Algorithmen zum gegenseitigen Ausschluß in verteilten Systemen

Diplomarbeit

Humboldt-Universität zu Berlin

Mathematisch-Naturwissenschaftliche Fakultät II

Institut für Informatik

Lehrstuhl für Softwaretechnik und Theorie der Programmierung II

Berlin, Dezember 1999

Autor:

Claus Wagner

Betreuer:

Dr. Frank Müller

Gutachter:

Pr. Dr. sc. nat. Klaus Bothe und Dr. Andreas Polze

Thesen

1. Algorithmen zum gegenseitigen Ausschluß in verteilten Systemen können durch ein Protokoll unterstützen werden, welches gleichzeitig mehreren Lesern oder einem Schreiber den Zutritt in die kritische Sektion gewährt.

2. Die in dieser Arbeit entwickelten Algorithmen zur Realisierung von Lese- und Schreibsperren arbeiten korrekt.

3. Der Gewinn durch den Einsatz von Algorithmen für Lese- und Schreibsperren bezüglich der Wartezeit und der Gesamtausführungszeit rechtfertigt ihren Einsatz.

4. Der in dieser Arbeit entwickelte Algorithmus zum gegenseitigen Ausschluß in verteilten Systemen kann Fehler tolerieren.

5. Der fehlertolerante Algorithmus arbeitet korrekt.

6. Netzwerk- und Knotenfehler werden erkannt und können behoben werden.

7. Die Fehlerbehebung beim Auftreten von Knotenfehlern ist in Abhängigkeit von der Fehlerrate und der logischen Netzwerktopologie effizient.

Ausarbeitung:

Der Stand der Forschung wurde anhand einer Literaturrecherche besprochen. Es erfolgte eine Einteilung von Algorithmen zum gegenseitigen Ausschluß in verteilten Systemen.

Entwickelt wurden zwei Algorithmen, die Schreib- und Lesesperren realisieren. Die Algorithmen wurden informell und anhand von Beispielen beschrieben. Die Komplexität sowie die Performance der Algorithmen wurde untersucht. Eine für die Untersuchung der Performance notwendige Testumgebung wurde geschaffen. Diese Testumgebung simuliert ein verteiltes System lokal. Mit Hilfe der Testumgebung wurde das Verhalten bei verschiedenen Eingabeparametern untersucht.

Es erfolgte eine Einteilung der in einem verteilten System möglichen und für die Algorithmen zum gegenseitigen Ausschluß relevanten Fehler. Als Grundlage diente das von F. Cristian entwickelte und heute als Quasistandard geltende Fehlermodell [46]. Realisierungsmöglichkeiten einer fehlertoleranten Kommunikation wurden diskutiert. Ein fehlertoleranter Algorithmus für den gegenseitigen Ausschluß wurde entwickelt und informell beschrieben. Die Komplexität des Algorithmus wurde untersucht. Bewiesen wurde die Korrektheit des Algorithmus nach einer erfolgten Fehlerbehandlung.

Die Testumgebung wurde um fehlertolerante Kommunikation erweitert. Danach folgte eine Untersuchung der Performance des fehlertoleranten Algorithmus. In den Testläufen wurden für ein verteiltes System typische Fehlerszenarien erzeugt.

Kurzfassung

Algorithmen zum gegenseitigen Ausschluß können ein Protokoll unterstützen, welches gleichzeitig entweder mehreren Lesern oder einem Schreiber den Zugriff auf Ressourcen in einer kritischen Sektion gestattet. Algorithmen, die solche Protokolle implementieren, müssen effizient arbeiten: Der Gewinn durch die Parallelisierung der Leseoperationen muß den algorithmischen Mehraufwand der Protokolle wettmachen.

Behandelt werden tokenbasierende Algorithmen. Der Besitz des Tokens sichert im Fall des Schreibens einen exklusiven Zugriff und im Fall des Lesens den Ausschluß von Schreibern zu. Tokenbasierende Algorithmen haben im besten Fall eine Komplexität der Nachrichten von 0. Es ist keine Kommunikation notwendig, wenn der Tokenbesitzer seine kritische Sektion betreten will. In dieser Arbeit werden zwei um Lese- und Schreibsperren erweiterte Algorithmen vorgestellt. Die Algorithmen haben eine Komplexität der Nachrichten in Abhängigkeit von der Größe des Systems und je kritischer Sektion von $\mathrm{O}(N)$ bzw. $\mathrm{O}(\log_2 N)$ im schlechtesten Fall. Die Messungen in einer Testumgebung, die ein verteiltes System lokal simuliert, ergaben folgendes: Die Wartezeit je kritischer Sektion nimmt mit der Erhöhung der als Lesezugriffe ausgezeichneten Anzahl der kritischen Sektionen linear ab, die Anzahl der Nachrichten nimmt zu. Die negative Steigung der mittleren Wartezeit nimmt mit der Erhöhung der Größe des Systems zu und bleibt gleich bei der Erhöhung der Anzahlen kritischer Sektionen. Die mittlere Wartezeit liegt bei 10% Lesesperren bis zu doppelt so hoch wie bei 90% Lesesperren. Die Anzahl der Nachrichten pro kritischer Sektion steigt um maximal eins.

In einem verteilten System können Fehler auftreten. Unterschieden werden Knoten- und Netzwerkfehler. Knotenfehler bedeutet den Ausfall eines Knotens ohne vorherige Ankündigung, wie z. B. ein Signal vom Betriebssystem an laufende Prozesse. Der Verlust von Nachrichten und die dauerhafte Störung von Nachrichtenverbindungen sowie semantische Veränderungen von Nachrichten werden unter dem Begriff Netzwerkfehler zusammengefaßt. Netzwerkfehler können, sofern dem System kein zuverlässiges Netzwerk zugrunde liegt, durch eine fehlertolerante Kommunikation erkannt und behoben werden. Die fehlertolerante Kommunikationseinheit nutzt Quittungen und Timeouts. Weiterhin sind die fehlertolerante Kommunikationseinheit bzw. das zuverlässige Netzwerk für das Erkennen der meisten Knotenfehler verantwortlich. Nach einer Anzahl von Versuchen eine nicht quittierte Nachricht erneut zu senden, wird kapituliert und die Behebung eines Knotenfehlers ausgelöst. Knotenfehler, die nicht durch die Kommunikationseinheit erkannt werden, werden durch eine weitere Einheit erkannt. Diese Einheit arbeitet wiederum mit Timeouts. Die Evaluierung der Timeoutintervalle gehört zum Feintuning eines verteilten Systems und hängt stark von der Größe des Systems, den Eigenschaften der kritischen Sektionen und der zugrunde liegenden Architektur ab.

Aufbauend auf ein zuverlässiges Netzwerk bzw. auf die fehlertolerante Kom-

munikation wird ein fehlertoleranter Algorithmus entwickelt. Er ist tokenbasierend und arbeitet im fehlerfreien Fall mit einer Komplexität der Nachrichten im mittleren Fall (ac) von $\mathrm{O}(\ln N)$. Für die Fehlertoleranz ist die Speicherung von zusätzlichen Informationen auf jedem Knoten notwendig, neue Nachrichten müssen eingeführt werden und die Datenstruktur muß erweitert werden. In dem vorgestellten fehlertoleranten Algorithmus wird eine weitere logische Nachrichtentopologie benutzt. Die Knoten des Systems werden, zusätzlich zu der von dem Basisalgorithmus benötigten Struktur, in einem doppelt verketteten Ring organisiert. Jeder Knoten erhält einen Vorgänger und einen Nachfolger. Daraus folgt ein linearer Speichermehraufwand. Im Fehlerfall wird der Ring als Nachrichtentopologie genutzt. Nach erfolgter Fehlerbehebung wird der durch den Fehler möglicherweise verletzte Ring wieder geschlossen. Der fehlertolerante Algorithmus hat einen linearen Speichermehraufwand pro Knoten und benötigt für die Behebung eines Knotenfehlers oder eines dauerhaften Netzwerkfehlers $\mathrm{O}(N)$ Nachrichten.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

1 Einführung

In den letzten Jahren hat das Gebiet des parallelen und verteilten Rechnens an Bedeutung gewonnen. Einige Probleme sind erst durch die Verarbeitung auf mehreren Prozessoren lösbar. Weiterhin zeichnet sich ein Trend von massiven Parallelrechnern zu verteilten Systemen ab, welche aus herkömmlichen Workstations oder PCs und einem Netzwerk bestehen. Die Lösung des Synchronisationsproblems ist eines der wichtigsten Probleme in einem verteilten System.

Ohne geeignete Synchronisation kann beispielsweise ein konsistenter Zustand der Daten des Systems nicht zugesichert werden. Ein Konzept zur Synchronisation ist der gegenseitige Ausschluß.

1.1 Allgemeines

Gegenseitiger Ausschluß ist ein wichtiges Konzept, um gemeinsam genutzte Ressourcen in verteilten Systemen zu schützen. Eine Anzahl von Prozessen [1] synchronisiert ihren Zugriff auf gemeinsame Daten durch den Eintritt in eine kritische Sektion.

Diese Arbeit beschäftigt sich mit tokenbasierenden Algorithmen, die den gegenseitigen Ausschluß in einer kritischen Sektion zusichern. Das Privileg, die kritische Sektion zu betreten, wird bei tokenbasierenden Algorithmen durch das sogenannte Token repräsentiert. Der Prozeß, der das Token besitzt, und nur dieser, hat uneingeschränkten Zugriff auf die gemeinsamen Ressourcen. Es existiert zu jedem Zeitpunkt genau ein Token im System.

Der Schwerpunkt dieser Arbeit liegt auf Algorithmen, die den Mehrfacheintritt in Abhängigkeit von den in der kritischen Sektion ausgeführten Operationen in die kritische Sektion zulassen. Leseoperationen auf einem Datum verlangen den Ausschluß von Schreiboperationen auf dem gleichen Datum, müssen jedoch nicht im gegenseitigen Ausschluß mit anderen Lesezugriffen ablaufen. Daher bietet sich ein Schreiber-Leser-Protokoll an, welches Lesezugriffe parallelisiert und Schreibern den exklusiven Zugriff sichert.

Der zweite Schwerpunkt liegt auf Algorithmen, die typische Fehler eines verteilten Systems tolerieren. Im wesentlichen sind das der Ausfall eines Knotens, Fehler des zugrundeliegenden Netzwerks und, daraus resultierend, die Zerlegung des Systems in Partitionen.

Die Algorithmen müssen korrekt und hinreichend effizient arbeiten. Bei der Effizienz ist in erster Linie die Anzahl der benötigten Nachrichten zu betrachten. Das zugrundeliegende Netzwerk bildet meist den Flaschenhals eines verteilten Systems. Der Nachrichtenverkehr ist soweit wie möglich zu minimieren.

[1] In der Literatur werden auch die Begriffe Nodes, Sites, Knoten oder Tasks verwendet.

1.2 Aufbau der Arbeit

Die vorliegende Arbeit ist wie folgt gegliedert:

Das Kapitel 2 beschreibt die Ideen, die verfolgt wurden und motiviert die Arbeit.

Bezeichnungen, grundlegende Erklärungen sowie Abkürzungen werden in Kapitel 3 (Terminologie) eingeführt.

Im Abschnitt 4 werden Hintergrundarbeiten vorgestellt, der Stand der Forschung beschrieben und eine Einteilung der Methoden zum gegenseitigen Ausschluß vorgenommen.

Die für die Untersuchung der Algorithmen im Hinblick auf die Performance und das allgemeine Verhalten notwendige Testumgebung wird in Kapitel 5 beschrieben.

Die Ausführungen zu Lese- und Schreibsperren befinden sich in Kapitel 6 (Algorithmen zur Realisierung von Lese- und Schreibsperren). Zwei Algorithmen für die Realisierung von Schreib- und Lesesperren werden eingeführt, deren Korrektheit bewiesen und eine Untersuchung der Komplexität und Performance durchgeführt.

Im zweiten Teil der Arbeit, Kapitel 7 ff., wird die Fehlertoleranz von Algorithmen zum gegenseitigen Ausschluß behandelt. Fehlerarten und Fehlermodelle werden besprochen und grundlegende Begriffe eingeführt. Es erfolgt eine Einteilung von Fehlern in einem verteilten System und deren Wirkungen auf die Algorithmen zum verteilten gegenseitigen Ausschluß. Aufbauend auf eine fehlertolerante Kommunikation wird ein um Fehlertoleranz erweiterter Algorithmus zum gegenseitigen Ausschluß informell beschrieben und anhand von Beispielen illustriert. Die Korrektheit des Algorithmus wird informell gezeigt und eine Analyse der Komplexität der Nachrichten durchgeführt.

Es folgen die Ergebnisse der Untersuchung des Algorithmus in der Simulationsumgebung.

Im Anschluß an die Hauptteile der Arbeit, Kapitel 6 und Kapitel 8, befindet sich jeweils eine kurze Zusammenfassung.

Abschließende Bemerkungen und Ausblicke findet der Leser in Abschnitt 9 (Ausblick).

2 Motivation

Eine Vielzahl von Algorithmen zum gegenseitigen Ausschluß in verteilten Systemen wurden in den letzten zwei Jahrzehnten veröffentlicht.

In den meisten der vorgestellten Arbeiten ging man davon aus, daß die Knoten des Systems exklusiven Zugriff auf Ressourcen in einer kritischen Sektion benötigen. Unbetrachtet blieb dabei, welche Operationen in der kritischen Sektion tatsächlich ausgeführt werden. Das exklusive Eintreten in die kritische Sektion impliziert jedoch eine Schreiboperation. Betrachtet man ein verteiltes System, so stellt man fest, daß es sich bei den meisten Zugriffen auf gemeinsame Daten um lesende Zugriffe handelt. In einem verteilten Datenbanksystem beispielsweise sind ca. 90 % aller Zugriffe Lesezugriffe [43]. Lesezugriffe jedoch erfordern nur den Ausschluß von gleichzeitigen Schreibzugriffen. Schreibzugriffe wiederum benötigen exklusiven Zugriff.

Dies motivierte z.B. dazu, die Posix-Thread-Interfaces um die Funktionen *pthread_rwlock_rdlock()* für einen Lesezugriff und *pthread_rwlock_wrlock()*, für einen Schreibzugriff zu erweitern.

Um den Gewinn durch die Parallelisierung der Leseoperationen zu nutzen, müssen Schreib- und Lesesperren effizient implementiert werden. Effizient bedeutet, daß die Anzahl der Nachrichten minimiert und die Wartezeit bis zum Betreten der kritischen Sektion (schreibend oder lesend) verkürzt wird. Der algorithmische Mehraufwand muß durch die Verkürzung der Wartezeit wettgemacht werden.

