler Schnappschuss berechnet werden und an den lokalen Zuständen der Prozesse abgelesen werden, ob die einzelnen Prozesse terminiert sind.
- **Verklemmung:** Wenn in einer Konfiguration γ einer verteilten Berechnung eine Teilmenge von Prozessen blockiert ist, weil ein Deadlock vorliegt, wird dies sich ohne einen Eingriff von außen auch in der Zukunft nicht ändern. Daher liegt auch hier ein stabiles Prädikat vor.
- **Tokenverlust:** Man nehme ein verteiltes System an, in dem Token weitergereicht und verbraucht werden können, jedoch nicht neu erzeugt werden. Dann ist die Eigenschaft „Es existieren höchstens k Token“ stabil.
- **Garbage Collection:** In einer objektorientierten Programmierumgebung kann eine Vielzahl von Objekten existieren. Dies halten wiederum Referenzen auf andere Objekte. Ein Objekt ist nur dann erreichbar, wenn eine Referenz darauf existiert. Ist ein Objekt nicht erreichbar, wird es als *Garbage* bezeichnet. Da auf ein solches Garbage-Objekt nie wieder eine Referenz existieren wird, lässt sich diese Eigenschaft als stabiles Prädikat ausdrücken.

An einen solchen Schnappschussalgorithmus zur Erkennung von stabilen Prädikaten gibt es zwei wesentliche Anforderungen [10]:

1) Wenn der Algorithmus ein stabiles Prädikat erkennt, gilt es zum aktuellen Zeitpunkt ebenfalls (safety)
2) Wenn das stabile Prädikat gilt, wird eine im aktuellen Zustand initiierte Ausführung des Schnappschussalgorithmus dies in endlicher Zeit feststellen (liveness)

Ein einfacher Algorithmus zur Erkennung eines stabilen Prädikates kann wie folgt angegeben werden [1]:

1) Berechne einen globalen Schnappschuss G
2) Überprüfe, ob die stabile Eigenschaft B in Zustand G erfüllt ist. Andernfalls: wiederhole dem Algorithmus nach einer gewissen Zeit.

Es lassen sich einige Einschränkungen beim Ermitteln von globalen Prädikaten mittels des Schnappschussalgorithmus feststellen [1]: Der Algorithmus ist nicht nutzbar für nicht-stabile Prädikate, da ein solches sich zwischen zwei Schnappschüssen ändern kann. Für viele Anwendungen ist es außerdem wünschenswert, den ersten Zustand zu ermitteln, der ein gegebenes Prädikat erfüllt. Bei Einsatz des Schnappschussalgorithmus von Chandy und Lamport wird unter Umständen das Nachrichtenaufkommen sehr groß, was bei oft wiederholten Ausführungen des Algorithmus ungünstig ist. In vielen Fällen lässt sich ein Problem, wie z.B. das der Terminierung effizienter ohne den Schnappschussalgorithmus lösen.

VI. Verteiltes Debugging

Im Gegensatz zu stabilen Prädikaten gibt es Eigenschaften, die nur kurzzeitig in einer Berechnung auftreten. Wenn für eine verteilte Berechnung *Debugging* durchgeführt werden soll, wird eine Aussage darüber benötigt, ob eine solche Eigenschaft jemals aufgetreten ist. Ein Beispiel hierfür ist verteiltes System, welches eine Fabrik steuert. Soll hier der Fall beobachtet werden, ob alle Ventile zu einem Zeitpunkt gleichzeitig geöffnet waren, kann diese Eigenschaft nicht mit dem Schnappschussalgorithmus für stabile Prädikate beobachtet werden. Ein anderes Beispiel für eine derartige Zusicherung ist die Anforderung $|x_i - x_j| < \delta$, wobei die Variablen x_i und x_j zu unterschiedlichen Prozessen gehören [11].

Die Herausforderung ist hier eine andere als bei stabilen Prädikaten: Es gilt hier, das System über einen Zeitraum hinweg zu überwachen und in Hinblick auf eine eventuell verletzte *safety*-Eigenschaft.

In [11] wird ein Algorithmus beschrieben, der derartige Informationen zentralisiert sammelt. Dieser entspricht dem Algorithmus von Marzullo und Neiger von 1991 [12]. Die

Prozesse des verteilten Systems senden ihre Informationen an einen *Monitor*-Prozess, der aus den Informationen jeweils einen konsistenten globalen Zustand errechnet. Dieser liegt außerhalb des Systems der eigentlichen Berechnung und fungiert nur als Beobachter. Ziel ist es nun, festzustellen, ob ein Prädikat ϕ in einer verteilten Berechnung zu einem Zeitpunkt tatsächlich $wahr$ war. Für nicht-stabile Prädikate kann dies nicht an einem einzelnen Schnappschuss abgelesen werden. In vielen Fällen ist es aber erstrebenswert, herauszufinden, ob ein Prädikat *möglicherweise* in der tatsächlichen Berechnung aufgetreten ist. Für den Algorithmus werden daher mit der Historie H der Berechnung zwei Notationen verwendet:

- **possibly** ϕ**:** Es existiert ein konsistenter Zustand S, für den in einer Linearisierung von H das Prädikat $\phi(S)$ gilt
- **definitely** ϕ**:** Für alle Linearisierungen L von H gilt, es existiert ein konsistenter Zustand S, sodass $\phi(S)$.

Der Monitor-Prozess sammelt die Zustands-Informationen, indem jeder andere Prozess ihm anfangs seinen initialen Zustand sendet. Anschließend senden alle Prozess von Zeit zu Zeit Statusmeldungen, die jeweils einen Zeitstempel enthalten. Für jeden Prozess wird der Zustand in eine Warteschlange abgespeichert. Das Versenden der Statusnachrichten findet verzögert zur eigentlichen Berechnung statt, beeinflusst diese selbst aber nicht.

Die Auswertung der lokalen Zustände durch den Monitor-Prozess führt nicht nur zu einem einzelnen konsistenten Schnappschuss, sondern zu einem Verbund aller möglichen konsistenten Schnappschüsse. Auf diese Weise kann die Gültigkeit eines nicht-stabilen Prädikates nach den Kriterien *possibly* ϕ und *definitely* ϕ gezeigt werden.

Der Nachteil an diesem Verfahren ist die hohe Komplexität. Allerdings müssen Statusmitteilungen auch nur dann an den Monitor-Prozess übermittelt werden, wenn eine Änderung des Zustandes vorliegt. Daher kann der Algorithmus in Systemen zum Einsatz kommen, in denen nur wenige Ereignisse eine Änderung des zu überprüfenden globalen Prädikates zur Folge haben [11].

VII. Fazit

In dieser Arbeit wurde die für verteilte Systeme zentrale Problemstellung beschrieben, dass Prozesse keinen Zugriff auf den globalen Zustand der Berechnung haben. Das Problem wurde mit den in diesem Bereich üblichen Definitionen formalisiert. Die grundsätzliche Lösung konnte für FIFO-Nachrichtenkanäle mit dem Schnappschussalgorithmus von Chandy und Lamport gezeigt werden. Da dieser jedoch nur für bestimmte Einsatzzwecke und Annahmen verwendet werden kann, wurden einige wichtige Erweiterungen des ursprünglichen Algorithmus beschrieben. Hierbei können bspw. eine höhere Effizienz gewährleistet werden oder auch Nicht-FIFO-Nachrichtenkanäle zum Einsatz kommen.

Ein wichtiger Anwendungsfall für Schnappschussalgorithmen ist die Erkennung von stabilen Prädikaten wie bspw. verteilte Deadlocks oder Terminierung. Aber auch nicht-stabile Prädikate können mit einer abgewandelten Form des Algorithmus beobachtet werden und so globales Debugging betrieben werden. Insgesamt ist der Einsatz eines Schnappschussalgorithmus stark von dem jeweiligen Anwendungszweck anhängig, sodass von Fall zu Fall unterschieden werden muss, welche Variante verwendet wird.

Literatur

[1] V. K. Garg, Ph.D., *Elements of distributed computing*. New York, NY, USA: John Wiley & Sons, Inc., 2002.

[2] L. Lamport, "Ti clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, pp. 558–565, July 1978. [Online]. Available: http://doi.acm.org/10.1145/359545.359563

[3] F. Mattern, "Efficient algorithms for distributed snapshots and global virtual time approximation," *Journal of Parallel and Distributed Computing*, vol. 18, pp. 423–434, 1993.

[4] K. M. Chandy and L. Lamport, "Distributed snapshots: determining global states of distributed systems," *ACM Trans. Comput. Syst.*, vol. 3, pp. 63–75, February 1985. [Online]. Available: http://doi.acm.org/10.1145/214451.214456

[5] V. K. Garg, *Concurrent and Distributed Computing in Java*. John Wiley & Sons, 2004, ch. 9.

[6] A. D. Kshemkalyani and M. Singhal, *Distributed Computing: Principles, Algorithms, and Systems*, 1st ed. Cambridge University Press, May 2008, ch. 4. [Online]. Available: http://www.worldcat.org/isbn/0521876346

[7] M. Spezialetti and P. Kearns, "Efficient distributed snapshots," in *Proceedings of the 6th International Conference on Distributed Computing Systems*, 1986, pp. 382–388.

[8] S. Venkatesan, "Message-optimal incremental snapshots," in *ICDCS*, 1989, pp. 53–60.

[9] T. H. Lai and T. H. Yang, "On distributed snapshots," *Inf. Process. Lett.*, vol. 25, pp. 153–158, May 1987. [Online]. Available: http://dx.doi.org/10.1016/0020-0190(87)90125-6

[10] G. Tel, *Introduction to Distributed Algorithms*, 2nd ed. New York, NY, USA: Cambridge University Press, 2001, ch. 10.

[11] G. Coulouris, J. Dollimore, and T. Kindberg, *Distributed systems: concepts and design*. Addison-Wesley, 2005. [Online]. Available: http://books.google.de/books?id=d63sQPvBezgC

[12] K. Marzullo and G. Neiger, "Detection of global state predicates." in *Proceedings 5th International Workshop On Distributed Algorithms*, S. Toueg, P. G. Spirakis, and L. M. Kirousis, Eds. Springer, 1991, pp. 254–272.

Betrachten globaler Prädikate

Helge Arjangui
Carl-von-Ossietzky Universität Oldenburg
Fakultät II, Department für Informatik
helge.arjangui@uni-oldenburg.de

***Abstract*—Die folgende Ausarbeitung befasst sich mit der Beobachtung globaler Prädikate in verteilten Berechnungen. Dazu müssen zunächst Annahmen getroffen werden, unter denen die Beobachtungen stattfinden. Dabei wird auf die Kernproblematiken, die beim Beobachten globaler Eigenschaften auftreten, eingegangen. Nachdem zwei verschiedene Arten von Prädikaten vorgestellt wurden, gibt es abschliessend ein Fazit sowie Ausblick.**

***Keywords*-Prädikat; global; beobachten; Kernproblematiken;**

I. Einleitung

Verteilte Systeme sind allgegenwärtig. Als Beispiel anzuführen wären Bankautomaten die innerhalb einer Stadt verteilt sind. Hierbei muss es möglich sein, dass einer Person an jedem Automaten der aktuell korrekte Kontostand angezeigt wird. Daher ist die Zuverlässigkeit solcher Systeme sehr wichtig. Desweiteren sind die Vorteile eines verteilten Systems vielfältig. Sie reichen über Ausfallsicherheit, geringe Fehleranfälligkeit bis zur Lastverteilung von Berechnungen. Dabei besitzt ein verteiltes System jedoch folgende drei Hauptproblematiken:

- Keine gemeinsame Uhr
- Kein gemeinsamer Speicher
- Multiple Prozesse

Diese werden im Rahmen der Ausarbeitung näher vorgestellt. Aufgrund dieser Probleme, kann bei einem laufenden verteilten System nie genau gesagt werden kann, in welchem Zustand es sich gerade befindet. In einem Prozess finden Ereignisse statt. Diese Versenden und Empfangen Nachrichten untereinander. "'Globalen Zustände bestehen dabei aus den einzelnen lokalen Prozesszuständen sowie allen Kanalzuständen"' [4]. Folgende Abbildung 1 verdeutlicht dabei die Problematik:

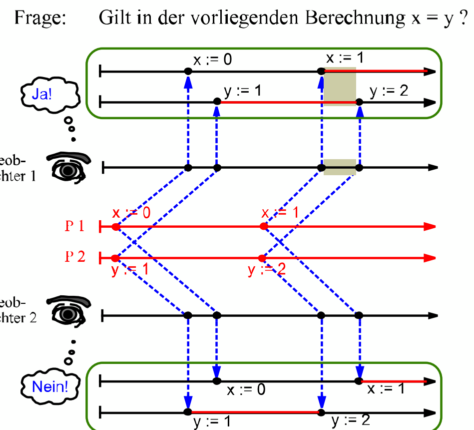


Figure 1. Probleme globaler Prädikate- Zustnde werden[4]

Die Problematik die bei dieser Abbildung dargestellt wird, ist das es je nach Betrachtungsweise eines Beobachters das System unterschiedliche globale Eigenschaften besitzt. Die Variablen x und y werden von zwei Prozessen P_1 und P_2 zu bestimmten Zeitpunkten gesetzt. Für den ersten Beobachter gilt die Berechnung von x=y, da er im "'richtigen"' Moment das System betrachtet. Für ihn gilt also die Aussage (das globale Prädikat) das der Wert von x gleich dem Wert von y ist. Für den zweiten Beobachter ist dies nicht so. Zu den Zeitpunkten, andenen er sich die Werte von x und y betrachtet, gilt nie x=y. Desweiteren lassen sich globale Zustände in konsistene und nicht konsistente unterteilen. Hierbei wird von einem *konsistenten*, bzw. *inkonsistenten* Schnitt gesprochen. Bei einem konsistenten Schnitt ist der *Empfang* einer Nachricht und somit auch das *Senden* dieser, enthalten. Die Richtung, in welche die Nachricht gesendet wird, ist durch eine entsprechende Pfeilrichtung dargestellt. Die folgende Abbildung 2 verdeutlicht dies:

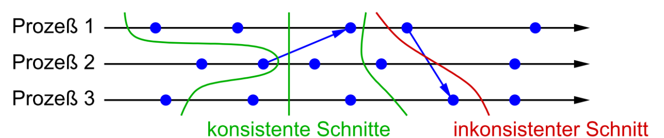


Figure 2. Konsistenter Schnitt[4]

Die rote Linie zeigt einen inkonsistenten Schnitt, da hier nur der Empfang aber nicht das Senden der Nachricht enthalten ist. Ein globales Prädikat beschreibt eine Eigenschaft eines verteilten Systems[**?**, mattern] In unserem Fall handelt es sich bei dabei um eine boolsche Funktionen, die den Wert *True* bzw. *False* annehmen kann. Zunächst wird in Tabelle I eine Übersicht über die verwendete Notation gegeben.

Table I
VERWENDETE NOTATION

S	Verteilte Berechnung
C(S)	Verband (Netz) aus konsistenten globalen Zuständen
G,H	Konsistente globale Zustände
B	Globales boolsches Prädikat
m	Anzahl der Zustandsintervalle pro Prozess
N	Anzahl der Prozesse
e	lokales Ereignis eines Prozesses

Desweiteren werden die Kernprobleme der Beobachtung globaler Prädikate dargestellt, und auf die einzelnen Prädikatstypen eingegangen. Unterschieden wird hier zwischen Linearen, Regulären sowie Konjunktiven Prädikaten. Im nächsten Abschnitt werden einschränkende Annahmen dargestellt, unter denen das Beobachten globaler Prädikate erfolgt.

II. ANNAHMEN DER BEOBACHTUNG

Bevor nun direkt auf die Beobachtung globaler Prädikate und dessen Problematiken eingegangen wird, müssen zur Eingrenzung einige Annahmen getroffen werden. Jeder Prozess P_i besitzt ein lokales Prädikat und kann dies auch selbstständig entdecken. Dieses wird als l_i bezeichnet. Dabei ist jedes lokale Prädikate definiert als jede beliebige bool'sche Eigenschaft eines lokalen Zustandes $S_i \in S$. Während einer lokalen Ausführung eines Prozesses definiert jeder Zustand $s \in S_i$ (mit **S** = $\{S_1, S_2, ..., S_N\}$), einen Wert für jede zugehörige Variable $x \in X_i$. Jedes Zustandspaar $s \in S_i$ und $t \in S_j$ definieren eine Nachrichtenqueue für den Kanal C_{ij} von Prozess P_i nach Prozess P_j.
Abbildung 3 zeigt den Zusammenhang zwischen Prozessen, Nachrichten und dem Nachrichtenkanal C_{ij}:

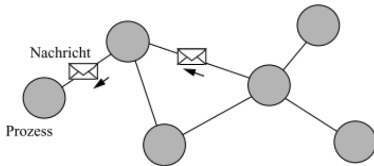


Figure 3. Nachrichten, die auf einem Kanal C_{ij} zwischen Prozessen verschickt werden[4]

Ein globales Prädikat ist nun eine booleanwertige Funktion **B** der Variablen in X_i und der Nachrichten in C_{ij}. B(G) bezeichnet den Wert des Prädikates B in einem Schnitt G=$\{s_1, s_2, ..., s_N\}$. Ein Schnitt ist dabei wie folgt definiert: "Ein Schnitt G ist Menge von Ereignissen, so dass bei jedem Prozess gilt: Falls ein Empfang einer Nachricht e' im Schnitt liegt, dann liegt auch das entsprechende Sendeereignis e im Schnitt"'[2]. Das Nachfolgeevent e' hängt dabei kausal von seinem Vorgängerevent e ab.

$$e \in G \wedge e' <_k e \Rightarrow e' \in G \quad (1)$$

Alle in der Teilmenge G befindlichen Zustände sind konsistent. Demnach ist ein globaler Zustand konsistent, wenn er genau die Ereignisse in einem konsistenten Schnitt berücksichtigt[2].
Globale Prädikate können nur über konsistente Zuständen sinnvoll bestimmt werden[4].
Das Ergebnis von B(G) für einen gegebenen Schnitt G definiert den Wert des globalen Prädikats mittels *True* bzw. *False*. Anstelle des konsistenten Schnittes eines Systems G, kann der bool'schen Funktion B aber auch eine Berechnung übergeben werden. Für diesen Fall wird die gegebene Definition von B(G) um die Berechnung erweitert. Unter einer Berechnung versteht man in dem Zusammenhang zwei Dinge: Zum einen wird so eine Menge zusammengehöriger Events in dem System bezeichnet, oder als Menge der lokalen Zustände des Systems bezeichnet werden. In userem Zusammenhang stützen wir uns auf letzteres. Für die Definition von B bietet es sich an, die von der Berechnung generierten konsistenten Schnitte zu beachten. C(S) bezeichnet den Verband aller globalen konsistenten Zustände aus S. Für ein globales Prädikat B ergeben sich dabei zwei Beobachtungsmöglichkeiten:

Definition: possibly:B
Ein globales Prädikat B ist möglicherweise (possibly) *True*, wenn es für einen Pfad in C(S) einen Zustand gibt, indem B den Wert *True* annimmt.

Definition: definitly:B
Ein globales Prädikat B ist definitv *True*, wenn es für jeden Pfad in C(S) einen Zustand gibt, indem B den Wert *True* annimmt.

Erweiternd zu *possibly* und *definitly* werden die Definitionen der *unveränderlichkeit*, sowie die der *kontrollierbarkeit* eingeführt.

Definition: invariant:B
invariant: B $\stackrel{\text{def}}{=}$ ¬*possibly* : ¬B Ein globales Prädikat B ist für alle Zustände unveränderbar(invariant), wenn es in allen konsistenen Schnitten G des Verbandes C(S), wahr ist.

Definition: controllable:B
kontrollierbar: B $\stackrel{\text{def}}{=}$ ¬*definitly* : ¬B
Ein globales Prädikat B ist kontrollierbar (controllable), wenn es einen Pfad vom initialen konsistenten Schnitt, bis zum finalen konsistenten Schnitt in C(S) gibt, indem B jederzeit, also für jeden dazischenliegenden Zustand, wahr ist.

Da in verteilten Systemen die erste Definition (possibly) am interessantesten ist, wird hier auch der Fokus gesetzt. Nach der Definition lässt sich folgendes Lemma schliessen:

Lemma 1 possibly: B für eine beliebige Berechnung S: existiert ein konsistenter Schnitt G ∈ C(S), dann ist B wahr in G.

Folglich ist es nicht nötig zu ermitteln ob ein globales Prädikat B möglicherweise whärend einer Berechnung wahr wird, sondern reicht es aus zu bestimmen, ob es einen konsistenten Schnitt gibt, indem es wahr ist. Das Problem hierbei hierbei liegt darin, dass es eine sehr große Anzahl von konsistenten Schnitten in S gibt. Im worst case handelt es sich dabei um m^N(vgl. Tabelle I) konsistente Schnitte insgesamt in S, sofern jede Berechnung S_i aus m Zuständen besteht.

Nach den vorgestellten Annahmen, wird auf die Kernproblematiken eingegangen, denen globale Prädikate unterliegen.

III. Kernprobleme in der Beobachtung globaler Eigenschaften

Im Wesentlichen stellen sich drei Probleme dar, die beim Beobachten globaler Prädikate auftreten können. Dies sind gleichzeitig auch die Schlüsselproblematiken die jedes verteilte System mit sich bringt. Grund für diese Schwierigkeiten sind (neben dem inhärenten Nichtdeterminismus) das Fehlen einer exakten globalen Zeit sowie das partielle Fehlverhalten von Komponenten im System. Stellt man sich nun ein beliebiges Programm vor, das seine Berechnungen durch die Verteilung von Teilberechnungen auf verschiedene Prozesse realisiert, kann dies als ein verteiltes System angenommen werden. Nun soll das Programm seine Berechnungen beispielsweise abbrechen, sofern das globale Prädikat B den Wert *True* annimmt. Um dies zu gewährleisten, muss nun aber das Prädikat B ermittelt werden. Folgenden Tabelle II stellt die Charakteristik eines verteilten Systems, das damit zusammenhängende Problem, sowie eines entsprechenden Lösungsansatzes, vor.

Table II
Probleme und Lösungen beim Beobachten globaler Prädikate

Charakteristik	Problematik	Idee
Keine gemeinsame Uhr	Ereignisse sortieren	Happened before(Bereits eingetreten)
Kein gemeinsamer Speicher	Nachrichten/ Zustandsänderungen	Monotonie (=Stabile Prädiakte)
Multiple Prozesse	Kombinatorische Explosion	Linearität

Die nächsten Abschnitte stellen die Probleme genauer vor.

A. Fehlen von gemeinsam genutzter Uhren

Das globale Prädikat B setzt sich aus allen lokalen Prädikaten zusammen. Stellt man sich zwei Prozesse P_1 und P_2, mit zugehörigen lokalen Prädikaten CS_1 sowie CS_2 vor, die beide den Wert *True* besitzen, so ist auch das globale Prädikat B *True*, da gilt:

$$B = CS_1 \wedge CS_2 \quad (2)$$

Dies kann aber nur dann Gültigkeit haben, wenn beide *lokale* Prädikate (CS_1 sowie CS_2), *gleichzeitig* den Wert True besitzen. Soweit funktioniert dies auch - zumindest in sequentiellen Systemen. In verteilten Systemen hingegen ist dies nicht so einfach möglich, da jeder Prozess eine eigene "'Uhrzeit"' besitzt und sich diese auch zumindest nicht perfekt synchronisieren lässt. Dies hat zurfolge, dass es im Prinzip unmöglich ist festzustellen, ob zwei Events gleichzeitig passiert sind oder eben nicht. Eine Möglichkeit, dieses Problem zu lösen, ist die sogenannte "Happened-Before-Relation"'(Lamport-Uhr) [4]. Sie wurde von Leslie Lamport formuliert wird verwendet, um die relative Zeit zwischen zwei Events herauszufinden. Dazu wird den versendeten Nachrichten der jeweils aktuelle Zeitstempel hinzugefügt. Auf diesem Weg werden die stattgefundenen Events also zeitlich sortiert. Aufgrund dieser Erkenntnis, wird die Definition, wann B *True* wird, präzisiert: Gegeben sind zwei Zustände s und t, mit zugehörigen (lokalen) Prädikaten CS_1 sowie CS_2. CS_1 sowie CS_2 sind *True* in s und t. Hinzukommt, das beide Zustände simultan aktiv sind. Demzufolge wird die Definition von *possibly B* (vgl.II) angepasst:

Definition possibly:B
Ein globales Prädikat B ist *True*, wenn es einen Pfad vom initialen bis zum finalen Zustand in C(S) gibt, indem B in einen dazwischenliegenden konsistenten Zustand *True* ist.

Das diese Definition auf der "'Happend-Before-Relation"' basiert, liefert einen entscheidenden Vorteil: Zur Bestimmung einer globalen Zeit können nun die sogenannten *Vektoruhren* verwendet werden. Vektoruhren sind eine Erweiterung der Lamport-Uhr. Wie der Name schon vermuten lässt, besteht die Uhr hier nur aus dem lokalen Wert, sondern

aus einem Array, dass *alle* bekannten Uhrzeiten der anderen Prozesse beinhaltet.

B. Fehlen von gemeinsamen Speicher

Ein weiteres Problem hat seinen Ursprung in dem Fehlen eines gemeinsam genutzten Speichers. Die Prozesse haben keinen gemeinsam verfügbaren Speicher und kommunizierenn daher über Nachrichten miteinander. Aufgrunddessen, fällt die Nachrichtenkomplexität, globale Eigenschaften des Systems zu beobachten, natürlich entsprechend höher aus, da auf keinen gemeinsamen Speicher zugegriffen werden kann. Ein Beispiel hierfür kann wie folgt dargestellt werden:
Stellt man sich zwei Prozesse P_1 und P_2, mit zugehörigen Variablen $x_1 \in P_1$ sowie $x_2 \in P_2$ vor, stellen diese das globale Prädikat B(x_1, x_2) dar. Vorrausgesetzt, dass eine Evaluation von x_2 jeden Wert von x_1 benötigt, so müssen alle Nachrichten die das betrifft, von P_1 zu P_2 geschickt werden. Es ist leicht vorstellbar, dass dies sehr unpraktisch innerhalb eines verteilten Systems ist.