Forschungen der letzten Jahre und eigene Untersuchungen zeigen, daß tokenbasierende Basisalgorithmen bei den relevanten Parametern der Performance, Anzahl der Nachrichten und mittlere Wartezeit, die besten Ergebnisse liefern. Dies motiviert eine Erweiterung tokenbasierender Algorithmen um Lese- und Schreibsperren. Zu untersuchen ist die Zunahme der Anzahl der Nachrichten und die mittlere Wartezeit im Vergleich zu den Basisalgorithmen, um den erhöhten algorithmischen Aufwand plausibel zu machen. Die erweiterten Algorithmen müssen korrekt sein, d.h. durch die Erweiterung darf weder der gegenseitige Ausschluß verletzt sein, noch dürfen Verklemmungen oder das Verhungern von Knoten auftreten.

Weiterhin wird in vielen Arbeiten zum gegenseitigen Ausschluß in verteilten Systemen von einem fehlerfreien verteilten System ausgegangen, das es quasi nicht gibt. Das Netzwerk, auf dem das verteilte System basiert, ist selten zuverlässig, d.h. Nachrichten können verloren gehen oder semantisch verändert werden. Weiterhin können Knoten (Rechner) ausfallen und Netzwerkverbindungen dauerhaft unterbrochen werden. In den meisten Fällen wird durch einen dieser Fehler die Korrektheit des Algorithmus verletzt. Fehlertoleranz ist in Echtzeitsystemen existenziell. Aussagen, wie: Beim Auftreten von n Fehlern ist das System weiterhin verfügbar oder beim Auftreten von k Fehlern muß kapituliert und gegebenenfalls neu gestartet werden, sind nötig. Die fehlertoleranten Algorithmen zum gegenseitigen Ausschluß müssen:

- im fehlerfreien Fall effizient arbeiten und
- im Fehlerfall die Fehlerbehebung, Erkennung und Behandlung hinreichend effizient realisieren.

Zu untersuchen ist somit die Performance im fehlerfreien Fall und die Performance beim Auftreten von Fehlern in einer bestimmten Frequenz. Wiederum motivieren die guten Ergebnisse der tokenbasierenden Algorithmen zu einer Erweiterung dieser um Fehlertoleranz, da die erste obengenannte Bedingung bereits erfüllt ist.

3 Terminologie

Im folgenden wird ein verteiltes System betrachtet. Ein *verteiltes System* besteht aus (von einander unabhängigen) Knoten (Workstations, PCs), die keinen gemeinsamen Speicher teilen und keine gemeinsame Uhr besitzen. Die Knoten kommunizieren ausschließlich über ein zugrundeliegendes Netzwerk. Das Netzwerk ist logisch abgeschlossen, d.h. jeder Knoten kann mit jedem kommunizieren, sofern die Adressen bekannt sind. Die Größe des verteilten Systems, d. h. die Anzahl der Knoten, wird mit N bezeichnet.

Die mittlere Anzahl der Nachrichten, die ein Knoten benötigt, um seine kritische Sektion zu betreten, wird mit nmE, die mittlere Wartezeit vom Zeitpunkt des Versuchs bis zum Eintritt in die kritische Sektion mit rt für response time abgekürzt. Auf den verschiedenen Knoten wird derselbe Programmcode abgearbeitet, der Kontrollfluß jedoch kann, im speziellen zeitlich, unterschiedlich sein. Verteilter Programmcode besteht aus kritischen und nichtkritischen Teilen. Im kritischen Teil befinden sich die Operationen, die im gegenseitigen Ausschluß abgearbeitet werden müssen. Der kritische Teil wird als *kritische Sektion* bezeichnet und mit cs abgekürzt.

Die Verweildauer in der kritischen Sektion wird mit Phi, die Zeit vom Verlassen bis zum Wiederanfragen der kritischen Sektion, der nichtkritische Teil, mit $Gamma$ bezeichnet.

ncs bezeichnet die Anzahl der kritischen Sektionen, die auf einem Knoten abgearbeitet werden. Die Anzahl der kritischen Sektionen kann auf den verschiedenen Knoten unterschiedlich sein, da der Kontrollfluß nicht identisch ist. Somit ist ncs als obere Schranke anzusehen.

Ein Algorithmus zum gegenseitigen Ausschluß in verteilten Systemen ist *korrekt*, wenn er

1. den gegenseitigen Ausschluß sichert (mutual exclusion),
2. frei von Verklemmungen ist (freedom of deadlock),
3. das Verhungern von Knoten vermeidet (absence of starvation) und
4. die Weitergabe des Tokens muß auf faire Weise erfolgen (fairness).

Dazu ist zu bemerken, daß die Bedingung der Fairness die Vermeidung des Verhungerns einschließt.

Ein Algorithmus für die Realisierung von Lese- und Schreibsperren ist *korrekt*, wenn Bedingung 2. und 3. erfüllt und wenn zu einem Zeitpunkt entweder maximal ein schreibender oder beliebig viele lesende Zugriffe gestattet werden.

4 Hintergrundarbeiten

Zum erstenmal wurde das Problem konkurrierender Zugriffe von Dijkstra 1965 aufgeworfen [2]. Später diskutierte ebenfalls Dijkstra das Problem anhand des Beispiels der "Speisenden Philosophen". Seitdem wurde eine Vielzahl von Lösungen für das Problem des gegenseitigen Ausschlusses in verteilten Systemen entwickelt und veröffentlicht.

4.1 Allgemeines

In den Arbeiten des letzten Jahrzehnts werden zwei Lösungsansätze unterschieden: **tokenbasierende** und **nicht tokenbasierende**. Bei den tokenbasierenden Algorithmen entscheidet der Besitz eines logischen Tokens über das Privileg, exklusiven Zugriff auf gemeinsame Ressource innerhalb einer kritischen Sektion zu haben. In einem System existiert zu jedem Zeitpunkt genau ein Token. Durch Weitergabe des Tokens, auf möglichst faire Weise, kann jeder Knoten irgendwann seine kritische Sektion betreten. Tokenbasierende Algorithmen werden in [3], [14], [16], [37] und [41] vorgestellt.

Der nicht tokenbasierende Ansatz verlangt von einem Knoten, der in die kritische Sektion eintreten will, die Erlaubnis dazu von einer Untermenge der Knoten des Systems einzuholen.[2] Zu den nicht tokenbasierenden Algorithmen zählen die in [9], [13], [29], [35], [38] und [40] vorgestellten Arbeiten. Eine weitere Einteilung von Algorithmen zum gegenseitigen Ausschluß kann in dynamisch und nicht dynamisch vorgenommen werden. Hängen die Aktionen eines Knotens, zum Betreten seine kritischen Sektion vom momentanen Zustand des Systems und der Variablenbelegung des Knotens ab, ist der Algorithmus dynamisch, anderenfalls ist er statisch.

Tokenbasierende Algorithmen können wiederum in zwei Klassen eingeteilt werden: auf logischen Strukturen basierende und Broadcast-basierende Algorithmen. Broadcast-basierende Algorithmen benötigen keine logische Struktur auf den Knoten und senden Anfragen parallel. Auf logischen Strukturen basierende Algorithmen haben eine Struktur, die vorgibt, ob, wie und wohin Nachrichten versendet werden und welche Aktionen auszuführen sind. Es gibt eine logische Konfiguration auf jedem Knoten. Die auf logische Strukturen basierenden Algorithmen unterteilen sich in zwei Gruppen: Es gibt Algorithmen, die auf statischen, und Algorithmen, die auf dynamischen, logischen Strukturen arbeiten. Bei den Erstgenannten ändert sich der Graph, der die Kommunikationswege zwischen den Knoten bestimmt, nicht. Lediglich die Richtungen der Kanten können sich verändern. Bei den auf dynamischen logischen Strukturen basierenden Algorithmen können sich die Kanten und die Richtungen der Kanten ändern. Einen Überblick über die Einteilung von Algorithmen zum gegenseitigen Ausschluß gibt der

[2]Diese Untermengen werden in der Literatur als Quorums bezeichnet.

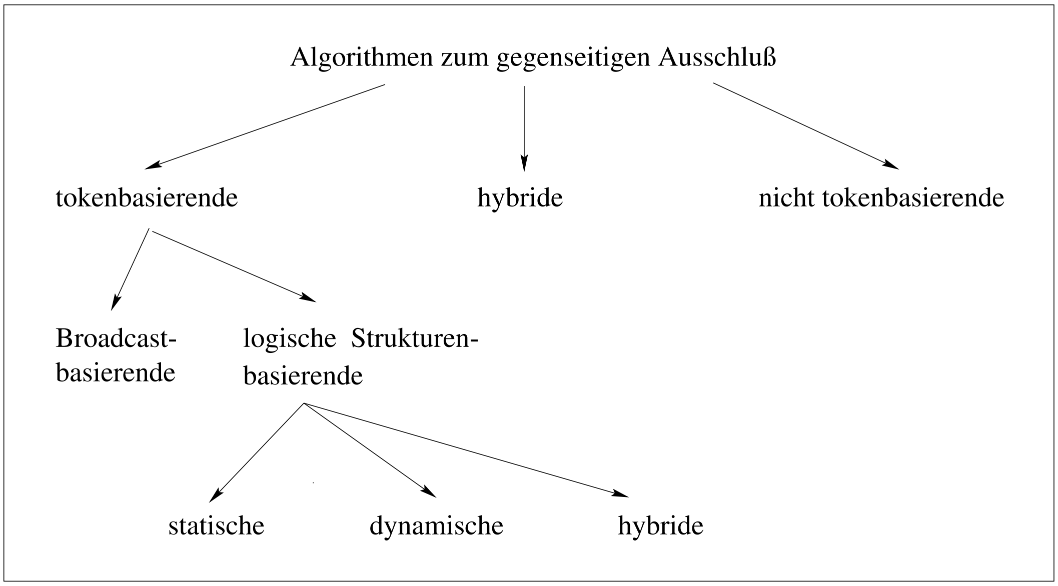


Abbildung 1: Einteilung von Algorithmen für den gegenseitigen Ausschluß.

Graph in Abbildung 1. Chang et al. verfolgten einen **hybriden** Ansatz [28]. Das Verfahren bildet Gruppen von Knoten. Die Knoten innerhalb der selben Gruppe verwenden einen anderen Algorithmus zum gegenseitigen Ausschluß als Knoten aus verschiedenen Gruppen. Die verwendeten Algorithmen können sowohl nicht tokenbasierend als auch tokenbasierend sein.

Ein Algorithmus, der O(N) Nachrichten für den Eintritt in eine kritische Sektion benötigt, wurde von Lamport vorgestellt [9]. Dieser Algorithmus nutzt globale, annähernd synchronisierte Uhren. Maekawas Algorithmus [35] braucht $O(\sqrt{N})$ Nachrichten. Es handelt sich um einen nicht tokenbasierenden Algorithmus, der den guten Wert der Anzahl der Nachrichten durch Gleichverteilung der Knoten über die Quorums erreicht. Kumars Algorithmus [33] ist ebenfalls nicht tokenbasierend und arbeitet mit $O(N^{0.63})$ Nachrichten. Aufbauend auf Maekawas Algorithmus, der jeden Knoten genau einer Teilmenge zuordnet, entwickelten Agrawal und El Abbadi ihren Algorithmus [8]. Knoten werden redundant den Teilmengen zugeordnet, erreicht wird damit die Grundlage für eine erhöhte Stabilität. Der Algorithmus arbeitet mit einer Komplexität der Nachrichten von O(N). Eine weitere Klasse von nicht tokenbasierenden Algorithmen öffnet der Algorithmus von Ricart und Agrawal [13]. Die Erlaubnis zum Betreten der kritischen Sektion hängt nur von dem lokalen Status eines Knotens ab, welcher jedoch nach dem Verlassen mit den Stati der anderen Knoten abgeglichen werden muß. Der Algorithmus arbeitet mit O(N) Nachrichten. Suzuki und Kasamis Algorithmus [16] verwendet Broadcast-Nachrichten. Ein Knoten, der seine kritische Sektion betreten will, sendet an alle anderen Knoten des Systems eine Anfrage. Die Erlaubnis, die kritische Sektion zu betreten, wird daraufhin vom Tokenbesitzer erteilt. Um

Fairness zu sichern, enthält die Tokennachricht eine systemweit gültige Warteschlange.

Chang, Singhal und Liu präsentierten einen Algorithmus [27], der die Anzahl von Nachrichten auf O(ln N) reduziert. Der einfache Algorithmus von Raymond, vorgestellt in [14], benötigt bei entsprechender Konfiguration[3] $O(\log_2 N)$ Nachrichten. Der Algorithmus von Neilsen und Mizuno [11] ist eine Weiterentwicklung von Raymonds Algorithmus. Er benutzt im Gegensatz zu Raymonds eine verteilte Warteschlange und optimiert damit den Nachrichtenverkehr. Die Algorithmen von Chang et al. bzw. Naimi und Trehel sowie der Algorithmus von Neilsen und Mizuno dienen als Grundlage für die Algorithmen der vorliegenden Arbeit und werden in den folgenden Abschnitten detailliert beschrieben. Die statische Struktur in Form eines Rings nutzt Martin [35] für seinen Algorithmus, der O(N) Nachrichten benötigt. Der Algorithmus ist ebenfalls tokenbasierend. Die Richtung des Tokens und die Richtung der Anfragen ist durch den Ring statisch vorgegeben. Es ändern sich weder die Kanten noch die Richtungen der Kanten.

4.2 Verwandte Arbeiten

K. Raymond stellte 1988 einen Algorithmus für den Eintritt mehrerer Knoten in eine kritische Sektion vor [14]. Der Basisalgorithmus ist nicht tokenbasierend. Ein Knoten, der seine kritische Sektion betreten möchte, benachrichtigt alle Knoten des Systems. Nachdem der Knoten von allen anderen Knoten des Systems eine Antwort erhalten hat, betritt er seine kritische Sektion.

Ein zweiter Algorithmus für den Mehrfacheintritt in eine kritische Sektion wird in [15] vorgestellt und bewertet. Braucht Raymonds Algorithmus $2(N\text{-}1)$ Nachrichten, reduziert dieser Algorithmus die Anzahl der Nachrichten auf (N + K - 1), wobei ($K \leq N$) ist und K die Anzahl der sich gleichzeitig in der kritischen Sektion befindenen Knoten bezeichnet. Dieser Algorithmus basiert auf Suzuki und Kasamis Algorithmus [16]. Ein Knoten muß ebenfalls alle Knoten des Systems informieren und eine Antwort abwarten, bevor er das Privileg, seine kritische Sektion zu betreten, erhält.

Die obengenannten Algorithmen betrachten jedoch nicht die in der kritischen Sektion ausgeführten Operationen und sind somit für die Realisierung von Schreib- und Lesesperren nicht anwendbar.

Tritt ein Fehler in einem verteilten System auf, sind drei Schritte notwendig: Der Fehler muß erkannt werden, die Fehlerbehandlung muß eingeleitet werden, und der Fehler muß behoben werden. Naimi und Trehel beschäftigen sich in [3] mit dem zweiten und dritten Schritt: Wie kann man in dem Algorithmus, vorgestellt in [5], eine Fehlerbehandlung einleiten und das Token regenerieren? Ein Knoten,

[3]Entsprechende Konfiguration bedeutet, daß die Anzahl der Nachrichten vom Durchmesser der zugrundeliegenden Struktur abhängt. Die Angabe $O(\log_2 N)$ bezieht sich auf einen binären Baum.

der mit der kleinsten ID der möglichen Knoten, wird dann den Fehler beheben. Es handelt sich um ein Mehrphasenprotokoll zur Fehlerbehebung, welches einen quittierten Broadcast nutzt. Der Algorithmus benutzt außerdem die Funktionen *start_timer()* und *reset_timer()* sowie ein *time_out_event* eines Zeitgebers des Betriebssystems. Der Zeitgeber wird vorausgesetzt und nicht näher beschrieben. Es werden keine Ausführungen zu Evaluierung der Timeouts gemacht.

Fehlertoleranz ist im allgemeinen nur durch redundante Speicherung von Informationen zu realisieren. Diese Redundanzen sind notwendig, um den Systemzustand zu rekonstruieren bzw. um Alternativen für ausgefallene Knoten zu bestimmen. Der Algorithmus von Chang et al. [6] erweitert Raymonds Algorithmus [22] um Fehlertoleranz dadurch, daß mehrere Pfade für Anfragen durch die logische Struktur existieren. Durch das Halten der Information, welcher Pfad der kürzeste ist, arbeitet der fehlertolerante Algorithmus im fehlerfreien Fall ebenso effizient wie der Basisalgorithmus. Um die redundanten Pfade zyklenfrei zu halten und somit weiterhin Verklemmungen auszuschließen, wird eine weitere Nachricht benötigt. Chang et al. machen keine Ausführungen, wie ein Fehler erkannt werden kann. Der Algorithmus behandelt Knoten- und Netzwerkfehler.

Agrawal und El Abbadi schlagen in [8] einen weiteren fehlertoleranten Algorithmus vor. Der Algorithmus behandelt ebenfalls Knoten- und Netzwerkfehler. Zur Fehlererkennung wird ein Timeout benutzt. Der Verlust des Tokens wird durch logische Uhren auf Sender- und Empfängerseite erkannt. Jede Anfrage erhält zusätzlich einen Zeitstempel [9]. Um in diesem Algorithmus das Verhungern zu vermeiden, welches durch nicht abgeschlossene Tokentransfers auftritt, ist es notwendig, schwachkonsistente Kopien von Warteschlangen der Anfragen zu halten. Der Algorithmus arbeitet mit einer Komplexität der Nachrichten von O(N) und einem Speichermehraufwand von $O(N)$ je Knoten.

Die von S. Misha und K. Srimani vorgestellten Algorithmen [17] tolerieren ausschließlich Knotenfehler. Der Algorithmus A ist tokenbasierend und eine Erweiterung des Algorithmus aus [16]. Algorithmus B dieser Veröffentlichung nutzt einen hybriden Ansatz, d. h. er ist tokenbasierend und teilweise zentralisiert. Ein Knoten des Systems, der sogenannte *Kontrollknoten*, übernimmt zu einem Zeitpunkt zentrale Aufgaben bezüglich der Fehlertoleranz. Der aktuelle Kontrollknoten wechselt nach bestimmten Zeitintervallen, unabhängig davon, ob ein Fehler aufgetreten ist oder nicht. Misha und Srimani machen keine Aussagen über die Komplexität ihrer Algorithmen.

4.3 Weitere Arbeiten

Einen guten Überblick über die verschiedenen Klassen und Methoden von Algorithmen zum verteilten gegenseitigen Ausschluß gibt die Veröffentlichung von Chang et al. [25]. In dieser Arbeit befindet sich auch ein theoretischer Vergleich der verschiedenen Verfahren. K. Scholtyssik versucht in seiner Diplomarbeit [44] die verschiedenen Methoden in ein allgemeines Schema zu bringen, sowie durch einen theoretischen und empirischen Vergleich Rückschlüsse auf die Anwendbarkeit auf bestimmte parallele Probleme zu ziehen. Weitere vergleichende Studien findet der Leser in [12], [30] und [32]. Alle diese Arbeiten, einschließlich der von K. Scholtyssik, kommen auf das Ergebnis, daß der Algorithmus von Chang et al. bzw. Naimi und Trehel und der Algorithmus von Neilsen und Mizuno die besten Ergebnisse bezüglich der Performance aufweisen.

Ein weiterer Schwerpunkt der Forschung auf diesem Gebiet betrifft Echtzeitanforderungen. Neben fehlertoleranten Algorithmen sind in diesem Zusammenhang Arbeiten über priorisierte Algorithmen von Interesse (siehe [23], [36] und [42]).

Das Kapitel 7 stützt sich auf die Dissertation von V. Hübsch [45]. Diese Arbeit gibt eine Einführung in das Thema Fehler und Fehlertoleranz bezüglich verteilter Systeme. V. Hübsch beschäftigt sich, nach der Einführung grundlegender Begriffe, mit einem fehlertoleranten Distributed Shared Memory System (DSM). Dieses DSM und speziell die Fehlertoleranz des Systems basiert auf dem Transaktionskonzept.

5 Die Testumgebung

Die Performance eines Algorithmus zum gegenseitigen Ausschluß wird durch die Anzahl der Nachrichten je kritische Sektion und die mittlere Anfragezeit bestimmt. Um die Algorithmen im Hinblick auf die obengenannten Parameter zu untersuchen, wird eine Testumgebung benutzt. Die in dem folgenden Abschnitt beschriebene Testumgebung ist die Grundlage für eine komparative Untersuchung verschiedener Algorithmen.

Die Testumgebung ist in C unter UNIX bzw. Linux implementiert und simuliert ein verteiltes System lokal. Die Knoten des verteilten Systems sind auf Linux- bzw. UNIX-Prozesse abgebildet. Die Knoten kommunizieren untereinander über TCP-Sockets.

5.1 Struktureller Aufbau

Ein Knoten der Testumgebung besteht aus mehreren Komponenten: einem Worker, einem Kommunikationsserver, einem Mutexserver und einem Barrierenserver. Der Worker arbeitet die Schleife mit den kritischen Sektionen ab. Sämtliche Nachrichten werden über den Kommunikationsserver empfangen. Der Mutexserver verarbeitet Nachrichten, die den Algorithmus zum gegenseitigen Ausschluß betreffen, d.h. in ihm sind die Nachrichtenhandler der verschiedenen Algorithmen implementiert. Die Knoten brauchen, um ordnungsgemäß zu terminieren, eine Barriere. Diese Barriere ist als binäre Baumbarriere implementiert. Die zur Barriere gehörenden Nachrichten werden vom Barrierenserver verarbeitet. Die einzelnen Komponenten der Knoten sind als Posix-Threads [18] implementiert und nutzen die POSIX-Thread-Bibliotheksfunktionen zur Synchronisation und Kommunikation. Verwendet wurde die Implementation der Florida State University [19].

Den strukturellen Aufbau eines Knotens der Testumgebung zeigt Abbildung 2. Die Rechtecke illustrieren die einzelnen Komponenten eines Knotens. Die Pfeile beschreiben die Kommunikationswege zwischen den Knoten und zwischen den Komponenten eines Knotens.

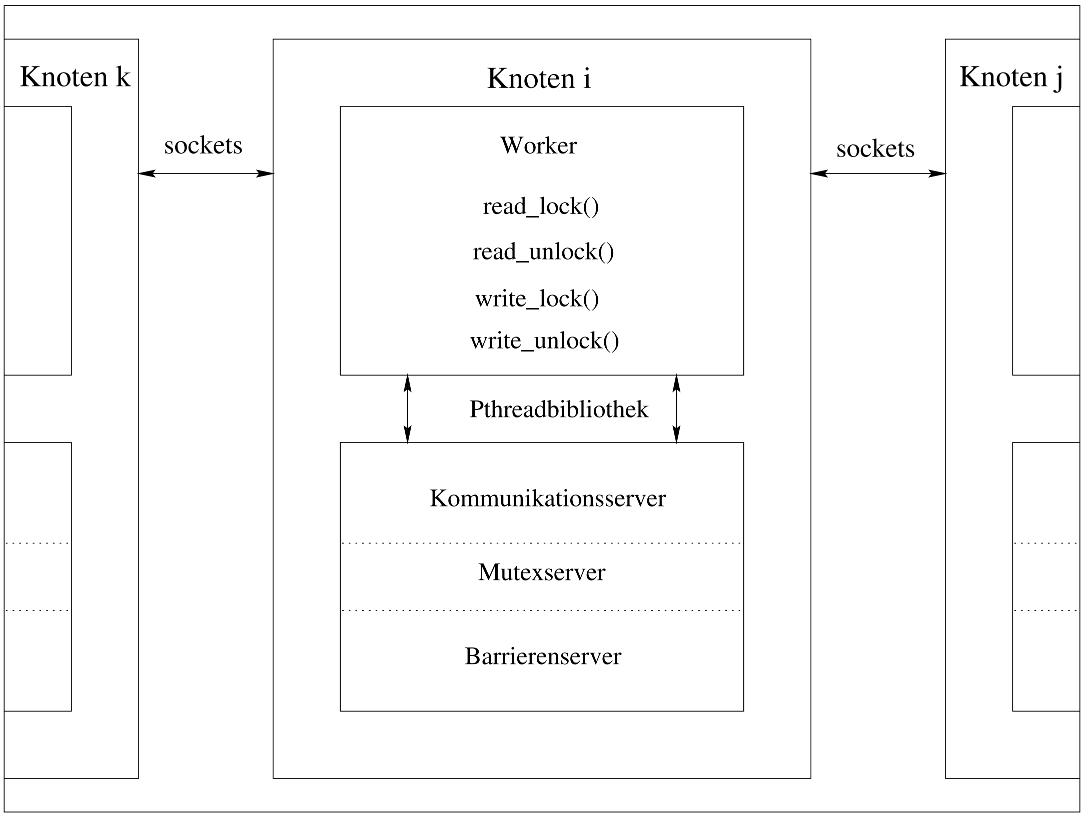


Abbildung 2: Struktureller Aufbau eines Knotens der Testumgebung.

Die Parameter eines Testlaufs, wie Größe des Systems, Anzahl der kritischen Sektion und Prozent der Nurlesezugriffe, werden über die Kommandozeile eingegeben. Hier der Aufruf für den Algorithmus von Chang et al. bzw. Naimi und Trehel für ein System von 32 Knoten, 64 kritische Sektionen, Phi = 200 μsec, Gamma = 2000 μsec und 50 % Leseanfragen:

% test_CLS 32 64 200 2000 50

Der Startknoten startet N Knoten und terminiert bzw. wartet auf das Terminieren der gestarteten Knoten und verarbeitet die Meßergebnisse. Die Knoten laufen ab diesem Zeitpunkt unabhängig.

Die Parameter Phi und Gamma werden randomisiert um $+/-10\%$ verändert, um einen realistisch gestreuten Ablauf zu erzeugen. Um das System einzupendeln und Anomalien zu vermeiden, werden vor den für die Messung relevanten kritischen Sektionen eine Anzahl weiterer kritischer Sektionen eingefügt. Diese Anzahl hängt von der Anzahl der Knoten N ab.

Die benötigte *await(Token)* Anweisung ist mit Hilfe von *pthread_cond_wait()* und *pthread_cond_signal()* implementiert. Das Verzögern innerhalb und außerhalb der kritischen Sektion ist mit *nanosleep()* implementiert. Sämtliche Zeitmessungen wurden mit *clock_gettime()* durchgeführt. *nanosleep()* und *clock_gettime()* sind Funktionen des Posix-Standards [21].

Die Testumgebung ist als C-Bibliothek verfügbar, die die notwendigen Funktionen, wie Starten von Knoten, Barrieren-Synchronisation usw. bereitstellt. Eine Schnittstellenbeschreibung der Bibliothek befindet sich im folgenden Abschnitt.

5.2 Interface der Testumgebung DSiM

Folgende Funktionen werden von der Testumgebung zur Verfügung gestellt:

Initialisierung

*int DSiM_init(DSiM_env_t**);*

Die Testumgebung wird initialisiert. Die Funktion erhält als Parameter eine Struktur, die u.a. die Portadresse des Initialknotens enthält, welcher als Nameserver fungiert.

Starten von Knoten

*int DSiM_create(DSiM_env_t,int,int *(func),void*);*

Die Funktion startet einen Knoten des Systems in einem UNIX- bzw. Linuxprozeß. Die Parameter sind eine ID, die auszuführende Funktion und eine Parameterliste dieser Funktion.

Barrierensynchronisation

int DSiM_barrier_init(barrier_t);*

Initialisierung der Barriere und Start des Barrierenservers.

int DSiM_barrier_wait(barrier_t);*

Die Funktion blockiert den Knoten bis alle weiteren Knoten, die an der Barriere teilnehmen, den Synchronisationspunkt erreicht haben.

int DSiM_barrier_destroy(barrier_t);*

Gibt den durch die Barriere allozierten Speicher frei.

Mutexsynchronisation

int DSiM_mutex_init(mutex_t);*

Initialisiert die benötigte Struktur und startet den Mutexserver.

int DSiM_mutex_lock(mutex_t);*

int DSiM_mutex_unlock(mutex_t);*

int DSiM_mutex_readlock(mutex_t);*

int DSiM_mutex_readunlock(mutex_t);*

int DSiM_mutex_writelock(mutex_t);*

int DSiM_mutex_writeunlock(mutex_t);*

Synchronisation durch gegenseitigen Ausschluß in einer kritischen Sektion (exklusiv, lesend und schreibend).

int DSiM_mutex_destroy(mutex_t);*

Gibt den durch die Mutexsynchronisation allozierten Speicher frei.

Terminierung

int DSiM_exit(int);

Beendet einen Knoten der Testumgebung. Im speziellen werden die Threads beendet und der allozierte Speicher freigegeben.

6 Algorithmen für Lese- und Schreibsperren

In einem verteilten System existieren auf Grund der Architektur keine atomaren Test-und-Set Operationen. Um atomare Operationen zu realisieren, wird der verteilte gegenseitige Ausschluß eingesetzt.