Um solche unpraktischen Funktionen zu unterbinden, wird eine Klasse von Funktionen erstellt, die durch Monotonität beschränkt werden. Diese beeinhaltet alle Funktionen, die höchstens einen Wert pro Event über Nachrichten versenden müssen. Formal gesehen ist ein Prädikat dann, monoton bezüglich einer Variablen x_1, wenn das Ersetzen von x_1 durch einen größeren Wert, whärend alle anderen Variablen gleichbleiben, den Wert des (globalen)Prädikates nicht verändert. Das globale Prädikat B ist monoton bezüglich x_1 wenn folgende Gleichung gilt:

$$\forall a, b, x_2 : (a < b) \land (B(a, x_2) \Rightarrow (B(b, x_2)). \quad (3)$$

Ein einfaches Beispiel verdeutlicht dies:
Sei B = $(x_1 > x_2)$ mit $x_1, x_2 \in N$. Dann ist B monoton bzgl. x_1, denn wenn B für einen bestimmten Wert von x_1 bereits größer als x_2 ist, dann wird dies auch für nachfolgende, der Relation entsprechende Werte von x_1 so bleiben. Werden Prädikate durch Monotonität beschränkt, genügt es das wir uns auf *Zustandsintervalle* begrenzen und nicht weiterhin Zustände betrachten müssen. Ein Zustandsintervall ist eine Sequenz von Zuständen zwischen zwei *externen* Events, also dem Senden, bzw. Empfangen einer Nachricht. Ein weiteres Beispiel ist der Beginn und das Ende eines Prozesses. Es wird also nicht länger ein Zustand sondern eine Reihe von Zuständen betrachtet. So reicht es auch für jede Variable den entsprechend erreichten Maximalwert zu kommunizieren. Es muss also *nicht mehr jeder angenommene Wert* einer Variablen gesendet werden. Die Anzahl der Zustandsintervalle ist in der Regel kleiner als die Anzahl aller auftretenden Events in einem System. Im folgenden werden die Begriffe *Zustand* und *Zustandsintervall* synonym verwendet.

C. Kombinatorische Explosion

Im folgenden wird angenommen das die vorher angesprochenen Restriktionen, das Fehlen von gemeinsam genutzter Uhren und Speichers, überwunden sind. Das Problem an diesem Punkt ist die Berechnungskomplexität. Hierbei tritt das Problem der Kombinatorischen Explosion auf. Es existieren N Prozesse, die maximal mögliche Anzahl an Zuständen ist m^N, wobei m die Anzahl der Zustandsintervalle in jedem Prozess ist. Mit jedem weiteren Zustand steigt die Berechnungskomplexität also exponentiell an. Betrachtet man ein globales, boolsches Prädikat B und angenommen, kein einziger Prozess würde mit einem anderen kommunizieren, (es werden also keinerlei Nachrichten versendet oder empfangen), ist das Problem, das globale Prädikat *possibly: B* zu entdecken NP-vollständig. Das Problem ist also nur von einer nichtdeterministischen Turingmaschine in polynomieller Zeit lösbar. Dies gilt sogar im minimalen Fall, indem, keinerlei Kommunikation zwischen den Prozessen stattfindet. Dies ist auch Grundlage des folgenden Beweises. Zur Vereinfachung werden folgende Annahmen getroffen:

- zwei Zustände S_1 und S_2
- drei Prozesse P_1, P_2 sowie P_3
- drei boolsche Variable x_i, mit i=1,2,3
- kein Prozess versendet Nachrichten

Folgende Abbildung 4 veranschaulicht die Annahmen.

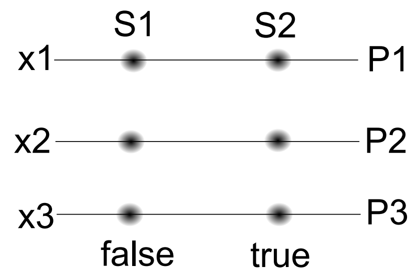


Figure 4. Deposet

Beweis: Entdecken globaler Prädikate ist NP-vollständig.
invariant: B $\stackrel{\text{def}}{=}$ ¬*possibly* : ¬B Ein globales Prädikat B ist für alle Zustände nicht mehr veränderbar(invariant), wenn es in allen konsistenen Schnitten G des Verbandes C(S), wahr ist.

Für jede Variable $u_i \in U$ wird ein Prozess definiert. In unserem Beispiel sind das drei. Die Variablen x_i werden in den zugehörigen Prozessen verwendet. Jeder Prozess erreicht genau zwei Zustände. Im ersten besitzt x_i den Wert False, im zweiten den Wert True. Damit ergeben sich beliebig viele konsistente Schnitte, indenen leicht

nachvollzogen werden kann, dass das globale Prädikat B den Wert *True* erhält, *genau dann wenn* der entsprechende Ausdruck x_i erfüllt ist. Das ist ein Problem, das aufzeigt, dass selbst einfachere verteilte Berechnungen unlösbar sind. Daher muss die Klasse der Prädikate eingeschränkt werden. Dieser Schritt ist nötig, um effizientere Entdeckungen zu erlauben. Zum einen gibt es *stabile* Prädikate. Ein Prädikat ist genau dann stabil, wenn es einmal den Wert *True* angenommen, diesen beibehält. Das bedeutet, das wenn ein Prädikat B seinen Wert von bspw. False auf True ändert, bleibt dieses auch True. Prozesse schicken sich Nachrichten zu. Prozess explodiert wenn er von einer Nachricht getroffen wird. Formal gesehen sieht das wie folgt aus:

$$\forall G, H : B(G) \land H \text{ ist erreichbar durch } G \Rightarrow B(H) \quad (4)$$

Ob ein Prädikat dabei wirklich stabil ist, hängt auch immer von dem System ab indem es sich befindet. Ein für ein System A stabiles Prädikat kann in einem anderen System B instabil sein. Desweiteren gibt es die Unterteilung in lineare und reguläre Prädikate. Diese werden in den kommenden Abschnitten näher vorgestellt.

IV. Prädikate

In den folgenden Abschnitten werden nun die zwei verschieden Typen von Prädikaten, die Linearen und Regulären, vorgestellt.

A. Lineare Prädikate

Bei der Klasse der Linearen Prädikate handelt es sich um die allgemeinste und damit einfachste Klasse von Prädikaten. Um einen effizienten Algorithmus zu entwickeln, der in der Lage ist, lineare Prädikate zu entdecken, wird die Definition des *verbotenen* Zustandes verwendet. Dazu werden folgende Annahmen getroffen:

- S (Menge aller Events),
- das globales Prädikat B,
- ein Schnitt $G \subset S$.

Der Schnitt G besteht dabei wieder aus beliebig vielen Zuständen, dabei muss es sich nicht zwingend um einen konsistenten Schnitt handeln. Ein Zustand i aus dem Schnitt G, (kurz $G[i]$) heißt dann *verboten*, wenn er unter Einbezug eines jeden (anderen) Schnittes H∈C(S), indem G≤H gilt, impliziert, dass das globale Prädikat B den Wert *False* für H annimmt , obwohl B für G *True* ist. Formal lässt sich ein verbotener Zustand also wie folgt formulieren:

$$forbidden(G, i) \stackrel{\text{def}}{=} \forall H \in C(S) : G \leq H : (G[i] \neq H[i]) \quad (5)$$

beziehungsweise

$$\neg B(H). \quad (6)$$

Ein Prädikat ist demnach genau dann *linear*, wenn gilt:

Definition: Lineares Prädikat

$$\forall G \in C(S) : \neg B(G) \Rightarrow \exists i : forbidden(G, i) \quad (7)$$

Ausformuliert bedeutet dies: Ein boolsches (globales) Prädikat B ist genau dann linear bzgl. S(der Menge aller Events), wenn es impliziert, das es für jeden Schnitt G, für den es den Wert *False* besitzt, G einen verbotenen Zustand enthält. Aus der Definition des linearen Prädikates ergeben sich folgende Eigenschaften eines linearer Prädikates:

- $B_1 \& B_2$ sind linear, dann auch $B_1 \land B_2$.
- B verwendet die Variablen eines einzelnen Prozesses $\Rightarrow$ B ist linear
- B(G) $\equiv \forall i, j G(i) \parallel G(j) \Rightarrow$ B ist lineares Prädikat

Zu beachten ist, dass die Linearität eines boolschen Prädikates von C(S), somit auch von S abhängt. Infolgedesssen ist es interessant einen minimalen Schnitt für lineare Prädiakte zu finden.

1) Minimale Schnitte für Lineare Prädikate: Jedes globale Prädikat stellt eine(ggfs. leere) Teilmenge von Schnitten $C_B S \subseteq C(S)$ dar. Hier gilt B für jegliche Schnitte in $C_B S$. Dies bedeutet, dass wenn B linear ist, dass $C_B S$ abgeschlossen bzgl. des Infimums ist. Als Folgerung davon gilt, dass es keine Menge in C(S) gibt, die einen kleineren Schnitt als $C_B S$, definiert. Nachfolgend wird gezeigt das auch der minimale konsistente Schnitt die Eigenschaften eines Linearen Prädiaktes erfüllt.

> ***Lemma 2: Sei $C_B S \subseteq C(S)$. $C_B S$ ist abgeschlossen bzgl. des Minimums, gdw B in Bezug zu C(S) linear ist.***
>
> **Beweisansatz:**
>
> Sei B nicht linear. Dann muss es einen konsistenten Schnitt G geben,so dass gilt:
>
> - $\neg B(G)$
>
> $$\forall i : \exists H_i \geq G : (G[i] = H_i[i]) \quad (8)$$
>
> - sowie $B(H_i)$
>
> Sei Y = $\bigcup_i H_i$. Desweiteren gilt $Y \in C_B S. \Rightarrow$ Das infimum von Y aus G liegt nicht in $C_B S$. $C_B S$ ist demnach nicht abgeschlossen bzgl. der Minimum operation.

Ein Beispiel wird in Abbildung 5 dargestellt:

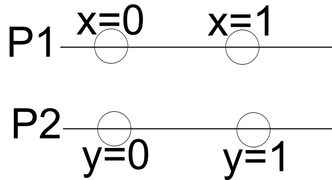


Figure 5. Nichtlineares Prädikat

Das Prädikat B(G) sagt aus, dass $x + y > 0$ gelten muss. Dies ist aber für genau den ersten Fall (x,y=0) nicht gegeben. Die restlichen Möglichkeiten erfüllen das Prädikat. nicht erfüllt. In manchen Fällen kann es durchaus sinnvoll sein das erste Vorkommen von B indem diese den Wert *True* besitzt zu kennen. Ein Beispiel hierfür ist, dass ein Entwickler versuchen kann zu bestimmen, welcher Programmierfehler das System in einen unerwünschten Zustand versetzt hat. Kann der Debugger nun also den ersten Zustand, indem B *True* war identifizieren, so ist es dem Programmierer leicht möglich den Fehler zu entdecken und zu korrigieren[3].
Im nachfolgenden Abschnitt wird kurz ein Algorithmus vorgestellt, der ein lineares Prädikat entdeckt.

2) Entdecken linearer globaler Prädikate: Hierfür existiert eine effizienter Algorithmus, der in polynomieller Zeit den verbotenen Zustand bestimmen kann. Dieser durchläuft die Menge der konsistenten Schnitte solange $/B(G)$ gilt. Dabei wird jeder enthaltene Zustand G[i] überprüft ob auf diesen die Defintion des verbotenen Zustandes zutrifft. Erreicht der Algorithmus den letzten Zustand ohne einen verbotenen Zustand zu finden, wird *False*, andernfalls *True* zurückgeliefert. Ist der erste verbotene Zustand bekannt, kann das entsprechende Prädikat (gemäß Definition der Linearität) schneller gefunden werden. Obwohl es eine exponentielle Anzahl von Schnitten im Ausführungsnetz gibt, werden bestenfalls $m \cdot N$ Schnitte entdeckt, wobei m die Anzahl der Zustände in jedem Prozesses sind.
Der folgende Abschnitt erläutert die regulären Prädikate.

B. Reguläre Prädikate

Reguläre Prädikate sind eine Teilmenge der Linearen Prädikate. Das hat zurfolge, dass die meisten linearen Prädikate auch regulär sind.

Definition: ***Reguläre Prädikate***
Sei C(E) das Ausführungsnetz der konsistenten Schnitte einer Berechnung(E,→), dann ist ein Prädikat regulär, gdw $\forall G, H \in C(E)$:

$$B(G) \wedge B(H) \Rightarrow B(G \cap H) \wedge B(G \cup H) \quad (9)$$

Desweiteren sind reguläre Prädikate abgeschlossen bzgl. Konjunktion. Ein (etwas ausführlicheres) Beispiel für reguläre Prädikate:

Gegeben ist ein Nachrichtenkanal C. Das Prädikat sagt nun folgendes aus: "‘Es befinden sich keine ausstehenden Nachrichten mehr auf C.”’ B verbleibt genau dann in einem konsistenten Schnitt G, wenn sich alle gesendeten Nachrichten mit ihren zugehörigen Empfängerevents ebenfalls in G befinden. Logisch gilt: Wenn $B(G) \wedge B(H) \Rightarrow B(G \cup H)$. Um zu zeigen das dies auch für $B(G \cap H)$ kann man sich folgendes überlegen: Sei *e* ein beliebig Senderevent in $G \cap H$ sein, und sei f das zugehörige Empfängerevent. Das Prädikat B(G) liefert, dass $f \in G$ und B(H) liedert, dass $f \in H$ liegt. Demnach gilt: $f \in G \cap H$.

Beweis: *Sind $B_1 sowie B_2$ regulär, dann ist auch $B_1 \wedge B_2$ regulär. Seien G und H konsistente Schnitte und gelte K = $G \cup H$. B_1 ist ein reguläres Prädikat für K, somit gilt auch $B_1(K)$. Analog für B_2. Also gilt auch $B_1 \wedge B_2$ für K.*

Nachfolgend wird ein Algorithmus vorgestellt, der lineare Prädikate entdecken kann.

V. Algorithmen

Ein effizienter Algorithmus der lineare Prädikate entdecken kann sieht wie folgt aus[1]:

var
G:array[1...N] $\forall i : G[i] :=$ *initial(i)*
while ¬ B(G) do
Let *i* be such that *forbidden*(G,i);
if (G[i]=*final(i)*) then return *false*
else G[i] = *next*(G[i]);
endwhile;
return true;

Durch die Linearität von B ist bekannt, dass wenn B in irgendeinem Zustand den Wert *False* besitzt, es einen verbotenen Zustand gibt. Mithilfe des verbotenen Zustandes wird nun in allen minimalen konsistenten Schnitten nach dem Zustand gesucht, indem das Prädikat den Wert *True* besitzt. Ist dies gefunden, wurde das Prädikat entdeckt. Dieser Algorithmus erreicht dies in polynomieller Zeit[1]

Ein weiterer Lösungsansatz globale Prädikate zu entdecken, ist der von Lamport und Chandy entwickelte globale Schnappschussalgorithmus. Allerdings lässt sich dieser nur auf stabile Prädiakte, wie beispielsweise Terminierung anwenden.

VI. Abschluss

Folgend wird ein Fazit und ein Ausblick auf weitere Prädikatsklassen gegeben.

A. Fazit

Um nachvollziehen zu können wie sich ein verteiltes System verhält, ist es sinnvoll dessen Eigenschaften zu bestimmten Zeitpunkten zu kennen. Konsistente globale Zustände geben einen Einblick was in einem System passiert sein könnte. Hilfreich dabei ist das entdecken bestimmter Eigenschaften(Prädikate) wie beispielsweise, "‘Prozess P_1

hat Nachricht A erhalten"' oder "'Prozess P_2 antwortet nicht"'. Ein konkretes Beispiel wurde in Abschnitt der linearen Prädikate gegeben.

B. Ausblick

Neben den linearen und regulären Prädikaten gibt es noch die Klasse der Konjunktiven Prädikaten. Diese sind eine Teilmenge der regulären Prädikate. Dabei handelt es sich um globale Prädikate, die aus der Konjunktion lokaler Prädikate besteht, beispielsweise

$$B = q_1 \wedge ... \wedge q_k; \qquad k \geq 1, \tag{10}$$

sofern q jeweils ein lokales Prädikat bezeichnet.

References

[1] Vijay K. Garg, *Observation of global properties in distributed systems*, Electrical and Computer Engineering Department, The University of Texas at Austin, Austin, TX 78712

[2] Universität München *Vorlesung Betriebssysteme II* http://inf3-www.informatik.unibw-muenchen.de/lehre/vorlesungen/ft2003/betrsys2/html/whiteboard/vsys5.4.4.2.html

[3] Vijay K. Garg and Craig M. Chase, *Detection of Global Predicates: Techniques and thier Limitations*, Electrical and Computer Engineering Department, The University of Texas at Austin, Austin, TX 78712

[4] Prof. Dr. F.Mattern *Verteilte Algorithmen*, Departement Informatik, ETH Zürich, 2005.

Message Orderings in Distributed Systems

FIFO, causal, synchronous and total ordering

Nam Thang Dinh
Carl von Ossietzky University Oldenburg
Faculty II: Informatik, Wirtschafts- und Rechtswissenschaften
Master of Science: Embedded systems and micro robotics
Email: nam.thang.dinh@uni-oldenburg.com

Abstract — **This assignment was in the context of the course "Fault Tolerance in Distributed Systems" by Prof. Dr.-Ing. Oliver Theel, C.v.O University of Oldenburg, which was created in the summer semester 2011. The report shows how the message orderings are assigned and edited, making the communication in distributed systems can be completed with higher performance and more correctness. Through various algorithms such as causal, synchronous or total ordering which allows messages to specific order of precedence.**

I. Introduction

Determinism is an important property that plays a big role in the communication of distributed systems. *Determinism* means that every output of a communicated system should remain constant values by the same input. Fulfilling this above mentioned feature is a difficult task, because of their *nondeterministic* nature, which was caused by various ordering of messages in different cases. In Figure 1, the first example shows that the received event r12 reaching process P1 before the received event r13. Meanwhile the second example with another ordering results a reserved effect: the received event r12 reaches P1 after the received event r13. In order to prevent these possible multiple behaviors it is necessary to implement methods, which effectuate the ordering of message in a system.

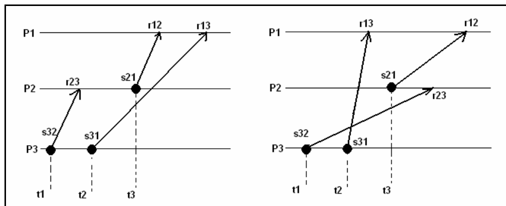


Figure 1. Different ordering of messages causes nondeterministic results

Message orderings are either *synchronous or asynchronous*. The main difference between a “synchronous” and “asynchronous” ordering is that a synchronous ordering blocks a process till a sending/receiving event completes, while an asynchronous ordering is non-blocking and only initiates the event. A process in an asynchronous ordering can not make any assumptions about the state of another process at any time, since all processes with different speeds are allowed to work and there is not a reliable global timer available. A message may need arbitrarily long time to attain their destination; a lost message has even endless transmission time, which will never reach its target. As a result of unknown reaching time, all messages can be delivered in any order. While asynchronous ordering is independent of physical time, all components of a synchronous ordering must be time-restricted. Figure 2 shows that, every send event in a synchronous ordering need to be finished after an amount of predictable time. The ideal synchronous can be realized, where all processes works without deviations, on the same rate and deliver messages without any delays. In other words, synchronous systems can be understood as a special case of asynchronous systems.

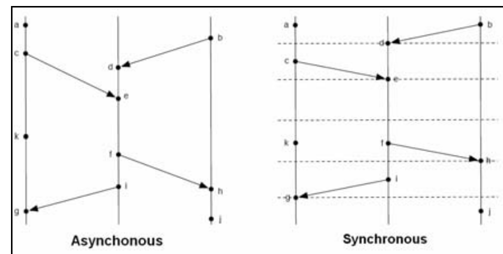


Figure 2. Asynchronous vs. synchronous ordering [1]

Moreover, it is necessary to determine the message orderings between different processes. Figure 3 shows an example of typical problem, which a replicated variable x need to be updated. As result of absence of an ordering between sending events in different processes, x may contain locally different values lately.

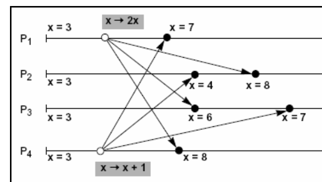


Figure 3. Problem – received data may be non-atomic [2]

II. FIRST-IN-FIRST-OUT

A fully asynchronous computation has a huge advantage: allowing maximum concurrency of processes to be executed. But fully asynchronous algorithms, in most cases, are very difficult to design, because they need to be realizable on every working ordering of messages. For this reason many systems restrict message orderings to specific sequences. One of the most well-known orderings is FIFO, which is based on a "***F**irst-**I**n-**F**irst-**O**ut*" assumption. Following this order, every message, which was generated by the same sender and be delivered to the same receiver, will receive a sequence number. These numbers will decide if the receiver will obtain an accordant message earlier or later as other upcoming messages.

Let us consider that every system is composed of *different processes*, which consist of a *sequence of events*. An event can, depending on the application, be the execution of a computer or a machine instruction. It is assumed that events are arranged sequentially within a process, such that: *a, b* are two events happened on the same process: if *a* occurs before *b*, then $a \prec b$. "$\prec$" stands for "*locally-precedes relation*"

Process P1 sends two messages S1 and S2 to process P2. R1 and R2 are the corresponding events at P2, which confirm that it has already obtained the messages. FIFO-ordering makes sure that, if the message S1 was sent out before S2, then in any case the corresponding event R2 cannot have occurred before R1.

FIFO: $S1 \prec S2 \Rightarrow \neg(R2 \prec R1)$

Satisfying the FIFO ordering, a system must *lose some of its concurrent feature,* because the messages from the same process must be ordered in a particular sequence. And if a message while sending causes an error, the whole processing will must be delayed.

III. CAUSAL ORDERING

A. Causal relation

In order to make stronger declarations about the relationship between different events in a distributed system, the term must be defined more precisely. This is how causal relation was designed for.

Causality is the relationship between a caused-event and an effected-event, where the second event is understood as a consequence of the first. Every modification of the caused-event leads to changing its effected-events. In other words: caused-event influences effected-event.

There are two events E1 and E2 of two any processes P1 and P2. E2 will be considered as causally dependent on E1, or "E1 → E2", if any of following conditions is satisfied:

- Cond. 1: P1 and P2 are the same process; and E1 happened before E2.
- Cond. 2: E1 is the sending event of a message, while E2 is the receiving event of same message.
- Cond. 3: There is an event E, so that: E1 → E and E → E2 (transitivity)

"→" stands for "*happened-before relation*".

B. Causal ordering - Principle

Generally causal ordering is an *improvement of FIFO* ordering, as is contains every properties of FIFO. In addition the causal ordering carries *one more condition*, which makes it stronger as FIFO. Causal ordering requires that single messages should not be overtaken by a sequence of messages. Sequence of messages is defined as messages, which causally relate to each other.

Let us consider that S1 and S2 are the two sending events to the same receiver in a distributed system and S2 is causally dependent on S1. If R1 and R2 are their two corresponding events, then R2 should not be occur before R1 by any process, in context of causal ordering.

CAUSAL ORDERING: $(S1 \rightarrow S2) \Rightarrow \neg(R2 \prec R1)$

Figure 4 shows an example, where the message orderings do or do not satisfy causal ordering, respectively. As we see, s13→s12 satisfies, because s13 happens before s12 on the same process P1. Meanwhile s12→r12 satisfies, because r12 is receiving event of sending event s12, and on process P2 r12 should be happened before s23, or r12→s23. It means s13→ s12→ r12→s23, or s13→s23. Following causal ordering, there is no way that the receiving event r13 of the sending event s13 could be happened before the receiving event r23 of the sending event s23. Therefore the first case does not satisfy causal ordering, while the second does.

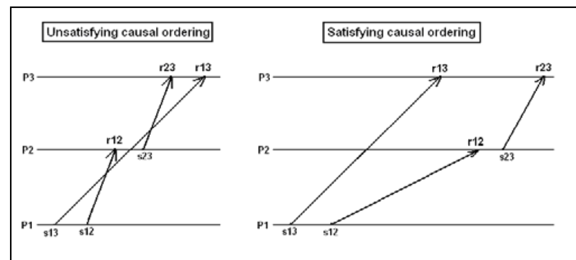


Figure 4. Causal ordering

C. Causal ordering – Algorithm

The condition is that a process can not send any messages to itself. This algorithm is using time vectors or matrixes, which are implemented at every process. Process P_i contains a matrix M[m, n] of integer, which records the number of messages sent from process P_m to process P_n.

Sending a message is very simple. Every time process P_i want to send a message to process P_k, the value of time vector M[i, j] will be increased by 1, and will be piggybacked in the message. The actual value of M[i, j] shows how many messages from P_i was sent and eventually still heading to P_k.

P_i: Send a message to P_k → *M[i,k] = M[i,k] + 1*

Receiving a message is a bit more complicated. Whenever a new message comes from P_j to P_i, it must be first checked for the correct sequence before can be able to be received. Let's say a message with its entry W[j, i] need to be delivered to P_i. Firstly it will be checked if the entry W[j,i] is one more than the entry M[j, i], to make sure that the message W is the next message from P_j to P_i. The second check is if there are any messages from other processes upcoming. If W[k, i] > M[k, i] for any process P_k ($k \neq j$) , it means that there are some messages, which had already sent from process P_k to P_i before, but have not arrived recently. In that case P_i must wait for these other messages, before it can accept the message W. If W[k, i] <= M[k, i], then it is possible now for the message to be delivered. After successfully capturing or denying a message P_i will update his matrix M with the value of the latest message's matrix W.

P_i: Receiving a message with entry W from P_J, if:
(W[j, i] = M[j, i] + 1) and
(with any ($k \neq j$): W[k, i] <= M[k, i]);
M[j, i] = max(M[j, i], W[j, i]);

IV. SYNCHRONOUS ORDERING

A. Syschronous ordering - Principle

As we see from the first three sections, the *unpredictable reaching time* of a sent message was the main problem of an asynchronous system. It takes normally an amount of time from the moment that a process sends a message to another, until this message reaches its destination. Therefore, it will be very difficult to put those messages in a fixed order as they should be, because if a sending event lasts long enough so that it is overtaken by other faster events, the final result would be changed. However it would be much easier if the travel time of messages could be considered as *instantaneous*, logically. This means, the receiver would obtain a message at the moment it was just sent. It is how the synchronous messages ordering are defined. Figure 5 is an example of a satisfied synchronous messages ordering following this conception, showing that the receiving events r23, r32 and r13 are designed to happen at the same moment as their sending events s23, s32 and s13.

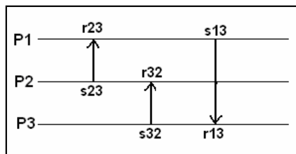


Figure 5. Satisfied synchronous messages ordering

The main ideal of this ordering, which makes communication easier to design, is to assume that every sending messages are instantaneous or as "*points*" rather then "*intervals*". In fact messages can never be instantaneous, as they always need some *amount of time* to finish broadcasting, respectively. In some certain situations, we can only consider that this requirement is valid, if the transformation from "*intervals*" to "*points*" does not affect the final result of the messages transaction.

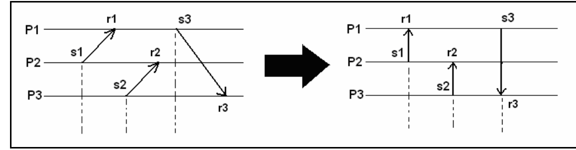


Figure 6. Casting to synchronous ordering

Let s be a sending event; r is its receiving event; ε is the set of all external events and T is the timestamp of the event, then a synchronous ordering is defined, if these two conditions are fulfilled:

While $s, r, e, f \in \varepsilon$:

- All transactions are instantaneous:
  $$s \mapsto r \Rightarrow T(s) = T(r)$$
- Earlier events should happen before later events
  $$e \prec f \Rightarrow T(e) < T(f)$$

It is easy to see, which two any external events e and f:

$$\boxed{(e \to f) \wedge \neg(e \mapsto f) \Rightarrow T(e) < T(f)}$$

In this assignment, "$\mapsto$" stands for "*remotely-precedes relation*", which defines that *a* occurs before *b* and a, b are two events on different processes.