Um Daten des verteilten Systems in einem konsistenten Zustand zu halten, müssen Schreibzugriffe im gegenseitigen Ausschluß erfolgen. Lesezugriffe verlangen lediglich den Ausschluß gleichzeitiger Schreibzugriffe, jedoch nicht den Ausschluß anderer Lesezugriffe. Somit bietet sich an, den gegenseitigen Ausschluß durch ein Protokoll zu realisieren, welches zu einem Zeitpunkt entweder mehreren Lesern oder einem Schreiber den Zugriff gestattet. Erreicht wird dadurch ein höherer Grad an Parallelität.

Die in den folgenden Abschnitten vorgestellten Algorithmen basieren auf dem Algorithmus von Chang et al. bzw. Naimi und Trehel [6, 7] [4] und dem Algorithmus von Neilsen und Mizuno [11]. Die Basisalgorithmen machen keine Unterscheidung zwischen Lese- und Schreibsperren.

Die Komplexität der Nachrichten je kritischer Sektion nmE beträgt bei Chang et al.'s $\mathrm{O}(\ln N)$ und bei Neilsen und Mizunos $\mathrm{O}(\log_2 N)$ im mittleren Fall . Eine ausführliche Beschreibung der Basisalgorithmen sowie eine Bewertung der Performance befindet sich in [12]. Die Erweiterung des Algorithmus von Chang et al. bzw. Naimi und Trehel wird im folgenden mit *DVARW* für auf dynamischer Struktur basierender Algorithmus mit verteilter Warteschlange, die Erweiterung des Algorithmus von Neilsen und Mizuno mit *SVARW* für auf statische Struktur basierender Algorithmus mit verteilter Warteschlange abgekürzt. Die Erweiterung *RW* zeigt an, daß es sich um Algorithmen handelt, die Lese- und Schreibsperren realisieren.

Die Algorithmen sind dezentralisiert, d.h. kein Knoten hat spezielle Operationen bezüglich des gegenseitigen Ausschlusses zu übernehmen, und tokenbasierend. Tokenbasierende Algorithmen zeichnen sich dadurch aus, daß das Privileg, die kritische Sektion zu betreten, durch das sogenannte Token repräsentiert wird. Der Knoten der das Token besitzt, und nur dieser, hat das Recht, seine kritische Sektion zu betreten. Bei den Algorithmen für die Realisierung von Schreib- und Lesesperren zeigt der Erhalt des Tokens an, daß der Knoten zum Lesen seine kritische Sektion betreten darf und während des Lesens keine Schreibzugriffe erfolgen.

Im besten Fall wird keine Nachricht benötigt, um die kritische Sektion zu betreten: Der Knoten, der das Token hält, will seine kritische Sektion betreten, testet das Token und kann, ohne mit einem anderen Knoten des Systems kommuniziert zu haben, in seine kritische Sektion eintreten.

Erweitert wurde die Struktur der Basisalgorithmen um den Boolean *Read_Re-*

[4] Der Algorithmus wurde unabhängig sowohl von Chang et al., als auch von Naimi und Trehel entwickelt.

quested, den Boolean *Last_Reader*, den Zeiger *Next_Reader* und um eine Warteschlange *Next_Last_Reader*. In *Next_Last_Reader* werden Knoten gespeichert. Die Warteschlange arbeitet nach dem FIFO-Prinzip. Tabelle 1 zeigt die vollständige Struktur, die die erweiterten Algorithmen benötigen, um korrekt zu arbeiten. Die fettgedruckten Teile der Tabelle kennzeichnen die Erweiterung der Struktur für die Realisierung von Lese- und Schreibsperren. Die Boolean *Write* und *Read* ersetzen die Variable *cs* der Basisalgorithmen, welcher angibt, ob sich ein Knoten in seiner kritischen Sektion befindet.

Name	Type	Beschreibung
Token	Boolean	Repräsentiert das Token auf einem Knoten.
Dir	Knoten	Bestimmt die Richtung einer Tokenanfrage.
Next	Knoten	Speichert den Knoten, der als nächstes das Token erhält.
Self	Knoten	Identifiziert den eigenen Knoten.
Read	Boolean	Zeigt an, ob der Knoten lesend in seiner kritischen Sektion ist.
Write	Boolean	Zeigt an, ob der Knoten schreibend in seiner kritischen Sektion ist.
Next_Reader	Knoten	Zeigt an, welcher Knoten der nächste Leser ist, falls dieser existiert.
Next_Last_Reader	FIFO für Knoten	Zeigt die Richtung, in der die Nachricht LAST_READER gesendet wird an.
Read_Requested	Boolean	Zeigt an, ob ein Knoten eine aktuelle Leseanfrage hat.
Last_Reader	Boolean	Zeigt an, ob ein Knoten der letzte Leser ist.
TOKEN	Nachricht	Das Token wird versendet. Die Nachricht führt ein Flag mit sich, welches Anzeigt, ob die Nachricht von einem lesenden Knoten gesendet wurde.
READ_REQUEST	Nachricht	Eine Leseanfrage, mitgeführt wird in ihr der Absender der Anfrage.
WRITE_REQUEST	Nachricht	Eine Schreibanfrage, mitgeführt wird in ihr der Absender der Anfrage.
LAST_READER	Nachricht	Der Empfänger ist der letzte Knoten in der Kette lesender Knoten.

Tabelle 1: Struktur für die erweiterten Algorithmen SVARW und DVARW.

6.1 DVARW

Der in [6, 7] vorgestellte und in [12] auf Performance und Korrektheit untersuchte Algorithmus von Chang et al. bzw. Naimi und Trehel ist tokenbasierend und arbeitet auf einer dynamischen Struktur.

Der Algorithmus ist in den Abbildungen 3 bis 11 im einheitlichen Pseudocode skizziert. Die Teile der Erweiterung des Algorithmus sind durch Fettdruck gekennzeichnet. Im Abschnitt 6.2 befinden sich Beispiele, die den Algorithmus illustrieren.

Initial sind die Knoten in Sternform angeordnet und ein ausgezeichneter Knoten ist der Tokenbesitzer (Abbildung 3 Zeile 4 bis 10). Das heißt, beim initialen Tokenbesitzer ist der Zeiger *Dir* auf NULL und der Boolean *Token* auf *TRUE* gesetzt, bei allen anderen Knoten ist *Dir* auf den initialen Tokenbesitzer und das *Token* auf *FALSE* gesetzt. *Next*, *Next_Reader*, *Read*, *Write* und *Next_Last_Reader* sind bei allen Knoten auf *NULL* bzw. *FALSE* gesetzt (Abbildung 3 Zeile 11 bis 15).

```
1   mutex_init(id)
2   begin
3       Self = node(id);
4       if (id == 0))
5           Token = TRUE;
6           Dir = NULL;
7       else
8           Token = FALSE;
9           Dir = node(0);
10      fi
11      Next = NULL;
12      Next_Reader = NULL;
13      Last_Reader = FALSE;
14      Read = FALSE;
15      Write = FALSE;
16  end mutex_init.
```

Abbildung 3: Initialisierung der Struktur beim DVARW.

Will ein Knoten zum Lesen seine kritische Sektion betreten, testet er zunächst, ob er der Tokenbesitzer ist (Zeile 19 in Abbildung 4). Ist dies der Fall, setzt er *Read* und *Last_Reader* auf *TRUE*. *Last_Reader* gleich *TRUE* zeigt an, daß sich keine weiteren Leser im System befinden. *mutex_read_lock()* kehrt zurück und der Knoten betritt seine kritische Sektion (Abbildung 4 Zeile 34).

Ist er nicht der Tokenbesitzer, sendet er eine Tokenanfrage (*READ_REQUEST*) in die Richtung, die ihm *Dir* weist, setzt *Dir* danach auf *NULL* und *Read_Requested* auf *TRUE*. Der Knoten blockiert dann auf *await(Token)*, bis er das Token erhält (Abbildung 4 Zeile 22 bis 25).

Hat der Knoten das Token erhalten, testet er mit *Next_Reader != NULL* zunächst, ob weitere Knoten eine Leseanfrage generiert haben (Abbildung 4 Zeile 27). Ist dies der Fall, sendet er das Token an *Next_Reader*, setzt *Token* auf *FALSE*, fügt *Next_Reader* in die Warteschlange *Next_Last_Reader* ein und setzt daraufhin *Next_Reader* auf *NULL* (Abbildung 4, Zeile 27 bis Zeile 31). Nachdem *Read* auf *TRUE* gesetzt wurde, kehrt *mutex_read_lock()* zurück und der Knoten betritt seine kritische Sektion.

```
17  mutex_read_lock
18  begin
19     if (Token == TRUE)
20        Last_Reader = TRUE;
21     else
22        send_read_request(Self) to Dir;
23        Read_Requested = TRUE;
24        Dir = NULL;
25        await(Token);
26        Read_Requested = FALSE;
27        if (Next_Reader != NULL)
28           send_token(0) to Next_Reader;
29           Token = FALSE;
30           push(Next_Reader,Next_Last_Reader);
31           Next_Reader = NULL;
32        fi
33     fi
34     Read = TRUE;
35  end mutex_read_lock.
```

Abbildung 4: Die Funktion *mutex_read_lock()* beim DVARW.

Ein lesender Knoten verläßt mit dem Ruf *mutex_read_unlock()* seine kritische Sektion.

Zunächst setzt der Knoten *Read* auf *FALSE*. Anschließend testet er, ob sich ein Knoten im System befindet der auf die Nachricht *LAST_READER* wartet und ob er der letzte Leser in der Kette lesender Knoten ist. Ist dies der Fall, entnimmt der Knoten den ersten Eintrag aus der Warteschlange *Next_Last_Reader*, sendet *LAST_READER* an den entnommenen Knoten und setzt *Last_Reader* auf *TRUE* (Abbildung 5, Zeile 39 bis Zeile 45).

Ist der Knoten der letzte Leser, ist *Next* ungleich *NULL* und ist er der Tokenbesitzer, sendet er das Token zu *Next*, setzt *Next* auf *NULL* und *Token* auf *FALSE* (Abbildung 5, Zeile 46 bis Zeile 54). Danach kehrt *mutex_read_unlock()* zurück.

```
36 mutex_read_unlock
37 begin
38    Read = FALSE;
39    if (Last_Reader == TRUE) and
40       (Next_Last_Reader != NULL)
41
42       Node = fetch(Next_Last_Reader);
43       Last_Reader = FALSE;
44       send_last_reader() to Node;
45    fi
46    if (Next != NULL) and
47       (Last_Reader == TRUE) and
48       (Token == TRUE)
49
50       send_token(0) to Next;
51       Token = FALSE;
52       Next = NULL;
53       Last_Reader = TRUE;
54    fi
55 end mutex_read_unlock.
```

Abbildung 5: Die Funktion *mutex_read_unlock()* beim DVARW.

Will ein Knoten seine kritische Sektion zum Schreiben betreten, führt er die in Abbildung 6 dargestellte Funktion *mutex_write_lock()* aus. Ist das Token lokal verfügbar, setzt der Knoten *Write* auf *TRUE*. Die Funktion kehrt zurück und der Knoten betritt seine kritische Sektion (Abbildung 6, Zeile 63). Ist das Token nicht lokal verfügbar, sendet der Knoten eine Tokenanfrage (*WRITE_REQUEST*) in Richtung *Dir*. Anschließend weist der Knoten *Dir* *NULL* zu und blockiert auf *await(Token)*. Hat der Knoten das Token erhalten, setzt er *Write* auf *TRUE* (Abbildung 6, Zeile 58 bis Zeile 63). Danach kehrt *mutex_write_lock()* zurück und der Knoten betritt seine kritische Sektion.

Mit dem Ruf *mutex_write_unlock()* verläßt ein Schreiber seine kritische Sektion (Abbildung 7). Nach dem der Knoten *Write* auf *FALSE* gesetzt hat, testet er, ob ein weiterer Knoten auf das Token wartet (*Next != NULL*). Ist dies der Fall,

```
56 mutex_write_lock
57 begin
58     if (Token != TRUE)
59         send_write_request(Self) to Dir;
60         Dir = NULL;
61         await(Token);
62     fi
63     Write = TRUE;
64 end mutex_write_lock.
```

Abbildung 6: Die Funktion *mutex_write_lock()* beim DVARW.

sendet der Knoten das Token an *Next* und weist *Token FALSE* und *Next NULL* zu (Abbildung 7, Zeile 67 bis Zeile 72).

Anderenfalls behält der Knoten das Token, ohne es zu nutzen. Die Funktion *mutex_write_unlock()* kehrt zurück.

```
65 mutex_write_unlock
66 begin
67     Write = FALSE;
68     if (Next != NULL)
69         send_token(0) to Next;
70         Token = FALSE;
71         Next = NULL;
72     fi
73 end mutex_write_unlock.
```

Abbildung 7: Die Funktion *mutex_write_unlock()* beim DVARW.

Erhält ein Knoten die Nachricht *WRITE_REQUEST*, sendet er diesen, insofern sein Zeiger *Dir* nicht *NULL* ist, weiter in Richtung *Dir* (siehe Abbildung 8, Zeile 77).

Ist *Dir* gleich *NULL*, testet er das Tokenflag, um festzustellen, ob er der momentane Tokenbesitzer ist. Ist er der Tokenbesitzer und befindet er sich nicht in seiner kritischen Sektion, sendet er das Token sofort an den Absender und setzt danach *Token* auf *FALSE* (Abbildung 8 Zeile 79 bis Zeile 82). Befindet er sich in seiner kritischen Sektion, weist er *Next* den Absender der Tokenanfrage zu (Abbildung 8 Zeile 84). Ist er nicht der Tokenbesitzer, befindet er sich am Ende der verteilten Warteschlange der Zeiger *Next* und wartet somit auf das Token. Der Knoten, der die Anfrage generiert hat, wird mit der Zuweisung *Next = requester* das neue Ende der verteilten Warteschlange. (Abbildung 8, Zeile 87).

In allen Fällen setzt der Knoten seinen Zeiger *Dir* auf den Knoten, der die Anfrage generiert hat, da dieser den kürzesten Weg für weitere Tokenanfragen vorgibt (Abbildung 8, Zeile 90).

```
74  Beim Empfang von write_request(requester)
75  begin
76     if (Dir != NULL)
77        send_write_request(requester,Self) to Dir;
78     else
79        if (Token == TRUE)
80           if (Read == FALSE) and (Write == FALSE)
81              send_token(0) to requester;
82              Token = FALSE;
83           else
84              Next = requester;
85           fi
86        else
87           Next = requester;
88        fi
89     fi
90     Dir = requester;
91  end.
```

Abbildung 8: Reaktion des DVARW auf die Nachricht *WRITE_REQUEST*.

Erhält ein Knoten die Nachricht *READ_REQUEST*, testet er zunächst den Zeiger *Dir*. Ist dieser nicht *NULL*, sendet er die Anfrage in Richtung *Dir* weiter (Abbildung 9, Zeile 94 und 95).

Ist *Dir* gleich *NULL*, testet der Empfänger, ob er der momentane Tokenbesitzer ist. Ist er der Tokenbesitzer und ist weder der Boolean *Write* noch der Boolean *Read* gleich *TRUE*, d. h. der Tokenbesitzer befindet sich weder lesend noch schreibend in seiner kritischen Sektion, setzt er den Boolean *Token* auf *FALSE* und sendet das Token direkt zu dem Knoten, der die Anfrage generiert hat (Abbildung 9 Zeile 97 bis 100).

Ist *Write* gleich TRUE, d.h. der Empfänger befindet sich schreibend in seiner kritischen Sektion, setzt er *Next* und *Dir* auf den Absender der Tokenanfrage (Abbildung 9 Zeile 107 und Zeile 119).

Befindet sich der Empfänger lesend in seiner kritischen Sektion, sendet er das Token an den Absender und setzt den Boolean *Token* auf *FALSE*. Zusätzlich fügt er den Absender in die Warteschlange *Next_Last_Reader* ein, weil nun ein Knoten im System existiert, der auf die Nachricht *LAST_READER* wartet (Abbildung 9

Zeile 103 bis 105). Die Tokennachricht führt in diesem Fall ein Flag mit sich, welches anzeigt, daß das Token von einem Leser gesendet wurde. Ist der Empfänger nicht der Tokenbesitzer und hat er eine aktuelle Leseanfrage (*Read_Requested == TRUE*) setzt er *Dir* und *Next_Reader* auf den Absender der Tokenanfrage. Zusätzlich schreibt der Knoten den Absender der Anfrage in die Warteschlange *Next_Last_Reader* (Abbildung 9 Zeile 111 bis Zeile 113).

In allen Fällen setzt der Knoten *Dir* auf den Knoten, der die Anfrage generiert hat (Abbildung 9 Zeile 119).

```
 92  Beim Empfang von read_request(requester)
 93  begin
 94     if (Dir != NULL)
 95        send_request(requester) to Dir;
 96     else
 97        if (Token == TRUE)
 98           if (Read == FALSE) and (Write == FALSE)
 99              send_token(0) to requester;
100              Token = FALSE;
101           else
102              if (Read == TRUE)
103                 send_token(1) to requester;
104                 push(requester,Next_Last_Reader);
105                 Token = FALSE;
106              else
107                 Next = requester;
108              fi
109           fi
110        else
111           if (Read_Requested == TRUE)
112              Next_Reader = requester;
113              push(requester,Next_Last_Reader);
114           else
115              Next = requester;
116           fi
117        fi
118     fi
119     Dir = requester;
120  end.
```

Abbildung 9: Reaktion des DVARW auf die Nachricht *READ_REQUEST*.

Erhält ein Knoten die Nachricht *TOKEN*, setzt er *Token* auf *TRUE*. Ist das

Flag, welches die Tokennachricht mit sich führt, gleich null, setzt der Knoten *Last_Reader* auf *TRUE*, anderenfalls setzt er *Last_Reader* auf *FALSE* (Abbildung 10).

```
Beim Empfang von token(flag)
begin
   Token = TRUE;
   if (flag == 0)
      Last_Reader = TRUE;
   else
      Last_Reader = FALSE;
   fi
end.
```

Abbildung 10: Reaktion des DVARW auf die Nachricht *TOKEN*.

Empfängt ein Knoten die Nachricht *LAST_READER*, setzt er *Last_Reader* auf *TRUE* (Abbildung 11 Zeile 132). Dann testet er, ob sich in der Warteschlange *Next_Last_Reader* ein Eintrag befindet. Ist dies der Fall, sendet er die Nachricht *LAST_READER* an den ersten Knoten in *Next_Last_Reader*, löscht diesen aus der Warteschlange und setzt *Last_Reader* wieder auf *FALSE* (Abbildung 11 Zeile 134 bis Zeile 136). Ist die Warteschlange *Next_Last_Reader* leer, testet er den Zeiger *Next*. Ist dieser nicht *NULL* und ist der Knoten der Tokenbesitzer, sendet er das Token zu *Next* und setzt dann *Token* auf *FALSE* (Abbildung 11 Zeile 143 bis Zeile 146).

```
Beim Empfang von last_reader()
begin
   Last_Reader = TRUE;
   if (Next_Reader != NULL)
      Node = fetch(Next_Reader);
      send_last_reader to Node;
      Last_Reader = FALSE;
   fi
   if (Next != NULL) and
      (Last_reader == TRUE) and
      (Token == TRUE) and
      (Read == FALSE)
      send_token(0) to Next;
      Token = FALSE;
      Next = NULL;
      Last_Reader = FALSE;
   fi
end.
```

Abbildung 11: Reaktion des DVARW auf die Nachricht *LAST_READER*.

6.2 Beispiel

Im folgenden wird ein System von vier Knoten, A, B, C und D betrachtet. In den grafischen Darstellungen beschreiben die Kreise die Knoten, das **T** bezeichnet das Token, durchgängig gezeichnete Pfeile bezeichnen die Zeiger *Dir*, gestrichelte Pfeile das System der Zeiger *Next*, und Pfeile, die mit Strich-Punkt-Linien dargestellt sind, bezeichnen die Zeiger *Next_Last_Reader*. Eine Schreibanfrage wird durch *Rw(Knoten)*, eine Leseanfrage durch *Rr(Knoten)*dargestellt. Das Senden des Tokens bezeichnet *T(Flag)*. Der Knoten A ist der initiale Tokenbesitzer. Sein Zeiger *Dir* ist null. Die Zeiger *Dir* der anderen Knoten sind auf A gesetzt.

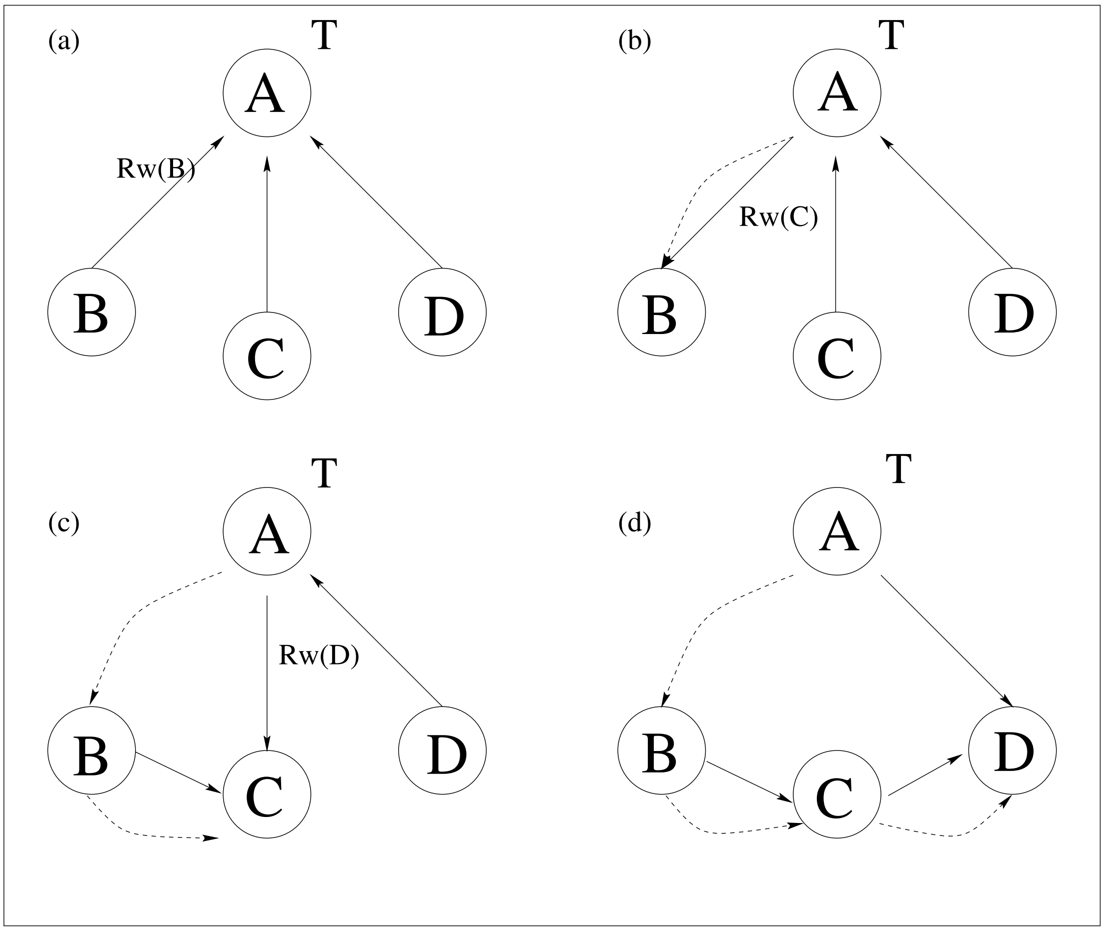


Abbildung 12: Beispielausführung beim DVARW.

Die Ausgangssituation zeigt Abbildung 12 (a). Knoten A betritt seine kritische Sektion. Nacheinander senden nun B, C und D eine Schreibanfrage in Richtung ihres Zeigers *Dir*. Die Anfrage von B erreicht A und bewirkt, daß A seinen Zeiger *Dir* und seinen Zeiger *Next* auf B setzt (Abbildung 12(b)). Die Anfrage von C wird von Knoten A in Richtung *Dir* weitergesendet und gelangt somit zu B. B setzt mit dem Empfangen der Nachricht *Next* und *Dir* auf Knoten C (Abbildung

12(c)). Die Schreibanfrage von Knoten D gelangt über A, der seinen *Dir* Zeiger auf D setzt, zu Knoten C. Knoten C setzt wiederrum *Dir* und *Next* auf Knoten D (Abbildung 12 (d)).

Nach und nach gelangt das Token entlang der *Next* Kette, nachdem A, B und C ihre kritische Sektion abgearbeitet haben, zu Knoten D (Abbildung 13). Verläßt Knoten D seine kritische Sektion, behält er das Token, bis ihn eine Anfrage erreicht, ohne es zu nutzen.

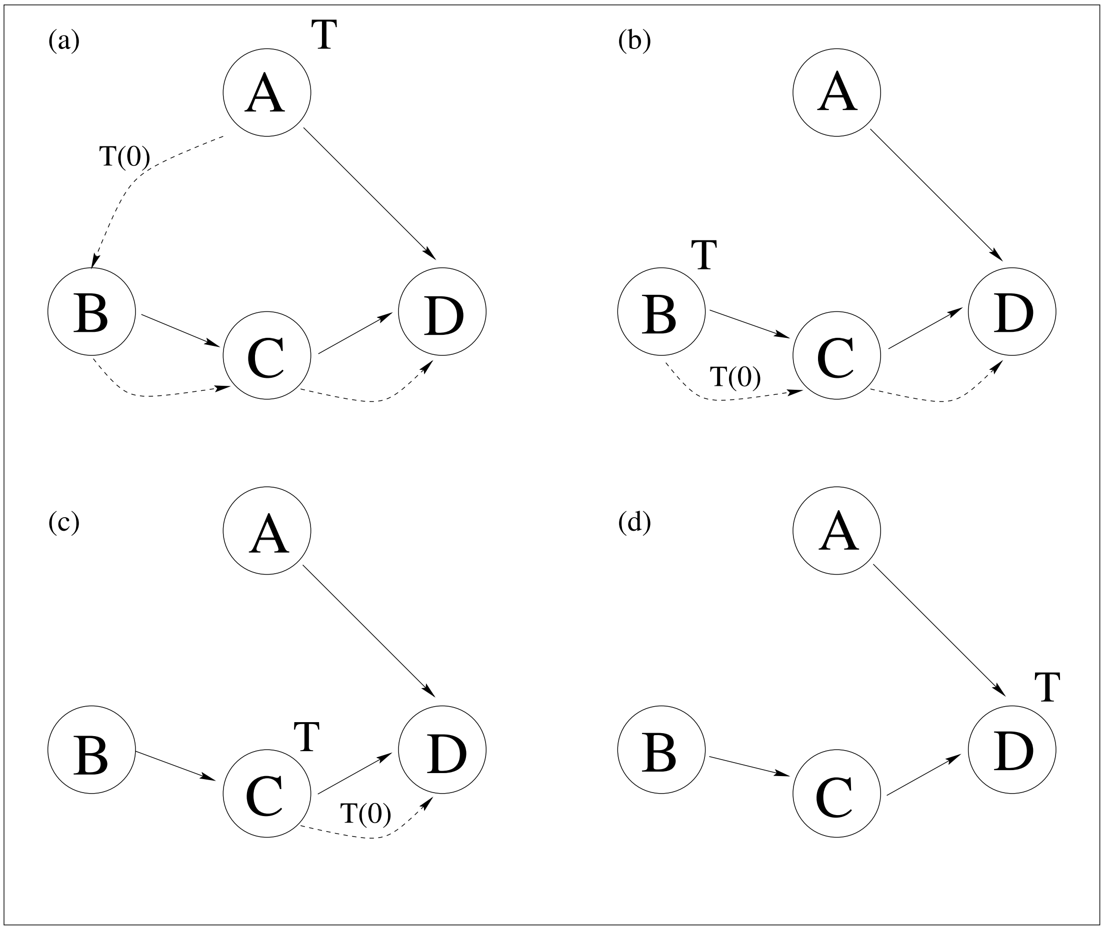


Abbildung 13: Beispielausführung beim DVARW.

Ausgehend von der gleichen Anfangssituation folgt nun ein Beispiel, in dem sowohl Leseanfragen als auch Schreibanfragen von Knoten generiert werden.

Der Tokenbesitzer, Knoten A, betritt sofort lesend seine kritische Sektion (Abbildung 14 (a)). Nacheinander stellen nun C und B eine Leseanfrage(Abbildung 14 (b, c)). Knoten D führt ein *mutex_write_lock()* aus und stellt, da er nicht im Besitz des Tokens ist, eine Schreibanfrage. Nacheinander erhalten Knoten B und Knoten C das Token, weil sich nur Leser im System befinden. Die Schreibanfrage von Knoten D gelangt zu Knoten C, der der Tokenbesitzer und das Ende der

Kette der Leser bildet (Abbildung 14 (c)). Knoten C setzt nach dem Empfang der Schreibanfrage seinen Zeiger *Next* auf Knoten D.