B. Synchronous Ordering - Crowns

Synchronous message orderings are very strong and have their big advantages, but it is not always easy to say, if an ordering can be *realized as synchronous or not*.

In Figure 7 is an example of a message ordering. As we see, this computation must always maintain some conditions: at process P1 the send event s1 must occur before the receive event r1 (s1 < r1); at process P2 the send event s2 must occur before the receive event r2 (s2 < r2) and at process P3 the send event s3 must occur before the receive event r3 (s3 < r3).

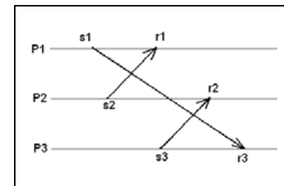


Figure 7. Impossible to be turned into synchronous ordering?

Now the ordering need to be reconstructed, by removing the "intervals" between sending events and their accordant receiving events and reconsider them as "points". In Figure 8 there are six alternative reconstructions.

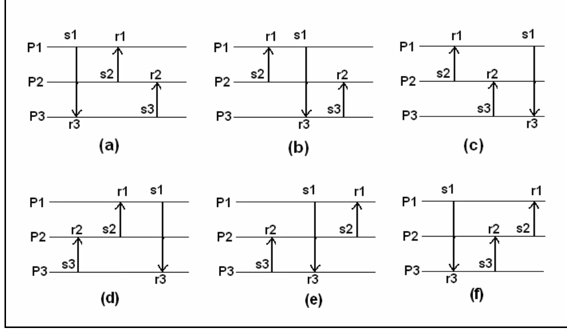


Figure 8. Casting to synchronous ordering

We would realize soon that *none* of these six fulfilled the requirements of original ordering: (a) – r3 occurs before s3; (b) – r3 occurs before s3 and r1 occurs before s1; (c) – r1 occurs before s1; (d) – r1 occurs before s1; (e) – r2 occurs before s2; (f) – r3 occurs before s3. It means, this computation structure is not able to be considered as "synchronous".

In distributed system this special structure has been called as "*crown*". ***"A crown (of size k > 1) was defined as a sequence [(s_i, r_i), i Є {0, 1, 2,…, k - 1}: s_i sends message to r_i] of pair of corresponding send and receive events such that***

$$s_0 \rightarrow r_1,\ s_1 \rightarrow r_2,\ \ldots,\ s_{k-2} \rightarrow r_{k-1},\ s_{k-1} \rightarrow r_0"$$

A computation is synchronous if there is no crown in it, obviously.

C. Synchronous ordering - Algorithm

Under the requirement, that a process can not send a message to itself, we assume that every send event is either from a big process called big(s), or from a small process called small(s). Furthermore a process can be found in two states: *active* or *inactive/passive*.

A process can send a message to a smaller process at anytime, but only when it is active. After sending the message the process turns to passive, not allowed to do the next send or to receive any messages, until it gets an *ack* message from the receiver of the sent message. In this case with any event e and a sending from a big process big(s):

$$(s \prec e) \wedge big(s) \Rightarrow (r \rightarrow e)$$

OR: $(s \rightarrow e) \wedge big(s) \Rightarrow (r \rightarrow e)$

If a process is going to send a message to a bigger one, it will need to ask for permission by requesting. The receiver can only grant permission, when it is active. After granting the permission it must turn to inactive and stay in this state until receiving the messages.

$$(e \prec r) \wedge small(s) \Rightarrow (e \rightarrow s)$$

Followed by

$$(e \rightarrow r) \wedge small(s) \Rightarrow (e \rightarrow s)$$

OR: $(s \rightarrow e) \wedge small(s) \Rightarrow \neg(e \rightarrow r)$

The final algorithm can be defined as:

$$(s \rightarrow e) \wedge big(s) \Rightarrow (r \rightarrow e)$$
$$(s \rightarrow e) \wedge small(s) \Rightarrow \neg(e \rightarrow r)$$

D. Synchronous ordering - Application

The synchronous ordering can be implicated following the algorithm in the last section. Let's consider that every process P_i contains two states: active and passive, which is initiated with active.

state: {active,passive} initially *active;*

Now process P_i wants to send a message to a "smaller" process P_J (j < i), which can only be executed if P_i is in active state. Once a process finishes sending a message, its state will be turn into passive.

Send *m* to P_J (j < i): *Send m* if (*state = active*); *state := passive;*

If process P_i receives a message from a bigger process P_J (j > i), the delivery only successes when P_i has a passive state. After receiving the message P_i sends an *ack* back to P_J.

Receive a message *m* from P_J (j > i): *Receive m* if (*state = active*); *send ack* to P_J;

If P_i receives an ack message, it turns its state into active.

Receive *ack*: *state := active;*

In order to send a message from P_i to a bigger process P_J (j > i), it need to make a request with the *message_id* to be accepted by P_J first.

Send a message (message_id, m) to P_J (j > i): *Send request(message_id)* to P_J;

If process P_i receives a request on receiving a message from a smaller process P_J (j < i), P_i will wait until its state is active and send a permission message back to P_J. After sending permission, P_i turns the state into passive.

Receive a request(message_id) from P_J (j < i):

Send permission(message_id) to P_J if *(state = active)*; *state := passive;*

By receiving permission on a message from a bigger process P_J, P_i waits until its state is active and sends a the permitted message to P_J.

Receive a permission(message_id) P_J (j > i): *Send* m to P_J if *state = active*;

After receiving a message from smaller process, P_i turns its state to active.

Receive m from P_J (j < i): state := active;

Figure 9 shows how a message can be sent from a big process to a smaller process and Figure 10 shows the inverse schedule.

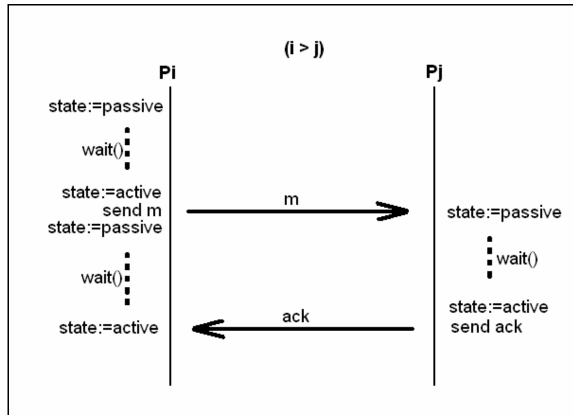


Figure 9. Sending a message to smaller process

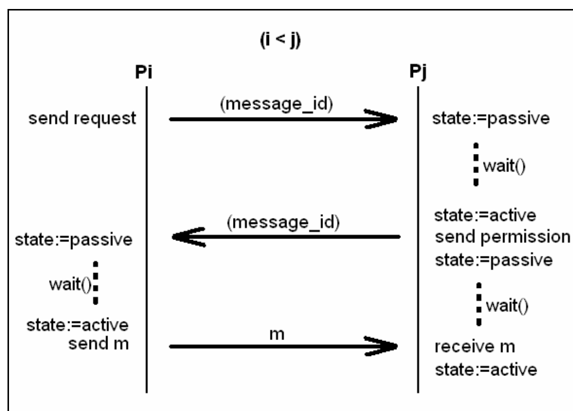


Figure 10. Sending a message to bigger process

V. Total Ordering

A. Total ordering for multicast messages

The main problem of casting messages to a process is solved with synchronous ordering so far, as we can see on the last section, there are many situations, where messages need to be delivered to *more than one process*. By *multicast* the same messages to different processes, it is necessary to keep all the messages approaching their destinations in the same order.

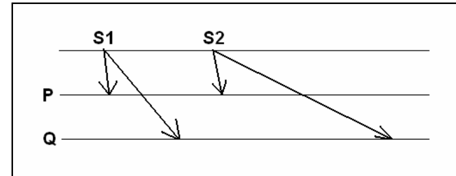


Figure 11. Total order for multicast messages

Let us consider that both messages S1 and S2 are sent to two processes P and Q (Figure 11). There is an assertion: if the message S1 reaches process P before S2, then process Q should not receive the message S2 before S1. This is also how the "*total order*" is defined.

There are two minor notes in this ordering. First, we would rather say "process Q should not receive the message S2 before S1", than saying "process Q should receive the message S1 before S2", because the first term will be still correct, even the message S1 is not sent to Q. Generally this concept is valid without a requirement that every single message must be broadcast to all processes. Moreover the order of receiving messages at all processes must not be the same as they were sent. If S1 was sent followed by S2, and processes P and Q received S2 before S1, the total ordering is still satisfied. Depending on the order of sending and receiving messages, in addition of total ordering we could have either "*FIFO total order*" or "*causal total order*".

B. Total ordering - Algorithm

There is some algorithms, which allows the total ordering to be realized, such as *centralized algorithm* (all sent messages are gathered in one center place before transmitted forward to destinations using mutual exclusion), or *Skeen's algorithm*. In this assignment we will discuss about an uncentred algorithm based on *Lamport's Distributed Mutual Exclusion Algorithm* for causal total ordering of messages.

Lamport's algorithm is an algorithm founded by computer scientist Leslie Lamport, which is intended to improve the safety in the usage of shared resources among multiple threads by means of mutual exclusion. We assume that all messages have FIFO ordering and every single message will be broadcast to all processes. Every process maintains a logical clock used for timestamps and a queue for storing undelivered messages. According to Lamport the algorithm can be done following these steps:

- Requesting critical section: a process broadcasts a message by sending it with timestamp to all other process includes itself.
- Waiting for critical section: the message after reaching other processes will be stored in their queue. After that the processes send an acknowledgement back to the sender.
- Executing in critical section: every time a process receives a message, which has a timestamp greater than the smallest one in its queue, the message with smallest timestamp will be removed.
- Releasing of critical section: when a process removes its own message, it is time for other processes to deliver their message.

By comparing the timestamps of messages and remove the smallest, it makes sure that every process maintains the latest message. After sending out any message including itself, the process waits until this message is removed because it is the smallest one in the queue. This guarantees that a sending event while processing will not be overtaken by other upcoming receiving events.

C. *Total ordering – Replicated state machines*

Replicated state machines are system for implementing a fault-tolerant service by replicating servers and coordinating client interactions with server replicas. These replicated servers work *independent* to each other. By requesting called inputs from client the inputs will be passed to all replicated server. It is very important, that the inputs need to be sent in the same order to all server replicas. After executing the request the system will respond to client with the outputs from state machines. If there is no majority of replicas with the same output, or if less than a majority of replicas returns an output, a system turns into "*crest failure mode*".

As we have noticed, in order to make sure that all replications are in the same state and work under the same condition while executing the requests, all inputs need to be ordered to each replica *identically*. In this case the total ordering of messages satisfies this property.

VI. Final Remarks

This assignment gives a general view, how messages in a distributed system should be ordered, by learning FIFO, causal, synchronous und total ordering systems. Each orderings relates to others in a tight relationship, in which:

$$\text{Synchronous} \subseteq \text{Causal} \subseteq \text{FIFO} \subseteq \text{Asynchronous}$$

The ordering likes synchronous ordering, which are *stronger* than other, contains also *more requirements* and need in most cases *more complex algorithm* in order to be applied, respectively. Because of various specified advantages and disadvantages all of messages ordering algorithms are still implemented in many distributed structure nowadays.

VII. Bibliography

[1] Scripts to distributed systems by Dipl.Inf. Philipp Schmidt and Prof. Dr.-Ing. Volker Roth, Free University Berlin.
http://www.inf.fu-berlin.de/lehre/SS11/VS/slides/VS-2011-05-03.pdf

[2] Scripts to distributed systems by Prof. Dr. Friedemann Mattern Prof. Dr. Gustavo Alonso, ETH Zurich.
http://www.vs.inf.ethz.ch/edu/WS0506/VS/slides/Vorl.VertSys05_06-4.pdf

Berechnung einer globalen Funktion

Tim Schmidt
Universität Oldenburg
Oldenburg, Germany
Email: tim.schmidt@uni-oldenburg.de

***Zusammenfassung*—Die gemeinsame Berechnung einer Funktion in einem verteiltem Netzwerk stellt Herausforderungen. Die Anfrage nach dem Zustand aller Knoten, dem globalen Zustand, soll möglich sein. Findet die globale Berechnung regelmäßig statt, so soll eine gleichmäßige Lastverteilung auf die einzelnen Knoten erfolgen. Ein Broadcast/Convergecast-Algorithmus bietet einen ersten Ansatz zur Bestimmung des globalen Zustands, benötigt aber einen Spannbaum. Eine weitere Möglichkeit, ohne Benutzung eines Spannbaumes, ist mittels Nachrichtenaustausch den globalen Zustand zu bestimmen. Jeder Knoten empfängt Nachrichten von fremden Knoten, leitet diese weiter und bestimmt daraus den globalen Zustand. Eine gleichmäßige Lastverteilung ist bei der Berechnung einer globalen Funktion zu erreichen, wenn in dem Netzwerk die Positionen der Knoten regelmäßig rotieren.**

I. Einleitung

Diese Ausarbeitung ist im Bereich der verteilten Systeme einzuordnen. Moderne Netzwerke berechnen komplexe Aufgaben oft nicht mehr lokal, sondern verteilen die Last auf verschiedene Knoten. Dabei ist es wichtig den Zustand aller Knoten, den globalen Zustand, erfassen zu können. Bei einer Aufgabe mit hoher Wiederholrate, z.B. das Zusammenfassen einer verteilten Datensammlung, soll auf die gleichmäßige Belastung aller Prozesse geachtet werden.

Der globale Zustand ist mittels dediziertem Root-Knoten berechenbar (Abschnitt III-A). Eine Alternative stellt ein nachrichtenbasierter Ansatz ohne Root-Knoten dar (Abschnitt III-D). Eine gleichmäßige Lastverteilung der Knoten ist realisierbar, wenn z.B. jeder Knoten einmal jede Position im Netzwerk einnimmt (Abschnitt IV).

II. Theoretische Grundlagen

Dieser Abschnitt stellt eine Arbeitsgrundlage für die nachfolgenden Kapitel dar. In einem verteilten Netzwerk kennt jeder Prozess seinen eigenen Zustand und besitzt eine eindeutige ID. Der Zustand aller Prozesse, auch *globaler Zustand* genannt, ist ihm unbekannt. Eine Funktion f, die den globalen Zustand eines Netzwerkes berechnet, heißt *globale Funktion*. Das Netzwerk von Prozessen wird durch einen ungerichteten zusammenhängenden Graphen repräsentiert. Ein *Kanal* oder Kante zwischen zwei Prozessen ist immer bidirektional. Die Übertragung einer Nachricht zwischen zwei Prozessen ist in Null-Zeit möglich.

Ein Spannbaum ist ein Baum, der alle Knoten enthält [CLR04, Seite 565]. Es wird unterschieden zwischen einem ungewichteten und gewichteten Spannbaum. Bei einem gewichteten minimalen Spannbaum soll die Summe der Kantengewichte minimal sein. Das Kantengewicht kann beispielsweise die Distanz zwischen zwei Prozessen modellieren. In den folgenden Abschnitten wird auf diese Notation zurückgegriffen:

N	Anzahl der Prozesse
m	Anzahl der Kanäle
D	Durchmesser des Netzwerkes
P, Q	Prozesse
$Z(P)$	Zustand von Prozess P
c	Kanal

Die globale Funktion kann mit dieser Beschreibung folgendermaßen beschrieben werden:

$$globaler\ Zustand = f\left(Z(P_1), Z(P_2), ..., Z(P_N)\right)$$

III. Berechnung einer globalen Funktion

In einem verteilten Netzwerk beschreibt eine globale Funktion f den Zustand aller Prozesse. Dieser Abschnitt liefert verschiedene Vorgehensweisen zur Berechnung von f. Die Unterschiede sind unter anderem die Voraussetzungen (gegebener Spannbaum etc.) und das Verhalten (z.B. phasenorientiertes). Die ersten Algorithmen benötigen einen Spannbaum und einen definierten Root-Prozess, d.h. einen dedizierten Prozess, der die globale Berechnung übernimmt. Von diesen Voraussetzungen soll schrittweise Abstand genommen werden. Der phasenorientierte Ansatz zur Berechnung von Routingtabellen benötigt weder Spannbaum noch Root-Prozess. Abschließend soll dieser Algorithmus für eine beliebige globale Funktion f verallgemeinert werden.

A. Umsetzung durch Broadcast- und Convergecast-Algorithmen

Eine Lösung zur Berechnung von f ist, alle Informationen an einen definierten Prozess zu schicken, der die globale Funktion berechnet. Im Anschluss wird das Ergebnis an jeden Prozess geschickt. Das ist mit einem Broadcast und Convergecast Algorithmus möglich. Voraussetzung ist ein Spannbaum mit einem definierten Root-Prozess.

Zuerst wird der Convergecast-Algorithmus aktiv. Jeder Prozess P sammelt von allen Kindern den Zustand ein.

Dem Prozess selber ist bekannt, wie viele Kinder er hat und wer der Vater-Prozess ist, sofern vorhanden. Sind alle Informationen der Kind-Prozesse vorhanden, werden diese Informationen zu dem Vater-Prozess Q geschickt. Sollte der Prozess P der Root-Prozess sein, dann wird anschließend der globale Zustand berechnet. Eine beispielhafte Umsetzung des Convergecast-Algorithmus ist in Algorithm 1 dargestellt.

Algorithm 1: Convergecast Algorithm

Data: *parent*: process id; //initialized based on spanning tree;
Data: *numchildren*: integer; //initialized based on spanning tree;
Data: *numreports*: integer initially 0;

On receiving a report from P_j
  $numreports \leftarrow numreports + 1$;
  if $numreports = numchildren$ **then**
    if *parent* = *null* **then**
      compute global function;
    else
      send report to *parent*;
    end
  end

Nachdem der Root-Prozess den globalen Zustand errechnet hat, kann die Information an alle Kind-Prozesse weitergereicht werden. Die Kind-Prozesse reichen die empfangende Nachricht ebenfalls an die Kinder weiter. Eine mögliche Umsetzung von einem Broadcast-Algorithmus ist in Algorithm 2 gegeben. Beide Algorithmen nutzen die Struktur von einem Spannbaum.

Algorithm 2: Broadcast Algorithm

if *P is root node* **then**
  send m to all children;
else
  on receiving a message m from *parent*;
  send m to all children;
end

Das Vorgehen beider Algorithmen ist an einem Beispiel in Abbildung 1 zu sehen.

Ein Spannbaum wird von dem Convergecast- sowie dem Broadcast-Algorithmus benötigt. Für die Berechnung eines Spannbaumes benötigt jeder Prozess das Wissen, ob der Vater-Prozess schon bestimmt ist (*notdone*). Diese Angabe ist bei allen Prozessen, bis auf den Root-Prozess, initial auf *false* gesetzt. Weiter wird gespeichert, wie viele Kind-Prozesse ein Prozess hat. Ist die Eigenschaft *notdone true*, dann sendet P Nachrichten an alle potentiellen Kinder aus der Menge der direkten Nachbarn.

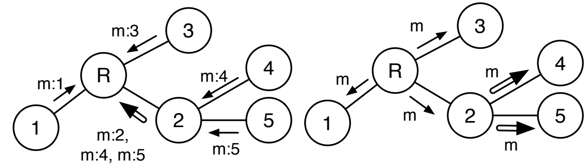


Abbildung 1. Dargestellt sind der Convergecast- (links) und Broadcast- (rechts) Algorithmus. Einfache Pfeile sind die erste Iteration, doppelte Pfeile stellen die zweite Iteration dar.

Ein Prozess P stellt mit dem Versenden einer Nachricht m an den Prozess Q die Anfrage, ob schon ein Vater-Prozess für Q definiert ist. Der Prozess Q fragt dazu *notdone* ab. Je nach Fall wird eine ablehnende oder akzeptierende Antwort verschickt. Im Fall einer akzeptierenden Antwort muss Q den Prozess P als Vater speichern. Gleichzeitig muss auch Q als Kind von P werden speichern und die Anzahl der Kinder um eins erhöhen.

Nachdem Q einen Vater-Prozess hat, wird an alle potentiellen Kinder eine Nachricht geschickt und die Anzahl der Kinder auf null gesetzt. Wenn die Anzahl der Antworten gleich der Anzahl der Nachbarn minus eins ist, dann stoppt der Algorithmus bei dem Prozess P. In diesem Fall haben alle potentiellen Kind-Prozesse geantwortet. Algorithm 3 fasst nun all diese Schritte zusammen. Ein Beispiel für das Vorgehen ist in Abbildung 2 zu sehen.

Die Anzahl der verschickten Nachrichten ist abhängig von der Kantenanzahl E: $\mathcal{O}(E)$.

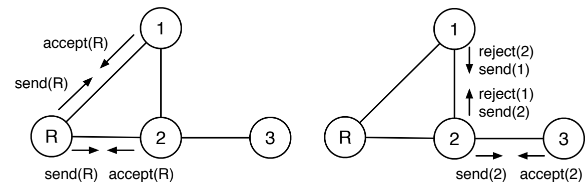


Abbildung 2. Zu sehen ist die Konstruktion eines Spannbaumes vom Prozess R aus. In der ersten Phase werden 1 und 2 Kinder von R. In der zweiten Phase wird 3 ein Kind von 2. Weiter schickt 2 an 3 und umgekehrt eine Ablehnung, da schon ein Vater-Prozess definiert ist.

B. Berechnung ohne Root-Prozess

Der vorgestellte Algorithmus zur Berechnung eines Spannbaumes benötigt einen Root-Prozess. Um dieses Problem zu lösen wird nachfolgenden ein Ansatz vorgestellt, der ohne Root-Prozess einen gültigen Spannbaum erstellt. Dazu wird vorausgesetzt, dass jeder Prozess eine eindeutige ID besitzt.

Zu Beginn geht jeder Prozess P davon aus, dass er der Root-Prozess ist und verschickt Nachrichten an alle Nachbar-Prozesse. Eine Nachricht m enthält dabei insbesondere die eigene ID. Jeder Empfänger Q vergleicht die eigene ID mit der ID aus der Nachricht m. Ist die enthaltene ID in

Algorithm 3: Konstruktion eines Spannbaums. Idee erstmals in [GHS83] vorgestellt.

Data: $parent$: process id initially $null$;
Data: $numchildren$: integer initially 0;
Data: $childrenlist$: list initially $null$;
Data: $numneighbors$: integer initally the number of neighbors;
Data: $numreports$: integer initially 0;
Data: $notdone$: boolean initially $true$ except for the root $false$;

On receiving a message m from P_j
  if $notdone$ **then**
    $parent \leftarrow P_j$;
    $notdone \leftarrow false$;
    send m to all neighbors except P_j;
    send ($parent$) to P_j;
  else
    send ($reject$) to P_j;
  end

On receiving a $parent$ message
  $numchildren \leftarrow numchildren + 1$;
  $append(childrenlist, P_j)$;
  $numreports \leftarrow numreports + 1$;
  if $numreports = numneighbors - 1$ **then**
    halt;
  end

On receiving a $reject$ message)
  $numreports \leftarrow numreports + 1$;
  if $numreports = numneighbors - 1$ **then**
    halt;
  end

m höher als die eigene wird der Sender als Vater akzeptiert. Andernfalls wird Q der Vater von P. Eine akzeptierende oder ablehnende Nachricht wird von Q an P zurück geschickt.

C. Berechnung von Routingtabellen

Die vorherigen Abschnitte haben gezeigt, dass ein Spannbaum für die Berechnung einer globalen Funktion erforderlich sein kann. Eine Lösung, die keinen Spannbaum benötigt, soll diskutiert werden. Der vorgestellte Algorithmus berechnet beispielhaft Routingtabellen. Das Prinzip wird im folgenden Abschnitt für eine globale Berechnung verallgemeinert. Analog zu den vorherigen Abschnitten wird der Algorithmus für jeden Prozess ausgeführt.

Ein Prozess kennt im Allgemeinen nur einen Teilgraphen, die direkten Nachbarn. Der Prozess P kann Q eine Nachricht schicken, wenn dieser ein direkter Nachbar ist. Das Verschicken einer Nachricht an Q, im Fall dass Q kein direkter Nachbar ist, soll mittels Routingtabellen gelöst werden. Jeder Prozess besitzt dazu an jedem anliegenden Kanal c eine Liste $route(c)$, in der angegeben ist, welche Nachrichten hier weitergeleitet werden sollen.

Die Berechnung der Routingtabellen erfolgt phasenorientiert lokal auf jedem Prozess. In der Phase i aktualisiert der Prozess P die Routingtabellen für Prozesse, die eine Distanz von i haben. Jeder Prozess besitzt eine vollständige Routingtabelle nach D Iterationen. Eine Iteration kann in drei Abschnitte aufgeteilt werden.

Der *erste* Schritt behandelt des asynchrone Versenden von Nachrichten. Für jeden Knoten, an dem der Kanal c anliegt, gibt die Menge $sent(c)$ die zu verschickenden Nachrichten über c an. Initial beinhalten alle Mengen $sent(c)$ eine Nachricht mit der eigenen ID. Weiter muss die Menge der neu angekommenen Nachrichten der aktuellen Iteration auf die leere Menge gesetzt werden.

Der *zweite* Schritt behandelt das Empfangen von Nachrichten. Eine Nachricht m' von einem Prozess Q', der bis dahin unbekannt war, muss in die Routing-Tabelle eingetragen werden. Er wird in die Routing-Tabelle des empfangenden Kanal c eingetragen. Alle Nachrichten m'' von Prozessen, die vorher noch nicht bekannt waren, sollen gespeichert werden. Die Nachrichten m''' von schon bekannten Prozessen werden verworfen. Sie enthalten keine neue Information für den Prozess.

Der *dritte* Schritt legt fest, über welche Kanäle welche Nachrichten in der nächsten Iteration verschickt werden sollen. Die Nachricht m'' sollen jeweils auf allen Kanälen bis auf den jeweiligen Empfangskanal weitergeleitet werden. Die zugehörigen Prozesse zu den Nachrichten m'' gehören ab jetzt zu der Menge der bekannten Prozesse.

Eine beispielhafte Umsetzung ist in Algorithm 4 zu finden. Die drei Schritte sind in Abbildung 3 dargestellt.

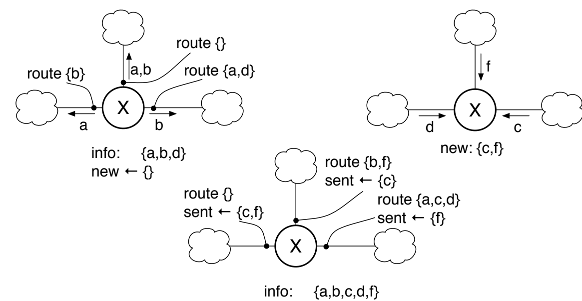


Abbildung 3. Oben links ist er *erste* Schritt zu sehen. Die Nachrichten der Mengen $sent$ werden verschickt. Im *zweiten* Schritt (oben rechts) kommen neue Nachrichten an. Eine Nachricht mit der Quell-ID d ist schon einmal angekommen und wird nicht weiter verarbeitet. Im *dritten* Schritt (unten) wird die Menge $sent$ neu gefüllt und die Menge $info$ aktualisiert.

D. Verallgemeinerung der Konstruktion

Der vorherige Abschnitt diskutierte die Berechnung von Routingtabellen. Voraussetzung war das Wissen über den

Algorithm 4: Berechnung von Routing-Tabellen. Zuerst veröffentlicht in [BKR88].

```
Data: d: integer initially 0;
Data: info: set of information initially {own node ID};
Data: sent(c): information initially info for all c
while d < D do
    d ← d + 1;
    send ⟨sent(c)⟩ on all channels c;
    new ← ∅;
    for every channel c do
        receive ⟨received(c)⟩ on c;
        for y ∈ received(c) − info − new do
            route(c) ← route(c) ∪ {y};
        end
        new ← new ∪ (received(c) − info);
    end
    info ← info ∪ new ;
    for every channel c do
        sent(c) ← new − received(c);
    end
end
```

Graphdurchmesser D. In jedem Durchlauf wurden Nachrichten geschickt oder empfangen, solange etwas zum verschicken vorhanden ist. Dem jetzt vorgestellten Algorithmus zur Berechnung einer globalen Funktion ist D nicht mehr bekannt. Allerdings soll auf die Grundprinzipien aus Algorithm 4 weiter zurückgegriffen werden.

Ohne die Angabe von D muss der Algorithmus ein neues Kriterium haben, ob ein neuer Schleifendurchlauf notwendig ist. Ein Kanal c heißt aktiv in der Phase i, wenn er in dieser Phase mindestens eine Nachricht verschickt oder empfängt. Die Berechnung ist so lange aktiv wie mindestens ein Kanal aktiv ist. Besitzt ein Prozess keine aktiven Kanäle mehr, so ist die Berechnung der globalen Funktion bei ihm lokal abgeschlossen. Dazu muss geklärt werden, wann ein Kanal schlussendlich nicht mehr aktiv ist.

P_1 und P_2 sind zwei benachbarte Prozesse. Kennen beide die gleiche Prozessmenge, so können sie nicht mehr von einander lernen. In Abbildung 4 tritt dieser Fall in der dritten Iteration ein. P_1 sendet an P_2 die Nachricht 5, genauso wie P_2 an P. Analoges Verhalten ist zu erkennen, wenn die Nachrichten von den Prozessen 6 bis 9 eintreffen. Der Kanal c zwischen P_1 und P_2 ist nicht weiter notwendig. P_1 und P_2 haben nach dem Empfangen und Versenden der Nachricht 5 nur noch einen aktiven Kanal. Der Algorithmus kann wie in Algorithm 5 umgesetzt werden. Die globale Funktionsberechnung basiert auf der Menge *info* und kann zur Laufzeit oder nach beenden des Algorithmus gestartet werden.

Für die Korrektheit soll nur eine Beweis-Skizze gegeben werden. Der vollständige Beweis ist in [Jal94, Seite 225] zu

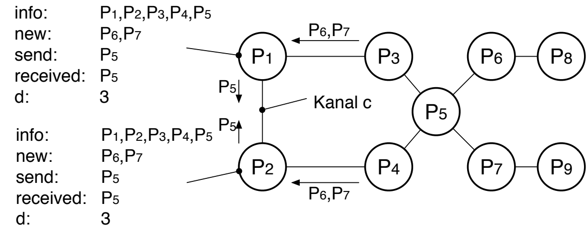


Abbildung 4. Zu sehen ist die dritte Iteration aus Sicht der Prozesse P_1 und P_2. Zu erkennen ist, dass Prozess P_1 und P_2 identische Nachrichten verschicken. Auch im weiteren Verlauf können sie nicht mehr voneinander lernen.

Algorithm 5: Allgemeine Funktion zum Berechnen der globalen Funktion

```
Data: d: integer; initially 0;
Data: info, new: set of information; initially {initial
      data};
Data: sent(c): information initially info for all c;
Data: active: set of channels initially all;
Data: received(c): information initially
      received(c) = ∅ for all channels;
while active ≠ ∅ do
    d ← d + 1;
    for c ∈ active do
        sent(c) ← new − received(c);
        send⟨sent(c)⟩ on c;
    end
    new ← ∅;
    for c ∈ active do
        receive⟨received(c)⟩ on c;
        if received(c) = sent(c) then
            active ← active − {c};
        end
        new ← new ∪ (received(c) − info);
        compute problem specific information
    end
    info ← info ∪ new;
end
```

finden. Folgende Definitionen sind für die Beweisstruktur notwendig:

$$V^i(P) = \text{Prozesse, die P in der i-ten Iteration kennen gelernt hat}$$

$$T^i(P) = \bigcup_{j \leq i} V^j(P)$$

Zuerst soll gezeigt werden, dass P und Q genau dann die gleiche Prozessmenge kennen, wenn P und Q über den Kanal c identische Nachrichten austauschen. Wir bemerken dazu zunächst folgenden Zusammenhang:

$$
\begin{aligned}
T^{d-1}(Q) &= T^{d-1}(P) \Leftrightarrow \\
V^{d-1}(P) - V^{d-2}(Q) &= V^{d-1}(Q) - V^{d-2}(P)
\end{aligned}
$$

Mittels Induktionsbeweis über die Phase d kann die folgende Aussage mit vorheriger Äquivalenz bewiesen werden.

$$
\begin{aligned}
send_d &= V^{d-1}(P) - V^{d-2}(Q) \\
received_d &= V^{d-1}(Q) - V^{d-2}(P) \\
new_d &= V^d(P) \\
active_d &= \{(P,Q) | T^{d-1}(P) \neq T^{d-1}(Q)\} \\
info_d &= T^d(P)
\end{aligned}
$$

Mit der $(D-1)$-ten Iteration hat jeder Prozess von jedem Prozess den Zustand. Das maximale Nachrichtenaufkommen liegt bei $2(D+1)m$ vielen Nachrichten pro Knoten und die Laufzeit bei $\mathcal{O}(D)$ [Jal94, Seite 228]

IV. Wiederholte globale Berechnung

In verteilten Netzwerken kann der globale Zustand von großem Interesse sein. Die wiederholte globale Berechnung wird angewendet wenn z.B. folgende Problemstellungen diskutiert werden:

- Deadlock-Erkennung
- eine globale Zustandsaufnahme
- Uhrzeit-Synchronisation

Dieser Abschnitt liefert eine Technik für den Umgang mit verteilten Datensammlungen. Bei der einmaligen globalen Berechnung wurde von der Baumstruktur mit dediziertem Root-Knoten auf Grund der hohen Fixkosten Abstand genommen. Für die wiederholte globale Berechnung bietet sie gewisse Vorteile an. Eine Umsetzung mittels Baumstruktur erfolgt, bei der angenommen wird, dass immer genau 2^n viele Prozesse existieren.

A. Anforderungen

Nachrichtenverkehr: Kein Prozess soll mehr als k Nachrichten in einem Zeitschritt gleichzeitig empfangen. Ziel ist es, spätere physikalische Eigenschaften des Netzwerkes abbilden zu können.

Hohe Parallelität: Unter Beachtung der ersten Forderung kann abgeleitet werden, dass die Kommunikation zwischen einem beliebigen Prozess und dem wechselnden Root-Prozess in $log_k(N)$ möglich ist. Die allgemeine Anforderung an den Algorithmus ist eine Laufzeit von $\mathcal{O}(log(N))$.

Gleiche Last: Jeder Prozess soll über die Dauer die gleiche Last bezüglich Rechenaufwand, empfangener und versendeter Nachrichten haben. Weiter soll jeder Prozess auf dem gleichen Instruktionssatz arbeiten.

B. Architekturmodelle

Mit den zuvor aufgestellten Anforderungen kann auf eine Architektur rückgeschlossen werden. Gegen eine Struktur mit *zentralisiertem* Root-Prozess, zu dem alle Prozesse verbunden sind, spricht die gleichmäßige Belastung aller Prozesse. Hinzu vergehen N/k viele Zeiteinheiten, bis der Root-Prozess jede Nachricht empfangen hat.

Ein *ringbasiertes* Verfahren mit definiertem Vor- und Nachfolger bietet geringe Parallelität. Es dauert $N-1$ Schritte, bis alle Nachrichten angekommen sind.

Der *hierarchische* Ansatz eines k-adischen Baumes (siehe Abbildung 5 links) stellt eine gute Ausgangsbasis dar. Bis die Nachrichten vom den untersten bis zu den obersten Prozessen weitergeleitet sind vergeht $\mathcal{O}(log(N))$ viel Zeit. Nachteil dieser Lösung ist die nicht gleichmäßige Lastverteilung.

C. Umsetzung

Die Architektur eines k-adischen Baumes bietet eine gute Ausgangssituation. Allerdings muss auf die Fairness geachtet werden: Der Rechenaufwand des Prozesses kann von der Position im Baum abhängen. Zum Beispiel verschickt ein Prozess auf Blattebene nur seine eigene Nachricht, wogegen ein innerer Prozess auch die Nachrichten seiner Kind-Prozesse weiterleitet. Weiter können auch die Distanzen zwischen Positionen variieren. Es soll ein Algorithmus entworfen werden, bei dem jeder Prozess genau einmal jede Position in N Durchläufen einnimmt.

Für einen Baum in Abhängigkeit seiner Blätteranzahl soll eine Abbildungsfunktion definiert werden. An der Position p sei zu dem Zeitpunkt t der Prozess P. Die Abbildungsfunktion $next$ legt fest an welcher Position der Prozess zum Zeitpunkt $t+1$ ist. Somit hat jede Position p eine eindeutige Folgeposition p'; Es dürfen nie zwei verschiedene Prozesse p als Folge-Position haben. $next$ muss also eine bijektive Funktion sein. Weiter soll $next$ die Anforderung erfüllen, dass jeder Prozess in N Durchläufen jede Position genau einmal einnimmt.

Ein k-adischer Baum liefert nach [GG94] eine Struktur mit der die Anforderungen an $next$ erfüllt werden können. Die Nummerierung der Positionen im Baum wird einmalig durch einen in-order Baumdurchlauf [CLR04, Seite 256] festgelegt. Semantisch soll die Folgeposition wie folgt bestimmt sein.

1) Innere Prozesse sollen von der Iteration i nach $i+1$ von der Ebene j nach $j+1$ wandern.
2) End-Prozesse im linken Teilbaum der Wurzel sollen End-Prozesse im rechten Teilbaum ihres Vaterprozesses werden.
3) End-Prozesse im rechten Teilbaum der Wurzel sollen innere Prozesse im rechten Teilbaum werden. Der Prozess x an der Position 2^n-1 (unten rechts) soll neuer Root-Prozess werden (Position 2^{n-1}).

Bei einem k-adischen Baum kann die erste Aufgabe erfüllt werden, indem jeder innere Prozesse P an die Stelle $P/2$ geschickt wird. Die Division mit zwei ist möglich, da alle inneren Positionen gerade sind (siehe Abbildung 5 links).

Die linken End-Prozesse sollen rechte End-Prozesse werden. Bei den linken End-Prozessen ist das höchstwertigste Bit immer 0, bei den rechten End-Prozessen ist es 1. Es bietet sich an, dass so lange nach links geschoben wird bis das höchstewertigste Bit 1 ist, was einer Multiplikation mit $2^{lead0(x)}$ (*lead0* ist die Anzahl der führenden Nullen) entspricht. Die Multiplikation ergibt eine gerade Zahl, eine Addition von eins ist notwendig.

Die rechten Prozesse sollen aufsteigen. Jeder Prozess hat genau eine nachfolgende Position im rechten Teilbaum die oberhalb seiner Schicht liegt. Das ist möglich, indem die Position um eins erhöht wird.

Das beschriebene Verhalten ist in Algorithm 6 dargestellt. Der Algorithmus ist erstmals in [GG94] veröffentlicht worden. Vorteil des Vorgehens ist, dass der linke innere Teilbaum zusammenhängend bleibt. Nur End- und Root-Prozess müssen neu gesetzt werden. Die Korrektheit für das Vorgehen ist ebenfalls [GG94] zu entnehmen.

Algorithm 6: Zu sehen ist die Umsetzung der *next*-Funktion, welche erstmals in [GG94] vorgestellt wurde.

```
next(x)
  if even(x) then                       // Type I
     x' = x/2;
  if odd(x) ∧ x < 2^(n-1) then          // Type II
     x' = x · 2^lead0(x) + 1;
  if odd(x) ∧ (x > 2^(n-1)) then        // Type III
     x' = x + 1;
     if x' = N+1 then                   // special case
        x' = x'/2;
```

In Abbildung 5 ist dargestellt, wie bei einem Netzwerk mit 15 Prozessen die Positionen neu berechnet werden.

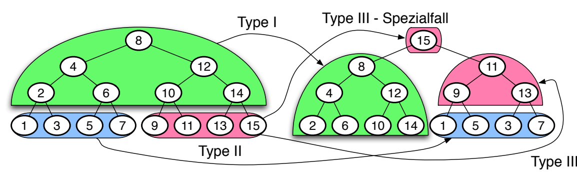


Abbildung 5. Dargestellt ist links ein k-adischer Baum mit $k = 2$ und 15 Prozessen. Der rechte Baum stellt den Wechsel der Prozess-Positionen nach einer Iteration dar. Die inneren Prozesse der Schicht i gehen in die Schicht $i + 1$. Die linken End-Prozesse wandern an die Stelle der rechten End-Prozesse. Die rechten End-Prozesse steigen wieder auf. Der End-Prozess an der Positon 15 wird neuer Root-Prozess. Die Kantenbeschriftung gibt an, welcher Fall aus Algorithm 6 dafür verantwortlich ist.

V. Zusammenfassung

Für die globale Berechnung ist ein erster Ansatz das Broadcast- und Convergecast-Verfahren. Hierbei wird jeweils ein dedizierter Root-Prozess benannt. Mittels Spannbaum können alle Prozesse Q an ihren Vater-Prozess P den Zustand schicken. Hat P den Zustand aller direkten Kinder wird dieser mit seinem eigenen Zustand weiter nach oben geleitet. Der Root-Prozess berechnet anschließend den globalen Zustand und verschickt ihn an alle seine Kinder, die ihn entsprechend weiterleiten.

Die Bestimmung eines Root-Prozesses in einem Spannbaum ist mittels ID möglich. Jeder Prozess P nimmt zu Beginn an er sei der Root-Prozess. Bei dem Empfangen einer Nachricht mit höherer ID wird der Sender Q neuer Vater-Prozess von P.

Die Berechnung der globalen Funktion ist auch ohne Spannbaum möglich. Das Prinzip ist für die Berechnung von Routing-Tabellen vorgestellt, was später für eine beliebige globale Funktion erweitert wurde. Die Idee dahinter ist, dass jeder Prozess zu Beginn an alle direkten Nachbarn eine Nachricht mit seiner eigenen ID und Zustand schickt. Jeder Prozess, der eine Nachricht erhält, leitet diese an alle direkten Nachbarn, bis auf den Sender, weiter. So propagieren sich alle Nachrichten zu jedem Knoten durch.

Eine wiederholte globale Berechnung wird genutzt, wenn z.B. ein Deadlock erkannt werden soll oder eine verteilte Datensammlung zusammengefasst wird. Dabei soll die Last in N Durchläufen gleichmäßig auf alle Knoten verteilt werden. Eine Lösung ist, dass jeder Prozess jede Position in dem Netzwerk genau einmal einnimmt. Ein Prozess-Netzwerk in Form von einem vollständigen k-adischen Baum bietet sich an. Durch eine Abbildungsfunktion muss zu Beginn für jede Position p einmalig berechnet werden, was die Folgeposition p' ist.

Literatur

[BKR88] Jean-Claude Bermond, Jean-Claude König, and Michel Raynal. General and efficient decentralized consensus protocols. In *Proceedings of the 2nd International Workshop on Distributed Algorithms*, pages 41–56, London, UK, 1988. Springer-Verlag.

[CLR04] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Algorithmen - Eine Einführung*. Oldenbourg Verlag München Wien, 2004.

[GG94] Vijay K. Garg and Joydeep Ghosh. Repeated computation of global functions in a distributed environment. *IEEE Transactions on Parallel and Distributed Systems*, 5, 1994.

[GHS83] Robert G. Gallager, Pierre A. Humblet, and Philip M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Program. Lang. Syst.*, 5(1):66–77, 1983.

[Jal94] Pankaj Jalote. *Fault Tolerance in Distributed Systems*. Prentice Hall, US edition, 1994.

Synchronisierer

Björn Wolff
Fakultät II: Department für Informatik
Carl von Ossietzky Universität
Oldenburg
bjoern.wolff@informatik.uni-oldenburg.de

***Zusammenfassung*—In dieser Ausarbeitung im Rahmen des Seminars „Konzepte und Methoden in verteilten Systemen“ werden die von Baruch Awerbuch im Jahre 1985 vorgestellten Synchronisierer beschrieben. Diese ermöglichen es synchrone Netzwerke durch asynchrone Netze zu simulieren. Dazu werden auch die graphentheoretischen Modelle erläutert, die für die Umsetzung der Synchronisierer notwendig sind.**

***Keywords*-Synchronisierer; Graphen; Bäume; Netzwerke**

I. EINLEITUNG

Das Entwerfen von verteilten Algorithmen ist weitaus einfacher, wenn davon ausgegangen werden kann, dass das zur Verfügung gestellte Netzwerk synchron arbeitet. Synchrone Algorithmen sind den asynchronen Algorithmen vor allem in der Entwicklung und Analyse überlegen. Sie sind meist weniger komplex, und auch das Verhalten ist leichter nachzuvollziehen. Somit stellt sich die Frage, ob es möglich ist mit asynchronen Netzwerken synchrone Netzwerke zu simulieren. In asynchronen Netzwerken ist es möglich, dass das Senden und Empfangen von Nachrichten zeitlich versetzt stattfindet, ohne das dieses Einfluss auf die korrekte Arbeitsweise hat. Dies würde in synchronen Netzwerken zu Problemen führen. Im Jahre 1985 stellte Baruch Awerbuch in einem Artikel [3] das Konzept der *Synchronisierer* vor. Diese Synchronisierer sind tatsächlich in der Lage, asynchronen Netzwerken ein synchrones Verhalten zu ermöglichen. Dies ist jedoch nur unter der Voraussetzung möglich, dass im Netzwerk keine Fehler oder Ausfälle auftreten.

In dieser Ausarbeitung werden zunächst Grundlagen zur graphentheoretischen Struktur von Netzwerken angesprochen. Aufbauend darauf werden verschiedene Synchronisierer-Algorithmen vorgestellt. Als Einstieg dient ein einfacher Synchronisierer, der die grundsätzlichen Eigenschaften erfüllt, um ein synchrones Netzwerk zu simulieren. Danach werden die von Awerbuch eingeführten α-, β- und γ-Synchronisierer behandelt. Dabei werden die jeweiligen Nachrichten- und Zeitkomplexitäten der unterschiedlichen Synchronisierer dargestellt. Nach einem kurzen Anwendungsbeispiel folgt abschließend das Fazit.

Als Quellen für diese Ausarbeitung dienten der Artikel von Baruch Awerbuch [3], das Buch von Vijay Garg [1] und das Buch von Ulrich Knauer [2].

II. GRUNDLAGEN

Im weiteren Verlauf der Ausarbeitung werden Netzwerke als graphentheoretische Strukturen dargestellt. Diese Strukturen sollen im folgenden erläutert werden. Zunächst werden dabei *Graphen* beschrieben, wobei speziell auf ungerichtete Graphen eingegangen wird. Um eine gewisse Reihenfolge innerhalb der Graphen zu erlangen, werden im Anschluss *Bäume* angesprochen, die es ermöglichen *Pfade*, in einem Graphen anzugeben. Die Definitionen zu Graphen und Bäumen orientieren sich an dem Buch von Ulrich Knauer [2].

A. Graphen

Ein Tupel $G = (V, E)$, wobei V eine nichtleere Menge von *Knoten* und E eine Menge von *Kanten* beschreibt, heißt

- *gerichteter Graph*, falls E eine Teilmenge von $V \times V$,
- *ungerichteter Graph*, falls E eine Teilmenge aller zweielementigen Teilmengen von V

ist. Wie der Name es schon verrät, besitzen die Kanten in einem ungerichteten Graphen keine Richtung, so dass die Kanten in beide Richtungen genutzt werden können.

Eine Abfolge von Knoten $v_0, ..., v_n$, wobei die jeweiligen Knoten v_i und v_{i+1} durch eine Kante verbunden sind, nennt man *Weg*. Ein solcher Weg heißt *Kreis*, wenn gilt $v_0 = v_n$. Die Länge des kürzesten Weges heißt *Abstand* von v_i nach v_{i+1}. Der *Durchmesser* eines Graphen gibt den größtmöglichen Abstand zwischen zwei Knoten an.

Ein ungerichteter Graph heißt *zusammenhängend*, wenn je zwei seiner Knoten durch einen Weg verbunden sind. Somit ist es möglich, dass von allen Knoten im Graphen ein Weg zu jedem anderen Knoten gefunden werden kann.

B. Bäume

Ein Graph ohne Kreise heißt *Wald* und ein zusammenhängender Wald heißt *Baum*. Ist $G = (V, E)$ ein zusammenhängender Graph, so ist der entsprechende Baum $T = (V, L)$ mit $L \subseteq E$ ein *Spannbaum* von G.

Besitzt ein Knoten des Baums keinen Folgeknoten mehr, so heißt dieser *Blatt*. Verfolgt man die eingehende Kante eines Knotens, so gelangt man zum sogenannten Vaterknoten. Im späteren Verlauf wird der Vaterknoten auch einfach als *Vater* bezeichnet.

Ein Baum, dessen Kanten eine Richtung besitzen und somit ein Knoten als Ursprungspunkt – beziehungsweise

Wurzel – bestimmt werden kann, nennt sich gewurzelter Baum. In dieser Ausarbeitung werden nur gewurzelte Bäume betrachtet, in denen jeder Knoten des Baums nur durch einen festgelegten Weg von der Wurzel aus zu erreichen ist.

Mit einem solchen Spannbaum ist es nun möglich, durch Bestimmung einer Wurzel, eine bestimmte Abfolge von Knoten in einem Graphen festzulegen.

III. Synchronisierer

Im Folgenden werden nun die aus Abschnitt II bekannten Strukturen benutzt um Netzwerkmodelle zu beschreiben.

Ein *asynchrones* Netzwerk ist ein Punkt-zu-Punkt-Netzwerk, welches als ungerichteter, zusammenhängender Graph $G = (V, E)$ beschrieben werden kann. Dabei werden Prozessoren des Netzwerks als Knoten dargestellt und die Kanten beschreiben die in beide Richtungen bestehenden Kommunikationskanäle zwischen den Prozessoren. Es existiert in diesem Modell kein gemeinsamer Speicher und jeder Knoten/Prozessor kann eindeutig identifiziert werden. Zusätzlich ist jeder Prozessor in der Lage, Nachrichten von seinen Nachbarn zu empfangen, lokale Berechnungen durchzuführen, sowie Nachrichten an seine Nachbarn zu verschicken. Man geht davon aus, dass all dies in vernachlässigbarer Zeit möglich ist. Alle Nachrichten besitzen eine feste Länge und haben einen festen Informationsgehalt. Die Übertragung einer Nachricht erfolgt innerhalb einer endlichen, jedoch nicht vorhersehbaren Zeit.

In einem *synchronen* Netzwerk ist es hingegen nur möglich, die Nachrichten innerhalb einer Zeiteinheit – eines sogenannten *pulse* – zu versenden. Auch ist die Anzahl der Nachrichten, die während eines *pulse* versandt werden darf, auf *eins* beschränkt. Die Übertragungsverzögerung einer Kante ist maximal so lang, wie eine Zeiteinheit der globalen Uhr.

Um eine Möglichkeit zu finden, die Komplexität der nachfolgenden Algorithmen zu bestimmen, werden die folgenden Bezeichnungen eingeführt. Die *Nachrichtenkomplexität* M gibt an, wie viele Nachrichten ein Algorithmus bei seiner Ausführung benötigt, wohingegen die Zeitkomplexität T untersucht, wie viele *pulses* zur Ausführung eines synchronen Algorithmus benötigt werden. Bei einem asynchronen Algorithmus gibt die Zeitkomplexität T die Zeit an, die im *worst-case* gebraucht wird. Dabei wird davon ausgegangen, dass die Verzögerung eines Nachrichtenaustausches, sowie die Verzögerung zwischen zwei aufeinanderfolgende Nachrichten einer Kante höchstens eine Zeiteinheit dauern. Dies dient allerdings nur der Evaluation der Leistung des Algorithmus; prinzipiell sollten die Algorithmen auch mit beliebigen Verzögerungen arbeiten können.

In Kommunikationsnetzwerken lässt sich häufig ein *trade-off* zwischen Nachrichten- und Zeitkomplexität beobachten.

Das Ziel eines *Synchronisierers* ist es, die Möglichkeit zu schaffen, dass synchrone Algorithmen in einem asynchronen Netzwerk benutzt werden können. Hierzu steuert der Synchronisierer den *pulse* jedes Knotens im Netzwerk. Der *pulse* wird dabei als Zähler genutzt, so dass sichergestellt werden kann, dass jede Nachricht, die in *pulse* i verschickt wurde auch vor *pulse* $i + 1$ empfangen wird. Somit wird erst ein neuer *pulse* generiert, wenn alle Nachrichten des aktuellen *pulse* von allen Nachbarn empfangen wurden. Um dies zu erreichen, ist es notwendig, zusätzliche Nachrichten über das Netzwerk zu versenden, die dafür sorgen, dass die Synchronisierung der Knoten funktioniert.

Damit Aussagen über die Komplexität getroffen werden können, muss der Sachverhalt zunächst abstrahiert werden: Sei ν ein beliebiger Synchronisierer. Die Nachrichten und die Zeit, die dieser Synchronisierer pro *pulse* benötigt, werden mit $M(\nu)$ und $T(\nu)$ bezeichnet. Viele Synchronisierer besitzen eine Initialisierungsphase, in der zunächst das Netzwerk vorbereitet werden muss. Beispielsweise gibt es Synchronisierer, die Spannbäume innerhalb des Graphen benötigen, welche zunächst bestimmt werden müssen. Dies wird im Folgenden mit $M_{init}(\nu)$ und $T_{init}(\nu)$ bezeichnet. Wenn nun ein synchroner Algorithmus T_{synch} Runden und M_{synch} Nachrichten für die Ausführung benötigt, so gilt für den asynchronen Fall

$$M_{asynch} = M_{synch} + T_{synch} \cdot M(\nu) + M_{init}(\nu),$$
$$T_{asynch} = T_{synch} \cdot T(\nu) + T_{init}(\nu).$$

Ein Synchronisierer wird laut Awerbuch als effizient angesehen, wenn die Parameter $M(\nu)$, $T(\nu)$, $M_{init}(\nu)$ und $T_{init}(\nu)$ „klein genug“ sind. Dies wird im späteren Verlauf noch deutlich. Besonders die Parameter für $M(\nu)$ und $T(\nu)$ sind entscheidend, da diese einen großen Einfluss auf die Komplexität besitzen.