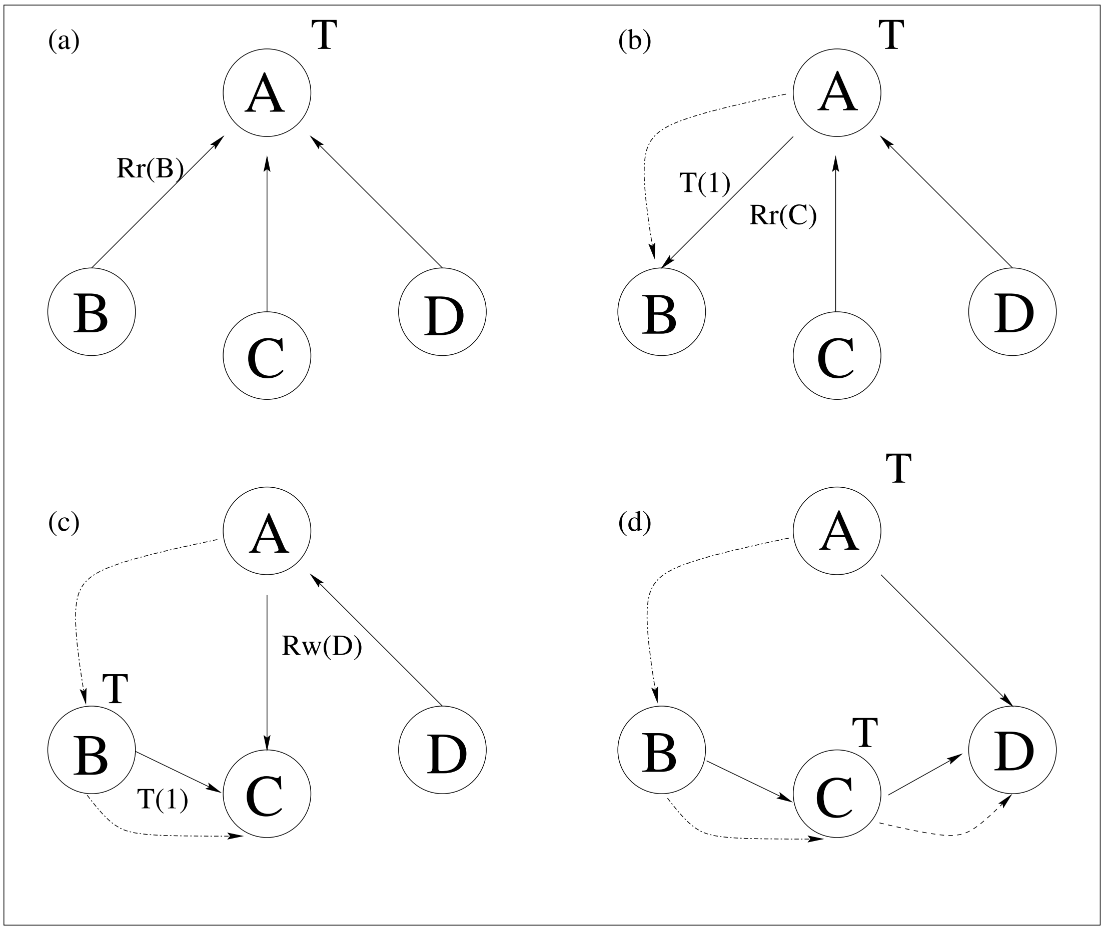


Abbildung 14: Beispielausführung beim DVARW.

Knoten C behält das Token solange bis er die Nachricht *LAST_READER* erhalten hat (Abbildung 15 (c)). Dabei spielt es keine Rolle, wer als erster Knoten, Knoten A oder Knoten B, seine kritische Sektion verläßt. Beim Empfang des Tokens mit dem *Flag == 1* wird *Last_Reader* auf *FALSE* gesetzt. Erst nach dem Empfang der Nachricht *LAST_READER* wird das Flag *Last_Reader* auf *TRUE* gesetzt. Erhält Knoten C die Nachricht *LAST_READER*, setzt er *Last_Reader* auf *TRUE*, testet seinen Zeiger *Next* und, da dieser nicht *NULL* ist, sendet Knoten C das Token zu *Next*, d.h. zu Knoten D (Abbildung 15 (d)). Knoten D erhält somit, erst nachdem alle lesenden Knoten ihre kritische Sektion verlassen haben, das Token und die Möglichkeit, seine kritische Sektion schreibend zu betreten.

Die Art der Anfragen hat keinen Einfluß auf die sich verzweigende Struktur der Anfragewege gegeben durch die Gesamtheit der Zeiger *Dir* des Systems. Der Erhalt des Tokens sichert einem Schreiber den exklusiven Zugriff und einem Leser

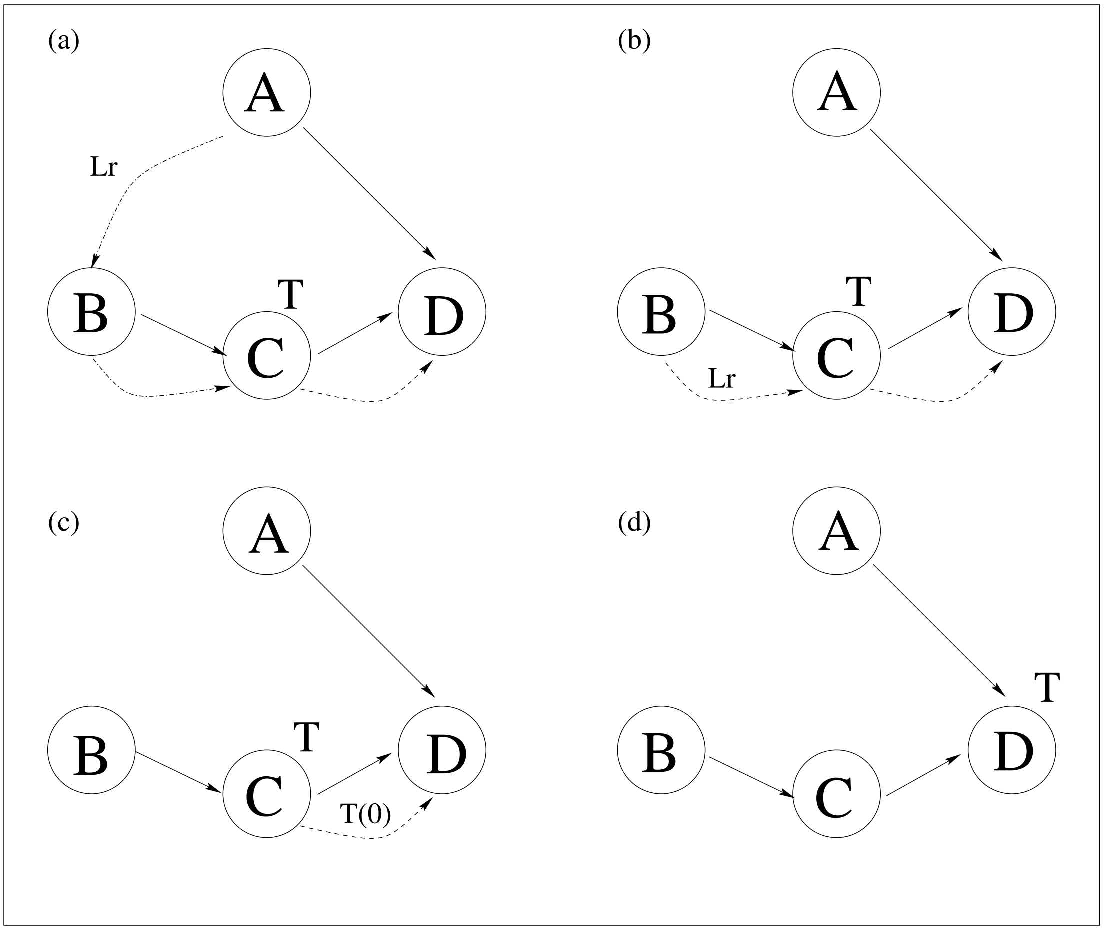


Abbildung 15: Beispielausführung beim DVARW.

den Ausschluß von gleichzeitigen schreibenden Zugriffen. Befindet sich ein Leser im System, erhalten alle Knoten, die ebenfalls lesend ihre kritische Sektion betreten wollen, sofort das Token. Die Leser bauen sukzessive eine Kette auf, die sich beim Verlassen der kritischen Sektion durch die einzelnen Knoten abbaut. Eine Schreibanfrage beendet diesen Mechanismus. Erst nachdem der Knoten, der die Schreibanfrage generiert hat, seine kritische Sektion abgearbeitet hat, erhalten die anderen Knoten die Möglichkeit, lesend ihre kritische Sektion zu betreten.

6.3 SVARW

Der SVARW ist ebenfalls ein tokenbasierender Algorithmus. Er arbeitet im Gegensatz zum DVARW auf einer statischen Baumstruktur. Der Basisalgorithmus wurde von Neilsen und Mizuno entwickelt [11]. Das System der Zeiger *Dir* der Knoten gibt wiederum den Weg von Tokenanfragen vor. Die Struktur kann beliebig, jedoch zyklenfrei, gewählt werden. Im Verlauf des Algorithmus ändert sich lediglich die Richtung der Zeiger *Dir*. Die Struktur wird deshalb als statisch bezeichnet. Bei den weiteren Ausführungen wird von einem binären Baum der Zeiger *Dir* ausgegangen. Das heißt, ein ausgezeichneter Knoten ist der Tokenbesitzer und somit die Wurzel des Baumes. Bei ihm ist $Token == TRUE$ und $Dir == NULL$ gesetzt. Bei allen anderen Knoten ist $Token == FALSE$ und *Dir* zeigt auf die Wurzel des Unterbaums, in dem sich das Token befindet. Die Zeiger *Dir* verweisen zu jedem Zeitpunkt des Verlaufs des Algorithmus auf die Wurzel des Unterbaums, in dem sich das Token (es existieren keine aktuellen Tokenanfragen im System) bzw. das Ende der verteilten Warteschlange (es existieren aktuelle Tokenanfragen im System) befindet. Die Richtung der Zeiger *Dir* wird mit dem Erhalt einer Schreib- bzw. Leseanfrage und deren möglicher Weitersendung verändert. Erhält ein Knoten die Nachricht *READ_REQUEST* bzw. *WRITE_REQUEST* und ist *Dir* nicht *NULL*, sendet er die Nachricht weiter in Richtung *Dir*. Im Unterschied zum DVARW setzt er nun nicht den Zeiger *Dir* auf den Knoten, der die Anfrage ursprünglich generiert hat, sondern auf den Knoten der ihm die Nachricht gesendet hat. Dieser ist ein Nachbar im binären Baum. Der Algorithmus ist in den Abbildungen 16 bis 24 im Pseudocode gegeben. Im Abschnitt 6.4 befinden sich Beispiele, die den Algorithmus illustrieren.

Der initiale Tokenbesitzer wird an die Funktion *mutex_init()* im Parameter *id* übergeben. Der zweite Parameter *up_node* bestimmt den Vater im binären Baum. Der Knoten mit der ID gleich *id* ist der initiale Tokenbesitzer und hat keinen Vater im binären Baum (Abbildung 16 Zeile 4 bis Zeile 6). Die Knoten, deren ID ungleich *id* sind, setzen ihren Zeiger *Dir* auf *up_node* und *Token* auf *FALSE* (Abbildung 16 Zeile 8 und 9). Den Flags bzw. Zeigern *Next*, *Next_Reader*, *Next_Last_Reader*, *Last_Reader*, *Read*, *Write* und *Read_Requested* wird bei allen Knoten *FALSE* bzw. *NULL* zugewiesen (Abbildung 16 Zeile 11 bis Zeile 17).

Ein Knoten, der seine kritische Sektion zum Lesen betreten will, ruft die Funktion *mutex_read_lock()*. In ihr wird zunächst getestet, ob das Token lokal verfügbar ist. Ist dies der Fall, wird das Flag *Last_Reader* auf *TRUE* gesetzt, weil sich kein weiterer Leser im System befindet. Der Knoten verläßt, nachdem er *Read* auf *TRUE* gesetzt hat, die Funktion *mutex_read_lock()* (Abbildung 17, Zeile 22 und Zeile 36). Ist das Token nicht lokal verfügbar, sendet der Knoten die Nachricht *READ_REQUEST* in Richtung *Dir*, setzt *Read_Requested* auf *TRUE* und *Dir* auf *NULL*. Danach blockiert der Knoten auf *await(Token)* (Abbildung

```
mutex_init(id,up_node)
begin
   Self = node(id);
   if (id == 0)
      Token = TRUE;
      Dir = NULL;
   else
      Dir = up_node;
      Token = FALSE;
   fi
   Next = NULL;
   Next_Reader = NULL;
   Next_Last_Reader = NULL;
   Last_Reader = FALSE;
   Read = FALSE;
   Write = FALSE;
   Read_Requested = FALSE;
end mutex_init.
```

Abbildung 16: Initialisierung der Struktur beim SVARW.

17, Zeile 24 bis Zeile 27). Kehrt *await(Token)* zurück, d. h. das Token hat den Knoten erreicht, setzt der Knoten zunächst *Read_Requested* auf *FALSE*. Danach kontrolliert er mit (*Next_Reader != NULL*), ob im System weitere Leseanfragen existieren. Wenn ja, sendet er das Token in Richtung *Next_Reader*, setzt *Token* auf *FALSE*, fügt *Next_Reader* in die Wartschlange *Next_Last_Reader* ein und setzt *Next_Reader* auf *NULL*. Die Tokennachricht führt in diesem Fall die Information mit sich, daß das Token von einem Leser versendet wurde. Ist *Next_Reader* gleich *NULL*, verläßt der Knoten, nachdem er *Read* auf *TRUE* gesetzt hat, die Funktion *mutex_read_lock()* (Abbildung 17, Zeile 28 bis Zeile 36).

Verläßt ein Leser seine kritische Sektion, führt er die Funktion *mutex_read_unlock()* aus. Zunächst wird *Read* auf *FALSE* gesetzt. Anschließend wird *Last_Reader* auf *TRUE* und *Next_Last_Reader* auf ungleich *NULL* getestet, um festzustellen, ob sich ein Knoten im System befindet, der auf die Nachricht *LAST_READER* wartet. Ist dies der Fall wird der erste Knoten aus der lokalen Warteschlange *Next_Last_Reader* entnommen, *Last_Reader* auf *FALSE* gesetzt und die Nachricht *LAST_READER* an den aus der Warteschlange entnommenen Knoten gesendet (Abbildung 18, Zeile 40 bis Zeile 47). Danach wird der Fall betrachtet, daß der Knoten der letzte Leser war und ihn während des Lesens eine Schreibanfrage erreichte. Ist dieser Fall gegeben, ist *Next* ungleich *NULL* und *Last_Reader* gleich *TRUE*. Somit ist der Knoten der Tokenbesitzer, sendet das Token zu *Next*, setzt

```
mutex_read_lock
begin
   if (Token == TRUE)
      Last_Reader = TRUE;
   else
      send_read_request(Self) to Dir;
      Read_Requested = TRUE;
      Dir = NULL;
      await(Token);
      Read_Requested = FALSE;
      if (Next_Reader != NULL)
         send_token(0) to Next_Reader;
         Token = FALSE;
         push(Next_Reader,Next_Last_Reader);
         Next_Reader = NULL;
      fi
   fi
   Read = TRUE;
end mutex_read_lock.
```

Abbildung 17: Die Funktion *mutex_read_lock()* beim SVARW.

daraufhin *Token* auf *FALSE* und *Next* auf *NULL*. Danach kehrt die Funktion *mutex_read_unlock()* zurück (Abbildung 18, Zeile 48 bis Zeile 55).