Für die Modellierung ist es wichtig, dass die Prozessoren niemals versagen, beziehungsweise keine Fehler auftreten können. Es ist nicht möglich, synchrone Netzwerke in einem asynchronen Netzwerk zu simulieren, wenn Prozesse fehlschlagen können. Außerdem müssen die Kanäle/Kanten zuverlässig sein und keine Störungen zulassen.

In den folgenden Unterkapiteln wird zunächst eine einfache Variante eines Synchronisierers vorgestellt, an Hand derer der *α-Synchronisierer* erläutert wird. Danach wird der *β-Synchronisierer* eingeführt. Diese beiden einfacheren Synchronisierer bilden die Basis für den *γ-Synchronisierer*.

Im Folgenden werden die Notationen

N: Anzahl der Knoten in einem Netzwerk,
E: Anzahl der Kanten in einem Netzwerk,
D: Durchmesser des Netzwerks

zur Bestimmung der Komplexität verwendet.

A. Ein einfacher Synchronisierer

Im einfachsten Fall kann ein Synchroniserer ϵ konstruiert werden, indem die Prozessoren pro *pulse* nur eine Nachricht an alle Nachbarn verschicken dürfen. Dies sorgt dafür, dass jeder Prozessor auch nur auf eine Nachricht pro Nachbar

warten muss, bevor ein nächster *pulse* starten kann. Hierfür ist es allerdings notwendig eine Art „Standardnachricht“ einzuführen, die versandt wird, wenn ein Prozess P_i im synchronen Netzwerk keine Nachricht an seinen Nachbarn P_j versenden würde. Sollte P_i im synchronen Netzwerk mehrere Nachrichten an P_j verschicken, so müssen diese im asynchronen Fall zusammengefasst und als eine Nachricht übermittelt werden.

Der Synchronisierer ϵ kann auf eine Nachricht pro Nachbar in *pulse* i warten und im Anschluss den nächsten *pulse* $i+1$ starten. Im Buch von Vijay Garg [1] wird dazu folgender Algorithmus für den Prozessor/Knoten P_j angegeben:

Algorithm 1 Implementierung eines einfachen Synchronisierer [1, S. 248]

P_j::
var
pulse: Integer mit Startwert 0;
Runde i:
pulse:= $pulse + 1$;
simuliere Runde i des synchronen Algorithmus;
sende Nachrichten an alle Nachbarn mit *pulse*;
warte auf genau eine Nachricht von jedem Nachbarn mit *pulse*=i;

Der in Algorithm 1 angegebene Algorithmus sorgt dafür, dass ein Prozessor in *pulse* i nur Nachrichten empfängt, die in *pulse* i gesendet wurden. Sollte ein Prozessor eine Nachricht aus *pulse* $i+1$ erhalten, so wird diese gepuffert und auf eine Nachricht aus *pulse* i gewartet. Da es innerhalb eines asynchronen Netzwerks mit einem einfachen Synchronisierer nur möglich ist, Nachrichten mit *pulse* i und *pulse* $i+1$ zu erhalten, kann der *pulse* durch ein Bit mit $i \pmod 2$ implementiert werden.

Zu erkennen ist, dass dieser Algorithmus keine spezielle Initialisierungsphase benötigt, da sobald der erste *pulse* gestartet wird, innerhalb von D Zeiteinheiten alle Prozessoren *pulse* 1 starten. Somit kann die Komplexität der Initialisierung bestimmt werden als

$$M_{init}(\epsilon) = 0; \quad T_{init}(\epsilon) = D.$$

Da für jeden *pulse* das Senden einer Nachricht in beide Kantenrichtungen nötig ist, erhält man

$$M(\epsilon) = 2E; \quad T(\epsilon) = 1.$$

B. Der α-Synchronisierer

Aufbauend auf dem einfachen Synchronisierer lässt sich der α - Synchronisierer definieren. Ein wichtiges Konzept, welches mit dem α - Synchronisierer eingeführt wird, ist das der *safety*. Ein Prozessor P ist *safe* in *pulse* i, wenn er weiß, dass alle seine Nachtrichten aus *pulse* i angekommen sind. Der α - Synchronisierer generiert erst dann einen neuen *pulse*, wenn bekannt ist, dass alle seine Nachbarn *safe* sind. Somit ist sicher gestellt, dass alle Nachrichten eines *pulse* übermittelt wurden.

Um den α-Synchronisierer implementieren zu können, ist es nötig ein Acknowledgement-Verfahren zu integrieren, so dass alle Nachrichten vom Empfängerknoten quittiert werden müssen. Damit weiß jeder einzelne Prozessor, wann er innerhalb eines *pulses safe* ist.

Mit diesem Acknowledgement-Verfahren ist es nun möglich, erst dann einen neuen *pulse* zu generieren, wenn alle Nachbarn gemeldet haben, dass sie *safe* sind.

Somit lässt sich die Nachrichten- und Zeitkomplexität des α-Synchronisierers als

$$M(\alpha) = \mathcal{O}(E); \quad T(\alpha) = \mathcal{O}(1)$$

festhalten. Da nun zusätzlich Nachrichten zur Quittierung eingeführt wurden, verändert sich die Nachrichtenkomplexität der Initialisierung, wohingegen die Zeit zur Initialisierung gleich bleibt. Somit gilt

$$M_{init}(\alpha) = D; \quad T_{init}(\alpha) = D.$$

C. Der β-Synchronisierer

Dieser Synchronisierer benötigt eine Initialisierungsphase, in welcher zunächst ein *leader* s des Netzwerks festgelegt und ein Spannbaum mit Wurzel s erstellt wird. Ist ein *leader* gefunden, ist dieser für die Regelung der *pulses* zuständig. Nachdem ein *pulse* gestartet wurde, erfährt s – nach einer gewissen Zeit – das alle Knoten im Netzwerk *safe* sind. Sobald dies geschehen ist, sendet er einen *broadcast* über den Spannbaum und benachrichtigt damit alle Knoten, dass ein neuer *pulse* gestartet werden kann. Die Rückrichtung eines *broadcast*, also die von den Blättern ausgehende Benachrichtigung der Wurzel, wird im folgenden als *convergecast* bezeichnet. Es wird in diesem Modell vorausgesetzt, dass jeder Knoten eine Benachrichtigung an den jeweiligen Vater versendet, sobald alle Knoten/Blätter, die unter ihm liegen, *safe* sind. Somit weiß der *leader* immer, wann es möglich ist einen neuen *pulse* einzuleiten.

Die Erstellung des in der Initialisierungsphase benötigten Spannbaums ist mit $\mathcal{O}(N \log N + E)$ Nachrichten und in $\mathcal{O}(N)$ Zeiteinheiten möglich. Während der Durchführung eines *pulse* werden die Nachrichten nur innerhalb des Spannbaums versandt, so dass die Nachrichtenkomplexität in $\mathcal{O}(N)$ liegt. Die Zeitkomplexität ist abhängig von der Tiefe des Spannbaums, die im ungünstigsten Fall auch in $\mathcal{O}(N)$ liegt. Also ist die Komplexität des β-Synchronisierers wie folgt:

$$M(\beta) = \mathcal{O}(N); \qquad T(\beta) = \mathcal{O}(N)$$
$$M_{init}(\beta) = \mathcal{O}(N \log N + E); \quad T_{init}(\beta) = \mathcal{O}(N).$$

D. Der γ-Synchronisierer

Im nächsten Unterkapitel wird der γ-Synchronisierer beschrieben, der eine Verallgemeinerung der beiden vorherigen Synchronisierer darstellt. Der γ-Synchronisierer benötigt eine Initialisierungsphase, in der das zugrunde liegende Netzwerk in *Cluster* eingeteilt wird. Diese Partitionierung kann als eine Aufteilung des Graphens (V, E) in mehrere spannende Wälder angesehen werden. Jeder Teilbaum des Waldes ist somit ein Cluster von Knoten und wird im Folgenden als *intracluster-Baum* bezeichnet. Zur Verbindung der benachbarten Cluster werden sogenannte *preferred links* eingeführt. Über diese speziellen Kanten können die verschiedenen Cluster miteinander kommunizieren. Analog zum β-Synchroniserer, wird innerhalb eines Clusters ein *leader* bestimmt. Dieser koordiniert den entsprechenden intracluster-Baum und gibt an, wann ein *pulse* gestartet werden kann. Auch dieser Synchronisierer benötigt die *safe*-Eigenschaft, so dass ein Cluster als *safe* bezeichnet wird, wenn alle seine Knoten *safe* sind.

Der Algorithmus des γ-Synchronisierers teilt sich in zwei Phasen auf. Während der ersten Phase wird das Prinzip des β-Synchronisierers innerhalb der Cluster benutzt. Sollte der *leader* eines Clusters erfahren, dass alle Knoten seines Clusters *safe* sind, so wird ein *broadcast* an alle Knoten des Clusters, sowie an die *leader* der benachbarten Cluster gestartet. Danach wird die zweite Phase gestartet. In dieser Phase wird darauf gewartet, dass alle benachbarten Cluster *safe* sind. Dieses wird mit der Vorgehensweise des α-Synchronisierers zwischen den Clustern kommuniziert. Sind alle Cluster *safe*, so kann ein neuer *pulse* gestartet werden.

Im Folgenden wird der Verlauf noch einmal genauer beschrieben. Damit ein *pulse* überhaupt gestartet werden kann, muss der *leader* eines Clusters einen *broadcast* senden, um die Knoten seines Clusters zu informieren, dass der nächste *pulse* gestartet werden muss. Nachdem alle Aufgaben eines *pulses* abgearbeitet wurden, begibt sich der Knoten in die erste Phase des γ-Synchronisierers und meldet sich mit Hilfe eines *convergecast* als *safe*. Dies startet bei den Blättern und wird über die jeweiligen Väter – den intracluster-Baum entlang nach oben – an den *leader* weitergegeben. Wenn alle Knoten des Clusters *safe* sind, sendet der *leader* einen *broadcast* um benachbarte Cluster zu informieren. Diese *broadcast*-Nachricht wird an alle Knoten eines Clusters geschickt, wodurch auch die *preferred links* informiert werden.

Nun wird die zweite Phase eingeleitet. In dieser Phase verhält sich der Cluster wie ein α-Synchronisierer, nur müssen die Knoten zwei Nachrichten senden und zwar

1) ob sie bereit sind in den nächsten *pulse* überzugehen und
2) ob die Nachbarcluster *safe* sind.

Die Information über die benachbarten Cluster wird durch die *preferred links* in den Cluster eingeführt. Sind alle Knoten bereit und hat sich die Information über die benachbarten Cluster verbreitet, so kann der *leader* den nächsten *pulse* einleiten. Hierzu wird erneut ein *broadcast* an alle Knoten geschickt.

Um die Komplexitäten bestimmen zu können, müssen erst neue Bezeichnungen eingeführt werden. Mit E_p wird die Menge aller Kanten, inklusive der *preferred links*, einer Partition P bezeichnet. Die maximale Höhe eines Baums von P wird mit H_p bezeichnet. Da während der Ausführung des γ-Synchronisierers höchstens vier Nachrichten über E_p versandt werden, gilt $M(\gamma) = \mathcal{O}(E_p)$. Damit ein Cluster feststellen kann ob er *safe* ist, benötigt dieser $\mathcal{O}(H_p)$ Zeiteinheiten und weitere $\mathcal{O}(H_p)$ Zeiteinheiten zur Bestätigung, dass die Nachbarcluster *safe* sind, so dass $T(\gamma) = \mathcal{O}(H_p)$ gilt. Wie zu erkennen ist, sind diese Parameter ausschließlich von der Struktur der Wälder abhängig, was dazu anregt, Partitionen zu finden, die für die Parameter E_p und H_p möglichst kleine Werte liefern. Es ist relativ einfach Partitionen zu finden, die dafür sorgen, dass einer der beiden Parameter klein ist. Zum Beispiel, wenn jeder Knoten einen Cluster bildet, dann gilt $H_p = 0$ und $E_p = E$. Des Weiteren könnte man den ganzen Graphen als einen Cluster nehmen, wodurch man $E_p = E$ und $H_p = \mathcal{O}(D)$ erhält. Mit diesen Partitionen erhält man also die Komplexitäten des α-Synchronisierers beziehungsweise des β-Synchronisierers.

Mit dem folgenden Satz aus [1] lässt sich zeigen, dass jeder Graph in Cluster aufgeteilt werden kann, so dass es auch immer zu einem *trade-off* zwischen E_p und H_p kommt.

Satz 1: Für jedes k, mit $2 \leq k < N$, existiert eine Partitionierung P, so dass $E_p \leq kN$ und $H_p \leq \frac{\log N}{\log k}$ gilt.

Beweis: In diesem Beweis wird eine explizite Konstruktionsvorschrift für die Partitionierung gegeben. In dieser Vorschrift werden nacheinander neue Cluster hinzugefügt. Angenommen, es wurden bereits r Cluster erstellt und es existieren weitere Knoten, die nicht Teil eines Clusters sind. Dann wird der nächste Cluster wie folgt erstellt:

Jeder Cluster C beinhaltet mehreren Stufen. Für die erste Stufe wird ein Knoten ausgesucht, der noch nicht Teil eines Clusters ist. Angenommen, dass i Stufen ($i \geq 1$) des Clusters C schon erstellt wurden. Sei S die Menge der Nachbarn des Knotens in Stufe i, die noch nicht Teil eines Clusters sind. Ist die Mächtigkeit von S mindestens $(k-1)$ mal so groß wie die Mächtigkeit von C, dann wird S als nächste Stufe des Clusters C hinzugefügt. Andernfalls ist die Konstruktion von C beendet. Nun können H_p und E_p für die Vorschrift bestimmt werden. Da jeder Cluster mit Stufe i mindestens k^{i-1} Knoten besitzt, folgt, dass H_p maximal $\frac{\log N}{\log k}$ groß ist. Der Parameter E_p besteht aus zwei Komponenten: Kanten und *preferred links*. Es existieren maximal N Kanten im Baum. Um die *preferred links* zu zählen, werden *preferred links* zwischen zwei Clustern und dem ersten konstruierten Cluster eingeführt. Da für einen Cluster C nach der Konstruktion gilt, dass er maximal $(k-1)|C|$ Nachbarknoten besitzt, kann

dieser maximal $(k-1)|C|$ *preferred links* besitzen. Werden alle Kanten zusammengezählt ergibt sich $(k-1)N$.

■

Als Nächstes wird der von Awerbuch [3] vorgestellte Algorithmus für den γ-Synchronisierer vorgestellt. Dieser kann auch zum Verständnis der beiden vorherigen Synchronisierer beitragen, da diese vom γ-Synchronisierer in seinen zwei Phasen genutzt werden. Der Algorithmus beschreibt die Vorgehensweise eines jeden Knoten i im Netzwerk, wenn dieser entsprechende Nachrichten von seinen Nachbarn erhält. Alle Nachrichten und Variablen besitzen eine Zuordnung zu der entsprechenden *pulse*-Nummer, was in der Beschreibung des Algorithmus nicht explizit angegeben wird. Der *pulse* kann, wie bereits erwähnt, durch ein Bit dargestellt werden, da jeweils nur *pulse* i und *pulse* $i+1$ relevant sind. Zur Veranschaulichung kann Abbildung 1 genutzt werden, die einen beispielhaften Aufbau eines Clusters zeigt.

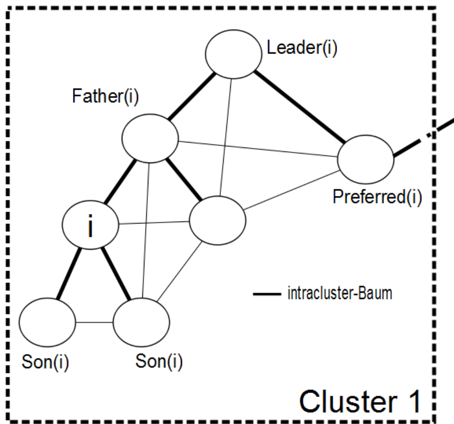


Abbildung 1. Beispielcluster zum γ-Synchronisierer

Nachrichten des Algorithmus:

ACK Acknowledgement, wird versandt, wenn eine Nachricht angekommen ist.
PULSE Nachricht, die den nächsten *pulse* auslöst.
SAFE Nachricht, die von Knoten/Blättern an die entsprechenden Väter versandt wird, wenn sie *safe* sind.
CLUSTER_SAFE Nachricht, die ein Knoten an seine Nachbarn und über *preferred links* versendet, wenn sein Cluster *safe* ist.
READY Nachricht, die ein Knoten an seinen Vater versendet, wenn alle Cluster, die über *preferred links* mit seinen Vorgängern verbunden sind, *safe sind.*

Variablen, die vom Algorithmus benutzt werden:

Safe(i, q) Ein Flag, für alle $q \in$ Sons(i), wobei 1 gesetzt ist, wenn die SAFE-Nachricht von q innerhalb des aktuellen *pulse* empfangen wurde. (Safe(i,q) = 0 oder 1).
Ready(i, q) Ein Flag, für alle $q \in$ Sons(i), wobei 1 gesetzt ist, wenn die READY-Nachricht von q innerhalb des aktuellen *pulse* empfangen wurde.(Ready(i,q) = 0 oder 1).
Dif(i, j) Der Zähler für alle $j \in$ Neighbors(i). Er zeigt die Differenz zwischen den versandten Nachrichten von i nach j und den entsprechenden Acknowledgements ACK von j nach i an. Am Anfang eines *pulse* gilt Dif(i,j) = 0. (Dif(i,j) = 0, 1, 2, ...).
cluster_safe(i, j) Ein Flag, für alle $j \in$ Sons(i) $\cup$ Father(i), wobei 1 gesetzt ist, wenn die CLUSTER_SAFE-Nachricht von j innerhalb des aktuellen *pulse* empfangen wurde.(cluster_safe(i,q) = 0 oder 1).

Methoden, die vom Algorithmus benutzt werden:

Safe_Propagation Methode, die *convergecasts* für die SAFE-Nachrichten durchführt.
Ready_Propagation Methode, die *convergecasts* für die READY-Nachrichten durchführt.

Der Algorithmus für den Knoten i:

```
for PULSE-Nachricht do
   Führe den nächsten pulse aus
   for all q ∈ Sons(i) do
      safe(i, j) ← 0 {Warten auf SAFE von q}
      sende PULSE an q
   end for
   for all j ∈ Neighbors(i) do
      Dif(i, k) ← 0
   end for
   for all k ∈ Preferred(i) do
      cluster_safe(i, k) ← 0
   end for
end for

for Gesendete Nachricht von i nach j do
   Dif(i, j) ← Dif(i, j) + 1
end for

for Empfangene Nachricht für i ausgehend von j do
   Sende ACK an j
end for

for ACK von j do
   Dif(i, j) ← Dif(i, j) - 1
   Rufe Safe_Propagation auf
end for

if Aktionen eines bestimmten pulse abgearbeitet then
   Rufe Safe_Propagation auf
end if

{Phase 1: siehe β-Synchronisierer}

Safe_Propagation: Methode
{Wird aufgerufen, sobald die Möglichkeit besteht, dass der Knoten i
und alle seine Vorgänger safe sind. Wird die Methode aufgerufen, wird
eine SAFE-Nachricht an den Vater gesendet}
if Dif(i, j) = 0 then
   for all j ∈ Neighbors(i) und safe(i, q) = 1 do
      for all q ∈ Sons(i) do
         if Leader(i) ≠ i then
            Sende SAFE an Father(i)
         else
            Sende CLUSTER_SAFE an i {Der leader i erfährt, dass
            sein Cluster safe ist und startet den broadcast für die
            CLUSTER_SAFE-Nachricht}
         end if
      end for
   end for
end if

for SAFE von q do
   safe(i, q) ← 1
   Rufe Safe_Propagation auf
end for

for CLUSTER_SAFE-Nachricht von j do
   if j ∈ Preferred(i) then
      cluster_safe(i, j) ← 1 {Der Cluster von j ist safe}
   end if
   if j ∈ Father(i) then
```

```
        {Der Cluster von i ist safe}
        for all q ∈ Sons(i) do
           Sende CLUSTER_SAFE an q
           ready(i, q) ← 0 {Warten auf READY von q}
        end for
        for all k ∈ Preferred(i) do
           Sende CLUSTER_SAFE an k {Informieren des Nachbarclusters, dass eigener Cluster safe ist}
        end for
     end if

     {Phase 2: siehe α-Synchronisierer}

     Rufe Ready_Propagation auf
  end for

  for READY von q do
     ready(i, q) ← 1
     Rufe Ready_Propagation auf
  end for

  Ready_Propagation: Methode
  {Wird aufgerufen, sobald die Möglichkeit besteht, dass alle Nachbarcluster mit Knoten i und dessen Vorgänger safe sind}
  if cluster_safe(i, j) = 1 then
     for all j ∈ Preferred(i) und ready(i, q) = 1 do
        for all q ∈ Sons(i) do
           if Leader(i) ≠ i then
              Sende READY an Father(i) {i ist kein leader}
           else
              Sende PULSE an i {Der leader i erfährt, dass sein Cluster, sowie alle Nachbarcluster safe sind und startet den nächsten pulse in seinem Cluster}
           end if
        end for
     end for
  end if
```

Einen entsprechenden Algorithmus zur Partitionierung lässt sich im Artikel von Awerbuch [3, S. 811] finden.

IV. Anwendungsbeispiel: Breitensuche-Algorithmus

Gegeben sei ein eindeutiger Knoten v. In diesem Beispiel soll nun ein Breitensuche-Baum erstellt werden, der als Wurzel v gesetzt hat. Ein synchroner Algorithmus ist für diese Aufgabe leicht zu implementieren. Dieser beginnt mit v (Ebene 0) und arbeitet sich nach und nach durch die nachfolgenden Knoten. Ein Knoten auf Ebene i muss dafür eine Nachricht an seine Nachbarn innerhalb des *pulse* i senden. Die Knoten, die eine Nachricht empfangen und noch keine Ebene zugeteilt bekommen haben, wählen den Vaterknoten aus und weisen sich die Ebene $i+1$ zu. Wenn ein zusammenhängender Graph vorliegt, ist klar, dass nach D *pulses* jeder Knoten einer Ebene zugeteilt ist – vorausgesetzt, dass jede Nachricht, die in *pulse* i gesendet wurde, in *pulse* $i+1$ empfangen wird (siehe Abbildung 2).

Will man dieses Verfahren mit einem asynchronen Netz simulieren, so kann jeder der bereits vorgestellten Synchronisierer genutzt werden, da alle die Voraussetzungen für eine solche *pulse*-Abfolge besitzen.

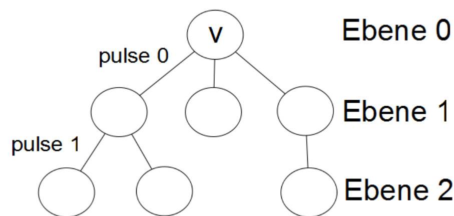


Abbildung 2. Beispielbaum zur Breitensuche mit Wurzel v

V. Fazit

In den vorherigen Kapiteln wurde erklärt, wie verschiedene Synchronisierer implementiert werden können. Dieses ist in den meisten Fällen auch relativ einfach umzusetzen. Auch zu erkennen war der *trade-off* zwischen Nachrichten- und Zeitkomplexität, der bei den jeweiligen Synchronisierern eingegangen werden muss. In seinem Artikel zeigt Awerbuch, dass der γ-Synchronisierer diesen *trade-off* bereits insofern berücksichtigt, dass je nach Anwendungsgebiet eine Regulierung – durch Wahl der Partitionen – der jeweiligen Komplexitäten möglich ist.

Somit kann der Eindruck entstehen, dass jedes synchrone Netzwerk durch ein asynchrones Netzwerk auf diese Art und Weise simuliert werden kann. Es muss klar sein, dass dies lediglich Modelle sind, die von einem fehlerfreien Netzwerk ausgehen. Es können auch nur synchrone Algorithmen simuliert werden, die darauf basieren, dass die Nachrichten in einer bestimmten Reihenfolge versandt werden (siehe Breitensuche-Algorithmus). Es ist einem asynchronen Netzwerk nicht möglich Algorithmen zu bewältigen, die zur korrekten Ausführung Zeitvorgaben besitzen. Ein Beispiel dafür ist das *k-session*-Problem. Bei diesem Problem muss jeder Prozessor k spezielle Vorgänge ausführen. Ein solcher Vorgang wird *flash* genannt. Dem Prozessor steht zur Ausführung eines *flash* nur ein gewisses Zeitfenster zur Verfügung, eine sogenannte *session*. Somit muss jeder Prozessor k *sessions* abarbeiten, um alle *flash*-Vorgänge abzuarbeiten. Ein synchroner Algorithmus würde für diese Aufgabe genau k Zeiteinheiten benötigen, wohingegen ein asynchroner Algorithmus mindestens $(k-1)D$ Zeiteinheiten [1] benötigen würde. Deshalb können die Synchronisierer für diese Aufgabe nicht verwendet werden, da bei dem *k-session*-Problem eben diese Zeitbeschränkungen für die korrekte Arbeitsweise wichtig sind.

Literatur

[1] Vijay K. Garg, *Elements of distributed computing*. New York, USA: John Wiley & Sons, Inc., 2002.

[2] Ulrich Knauer, *Diskrete Strukturen - kurz gefasst*. Heidelberg, Deutschland: Spektrum, 2001.

[3] Baruch Awerbuch, *Complexity of network synchronization*. *J. ACM*, 32(4):804–823, 1985.

Distributed Shared Memory

Volker Gollücke
Modul: Fehlertoleranz in verteilten Systemen
SS 2011
Carl von Ossietzky Universität Oldenburg
volker.golluecke@uni-oldenburg.de

Abstract

In dieser Arbeit wird das Konzept des Distributed Shared Memory (DSM) *vorgestellt. Dabei werden zunächst Grundlagen beschrieben, die für das Verständnis der Arbeit wichtig sind. Diese bestehen zum einen aus der Vorstellung verteilter Systeme, sowie einer Kurzerläuterung von Distributed Shared Memory. Im Anschluss daran werden Herausforderungen, die bei der Entwicklung und Verwendung von* DSM *auftreten, vorgestellt. Der Hauptteil besteht in der Erklärung was* DSM *ist und der Erklärung der wichtigsten Grundbausteine (Konsistenz-Modelle, Synchronisationmodelle, Aktualisierungsoptionen). Zum Abschluss wird ein Fazit gegeben und eine Wertung zu* DSM *gegeben.*

1. Einführung

Verteilte Systeme haben heute eine hohe Anzahl von Ausprägungen, die von verteilten Plattformen für das Internet bis hin zu eingebetteten Steuersystemen in Geräten, Flugzeugen oder Automobilen reichen.

Verteilte Systeme sind dabei eine Sammlung voneinander unabhängiger Computer, die für den Benutzer als eine einzige Einheit erscheinen. Der Grund für die Nutzung eines verteilten Systems liegt dabei auf den Vorteilen eines Rechnerverbundes. So können anfallende Lasten auf verschiedene Rechner verlagert werden und es kann eine gleichmäßige Auslastung verschiedener Ressourcen erfolgen. Die wichtigsten Eigenschaften eines verteilten Systems sind dabei Nebenläufigkeit, Skalierbarkeit und Fehlertoleranz.[5] Bezogen auf den verteilten Speicher lassen sich noch eine Steigerung der Verlässlichkeit eines Dienstes bzw. der Verfügbarkeit eines Datums erwähnen. Auch eine mögliche Steigerung der Leistungsfähigkeiten kann durch die Verteilung der Daten erreicht werden. Durch die ständig steigenden Anforderungen an Computer, werden Mehrprozessorsysteme eine Notwendigkeit. Die Programmierung solcher Systeme verlangen jedoch einen höheren Arbeitsaufwand und Fertigkeiten im Umgang mit Mehrprozessorsystemen.[6]

Die Lücke zwischen Prozessoren und Speichergeschwindigkeit wird immer größer und dies ist der Grund warum Speichersystemorganisation eine der kritischsten Designentscheidungen geworden ist. Abhängig von der Speichersystemorganisation können Systeme mit mehreren Prozessen in zwei Gruppen unterteilt werden. *Shared Memory Systems (SMS)* und *Distributed Shared Memory Systems (DSMS)*.

In einem *SMS* gibt es einen einzigen globalen physikalischen Speicher, der für alle Prozessoren gleich zugänglich ist.

Ein *DMS* besteht aus einer Sammlung von autonomen verarbeitenden Knoten, die eine unabhängige Ablaufsteuerung ,sowie lokale Speichermodule besitzen. Hierbei wird die Kommunikation zwischen Prozessen, die auf verschiedenen Knoten laufen, durch ein *Message-Passing*-Modell durchgeführt. Diesen Systemen wird auf der einen Seite eine gute Skalierbarkeit nachgewiesen, können auf der anderen Seite aber schnell einen Overhead bei der Implementierung erzeugen. Als Mischung aus *SMS* und *DMS* gibt es noch die *Distributed Shared Memory Systems (DSMS)*, diese versuchen das Beste der zwei vorher genannten Ansätze zu vereinen.[6]

2. Was ist Distributed Shared Memory ?

Ein *Distributed Shared Memory System* (DSMS) implementiert ein *shared memory model* in einem System, welches physikalischen verteilten Speicher verwendet. Dieser Ansatz versteckt die Kommunikationsmechaniken zwischen verteilten Speichermodulen vor dem Programmierer einer Anwendung. Hierdurch wird die Programmierung und Portierbarkeit erleichtert, wie es von *SMS* gewollt ist und gleichzeitig die Skalierbarkeit von *DMS* wird erreicht. *DSM* ist hauptsächlich ein Werkzeug für parallele Anwendungen oder für beliebig verteilte Applikationen, für deren einzelne, gemeinsam genutzte Datenelemente ein direkter Zugriff er-

forderlich ist. *DSM* ist eine Abstraktion, die für die gemeinsame Nutzung von Daten zwischen Prozessen in Rechnern, die keinen gemeinsamen physischen Speicher besitzen, verwendet wird. Dabei greifen Prozesse lesend und schreibend auf Speicher/Speicherseiten zu, wobei dieses wie *normaler* Speicher innerhalb des Adressraums, der Prozesse, genutzt werden kann. Speicherseiten(Kacheln) die in DSM-Systemen öfter verwendet werden, sind dabei eine durch die Rechnerarchitektur und das Betriebssystem festgelegte Anzahl von direkt aufeinanderfolgenden Speicherstellen im logischen Speicherraum. Obwohl der physische Speicher auf verschiedenen Rechnern verteilt ist, lässt *Distributed Shared Memory* die verschiedenen Prozesse darauf zugreifen als wenn Sie auf einen alleinigen, gemeinsam verwendeten Speicher zugreifen würden. In Abbildung 1 ist diese Idee anhand einer Architektur des *DSM* zu sehen. Das gesammte System kann dabei als Sammlung von Objekten gesehen werden, ohne Rücksicht auf die aktuelle Position/Lage der Objekte. Die Umsetzung der Datenzugriff $\leftrightarrow$ Netzwerkkommunikation erfolgt dabei automatisch durch die Komponenten des *DSM*. Die Prozesse in diesem Netzwerk kommunizieren dabei nicht über Nachrichten die untereinander verschickt werden. Sie synchronisieren über die *shared objects*. Ein Prozess kann dabei auf ein Objekt zugreifen, in dem es eine Methode auf ihm aufruft.[4] [5]

Eine wichtige zu beachtende Eigenschaft bei *DSM*-Systemen ist die Konsistenz. Was passiert wenn mehrere Prozesse gleichzeitig eine Methode auf einem Objekt aufrufen ? Hierfür werden Konsistenzmodelle, Synchronisationsarten und Aktualisierungsoptionen benötigt.[5] Die Erläuterung und Vorstellung dieser erfolgt in den nächsten Abschnitten.

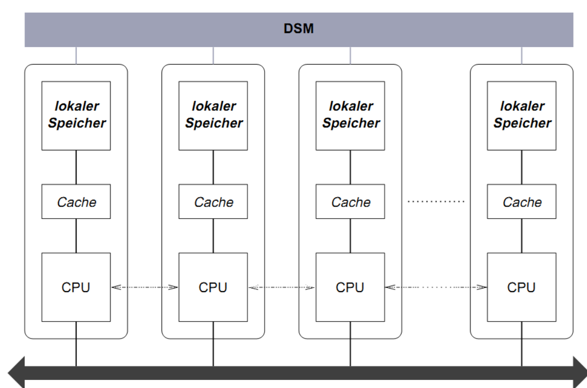


Abbildung 1. Mögliche DSM Architektur [9]

3. Herausforderungen bei Distributed Shared Memory Systemen

Die Hauptprobleme, die jedes *Distributed Shared Memory System* lösen muss, bestehen zum Einen im zuweisen eines logischen geteilten Adressbereichs auf den physikalischen verteilten Speicher der Module, zum Anderen im Finden und Zugreifen auf ein benötigtes Speicherdatum. Ein weiteres zu betrachtendes Problem besteht darin, eine kohärente Sicht auf den ganzen geteilten Addressraum zu halten. Ein wichtiges Ziel bei der Lösung dieser Probleme ist die Minimierung der Zugriffszeit auf die verteilten Daten. In den meisten Fällen werden folgende zwei Strategien für die Verteilung der Daten verwendet: *Replikation* und *Migration*.

Replikation erlaubt mehrere Kopien der selben Daten, die in unterschiedlichen lokalen Speichern liegen können.

Migration erlaubt eine einzige Kopie eines Datums, welches zu der zugreifenden Seite verschoben werden muss.

Genau wie in *SMS* mit privaten Caches, müssen *DSMS* mit dem Konsistenzproblem umgehen, wenn mehrere Kopien der selben Daten im System existieren. Um die kohärente Sicht auf den gemeinsamen Speicheradressraum zu wahren, muss eine Leseoperation den als letzten geschriebenen Wert zurückgeben. Dadurch muss dafür gesorgt werden, dass wenn eine von mehreren Kopien der Daten geschrieben wird, die anderen ungültig oder aktualisiert werden, abhängig vom genutzten Kohärenzprotokoll. Obwohl eine strikte Konsistenz die natürlichste Sicht auf den gemeinsamen Adressraum zulässt, können schwächere Konsistenzmodelle genutzt werden, um die Latenzzeiten zu verringern falls welche vorhanden sind.

4. Konsistenzmodelle

Die gleichzeitige Existenz mehrerer Kopien der selben Daten bei manchen *Distributed Shared Memory*-Algorithmen, ergeben ein zusätzliches Problem bei der Konsistenthaltung der Kopien. Um diesen Problem zu begegnen, gibt es mehrere Konsistenzmodelle, von denen in diesem Abschnitt drei vorgestellt werden sollen.

Strikte/Atomare Konsistenz: Das Modell der strikten Konsistenz besagt, das jede Read-Operation als Ergebnis den Wert der global „zuletzt“ geschriebenen Write-Operation liefert unabhängig davon, welcher Prozess geschrieben hat. Das größte Problem bei dieser Konsistenz ist, das es im Netz keine globale Zeit gibt und der Begriff „zuletzt“ sehr unscharf ist. In Abbildung 2 sind zwei Beispiele(korrekt und inkorrekt) für strikte Konsistenz zu sehen.

Sequentielle Konsistenz: Die sequentielle Konsistenz ist ein schwächeres Modell als die strikte Konsistenz. Das

Abbildung 2. Beispiel für korrekte(a) und inkorrekte(b) Operationsfolge bei strikter Konsistenz [9]

Modell besagt dabei, dass wenn mehrere nebenläufige Prozesse auf Daten zugreifen, dann ist jede gültige Kombination von Read- und Write-Operationen akzeptabel, solange alle Prozesse diesselbe Folge sehen. Das Ergebnis einer Ausführung ist dabei dasselbe, wie wenn alle Lese- und Schreiboperationen aller Prozesse in einer bestimmten Reihenfolge ausgeführt werden. Alle Prozesse sehen alle Schreiboperationen in der gleichen Reihenfolge. Während bei der strikten Konsistenz nicht offen ist wann eine Änderung sichtbar wird bleibt es dies bei der sequentiellen Konsistenz. Die Zeit spielt in diesem Modell keine Rolle. In Abbildung 3 sind zwei Beispiele(korrekt und inkorrekt) für sequentielle Konsistenz zu sehen. Währen das korrekte Beispiel sequentiell korrekt ist, da alle Prozesse die Schreibzugriffe in der selben Reihenfolge sehen, besteht keine strikte Konsistenz da P3 und P4 veraltete Werte lesen. Im inkorrekten Bespiel herscht keine sequentielle Konsistenz, da P3 und P4 den Schreibzugriff von P1 und P2 in unterschiedlicher Reihenfolge sehen.

Abbildung 3. Beispiel für korrekte(a) und inkorrekte(b) Operationsfolge bei sequentieller Konsistenz [9]

Kausale Konsistenz: Die kausale Konsistenz ist ein schwächeres Modell als die sequentielle Konsistenz. Die Regel dabei besagt, dass Write-Operationen, die potentiell in einem kausalen Verhältnis zueinander stehen, bei allen anderen Prozessen in derselben Reihenfolge gesehen werden müssen. Für nicht in dieser Beziehung stehende Operationen ist die Reihenfolge auf verschiedenen Rechnern gleichgültig. Nur kausal abhängige Schreibzugriffe sind hierbei zeitlich geordnet und werden von allen Prozessoren in derselben Reihenfolge gesehen. Zwei Write-Operationen sind auch potentiell kausal abhängig, wenn vor der zweiten Schreiboperation eine Leseoperation stattfindet, die den Wert der zweiten Schreiboperation beeinflusst haben kann. In Abbildung 4 sind zwei Beispiele(korrekt und inkorrekt) für kausale Konsistenz zu sehen.

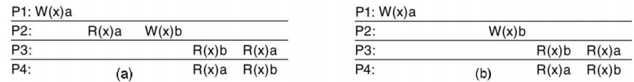


Abbildung 4. Beispiel für korrekte(a) und inkorrekte(b) Operationsfolge bei kausaler Konsistenz [9]

Auf der einen Seite bietet die Nutzung schwächerer Konsistenzmodelle eine Möglichkeit, die Performance von *DSMS* zu steigern, da dies die Überlappung und das Neuordnen von Speicherzugriffen erlaubt. Auf der anderen Seite führt dies auch zu einer Erschwerung bei der Implementierung, da vermehrt auf Synchronität geachtet werden muss.

Weitere Konsistenzmodelle wie *Pipelined RAM* und *Release-Konsistenz* können ebenfalls für *DSM* verwendet werden.In folgenden Quellen kann sich über diese Informiert werden: [6].

Bei der Wahrung der Konsistenz gibt es zwei Fehler zu unterscheiden: Lesefehler und Schreibfehler.

Lesefehler: Lesefehler treten auf, wenn ein Prozess versucht, von einer Seite zu lesen, für die der Prozess keine Leseberechtigung hat.

Beispiel: Seite p wird vom Besitzer von p nach Pr kopiert und mit Leseberechtigung versehen im Adressraum von Pr gelagert. Ist der Besitzer von p ein einzelner Schreiber, so wird das Schreibrecht entzogen aber nicht das Eigentumsrecht. Die Fehlerbehandlung schließt mit einem Neustart des fehlerverursachenden Befehls.

Schreibfehler Schreibfehler treten auf, wenn ein Prozess versucht, auf eine Seite zu schreiben, für die keine Zugriffs- oder Leseberechtigung besitzt.

Beispiel: Seite p wird an Pw übertragen und mit Lese- und Schreibberechtigung versehen im Adressraum von Pw gelagert. Die anderen Kopien werden entwertet, indem den in copy set enthaltenen Prozessen die Zugriffsberechtigung auf die Seite p entzogen werden. Die Fehlerbehandlung schließt dabei mit einem Neustart des fehlerverursachenden Befehls.

Um das zu verwendende Konsistenzmodell für ein *DSMS* auszuwählen, ist eine Festlegung auf ein zu verwendendes Zugriffsmuster und den Speichertyp notwendig. In den nächsten beiden Abschnitt werden die geläufigsten vorgestellt.[8]

5. Speichertypen bei der Nutzung von DSM

In diesem Abschnitt sollen verschiedene Speichertypen vorgestellt werden, die von Distributed Shared Memory genutzt werden können. Hierbei werden folgende Arten betrachtet: fortlaufende Bytes, Seiten, Objekte auf Sprachebene und unveränderliche Daten.

Fortlaufende Bytes: Bei fortlaufenden Bytes erfolgt der Zugriff wie auf generellen virtuellen Speicher. Dabei sind gemeinsam genutzte Objekte, direkt adressierbare Speicherpositionen. Bytefolgen sind feingranular, haben also auf der einen Seite einen kleineren Übertragungsaufwand, auf der anderen Seite jedoch einen hohen Verwaltungsaufwand.

Seiten/Kacheln: Seiten/Kacheln sind grobgranular, haben im Gegensatz zu Bytefolgen einen hohen Übertragungsaufwand aber einen vergleichbar kleinen Verwaltungsaufwand.

Objekte auf Sprachebene: Bei der Verwendung von Objekten auf Sprachebene kann eine Objektsemantik genutzt werden, um Konsistenz zu erzwingen. Die Serialisierung der Operationen ist dabei objektspezifisch und nicht kachel(seiten)spezifisch. Im Allgemeinen sind die Objekte von höherer, anwendungsbezogener Semantik.

unveränderliche Daten: Bei dieser Zusammensetzung können Prozesse Datenelemente lediglich hinzufügen, lesen und entfernen. Dabei wird *DSM* als Tupelraum genutzt. Die Tupel stellen dabei eine Folge typisierter Datenfelder. Tupel werden durch write() hinzugefügt, read() ausgelesen und durch take() entfernt. Es gibt jedoch keine Möglichkeit die Tupel im Tupelraum zu verändern.

Ein Manager oder ein Broadcast kann herausfinden, welcher Prozess zum Beispiel die aktuellste Version einer Seite hat und diese zurückliefern (*owner*). Um einen solchen Manager zu implementieren gibt es zentrale und verteilte Ansätze. Dabei kann der Manager selbst ein Teil der Anwendung oder ein externer Anbieter sein. Der Manager empfängt dabei auch Seitenfehler und und leitet diese weiter. Die Fehlerverursacher verarbeiten die Fehler dann selbständig weiter. Weiterhin gibt es die Möglichkeit, die Menge der Prozesse zu erhalten, die eine Kopie einer Seite haben (*copy set*). Diese Informationen werden beim Eigentümer gespeichert und bei einem Schreibfehler zugestellt. Weiterhin enthalten sind darin die Identifikationen und Transportadressen der teilnehmenden Prozesse. Der Prozess, der den Schreibfehler verursacht hat, entwertet die Daten per Multicast.[9]

6. Zugriffsmuster

In diesem Abschnitt werden zwei für *Distributed Shared Memory* in Frage kommende Zugriffsmuster vorgestellt werden: *Multiple Read Single Write (MRSW)* und *Multiple Read Multiple Write (MRMW).*

Beim *MRSW*-Zugriffsmuster legt eine Lese-Operation eine lokale Kopie, die nur Lesezugriffsrecht hat, an und entfernt das Schreibzugriffsrecht der Referenz. Bei einer Schreiboperation werden alle vorhandenen Kopien verworfen und ein neues Referenzobjekt angelegt mit Lese-/Schreibzugriffsrechten.

Beim *MRMW*-Zugriffsmuster legt eine Lese-Operation ebenfalls eine lokale Kopie, die nur Lesezugriffsrecht hat, an. Dabei wird jedoch nichts an den möglicherweise vorhandenen Schreibzugriffsrechten geändert. Beim schreibenden Zugriff wird gegebenenfalls eine lokale Kopie vom Referenzobjekt erzeugt, die Lese-/Schreibzugriffsrechte hat.

Der Schreibvorgang auf ein Replikat eines *DSM*-Objektes muss die Aktualisierungen aller anderen Exemplare desselben Replikats nach sich führen. Zur Weitergabe der Aktualisierungen von einem Prozess an die anderen Prozesse gibt es zwei Ansätze, die im nächsten Abschnitt erläutert werden.[10]

7. Aktualisierungsoptionen

In diesem Teil der Arbeit werden die zwei Aktualisierungsoptionen: *Schreiben-Aktualisieren* und *Schreiben-Entwerten* vorgestellt werden.

Schreiben-Aktualisieren: Die Änderungen erfolgen lokal und werden allen anderen Repliken per Multicast mitgeteilt. Bei der Verwendung von vollständig sortierten Multicasts ist es möglich, auch bei mehreren Schreibern sequentielle Konsistenz aufrechtzuerhalten. Schreiben-Aktualisieren unterstützt dabei sowohl das Zugriffsmuster *Multiple Read Single Write* als auch *Multiple Read Multiple Write*.[3]

Schreiben-Entwerten: Um die Kosten für die vollständige Sortierung zu reduzieren, wird bei dem Verfahren des Schreiben-Entwertens nur eine Schreibzugriff pro Zeiteinheit zugelassen. Hierbei kann ein Datenatom zu jedem Zeitpunkt von einem oder mehreren Prozessen gelesen oder von einem Prozess gelesen und geschrieben werden. Ein Element, das gerade gelesen wird, kann beliebig oft schreibgeschützt kopiert werden. Will nun ein Prozess dieses Element schreiben, so werden alle anderen Leser über Multicast benachrichtigt. Der Prozess beginnt dabei erst dann zu schreiben, wenn er von allen anderen Prozessen eine Rückmeldung erhalten hat. Dieser Ansatz ist bei

vielen Lesern und wenigen Schreibern sehr effizient. Wenn das Verhältnis von Lesern und Schreibern kippt, ist der Schreiben-Aktualisieren Ansatz vorzuziehen. Schreiben-Entwerten unterstützt dabei die Zugriffsmuster *Single Read Single Write* und *Multiple Read Single Write*.[3]

Beim Schreiben-Entwerten kann dabei das Problem des *Seitenflatterns* auftreten. Dies tritt auf, wenn die Laufzeitumgebung einen übermäßigen Zeitanteil damit verbringt, gemeinsam genutzte Daten zu entwerten und zu übertragen. So kann es sein, das ein Datum beim Leser andauernd invalidiert wird. Dies geschieht dadurch, dass das Datum laufend aktualisiert und so vom Schreiber ständig übertragen wird. Um dieses Problem abzuschwächen, kann den Datenelementen eine Mindesverweildauert mitgegeben werden.[7]

Um diese Konsistenskriterien einfacher zu halten, werden Objekte üblicherweise nur *read-only* repliziert. Dies bedeutet, dass Schreiboperationen nur auf dem Original, welches der Eigentümer hält, durchgeführt wrden können. Möchte ein anderer Prozessor auf ein Objekt schreiben, muss beim Eigentümer ein *write-exclusive* angefordert werden. Ist dies möglich, wird der anfragende Prozessor zum neuen Eigentümer und der alte Eigentümer markiert seine Kopie als *read-only*. Diese Vorgehensweise setzt jedoch voraus, daß der Eigentümer auffindbar ist. Durch Broadcasts lässt sich dies erreichen, was allerdings sehr ineffizient ist, da jeder Prozessor auf einen Broadcast reagieren muss.[2]

8. Implementierungsansätze

Im folgenden Abschnitt werden zwei Implementierungsansätze vorgestellt, die *Distributed Shared Memory* verwenden. Dazu gehören zum einen *Non-Uniform Memory Arhitecture (NUMA)* und *Paged Virtual Memory*.

8.1. NUMA

Non-Uniform Memory Architecture(NUMA) ist eine Architektur für Computer-Speicher bei Multiprozessorsystemes. Hier hat jeder Prozessor einen eigenen, lokalen Speicher. Jeder Prozessor lässt jedoch anderen Prozessoren über einen gemeinsamen Adressraum direkten Zugriff auf diesen. Der Zugriff auf den Speicher in diesem Verbund ist abhängig davon, ob sich die Speicheradresse im lokalen oder in einem fremden Speicher befindet.

Durch Anpassungen, beim *Paging* des virtuellen Speichers im Betriebssystem, besteht die Möglichkeit Rechnernetzweite Adressräume zu implementieren, und damit *NUMA in Software* umzusetzen. [6]

8.2. Paged Virtual Memory

Unter Paged Virtual Memory (gekachelter virtueller Speicher) wird eine festgelegte Anzahl von direkt aufeinanderfolgenden Speicherstellen im Speicherraum bezeichnet. Beim *gekachelten virtuellen Speicher* nimmt *DSM* einen festgelegten Bereich im virtuellen Speicher ein. Dies ist für jeden teilnehmenden Prozess derselbe Adressbereich im Adressraum. [1]

9. Zusammenfassung und Fazit

In dieser Arbeit wurde das Konzept des *Distributed Shared Memory (DSM)* vorgestellt. Dabei wurde zunächst auf die Grundlagen eingegangen, die für das Verständnis der Arbeit wichtig sind. Im Hauptteil der Arbeit wurde aufgezeigt, was DSM ist und eine Erklärung der wichtigsten Grundsteine (Konsistenz-Modelle, Aktualisierungsoptionen etc.) gegeben. Nach den ersten Umsetzungen, die sich noch an den nur begrenzt leistungsfähigen *shared-memory* Multiprozessor-Architekturen orientierten, könnten unter Anderem durch die Verwendung schwächerer Konsistenz-Modelle leistungsfähigere Systeme konzipiert werden. Um so schwächer das verwendete Konsistenz-Modell ist, um so kleiner sind die Verluste durch Netzwerk-Verzögerungen, angenommen es wurden keine Fehler bei der Umsetzung begangen. Durch die Verwendung schwächerer Konsistenzmodelle muss aber ein komplizierteres Programmiermodell genutzt werden, so dass zwischen Programmierbarkeit und Systemperformanz abgewogen werden muß, um eine Entscheidung zu treffen. Abschließend kann gesagt werden, dass *Distributed Shared Memory-Systeme* dem Programmierer ein einfacheres Programmiermodell bieten, dass einen virtuellen Adreßraum zur Verfügung stellt und damit insbesondere die Aufgabe des expliziten Nachrichtenaustauschs abnimmt.

Literatur

[1] H. G. Cragon. *Computer Architecture and Implementation*. Cambridge University Press, 2000.

[2] W. Eberling. Distributed shared memory (grundlagen), 1998.

[3] M. Galochino. Eine hierarchie ueber konsistenzmodelle, 2004.

[4] V. K. Garg. *Elements of Distributed Computing*. Wiley, 2002.

[5] T. K. George Coulouris, Jean Dollimore. *Verteilte Systeme - Konzepte und Design*. Pearson Education, 2002.

[6] V. M. Jelica Protic, Milo Tomasevic. *Distributed Shared Memory-Concepts and Systems*. IEEE Computer Society, 1998.

[7] C. R. Kime. *Logic and computer design fundamentals*. Pearson/Prentice Hall, 2004.

[8] B. Ramm. Speicherkonsistenz, 2002.
[9] W. Schroeder-Preikschat. Verteilter gemeinsamer speicher, 2006.
[10] H.-M. Windisch. *Speicherverwaltung fuer konzeptionell strukturierte verteilte Systeme*. Herbert Utz Verlag Wissenschaft, 1996.

Selbststabilisierung

Philipp Ittershagen
Abt. Systemsoftware und verteilte Systeme
Carl von Ossietzky Universität Oldenburg
Philipp.Ittershagen@Informatik.Uni-Oldenburg.DE

***Zusammenfassung*—In dieser Ausarbeitung wird das Prinzip Selbststabilisierung in verteilten Systemen betrachtet, welches erstmals von Edsger W. Dijkstra 1974 in [1] untersucht wurde. Dazu wird anhand eines abstrakten Modells eines verteilten Systems eine Definition für ein selbststabilisierendes System vorgestellt. Anschließend wird das Prinzip Selbststabilisierung als Ansatz zur Fehlertoleranz erläutert und grundlegende Unterschiede hervorgehoben. Dieser Ansatz zeigt, dass im Gegensatz zur Replikatebasierten Fehlertoleranz eine garantierte Eliminierung von Fehlern in endlichen Schritten möglich ist. Der wechselseitige Ausschluss und ein Wahlverfahren in einem Ring-basierten verteilten System werden in dieser Ausarbeitung als Beispiel zur Anwendung selbststabilisierender Algorithmen näher erläutert.**

***Keywords*-Selbststabilisierung, K-Zustandsautomat, vorübergehende Fehler, Hülle, Konvergenz**

I. Einleitung

Im Jahre 1974 veröffentlichte Edsger W. Dijkstra das Paper „Self-stabilizing systems in spite of distributed control“ [1] und eröffnete damit ein neues Forschungsgebiet im Bereich der verteilten Systeme. Dieser neue Ansatz zur Fehlertoleranz bietet im Gegensatz zur Maskierung von Fehlern eine neue Möglichkeit, in einem verteilten System vorübergehende Fehler zu tolerieren.

Selbststabilisierung ist die Fähigkeit eines Systems, sich von einem beliebigen Startpunkt aus in einen erwünschten Zustand zu versetzen. Das in [2] angegebene Beispiel für Selbststabilisierung ermöglicht ein erstes Verständnis für das Verhalten eines selbststabilisierenden Systems:

Ein Orchester spielt an einem windigen Abend draußen. Da kein Dirigent vorhanden ist, muss jeder Spieler auf die benachbarten Spieler hören, damit das Lied harmonisch klingt. Die Noten zum Lied liegen jedem Musiker in einem Notenbuch vor. Durch den Wind kann es nun allerdings passieren, dass bei einigen Musikern unwillkürlich die Seiten des Buchs gewendet werden, sodass die Musiker nun nicht mehr in Harmonie spielen können. Der Ansatz einer selbststabilisierenden Lösung ist nun, dass jeder Musiker sich an den Nachbar-Musiker orientiert, der die jüngste Seite aufgeschlagen hat. Sollte nun der Wind einzelne Notenbücher dazu bringen, dass die Seiten umgeblättert werden, orientieren sich die betroffenen Musiker an den Seitenzahlen ihrer Nachbarn und werden somit irgendwann wieder in Harmonie spielen.

Eine wichtige Beobachtung ist, dass das Publikum das Auftreten der vorübergehend fehlenden Harmonie erfährt, diese allerdings irgendwann wieder hergestellt wird. Genau dies ist das Prinzip eines selbststabilisierenden Algorithmus. Vorübergehende Fehler führen dazu, dass das System sich nach endlichen Schritten wieder in einem legalen Zustand befindet.