Die Funktion *mutex_write_lock()* ist identisch der Funktion *mutex_lock()* des Basisalgorithmus. Ist das Token lokal verfügbar, setzt der Knoten *Write* auf *TRUE* - *Write* ersetzt die Variable *cs* des Basisalgorithmus - und verläßt die Funktion *mutex_write_lock()* (siehe Abbildung 20, Zeile 58 und Zeile 63). Ist das Token nicht lokal verfügbar, sendet der Knoten eine Tokenanfrage in Richtung *Dir*, setzt *Dir* auf *NULL* und blockiert auf *await(Token)* (siehe Abbildung 20, Zeile 59 bis 61). Kehrt *await(Token)* zurück, wird *Write* auf *TRUE* gesetzt und der Knoten verläßt die Funktion *mutex_write_lock()* (siehe Abbildung 20, Zeile 63).

Verläßt ein Schreiber die kritische Sektion, ruft er die Funktion *mutex_write_unlock()*. Zunächst wird *Write* *FALSE* zugewiesen. Danach wird getestet, ob den Knoten während der Ausführung der kritischen Sektion eine Tokenanfrage erreicht hat. In diesem Fall ist *Next* ungleich *NULL* (siehe Abbildung 20, Zeile 68) und der Knoten sendet das Token in Richtung *Next*. Nach dem Versenden des Tokens wird *Token* *FALSE* zugewiesen und *Next* auf *NULL* gesetzt. Der Knoten verläßt die Funktion *mutex_write_unlock()* (siehe Abbildung 20, Zeile 69 bis Zeile

```
mutex_read_unlock
begin
   Read = FALSE;
   if (Last_Reader == TRUE) and
      (Next_Last_Reader != NULL)

      Node = fetch(Next_Last_Reader);
      Last_Reader = FALSE;
      send_last_reader() to Node;
   fi
   if (Next != NULL) and
      (Last_Reader == TRUE)

      send_token(0) to Next;
      Token = FALSE;
      Next = NULL;
   fi
end mutex_read_unlock.
```

Abbildung 18: Die Funktion *mutex_read_unlock()* beim SVARW.

72). Erreichte den Knoten keine Anfrage kehrt *mutex_write_unlock()* zurück und der Knoten behält das Token, ohne es zu nutzen.

Abbildung 21 zeigt den Handler für die Nachricht *TOKEN*: Der Knoten setzt *Token* auf *TRUE*. Danach überprüft er ob die Nachricht von einem Leser versendet wurde. (Abbildung 21, Zeile 77 bis Zeile 81). Wurde das Token von einem Leser versendet, wird *Last_Reader* *TRUE* zugewiesen, andernfalls *FALSE*. Zugesichert ist damit, daß das Token nicht zu einem Knoten der schreibend seine kritische Sektion betreten will gesendet wird, solange Leser im System existieren.

Empfängt ein Knoten die Nachricht *WRITE_REQUEST*, führt er den Nachrichtenhandler, dargestellt in Abbildung 22, aus. Ist der Zeiger *Dir* nicht *NULL*, sendet er die Nachricht in Richtung *Dir* weiter. Ist *Dir* *NULL*, befindet sich der Empfänger entweder in seiner kritischen Sektion, wartet auf das Token oder hält das Token ohne es zu nutzen. Der Knoten testet als erstes, ob er im Besitz des Tokens ist (Abbildung 22, Zeile 88). Ist er im Besitz des Tokens und weder schreibend noch lesend in seiner kritischen Sektion, sendet er das Token sofort an den Initiator der Tokenanfragen und setzt danach *Token* auf *FALSE*. Befindet er sich in seiner kritischen Sektion, weist er *Next* den Parameter *sender* zu. Der anfragende Knoten befindet sich daraufhin am Ende der verteilten Warteschlange (siehe Abbildung 22, Zeile 93). Ist der Empfänger nicht im Besitz des Tokens, be-

```
mutex_write_lock
begin
   if (Token != TRUE)
      send_write_request(Self) to Dir:
      Dir = NULL;
      await(Token);
   fi;
   Write = TRUE;
end mutex_write_lock.
```

Abbildung 19: Die Funktion *mutex_write_lock()* beim SVARW.

```
mutex_write_unlock
begin
   Write = FALSE;
   if (Next != NULL)
      send_token(0) to Next;
      Token = FALSE;
      Next = NULL;
   fi;
end mutex_write_unlock.
```

Abbildung 20: Die Funktion *mutex_write_unlock()* beim SVARW.

findet er sich wartend am Ende der *Next*-Warteschlange und weist somit seinem Zeiger *Next* wiederum *sender* zu (siehe Abbildung 22, Zeile 96).

In allen Fällen weist der Empfänger seinem Zeiger *Dir* den Absender der Nachricht zu, weil dieser den Weg für weitere Anfragen vorgibt (Abbildung 22, Zeile 99).

Abbildung 23 zeigt den Nachrichtenhandler der Nachricht *READ_REQUEST*. Zunächst testet der Knoten seinen Zeiger *Dir*. Ist dieser nicht *NULL*, sendet er die Anfragen in Richtung *Dir* weiter.

Anderenfalls testet er, ob er im Besitz des Tokens ist. Ist das Token lokal verfügbar und befindet sich der Knoten nicht in seiner kritischen Sektion, sendet er das Token an den Absender der Tokenanfrage und weist danach *Token FALSE* zu. Befindet sich der Empfänger in seiner kritischen Sektion, sind zwei Fälle zu unterscheiden: der Knoten ist ein Leser oder ein Schreiber.

Im Fall des Lesens sendet er das Token an den Absender der Anfrage, setzt *Token* auf *FALSE* und fügt den Initiator der Nachricht in die Warteschlange *Next_Last_Reader* ein. Die Tokennachricht führt in diesem Fall den Vermerk mit

```
Beim Empfang von Token(flag)
begin
   Token = TRUE;
   if (flag == 0)
      Last_Reader = TRUE;
   else
      Last_Reader = FALSE;
   fi
end.
```

Abbildung 21: Reaktion des SVARW auf die Nachricht *TOKEN*.

sich, daß das Token von einem Leser verschickt wurde (siehe Abbildung 23, Zeile 111 bis Zeile 114). Im Fall das Schreibens weist der Empfänger *Next* den Initiator der Nachricht zu, der somit der nächste Knoten in der verteilten Warteschlange ist.

Ist der Empfänger nicht der momentane Tokenbesitzer, testet er, ob er eine aktuelle Leseanfrage generiert hat (Abbildung 23, Zeile 119). Ist dies der Fall, weist er *Next_Reader* den Initiator der Nachricht zu und fügt ihn zusätzlich in die lokale Warteschlange *Next_Last_Reader* ein.

Als letztes wird der Fall betrachtet, daß der Knoten auf das Token wartet und sich somit am Ende der verteilten Warteschlange befindet: Der Knoten setzt in diesem Fall *Next* auf den Initiator der Nachricht der Tokenanfrage (Abbildung 23, Zeile 123). In allen Fällen weist der Empfänger der Variablen *Dir* den Absender der Anfrage zu (Abbildung 23, Zeile 127).

Empfängt der Knoten die Nachricht *LAST_READER*, wird der Nachrichtenhandler dargestellt in Abbildung 24 aufgerufen. Der Knoten setzt zunächst *Last_Reader* auf *TRUE*. Danach testet er, ob die Nachricht weitergesendet werden muß (Abbildung 24, Zeile 132). Befinden sich Knoten im System, die auf die Nachricht *LAST_READER* warten, ist die Warteschlange *Next_Last_Reader* nicht leer und der Knoten entfernt den ersten Eintrag aus der Warteschlange, sendet *LAST_READER* an den entnommenen Knoten und setzt *Last_Reader* wieder auf *FALSE*. Danach verläßt der Knoten den Handler (Abbildung 24, Zeile 132 bis Zeile 135).

Ist die Warteschlange *Next_Last_Reader* leer und befindet sich ein auf das Token wartender Knoten im System (*Next != NULL*), sendet er das Token in Richtung *Next* weiter und setzt daraufhin *Token* auf *FALSE* (Abbildung 24, Zeile 138 bis Zeile 145). Der Knoten verläßt danach den Nachrichtenhandler.

```
Beim Empfang von write_request(requester,sender)
begin
    if (Dir != NULL)
        send_write_request(requester,Self) to Dir;
    else
        if (Token == TRUE)
            if (Read == FALSE) and (Write == FALSE)
                send_token(1) to requester;
                Token = FALSE;
            else
                Next = requester;
            fi
        else
            Next = requester;
        fi
    fi
    Dir = sender;
end.
```

Abbildung 22: Reaktion des SVARW auf die Nachricht *WRITE_REQUEST*.

```
Beim Empfang von read_request(requester,sender)
begin
  if (Dir != NULL)
    send_read_request(requester,sender) to Dir;
  else
    if (Token == TRUE)
      if (Write == FALSE) and (Read == FALSE)
        send_token(0) to requester;
        Token = FALSE;
      else
        if (Read == TRUE)
          send_token(1) to requester;
          Token = FASLE;
          push(requester,Next_Last_Reader);
        else
          Next = requester;
      fi
    else
      if (Read_Requested == TRUE)
        Next_Reader = requester;
        push(requester,Next_Last_Reader);
      else
        Next = requester;
      fi
    fi
  fi
  Dir = sender;
end.
```

Abbildung 23: Reaktion des SVARW auf die Nachricht *READ_REQUEST*.

```
Beim Empfang von last_reader()
begin
   Last_Reader = TRUE;
   if (Next_Reader != NULL)
      Node = fetch(Next_Reader);
      send_last_reader() to Node;
      Last_Reader = FALSE;
   fi
   if (Next != NULL) and
      (Last_Reader == TRUE) and
      (Token == TRUE) and
      (Read == FALSE)
      send_token(0) to Next;
      Token = FALSE;
      Next = NULL;
      Last_Reader = FALSE;
   fi
end.
```

Abbildung 24: Reaktion des SVARW auf die Nachricht *LAST_READER*.

6.4 Beispiel

Wiederum wird ein System von vier Knoten, A, B, C und D betrachtet. Für die Grafiken gelten die selben Bezeichnungen wie in den Beispielen des Algorithmus DVARW in Abschnitt 6.2. Knoten A ist der initiale Tokenbesitzer. Die Knoten sind in einem binären Baum der Zeiger *Dir* organisiert. Knoten A ist die Wurzel des Baumes. Abbildung 25 (a) zeigt den Initialzustand des Systems.

Knoten A betritt schreibend seine kritische Sektion. Knoten B führt ein *mutex_writelock()* aus, stellt fest, daß er nicht der Tokenbesitzer ist und sendet somit eine Schreibanfrage an Knoten A (Abbildung 25 (a)). Knoten A empfängt die Anfrage und setzt, weil er sich in seiner kritischen Sektion befindet, seinen Zeiger *Next* auf Knoten B. Zusätzlich setzt er seinen *Dir* Zeiger ebenfalls auf Knoten B, das Ende der verteilten Warteschlange (Abbildung 25 (b)).

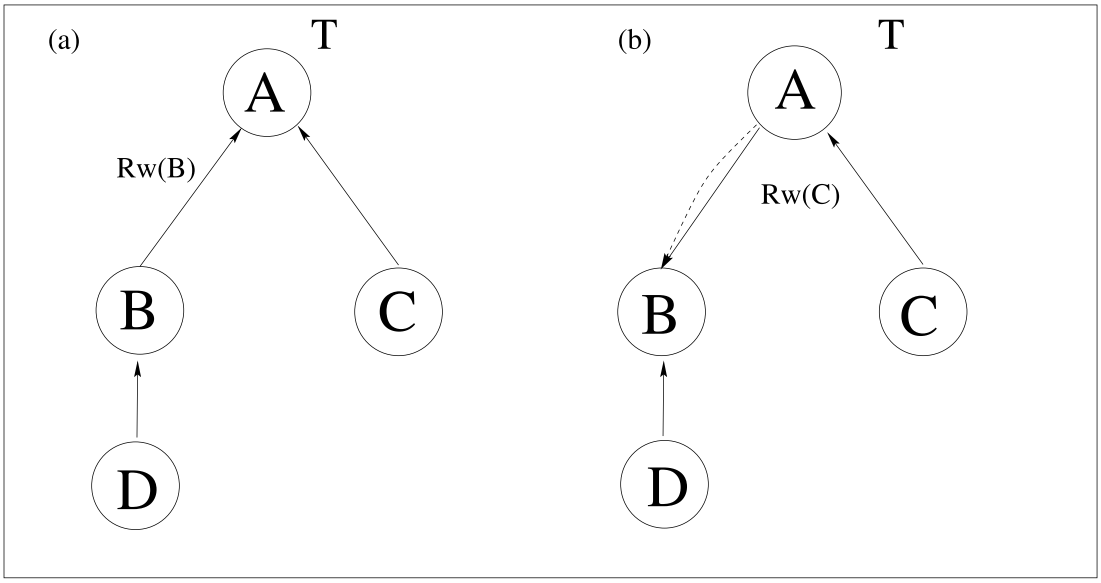


Abbildung 25: Beispielausführung beim SVARW.

Als nächstes sendet Knoten C eine Schreibanfrage (Abbildung 25 (b)) . Diese Anfrage gelangt über Knoten A, der seinen Zeiger *Dir* auf Knoten C setzt, zu Knoten B. Knoten B setzt seinen *Dir* Zeiger auf den direkten Absender der Anfrage, Knoten A. Danach setzt er den Zeiger *Next* auf den Knoten, der die Anfrage generiert hat, Knoten C. Der ursprüngliche Absender einer Anfrage wird mit einer Anfrage versendet. Knoten C bildet nun das Ende der Warteschlange und Anfragen aller Knoten gelangen auf dem Pfad der *Dir* Zeiger des Systems zu ihm (Abbildung 26 (a)).