Für die nähere Betrachtung von selbststabilisierenden Systemen und Algorithmen wird in Abschnitt II zunächst ein Berechnungsmodell vorgestellt, welches im späteren Verlauf erlaubt, eine abstrakte Sicht auf verteilte Systeme zu erlangen und das Verhalten von selbststabilisierenden Systemen zu beobachten. Abschnitt III geht dann mit Hilfe dieses Modells auf die Definition eines selbststabilisierenden Systems ein. In Abschnitt IV wird beschrieben, wie das Prinzip der Selbststabilisierung einen Ansatz zur Fehlertoleranz darstellt. Abschnitt V-A und V-B zeigen zwei Anwendungsbeispiele für einen selbststabilisierenden Algorithmus. Nach dieser Betrachtung schließt die Ausarbeitung mit einem Fazit in Abschnitt VI ab.

II. Definition eines verteilten Systems

Ein verteiltes System kennzeichnet sich durch eine Menge von Prozessen, die in der Lage sind, mit anderen im System befindlichen Einheiten zu kommunizieren.

Die Netztopologie der verbundenen Prozesse kann verschiedenen Strukturen entsprechen. Einerseits können die im Netz befindlichen Einheiten in einem Ring verbunden sein, es sind aber auch andere Hierarchien wie zum Beispiel Baumstrukturen möglich. Bei dem im Folgenden aus [2] vorgestellten Modell werden Prozesse eines verteilten Systems als Knoten dargestellt und deren Kommunikationskanäle durch Kanten (Abbildung 1). Prozesse in verteilten Systemen können einerseits mittels Nachrichtenaustausch über geeignete Kommunikationskanäle kommunizieren, oder aber über einen gemeinsamen Speicher Informationen untereinander austauschen.

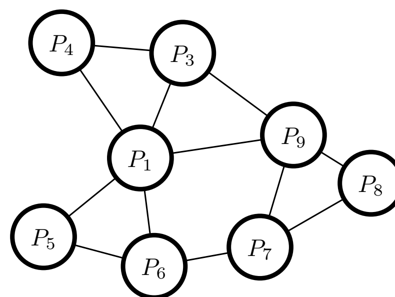


Abbildung 1. Eine allgemeine Netztopologie mit den Prozessen $P_1, \ldots, P_9$. Die Kanten im Graphen beschreiben die vorhandenen Kommunikationskanäle zwischen den Prozessen.

Der Nachrichtenaustausch im Modell geschieht über First-In, First-Out (FIFO) Kanäle, wodurch eine Nachricht versendet und dann zu einem späteren Zeitpunkt unabhängig vom Sender empfangen werden kann. Durch dieses Kommunikationsmodell können beispielsweise Wide-Area-Networks (WAN) beschrieben werden, deren Prozesse räumlich weit voneinander entfernt sind.

Geschieht die Kommunikation über gemeinsam genutzten Speicher, wird dies durch spezielle Kommunikationsregister modelliert. Jeder Prozess ist in der Lage, in ein oder mehrere solcher Register zu schreiben und aus einem oder mehreren Registern den aktuellen Inhalt zu lesen. Dieses Kommunikationsmodell bietet sich an, um Systeme zu modellieren, deren Prozesse sich beispielsweise auf der gleichen Platine befinden.

Jede Einheit besitzt einen lokalen Zustand, dessen Änderung vor allem durch das Empfangen und Senden von Nachrichten verändert wird. Die Summe aller lokalen Zustände der einzelnen Einheiten bildet den globalen Zustand eines verteilten Systems. Der globale Zustand enthält bei Betrachtung von verteilten Systemen mit FIFO-Kommunikationskanälen auch die bereits versendeten, aber aufgrund der Asynchronität noch nicht empfangenen Nachrichten. Diese Nachrichten werden in den Warteschlangen der FIFO-Kanäle gespeichert.

Durch die Definition eines globalen Zustands können nun Aussagen über das Verhalten eines Systems getroffen werden. Das Verhalten eines Systems ist definiert durch eine Transitionsrelation, die ein System von einem Zustand in einen anderen überführen. Prozesse können aufgrund ihrer Modellierung als Zustandsautomaten mit lokalem Speicher ihre Zustände ändern. Wird somit ein lokaler Zustand verändert, verändert dies auch gleichzeitig den globalen Zustand des Systems, da dieser die Menge aller lokalen Zustände umfasst. Ein Zustandsübergang eines lokalen Prozesses verändert somit den globalen Zustand des Systems. Das Verhalten eines Systems ist also abhängig von den lokalen Zustandsänderungen der Prozesse. Ein System ändert den Zustand genau dann, wenn sich der lokale Zustand eines Prozesses verändert.

III. Definition Selbststabilisierung

Um eine Definition der Selbststabilisierung zu erreichen, werden zwei Eigenschaften der im vorigen Abschnitt beschriebenen globalen Zustände vorgestellt, die ein System erfüllen muss, um selbststabilisierend zu sein. Dabei wird dem Übergang eines Systems von einem Zustand in einen anderen besondere Beachtung zugetragen.

Die Definition eines selbststabilisierenden Systems umfasst legale und illegale globale Zustände. Ein legaler Zustand ist dabei ein globaler Zustand des Systems, der sich dadurch auszeichnet, dass eine bestimmte Eigenschaft erfüllt ist. Diese Eigenschaft kann je nach Einsatzgebiet des Algorithmus variieren, im einleitenden Beispiel des draußen spielenden Orchesters ist die Eigenschaft das korrekte Zusammenspiel aller Musiker. Illegale Zustände sind somit globale Zustände, die diese Eigenschaft nicht erfüllen.

A. Hüllenbildung und Konvergenz

Die Eigenschaften der *Hülle*nbildung und *Konvergenz* im Bezug auf die Zustandsübergänge zwischen legalen und illegalen Zuständen eines selbststabilisierenden Systems werden im Folgenden erläutert.

Hülle. *Sobald das System einen legalen globalen Zustand erreicht hat, wird es diesen durch weiter Zustandsübergänge nicht mehr verlassen.*

Konvergenz. *Von einem beliebigen Startpunkt aus wird das System in einer endlichen Anzahl an Transitionen einen legalen globalen Zustand erreichen.*

Diese beiden Eigenschaften kombiniert veranlassen ein selbstabilisierendes System dazu, in jedem Fall einen legalen Zustand zu erreichen (Konvergenz) und diesen dann nicht mehr zu verlassen (Hüllenbildung).

B. Faktoren, die Selbststabilisierung verhindern

Nicht in jedem verteilten System können selbststabilisierende Algorithmen angewendet werden. Eine wichtige Eigenschaft, die eine verteiltes System besitzen muss, um Selbststabilisierung zu ermöglichen, ist die *Asymmetrie*. Ein verteiltes System ist asymmetrisch, wenn die Prozesse untereinander unterscheidbar sind. Dies kann zum Beispiel dadurch erreicht werden, dass jeder Prozess eine lokale eindeutige Identitätszahl erhält. Dann kann jeder Prozess auch den gleichen Programmcode ausführen, die Einheiten untereinander unterscheiden sich durch ihre Identitäten.

Die *Terminierung* gilt als Verhinderung von Selbststabilisierung, da durch das Terminieren eines Algorithmus dieser nicht mehr den in der Definition der Selbststabilisierung gestellten Anforderungen gerecht wird. So kann ein Algorithmus, der in einem illegalen Zustand terminiert, nicht wieder einen legalen Zustand erreichen. In verteilten Systemen sind Algorithmen im Allgemeinen allerdings nichtterminierend, da sie gewisse Eigenschaften, die das verteilte System besitzen soll, stets sicherstellen sollen. Daher ist ein Deadlock in einem verteilten System einer Terminierung bei sequentiellen Algorithmen gleichzusetzen. Deadlocks in verteilten Systemen stellen eine Art von Terminierung dar, da die verteilten Algorithmen zwar nicht terminieren, eine weitere Zustandsänderung allerdings nicht vollzogen werden kann, da die Prozesse im System auf ein Ereignis warten, welches durch einen Deadlock nicht eintreten wird.

Vorübergehende Fehler, die einen Nachrichtenversand unterdrücken, können zu einem Deadlock führen. Es kann zum Beispiel sein, dass ein Prozess auf eine Nachricht eines anderen Prozesses wartet, bevor weitere Aktionen getätigt werden. Sollte nun diese Nachricht durch einen Fehler nicht beim Empfänger ankommen, wartet dieser Prozess auf die Nachricht, die allerdings bereits versendet wurde. Der Prozess kann also nicht feststellen, ob die Nachricht bereits gesendet wurde. Eine Möglichkeit, dieses Problem zu umgehen, wurde in [3] untersucht und die Lösung vorgestellt, dass solche Nachrichten öfter gesendet werden, bis eine Empfangsbestätigung beim Sender ankommt. Diese Erweiterung des Nachrichtenaustauschs

verhindert Deadlocks in verteilten Systemen, in denen Algorithmen zur Selbststabilisierung eingesetzt werden.

In [4] wurde ein weiterer Faktor vorgestellt, der Selbststabilisierung verhindert. Diese *Isolation* wird in [5] an einem simplen Beispiel dargestellt. Gegeben sind drei Prozesse, die in einer Baumstruktur verbunden sind. Es existiert somit ein Wurzel-Prozess und zwei Kind-Prozesse. Jeder Prozess besitzt einen lokalen Zähler. Die Wurzel kann eine Aktion ausführen, welche den lokalen Zähler inkrementiert und eine Nachricht an einen der beiden Kind-Prozesse sendet. Sobald ein Kind-Prozess eine Nachricht empfangen hat, erhöht dieser seinen eigenen Zähler. Ein legaler Zustand wird nun definiert als der Zustand, in dem die Summe der Zähler beider Kind-Prozesse gleich der Summe des Zählers des Wurzel-Prozesses und der Anzahl der in den Kanälen enthaltenen Nachrichten ist. Es gibt also für die Kind-Prozesse keine Möglichkeit zu kommunizieren und nur der Wurzel-Prozess initiiert Zustandsänderungen bei den Kind-Prozessen. Geschieht nun durch einen vorübergehenden Fehler beispielsweise eine Veränderung eines Zählers ohne entsprechende Nachrichtenübermittlung, befindet sich das System in einem illegalen Zustand. Diesen illegalen Zustand wird es nicht mehr verlassen können, da die Kind-Prozesse keine Möglichkeit haben, ihre Zähler zu korrigieren. Obwohl die Kind-Prozesse sich nach einem vorübergehenden Fehler in einem legalen lokalen Zustand befinden, kann der globale Zustand nicht mehr in einen legalen Zustandsraum konvergieren.

IV. Selbststabilisierung als Ansatz zur Fehlertoleranz

Es ist leicht festzustellen, dass mit Selbststabilisierung nicht jeder beliebig schwerwiegende Fehler korrigiert werden kann. Wird ein System in der Art korrumpiert, dass das Verhalten nicht mehr dem eingangs definierten entspricht, so kann ein Algorithmus zur Selbststabilisierung nicht mehr korrekt ablaufen.

Stattdessen wird der Begriff „vorübergehender Fehler“ (transient failure) verwendet, um die Art Fehler zu klassifizieren, die zwar den Zustand einzelner Komponenten und damit den globalen Zustand des Systems ändern können, nicht aber deren Verhalten. Dies bedeutet insbesondere, dass ein selbstabilisierendes System an die Grenzen der Fehlertoleranz gelangt, sobald sich Teile des Systems nicht mehr konform verhalten.

Eine solche Art von Fehlern kann entstehen, wenn der lokale Speicher oder der Programmzähler des lokalen Algorithmus verändert werden. Die Prozesse sind als Zustandsautomaten modelliert und eine Veränderung des Programmzählers oder lokalen Speichers entspricht einer unerwarteten Zustandsänderung, da der Speicher eines Prozesses auch den lokalen Zustand dieser Einheit repräsentiert.

Es wird bei vorübergehenden Fehlern davon ausgegangen, dass sie nicht persistent sind. Dies ist eine wichtige Eigenschaft, denn durch das Vorübergehen der Fehler kann das System durch Selbststabilisierung den korrekten Ablauf (einen legalen globalen Zustand) wieder erreichen. Tritt dann wieder ein vorübergehender Fehler im System auf, erreicht ein selbststabilisierendes System durch die in Abschnitt III eingeführten Eigenschaften der Konvergenz und Hüllenbildung in endlichen Schritten wieder einen legalen Zustand.

Betrachtet man diese Fehlertoleranz als einen integralen Bestandteil eines Systems, fällt die Behandlung von konkreten Problemen weg. Somit kann die explizite Fehlerbehandlung von Übertragungsfehlern, inkonsistenter Initialisierung der einzelnen Prozesse, Ausfällen von Prozessen oder auch Speicherfehlern in einem verteilten System gleichzeitig mit einem selbststabilisierenden System vollzogen werden.

Üblicherweise wird bei der Behandlung von vorübergehenden Fehlern ein konkreter Fall behandelt. Es kann zum Beispiel zur Verhinderung von Speicherproblemen auf Replikate zurückgegriffen werden, falls die gewünschte Speicheradressierung aufgrund von temporären Ausfällen nicht erfolgen konnte. Diese Herangehensweise stellt somit eine Fehlermaskierung dar, da der Fehler durch Redundanz und geeignete Algorithmen „verschleiert“ wird. Selbststabilisierung garantiert vielmehr die Wiederherstellung des Systems in einen legalen Zustand. Dadurch behandelt Selbststabilisierung konkret auftretende Fehler in einer zu den üblichen Ansätzen komplementären Art, da Fehler nicht vermieden werden, sondern minimiert werden, bis sie schließlich durch die Konvergenzeigenschaft eliminiert werden.

A. Mächtigkeit der Selbststabilisierung

Die Fähigkeit der Selbststabilisierung, nicht nur mit einer konkreten Fehlerklasse umgehen zu können, sondern einen allgemeinen Ansatz zur Fehlertoleranz von vorübergehenden Fehlern darzustellen, stellt ein sowohl mächtiges als auch komplexes Konzept dar und ist daher auch als Kritikpunkt zu sehen (vgl. [5]). Um diese Komplexität zu verringern, kann die Betrachtung der Wiederherstellung eines legalen Systemzustands aus einer begrenzten Menge von vorübergehenden Fehlern durchgeführt werden. Man betrachtet also nicht alle möglichen illegalen Zustände, sondern nur diejenigen, die bei einem konkreten vorübergehenden Fehler auftreten können. Dadurch wird die Mächtigkeit der Selbststabilisierung zwar verringert, liefert aber dennoch für den betrachteten vorübergehenden Fehler eine garantierte Lösung.

V. Anwendungsbeispiele

A. Mutual Exclusion mit dem K-State Algorithmus

Der folgende Algorithmus zur Selbststabilisierung wurde von Dijkstra 1974 zusammen mit der Einführung in dieses Forschungsgebiet in [1] zuerst vorgestellt. Der Algorithmus dient zum wechselseitigen Ausschluss von K Prozessen in einer ringbasierten Topologie. Jeder Prozess hat einen linken und einen rechten Nachbarn, mit dem er kommunizieren kann. Das *Privileg* ist definiert als eine boolesche Funktion, die den Zustand des eigenen Prozesses und den der benachbarten Prozesse beinhaltet. Dieses

Privileg drückt aus, ob ein Prozess einen bestimmten lokalen Zustandswechsel vollziehen darf oder nicht.

1) Eigenschaften legaler und illegaler Zustände: Um im globalen Systemmodell Aussagen über legale und illegale Zustände zu treffen, wurden dann vier Kriterien für das Vorhandensein eines legalen Zustands festgelegt:

1) In jedem legalen globalen Zustand sind ein oder mehrere Privilegien vorhanden.
2) In jedem legalen Zustand führt jeder weitere Zustandswechsel wieder in einen legalen Zustand.
3) Jedes Privileg muss in mindestens einem legalen Zustand vorhanden sein.
4) Für jedes Paar von legalen Zuständen existieren ein oder mehrere Zustandsübergänge, die das System vom einen zum anderen Zustand überführen.

In [5] wurden diese vier Kriterien unter Berücksichtigung der eingeführten Konvergenzeigenschaft und der Hüllenbildung näher betrachtet. Die von Dijkstra aufgestellten Kriterien wurden mit dem Hintergrund des wechselseitigen Ausschlusses bestimmt und sind daher restriktiver als die in Abschnitt III eingeführten Definitionen.

2) Vergleich der Definition mit Konvergenzeigenschaft und Hüllenbildung: Das erste Kriterium bestimmt, dass das System einem Fortschritt unterlegen ist, also kein Prozess im System terminiert. Die zweite Bedingung ist bereits vorgestellte Definition der Hüllenbildung. In der dritten Bedingung wird zwar eine Bedingung an das Programm gestellt, allerdings nicht an der Definition von legalen Zuständen. Die vierte Bedingung verhindert, dass legale Zustände in mehr als zwei disjunkten Mengen auftreten können und Zustandswechsel zwischen diesen Mengen nicht möglich sind. Diese Bedingung ist restriktiver als die oben angegebene Hüllenbildung, da diese nicht das Auftreten von disjunkten legalen Zustandsmengen verhindert, sondern lediglich fordert, dass ein sich in einem legalen Zustand befindliches System nach Zustandswechsel weiterhin in einem legalen Zustand befindet.

3) Ablaufschritte des Algorithmus: In einem bidirektionalen Ring existieren N Prozesse $P_0, \ldots, P_{N-1}$. Namensgebend für diesen Algorithmus ist, dass jeder Prozess $K > N$ Zustände besitzt. Der linke Nachbar eines Prozesses P_i wird $L = P_{i-1 \bmod N}$ genannt und der momentan betrachtete Prozess $S = P_i$.

Die folgenden Bedingungen und Transitionen definieren, ob ein Prozess ein Privileg besitzt oder nicht: Ist der

Algorithm 1 Privileg eines Prozesses

if $L \neq S$ **then**
  $S \leftarrow L$
end if

eigene Zustand also nicht gleich dem des linken Nachbarn, dann besitzt S momentan ein Privileg und die angegebene Transition ist aktiv. Führt S dann diese Transition durch, verliert es das Privileg, weil nach der Transition $S = L$ gilt und somit die eingehende Bedingung falsch ist.

Die im Vorfeld betrachtete Asymmetrie wird durch die Hervorhebung des Prozesses P_0 sichergestellt, welcher auch als der „Bottom-Prozess“ bezeichnet wird. Der Bottom-Prozess wird speziell behandelt und besitzt die folgende Bedingung für ein Privileg: Der Bottom-Prozess

Algorithm 2 Privileg des Bottom-Prozesses

if $L = S$ **then**
  $S \leftarrow S + 1 \bmod K$
end if

besitzt also nur ein Privileg, wenn sein Zustand gleich dem seines linken Nachbarn ist. Auch hier wird dieses Privileg eliminiert, sollte der Prozess die Transition durchführen. Der eigene Zustand wird bei der Transition um 1 inkrementiert und die Anfangsbedingung ist dann nicht mehr wahr.

Das folgende Beispiel aus [6] verdeutlicht den Ablauf des K-State-Algorithmus. Abbildung 2 zeigt auf der lin-

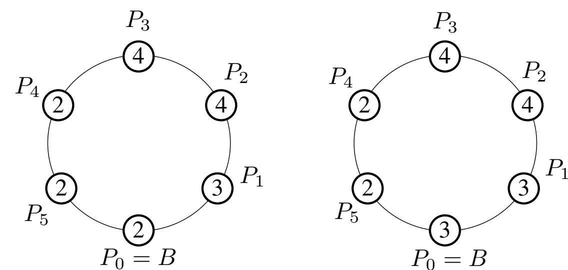


Abbildung 2. Ablauf des K-State Algorithmus durch Zustandswechsel des Bottom-Prozesses. Die Zahlen in den Prozessen stellen den lokalen Zustand dar (aus [2]).

ken Seite, dass der Bottom-Prozess ein Privileg besitzt, da der aktuelle Zustand dem seines linken Nachbarn entspricht. Nimmt der Bottom-Prozess nun die Transition, ändert dies seinen eigenen Zustand (Abbildung 2 rechts) und er besitzt das Privileg nicht mehr. Es ist zu erkennen, dass mehrere Prozesse ein Privileg besitzen können. In Abbildung 2 links besitzen neben dem Bottom-Prozess P_0 auch die Prozesse P_1, P_2 und P_4 ein Privileg, da bei ihnen die Bedingung $L \neq S$ erfüllt ist.

Da dieser selbststabilisierende Algorithmus allerdings den wechselseitigen Ausschluss garantieren soll, führen mehrere Privilegien zur Verletzung dieser Eigenschaft, da nun mehrere Prozesse den kritischen Abschnitt betreten können. Es existiert daher eine zentrale Komponente zur Überwachung, die stets einen Prozess auswählt, welcher dann einen Zustandswechsel durchführt.

Zusammen mit den erlaubten Zustandsübergängen und den Bedingungen (1) und (2) kann allerdings nur ein Prozess den kritischen Abschnitt betreten, da von der zentralen Komponente nur ein Prozess für einen Zustandswechsel ausgewählt wird. Dieser ausgewählte Prozess wird das Privileg beim Verlassen des kritischen Abschnitts wieder entfernen, da er die Zuweisung $S := L$ beziehungsweise $S := S + 1 \bmod K$ ausführt.

Auch wenn zunächst mehrere Prozesse ein Privileg besitzen können, so wird das System sich nach endlichen Schritten in einem globalen Zustand befinden, bei dem

global nur ein Privileg existiert. Die Zustandsübergänge der Prozesse $P_1, \ldots, P_N$ werden durch die Zuweisung $S := L$ nach einem Zustandsübergang irgendwann die gleichen Zustände besitzen. Dann wird nur der Bottom-Prozess P_0 im System das Privileg besitzen und dies wird wie ein Token durch den Ring wandern. Das System wird sich somit stabilisieren.

4) Erweiterung 3-State Algorithmus: Die Forderung $K > N$, also das Vorhandensein von mehr Zuständen als Prozessen stellt vor allem bei dynamischen verteilten Systemen mit wechselnder Anzahl an Prozessen ein Problem dar. Aus diesem Grund existiert der 3-State Algorithmus, welcher als Erweiterung des K-State Algorithmus angesehen werden kann, da er nur drei Zustände pro Prozess benötigt.

In Anlehnung an die im vorigen Abschnitt gegebene Definition, wird beim 3-State Algorithmus ebenfalls zwischen den Prozessen unterschieden. Allerdings besitzt der 3-State Algorithmus drei unterschiedliche Prozessklassen. Neben dem Bottom-Prozess und anderen Prozessen wird nun zusätzlich der Top-Prozess hervorgehoben. Analog zum Bottom-Prozess P_0 ist der Top-Prozess der Prozess P_N, welcher auch als T bezeichnet wird. Der Bottom-Prozess besitzt folgendes Kriterium:

Algorithm 3 Privileg Bottom-Prozess

```
if S + 1 = R then
    S ← S + 2
end if
```

Der Bottom-Prozess besitzt also genau dann ein Privileg, wenn sein eigener Zustand sich um 1 vom rechten Nachbarn unterscheidet. Durch das Aktivieren der Transition verliert der Bottom-Prozess wieder das Privileg, da die Transitionsbedingung nun nicht mehr gilt. Alle arithmetischen Operationen sind modulo 3 zu verstehen.

Der Top-Prozess besitzt zwei Bedingungen für ein Privileg: Wenn beide Nachbarn des Top-Prozesses den

Algorithm 4 Privileg Top-Prozess

```
if L = B and S ≠ B + 1 then
    S ← B + 1
end if
```

gleichen Zustand besitzen und der eigene Zustand sich nicht um 1 vom Bottom-Prozess unterscheidet, besitzt der Top-Prozess das Privileg. Wird die Transition aktiviert, verliert der Top-Prozess das Privileg wieder, da der eigene Zustand sich dann genau um einen Wert vom Zustand des Bottom-Prozesses unterscheidet.

Algorithm 5 Privileg andere Prozesse

```
if L = S + 1 or R = S + 1 then
    S ← S + 1
end if
```

Die übrigen Prozesse orientieren sich an ihren Nachbarn und wenn einer der Nachbarzustände sich um 1 unterscheidet, besitzt der aktuelle Prozess das Privileg. Die Zustandsdifferenz wird nach Aktivierung der Transition eliminiert und der Prozess verliert das Privileg. Um das Wechseln der Privilegien darzustellen, bietet sich die in [2] angegebene Schreibweise für den globalen Zustand an. Das System wird als Strahl gezeichnet, beginnend mit dem Bottom-Prozess und gefolgt von allen anderen Prozessen. Am Ende des Strahls befindet sich der Top-Prozess. Der Zustand zweier Prozesse ist entweder gleich oder aber unterscheidet sich um 1. Somit reicht es aus, diese Differenz zu zeigen. Sie wird in Abbildung 3 mit einem Pfeil dargestellt. Der Pfeil zeigt in die Richtung des Prozesses, dessen Zustand um $1 \pmod 3$ geringer ist als der eigene.

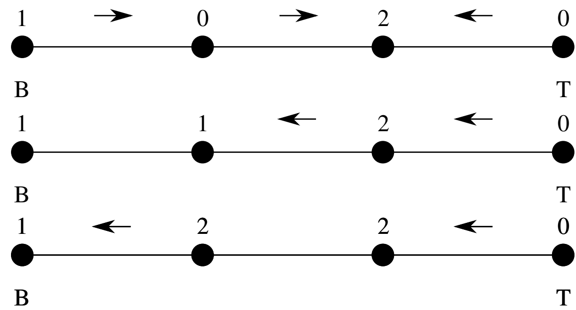


Abbildung 3. Ablauf des 3-State Algorithmus (aus [2]). Die Pfeile zeigen auf Prozesse, die gerade ein Privileg besitzen.

Dieses System wird nach einer endlichen Zahl von Zustandsübergängen nur noch einen Pfeil im gesamten Strahl besitzen (siehe dazu auch [2], [7]). Durch die Konsumierung der Privilegien durch die Prozesse werden diese auch bei jedem Zustandswechsel wieder abgegeben, denn die Nachbedingungen der Zustandswechsel in (3), (4) sowie (5) sorgen dafür, dass der aktuelle Prozess ein Privileg konsumiert und ein benachbarter Prozess dieses wieder erhält.

B. Leader Election in Ringen

Eine Leader Election ist ein Verfahren, um aus einer Menge an Prozessen einen speziellen auszuwählen, welcher dann Koordinationsaufgaben bewältigen kann. Dabei ist es wichtig, dass jeder Prozess über die Identität des gewählten Prozesses informiert ist, um eventuelle Anfragen an den Koordinator stellen zu können. Der folgende Algorithmus von Lin und Ghosh wurde in [8] vorgestellt. Er wählt einen Koordinator aus N Prozessen, die in einem Ring angeordnet sind. Jeder Prozess besitzt eine eindeutige ID, die ID des Prozesses i wird als id_i bezeichnet. Ziel ist, den Prozess mit der maximalen ID als Koordinator festzulegen. Um den Prozess mit der maximalen ID zu finden, kann ein Prozess mit seinem linken Nachbarn kommunizieren. Die Kommunikation erfolgt dabei durch das gegenseitige Lesen von Variablen des Nachbarprozesses und durch Setzen eigener Variablen. Jeder Prozess besitzt die Variablen max_i und $dist_i$.

Die maximale ID in einem Ring wird mit K bezeichnet. Ein stabiler Zustand ist definiert durch das Erfüllen zweier

Bedingungen. Die erste Bedingung lautet $max_i = K$, jeder Prozess kennt also die maximale ID. Die zweite Bedingung beschreibt die gültige Belegung von $dist_i$. Für den Koordinator j (der Prozess mit der ID K) gilt $dist_j = 0$, für alle anderen Prozesse i gilt $dist_i = 1 + dist_{(i-1) \bmod N}$. Die Variable $dist$ speichert also die Entfernung eines Prozesses zum Koordinator. Der Ablauf des Algorithmus für einen Prozess i ist folgendermaßen definiert. Der Index l kennzeichnet die $dist$-, id- und max-Variable des linken Nachbarn. id, max und $dist$ bezeichnen die eigenen Variablen.

Algorithm 6 Lin und Ghosh Algorithmus zur Leader Election in Ringen

loop
  if $id > max$ **or** $(id = max$ **and** $dist \neq 0)$
  or $(id \neq max$ **and** $dist = 0)$ **then**
    $max \leftarrow id$
  end if
  if $dist_l + 1 < N$ **and** $id < max_l$
  or not $(max = max_l$ **and** $dist = dist_l + 1)$ **then**
    $max = max_l$
    $dist = dist_l + 1$
  end if
  if $dist_l + 1 \geq N$ **and** $id > id_l$ **and** $id > max_l$
  and not $(max = id$ **and** $dist = 0)$ **then**
    $max = id$
    $dist = 0$
  end if
end loop

Die erste Bedingung aktualisiert den max-Wert, falls der aktuelle Prozess der ihm maximal bekannte ist. Die zweite Bedingung setzt die max- und $dist$-Werte korrekt, falls der linke Nachbar der Prozess mit der maximal bekannten ID ist. Die dritte Bedingung setzt den Prozess als Koordinator, falls der linke Nachbar angibt, die maximale Entfernung zum Koordinator zu besitzen oder falls die eigene ID größer ist, als die dem linken Nachbarn bekannte maximale ID und die ID des linken Nachbarn gleichzeitig kleiner ist als die eigene.

Man kann erkennen, dass jeder Prozess von den Variablen seines linken Nachbarn abhängig ist. Die $dist$-, id- und max-Werte des linken Prozesses werden mithilfe der eigenen Informationen bewertet und entscheiden, ob der aktuelle Prozess der Koordinator ist, oder ob er seinem rechten Nachbarn signalisieren muss, dass der Koordinator noch nicht ermittelt wurde. Dieses Verfahren sorgt dafür, dass sich das System nach endlich vielen Schritten in einem legalen Zustand befindet, nachdem es von einem beliebigen Zustand aus gestartet ist. Vorübergehende Fehler werden aktiv behandelt, indem ständig die Werte des linken Nachbarn betrachtet werden. Da jeder Prozess im Ring diese Betrachtung durchführt, wird das System vorübergehende Fehler korrigieren können.

VI. Fazit

Die in dieser Ausarbeitung vorgestellten Algorithmen und Prinzipien zur Selbststabilisierung zeigen, dass Selbststabilisierung ein wichtiges Forschungsgebiet im Bereich verteilter Systeme darstellt. Das Prinzip der Selbststabilisierung wurde als Ansatz zur Fehlertoleranz erläutert. Dabei wurde klar, dass Selbststabilisierung nicht den Ansatz der Maskierung von Fehlern verfolgt, sondern vorübergehende Fehler aktiv und garantiert eliminiert werden. Selbststabilisierung stellt daher einen allgemeinen Ansatz zur Fehlertoleranz in verteilten Systemen dar.

Die in Abschnitt V-A und Abschnitt V-B erläuterten Anwendungsbeispiele (Mutual Exclusion mit K-State Algorithmus, Leader Election in Ring-basierten Topologien) zeigen die vielfältigen Anwendungsgebiete für selbststabilisierende Algorithmen.

Darüber hinaus existieren Beweistechniken, um die Korrektheit von selbststabilisierenden Algorithmen formal beweisen zu können. In [2] wird unter anderem die Möglichkeit gezeigt, über eine Funktion, die über die Konfigurationsmenge eines Algorithmus definiert ist, die Konvergenzeigenschaft eines selbststabilisierenden Algorithmus zu zeigen. Eine solche Funktion steigt oder fällt monoton, wenn das System einen Berechnungsschritt durchgeführt hat. Erreicht die Funktion einen bestimmten Grenzwert, befindet sich das System in einer sicheren Konfiguration. Im Kontext des vorgestellten K-State Algorithmus kann diese Funktion beispielsweise für jede Konfiguration die Anzahl der aktuellen Prozesse liefern, die ein Privileg besitzen. Diese Funktion fällt monoton und erreicht nach einer endlichen Anzahl an Berechnungsschritten den Grenzwert 1, das System befindet sich nun in einer sicheren Konfiguration.

Literatur

[1] E. W. Dijkstra, “Self-stabilizing systems in spite of distributed control,” *Commun. ACM*, vol. 17, pp. 643–644, November 1974.

[2] S. Dolev, *Self-Stabilization*. MIT Press, 2000.

[3] S. Katz and K. J. Perry, “Self-stabilizing extensions for message-passing systems.” *Distributed Computing*, pp. 17–26, 1993.

[4] M. G. Gouda, R. R. Howell, and L. E. Rosier, “The instability of self-stabilization,” *Acta Informatica*, vol. 27, pp. 697–724, 1990, 10.1007/BF00264283.

[5] M. Schneider, “Self-stabilization,” *ACM Comput. Surv.*, vol. 25, pp. 45–67, March 1993.

[6] V. K. Garg, *Elements of distributed computing*. New York, NY, USA: John Wiley & Sons, Inc., 2002.

[7] N. Mittal and V. K. Garg, “A rigorous proof of $O(n^2)$ bound for Dijkstra’s 3-state algorithm,” University of Texas at Austin, Austin, TX, Technical Report ECE PDSLAB 2001, Jul. 2001, electrical and Computer Engineering Department.

[8] S. Ghosh and A. Gupta, “An exercise in fault-containment: Self-stabilizing leader election,” *Information Processing Letters*, vol. 59, no. 5, pp. 281 – 288, 1996.

Failure Detectors

Malin Gandor
Universität Oldenburg
Department für Informatik
Oldenburg, Germany
malin.gandor@uni-oldenburg.de

***Abstract*—These days, distributed systems deliver important services. A system crash can be expensive for the person who runs the system and often a crash can also be dangerous. As a consequence, distributed systems need to detect crashed processes.**

***Keywords*-failure detector; fault tolerance; distributed systems; failure models**

I. Introduction

A distributed system is defined in [1] as a "computer system that contains multiple processors connected by a communication network". Using a distributed system means one do not has to pool all processors in the same place. Therefore, distributed systems are often used these days to deliver services like air traffic control or patient monitoring. For these systems, it is very important to know if the system is available or not.

This paper describes the idea behind a so called "fault-tolerant distributed system" by defining the model behind such a system and which types of failures can occur in the system. After defining the system and possible failures, an abstract idea to detect the different types of failures with a so called *Failure Detector* module is presented.

A. Fault-tolerant distributed system model

To define the meaning of "fault tolerance", F.Cristian [9] declares a system as fault-tolerant, if there are, for a number well-defined failure behaviours to prevent a complete system crash. The well-defined failure behaviours specify a system behaviour, if a detected failure belongs to a defined number of tolerated failures. The tolerated failures are defined by a formal description and a classification. To describe the possible failure state of the system at any time, a failure model is needed, which is split into a structural model and a functional model [3]. The structural model declares the affected processes whereas the functional model defines the arising types of failures. For this paper, mainly the information from the functional failure model about the failures is needed. These can occur in different characteristics like

crash fault if the process stops without a reason [10],
omission fault if the process omits to respond to an input [9],
timing fault if the process needs too much or too less time to respond and the response is outside the specified time-interval (regardless whether the response was correct or incorrect)[9], and
Byzantine fault the process acts unpredictable [10].

As an abstract description, a distributed systems consists of processes which are connected with each other through channels. To implement tasks, the processes send messages to each other through these channels. In the theory of distributed systems, a crashed component does not mean automatically, that it has to be a process. Not receiving a message can also be the result of a crashed channel, which has stopped sending messages although the sending process works correctly. In this paper, we assume that channels are perfect can never crash. So, if a failure occurs, it has to be a process.

With information about the crashed process and the classification of the fault, the system theoretically can start its failure behaviour, so it has not to stop working on its tasks. If a failure occurs, the information about the failure are located in the process, where the failure appeared. Under obedience of *separation of concerns*, the idea is to collect all informations about the structural and functional failures in one external module. If, for example, the module holds all details of time-outs in the system, no process has to wait for messages delivered over channels which will arrive never or too late. This technique is called *Failure Detector*.

The following section describes the idea behind *Failure Detectors* and gives a definition of each Failure Detector class based on the properties *Completeness* and *Accuracy*.

II. (Un)reliable Failure Detector

A Failure Detector, as described by Jalote [1], is a from the system separated module which prevents a process from waiting for a message from a process which already is crashed. It consists of a number of failure detector modules so that each process has its own failure detector module. For example, if a process p_j will receive for a message from p_i, p_j is blocked until it receives the message. If p_i is crashed, process p_j will be blocked permanently. To prevent this, the failure detector module modifies the blocking receive

$$(wait \text{ for message } m \text{ from } p_i)$$

to

$$(wait \text{ for message } m \text{ from } p_i) \text{ or } (p_i \text{ is suspected}).[1]$$

This means that a process waits a certain timespan for the message from process p_i and if the message does not arrive, the failure detector *suspects* the process to be crashed. In a synchronous system with a shared clock, each failure detector module periodically sends, for example, a "heartbeat"-message to all other processes and waits for answers. If a process does not answer in a certain time period, the process is suspected [2]. But failure detector modules are not perfect and can make mistakes. A *unreliable failure detector* module can suspect an process to be failed although it has not (illustrated in Figure 1).

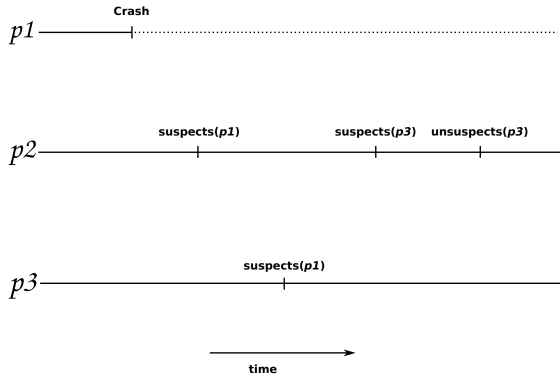


Figure 1. Illustration of an unreliable failure detector with 3 processes. Process 1 crashes and is suspected by process 2 and process 3. Process 2's failure detector suspects process 3 to be crashed too and revises its suspicion [2].

The system consists of three processors $p1, p2$ and $p3$ with its failure detector modules. Process $p1$ crashes and is as a result suspected by $p2$ and $p3$. Because process $p3$ works more slowly than $p2$, $p3$ is as well suspected by $p2$. After a certain time, process $p3$ sends its answer to $p2$, and $p2$ revises its suspicion. To revise its suspicion, the failure detector module removes the process from the list of suspected processes by *unsuspecting* the process. Each module can repeatedly add and remove processes from its list of suspects. So, at any given time, the two failure detector modules from process *p2* and *p3* can have different lists of suspects. For formal definitions the notation introduced in [1] and summarized in Figure II is used.

Based on the notation, in the following different classes of Failure Detectors are presented.

S	Set of States
s, t	States
Π	Set of all processes
i, j	Process indices
Π_c	Set of correct processes
C	Set of States on correct processes
C_j	States on a correct process
$failed(i)$	Process p_i has failed in the given run
$suspects(s, i)$	Process p_i is suspected in the state s
$permsusp(s, i)$	Process p_i is permanently suspected in state s

Figure 2. Notation

A. Classes of Failure Detectors

For process crashes, Chandra and Toueg[4] specify two abstract properties, *completeness* and *accuracy*, a Failure Detector must satisfy. A process is suspected by the Failure Detector if any of its failure detector modules suspects the process to be crashed.

Completeness means that a Failure Detector eventually suspect every process that has actually crashed and *accuracy* restricts the false suspicions a Failure Detector can make. Failure Detectors can satisfy two completeness properties [1]:

> *Strong Completeness* A failed process is eventually suspected by *all* correct processes.

Formally,

$$\forall i : \langle failed(i) \wedge \neg failed(j) \Rightarrow \exists_s \in C_j : permsusp(s, i) \rangle.$$

A failure detector can also satisfy *weak completeness* as defined in:

> *Weak Completeness* Every failed process is eventually permanently suspected by *some* correct process.

Formally,

$$\forall i : \langle failed(i) \Rightarrow \exists_s \in C : permsusp(s, i) \rangle.$$

However a failure detector with only one property is impractical. For example, a failure detector causes every correct process in the system to suspect every crashed process in the system. In this case the failure detector has a strong completeness but it is clearly useless since it provides no information about failures or false suspicions.
Failure Detectors also need accuracy to restrict the mistakes that they can make by false suspicions. There are four variants of *accuracy*: strong, weak, eventually strong and eventually weak.

> *Strong Accuracy* No process is suspected before it crashes.

Formally,

$$\forall i \in \Pi_c, \forall s \in C : \neg suspects(s, i).$$

In many practical systems, it is not possible to achieve strong accuracy, so there is a definition of *weak accuracy* too.

> *Weak Accuracy* Some correct process is never suspected.

Formally,

$$\exists i \in \Pi_c, \forall s \in C : \neg suspects(s, i).$$

Satisfying weak accuracy means that at least one correct process is never suspected. But just like strong accuracy, this property is difficult to achieve. So there are two subclasses of strong and weak accuracy, which can simply be described as strong/weak accuracy holds eventually.

> *Eventual Strong Accuracy* Eventually all correct processes are never suspected by any correct process.

Is formally defined as

$$\forall i \Pi_c, \forall j \in \Pi_c, \exists s \in C_j, \forall t \geq s : \neg suspects(t, i).$$

Similarly, *eventual weak accuracy* is defined as:

> *Eventual Weak Accuracy* Some correct process is eventually never suspected by any correct process.

Formally, a Failure Detector satisfies eventually weak accuracy if

$$\exists i \in \Pi_c, \forall j \in \Pi_c, \exists s \in C_j : \forall t \geq s : \neg suspects(t, i).$$

All failure detector classes are formed by combining completeness and accuracy properties as shown in Figure 3. For example, a Failure Detector is called *Perfect* if it satisfies strong completeness and strong accuracy. Such a *Perfect Failure Detector* makes no false suspicions and detects every failure. It is denoted by $\mathcal{P}$. The class of *eventually strong failure detectors* ($\Diamond\mathcal{S}$) is defined by strong completeness and eventual weak accuracy.

Completeness	**Accuracy**			
	Strong	Weak	E. Strong	E. Weak
Strong	Perfect $\mathcal{P}$	Strong $\mathcal{S}$	E. Perfect $\Diamond\mathcal{P}$	E. Strong $\Diamond\mathcal{S}$
Weak	$\mathcal{L}$	Weak $\mathcal{W}$	$\Diamond\mathcal{L}$	E.Weak $\Diamond\mathcal{W}$

Figure 3. Different classes of failure detectors characterised by completeness and accuracy properties [2]

Failure Detectors from different classes can be transformed into another class. The relationship among the classes and an example, how a Failure Detector from one class is used to implement a Failure Detector from another class, is presented in the following.

B. Relationship Among the Failure Detectors

The presented Failure Detector classes are related and in some ways one Failure Detector can be used to implement a Failure Detector from another class. If in a system exists a transformation which uses information from Failure Detector X to implement Failure Detector X', it means that X' is *weaker than* X. Let $X \succeq X'$ be the formal definition of *X' is weaker than X*, the relations between classes of Failure Detectors are defined as

$$\mathcal{P} \succeq \mathcal{L},\ \mathcal{S} \succeq \mathcal{W},\ \Diamond\mathcal{P} \succeq \Diamond\mathcal{L},\ \Diamond\mathcal{S} \succeq \Diamond\mathcal{W} \text{ [4]}.$$

For example, $\Diamond\mathcal{S}$ is an *eventually strong* detector and $\Diamond\mathcal{W}$ is an *eventually weak* detector. As shown in Figure 3, $\Diamond\mathcal{S}$ provides strong completeness and eventual weak accuracy and $\Diamond\mathcal{W}$ weak completeness and eventual weak accuracy. After the definition of strong and weak, $\Diamond\mathcal{W}$ is obviously weaker than $\Diamond\mathcal{S}$. But on the other hand, Jalote [1] shows $\Diamond\mathcal{S}$ is also weaker than $\Diamond\mathcal{W}$. Figure 4 shows an implementation of a $\Diamond\mathcal{S}$ detector. *ES.suspects* and *EW.suspects* are the set of processes suspected by $\Diamond\mathcal{S}$ and $\Diamond\mathcal{W}$. The detector works with two activities:

```
Pj::
var
       ES.suspects: set of processes initially EW.suspects
(I1)   send EW.suspects to all processes infinitely often;
(I2)   On receiving m.suspects from p_i;
       ES.suspects := ES.suspects ∪ m.suspects − {i};
```

Figure 4. Implementation of a $\Diamond\mathcal{S}$ detector using a $\Diamond\mathcal{W}$ detector [1]

- Each process sends his suspicions to all other processes.
- If a process receives a message from p_i, it queries its $\Diamond\mathcal{W}$ detector and joins its list of suspicions with the send list of suspicions from p_i and removes p_i itself from the list.

The correctness of the transformation is proven in [1]. Chandra et al. [4] describe an algorithm, which can transform a Failure Detector $\mathcal{D}$ into a Failure Detector $\mathcal{D}'$ (this algorithm is called *reduction algorithm*). The reduction algorithm shows, that each Failure Detector from the second row in Figure 3 is equivalent to the class above it. For example, the reduction algorithm transforms a Failure Detector from *weak completeness to strong completeness*, by strengthening Completeness and preserving Accuracy. Figure 5 shows the relation between all classes of Failure Detectors under the reducibility relation. The relations are defined as follows:

- An *undirected edge* between two classes implies, that both classes are equivalent, and
- a *directed edge* from class C to C' implies, that C is *weaker* than C.

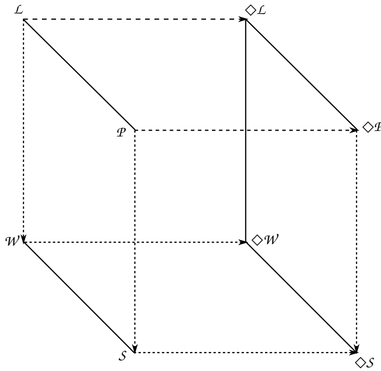


Figure 5. Comparing the eight Failure Detector classes by reducibility [4]

The question is, even though $\Diamond\mathcal{S}$ is weaker than $\mathcal{P}, \mathcal{S}$ and $\Diamond\mathcal{P}$, is $\Diamond\mathcal{S}$ still "strong enough" to solve common problems in distributed systems? The following section presents common problems in distributed systems, and gives an answer to this question.

III. Application of Failure Detectors to Solve Common Problems in Distributed Systems

After having described the basic ideas and classes of Failure Detectors, common problems in distributed systems and a Failure Detector solving *Consensus* is presented. The following problems are mentioned exemplarily as common problems in distributed systems:

Leader Election [6] After a process fails the system needs to reorganize itself. To keep the reorganization process as simple as possible, it is managed by a single process, called coordinator. As the name *Leader Election* implies, the coordinator is elected by all active/correct processes.

Mutual Exclusion [7] In this problem, the system is described as a collection of asynchronous processes, each executing critical and non critical sections. The question is how to synchronize all processes such that no two processes ever executes their critical sections concurrently.

Reaching Consensus [3] Consensus describes a form of agreement among the processes of a distributed system. To reach these consensus agreements, the processes require a form of global consistency (for example by synchronising the clocks or contents of its communication).

Obviously, *Reaching Consensus* is a fundamental problem, as e.g, a system cannot elect a leader without making an agreement about the coordinator. Mutual Exclusion also needs consensus to synchronize its processes. The following section takes a detailed consideration of the consensus problem and describes a Failure Detector solving it in a weak asynchronous system [1].

A. Failure Detector solving Consensus

To understand the failure detector solving consensus, an understanding of the consensus problem itself is needed. In fault-tolerant distributed computing, consensus is given by a property, which is divided in one liveness and two safety properties. These properties are defined in [2] as follows:

Uniform Termination: Every process eventually decides on some value.

Uniform Agreement: No two processes decide differently.

Validity: If a process decides a value v, then v was supposed by some process.

Uniform Termination and Uniform Agreement are hard or, in many cases, impossible to ensure in the presence of faults. For example, process p_i crashes before it can make his decision about value v. Because p_i crashes, it cannot decide anymore and the *Uniform Termination* property is never reached. An idea to ensure these properties also in presence of crash failures is to weaken the problem by allowing a substitution from *uniform termination* to *termination* and from *uniform agreement* to *agreement*. These weakened properties are defined as:

Termination: Every correct process eventually decides on some value.

Agreement: No correct two processes decide differently.

So the consensus problem is defined by the intersection from these three properties *termination, agreement* and *validity*.

Figure 6 shows a possible procedure of reaching consensus with one crash failure. First, each correct process

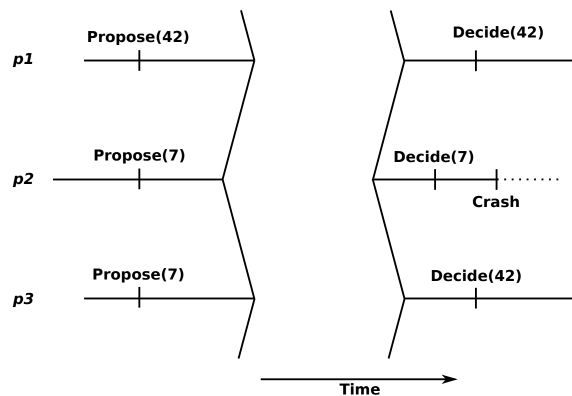


Figure 6. Reaching consensus wit h one crash failure. [2]

proposes a value (42 and 7) so the procedure satisfies *termination*. After having proposed their values, each process

makes its decision. Process $p2$ and $p3$ both decide on value 42, so agreement is also satisfied. Process $p2$ makes a different decision than $p1$ and $p3$ but since $p2$ is the faulty process in this scenario, $p2$'s decision will not be important for reaching consensus.

A failure detector solving consensus has to satisfy these three properties. Jalote [1] shows that an algorithm can solve the consensus in a weakened asynchronous environment with a $\Diamond S$ detector by using the *rotating coordinator paradigm*. Fisher et. al have shown in [3] that it is impossible to solve consensus with at least one faulty process in a pure asynchronous system without any timing assumptions. A pure asynchronous system has no shared clock or an uniform definition for a timespan (e.g. rounds) to differentiate crashed from slow processes. Therefore, the algorithm in [1] needs a weakened asynchronous system with a well-defined timespan or round.
For reaching consensus, the algorithm needs at least a majority of correct processes in the system. Let n be the number of processes in the system. Then, majority means that $\lceil(n + 1)/2\rceil$ processes have to be correct. In the following algorithm an iteration is defined as round r. In each round, the system has to choose a coordinator for the following phases. Different from the leader election problem, every process knows the actual number r. The coordinator for this round is the process with its process number $p = r \; mod \; n$.

After choosing a coordinator, a round consists of four phases:

Phase 1: Each process makes its estimation and sends it to the coordinator.
Phase 2: The coordinator collects the estimation since it has as much estimations as the majority of the processes in the system. After receiving enough estimations, the coordinator proposes a new estimation.
Phase 3: Each process waits for the coordinator's estimation. If it receives a value from the coordinator it sends back a positive message as accepting message. Otherwise the process sends back a negative message this implies that the process suspects the coordinator to have failed.
Phase 4: The coordinator waits until it gets the majority of answers from the processes. If the majority of answers are with positive context, the coordinator decides on this estimation value and sends it to all processes. If not, the algorithm increases r and starts the iteration with another coordinator again. On receiving the value, the processes decide on this value and stop.

With this algorithm the consensus problem can be solved in a weakened asynchronous system, because the coordinator decides on one value, sends it to all correct processes, and they decide on this value. But, if there are m crashed processes with $m > \lceil(n + 1)/2\rceil$, none of the coordinators get a majority of positive messages and the algorithm never terminates.

IV. Conclusion

In a distributed system different types of failures can occur. The paper has described common types of failures, the concept of Failure Detectors and an application to solve problems in presence of faulty processes. To ensure fault-tolerance in a distributed system, it is important to locate and identify the faulty processes. A crashed process can be detected in a synchronous system by time-outs. But if the process acts unpredictable, like Byzantine Faults, the process is hardly identifiable.

As described, Failure Detectors allow to encapsulate timing assumptions. If a distributed system does not need time constraints, the time assumption can completely used for failure detection. Failure Detectors can be used to solve common problems in a synchronous system, if there is at least a majority of correct processes. To detect failures in an asynchronous system there need to be some time assumptions to weaken system. In a pure asynchronous system the consensus problem can never solved with at least one faulty process. Without a homogeneous timespan, the Failure Detector cannot differentiate between crashed and slow process. As presented, a Failure Detector can only solve the Consensus Problem in an asynchronous system, if the system is weakened by defining rounds.

References

[1] P. Jalote, *Fault Tolerance in Distributed Systems* US Edition, Prentice-Hall, 1994

[2] T. Warns, *Structural Failure Models for Fault-Tolerant Distributed Computing*, 1st ed. Vieweg+Teubner, 216 Pages, 2009.

[3] M.J Fischer, N.A. Lynch, M.P. Paterson, *Impossibility of Distributed Consensus with One Faulty Process* Journal of the ACM Vol. 32, No. 2, Page 374-382, 1985

[4] T.D. Chandra and S. Toueg, *Unreliable failure detectors for reliable distributed systems* Journal of the ACM, Vol. 43, No. 2, Page 225-267, 1996.

[5] T.D Chandra, V. Hadzilacos and S. Toueg, *The Weakest Failure Detector for Solving Consensus* Journal of the ACM Vol. 43, No. 4, Page 685-722, 1996.

[6] H. Garcia-Molina, *Elections in a Distributed Computing System* IEEE Transactions on Computers, Vol. C-31, Pages 48-59, 1982.

[7] L. Lamport, *The Mutual Exclusion Problem, Part I & II* Journal of the ACM Vol. 33, No. 2, Page 313-348, 1986

[8] C. Echtle, *Fehlertoleranzverfahren* Springer, 343 Pages, 1990

[9] F. Cristian, *Understanding Fault-tolerant Distributed Systems* Communications of the ACM, Vol. 34, No. 2, Page 56-78, 1991

[10] O. Theel, *Fehlertoleranz in verteilten Systemen* Vorlesungsfolien, Carl-von-Ossietzky Universität Oldenburg, Abteilung Systemsoftware und verteilte Systeme, Sommersemester 2011