Die Anfrage von Knoten D gelangt über Knoten B und Knoten A, welche ihre Zeiger *Dir* umsetzen, zu Knoten C. Knoten C setzt *Dir* auf Knoten A und *Next*

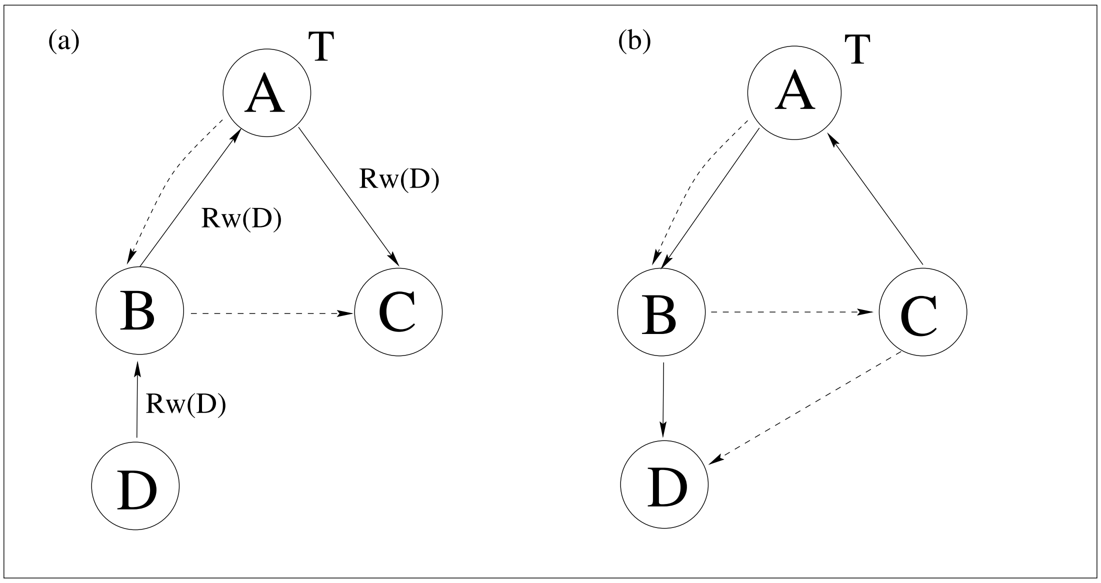


Abbildung 26: Beispielausführung beim SVARW.

auf Knoten D (Abbildung 26 (b)).

Verläßt nun Knoten A seine kritische Sektion, testet er den Zeiger *Next*. *Next* ist nicht *NULL*, sondern verweist auf Knoten B. Somit sendet Knoten A das Token zu Knoten B und löscht daraufhin seinen Zeiger *Next*. Wenn Knoten B die Tokennachricht von Knoten A empfängt, betritt er seine kritische Sektion (Siehe Abbildung 27 (a) und Abbildung 27 (b)).

Nacheinander können nun Knoten B, Knoten C und Knoten D das Token konsumieren. Die Kette der Zeiger *Next* baut sich ab. Nachdem Knoten D seine kritische Sektion verlassen hat, testet er seinen Zeiger *Next*. Dieser ist Null, somit behält Knoten D das Token, ohne es nutzen (Abbildung 28 (a) und (b)).

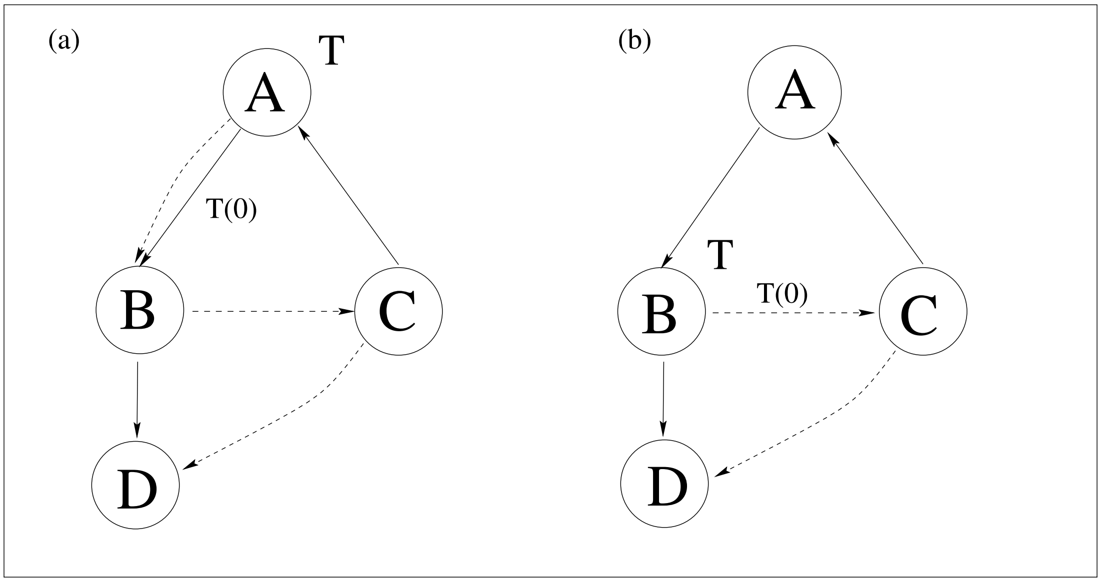


Abbildung 27: Beispielausführung beim SVARW.

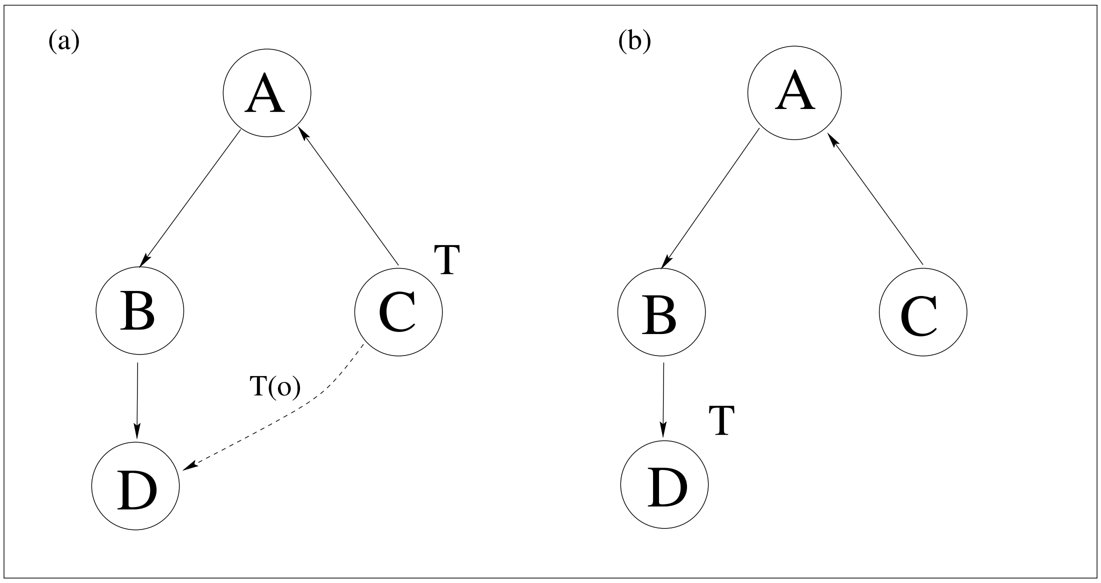


Abbildung 28: Beispielausführung beim SVARW.

Die Abbildungen 29 ff. sollen den Fall, daß Schreib- und Leseanfragen von den Knoten generiert werden, illustrieren. Die Ausgangssituation ist identisch den vorangegangenen Beispielen.

Knoten A ist der initiale Tokenbesitzer und betritt als dieser, mit einem *mutex_readlock()*, sofort lesend seine kritische Sektion. Während der Abarbeitung erreicht Knoten A eine Leseanfrage von Knoten B. Knoten B hat ebenfalls ein *mutex_readlock()* ausgeführt und nach dem Test des Tokens eine Anfrage in Richtung *Dir* gesendet. Knoten A sendet das Token, da es sich um eine Leseanfrage handelt und Knoten A selbst lesend in der kritischen Sektion ist, sofort an Knoten B und setzt *Next_Last_Reader* und *Dir* auf Knoten B. Die Tokennachricht führt in diesem Fall das Flag 1 mit sich, welches anzeigt, daß das Token von einem lesenden Knoten versendet wurde (siehe Abbildung 29 (a) und (b)). Dieses Flag bewirkt, daß Knoten B sein Flag *Last_Reader* auf *FALSE* setzt.

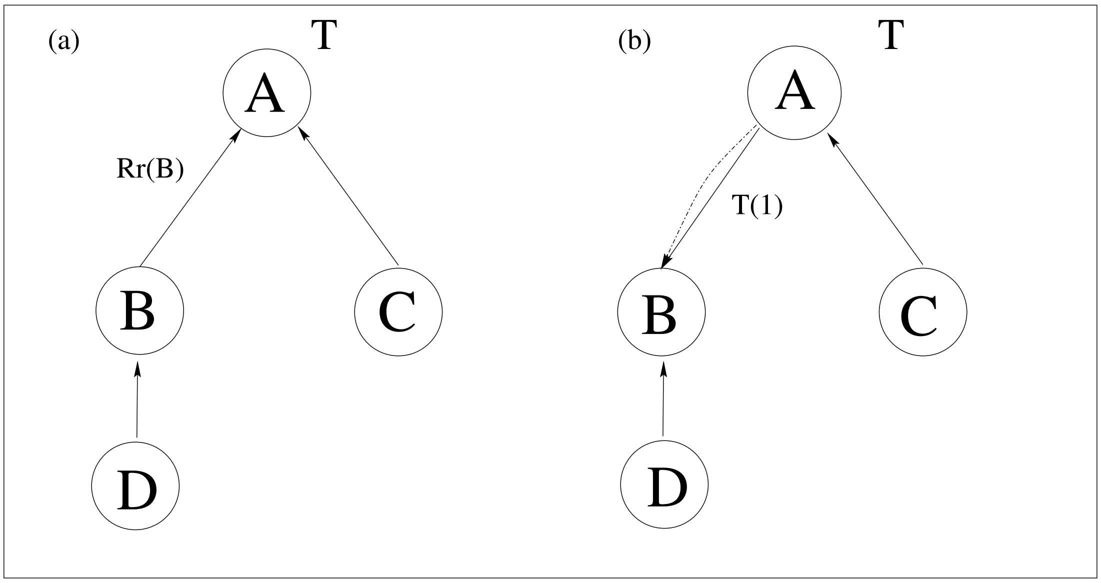


Abbildung 29: Beispielausführung beim SVARW.

Als nächstes erreicht Knoten B eine Leseanfrage von Knoten C. Diese Anfrage wurde über Knoten A transportiert. Knoten A hat somit seinen *Dir* Zeiger auf Knoten C gesetzt. Knoten B setzt seinen Zeiger *Dir* mit dem Empfang der Nachricht auf Knoten A, welcher der direkte Absender der Nachricht ist. Danach sendet Knoten B das Token sofort an den in der Nachricht mitgeführten Initiator. Danach fügt er Knoten C in die Warteschlange *Next_Last_Reader* ein. Das Token führt wiederum ein Flag mit sich, welches angibt, daß das Token von einem Leser versendet wurde. Knoten C empfängt das Token, setzt *Last_Reader* auf *FALSE* und betritt seine kritische Sektion (Abbildung 30 (a) und (b)) .

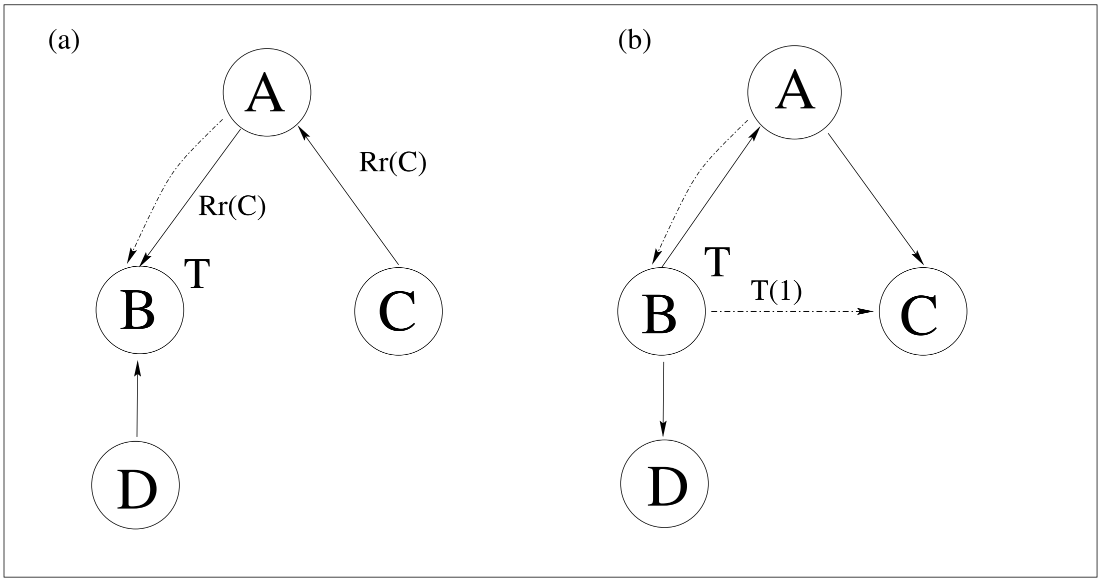


Abbildung 30: Beispielausführung beim SVARW.

Nun führt Knoten D ein *mutex_writelock()* aus. Nachdem der Test des Tokens negativ ist, sendet Knoten D eine Schreibanfrage in die Richtung die ihm *Dir* weist und setzt danach *Dir* auf NULL. Knoten B und Knoten A senden die Schreibanfrage von Knoten D weiter in Richtung ihrer *Dir* Zeiger und setzen danach *Dir* auf den direkten Absender der Anfrage von Knoten D (siehe Abbildung 31 (a)). Wenn Knoten C die von Knoten D generierte Schreibanfrage erhält, setzt er zunächst seinen Zeiger *Dir* auf Knoten A. Knoten C ist der Tokenbesitzer und befindet sich lesend in seiner kritischen Sektion. Sein Flag *Last_Reader* ist *FALSE*. Daraus folgt, da es sich um eine Schreibanfrage handelt, daß Knoten C seinen *Next* Zeiger auf Knoten D setzt (siehe Abbildung 31 (a)).

Knoten C behält das Token so lange, bis er die Nachricht *LAST_READER* erhält, dabei spielt es keine Rolle, ob Knoten C seine kritische Sektion verlassen hat oder in welcher Reihenfolge Knoten A und Knoten B ihre kritische Sektion verlassen. Hat nun Knoten C die Nachricht *LAST_READER* erhalten und seine kritische Sektion verlassen, sendet er das Token zu Knoten D und löscht seinen *Next* Zeiger.

Knoten D erhält somit das Token und die Möglichkeit seine kritische Sektion zu betreten (siehe Abbildung 32 (a) und (b)).

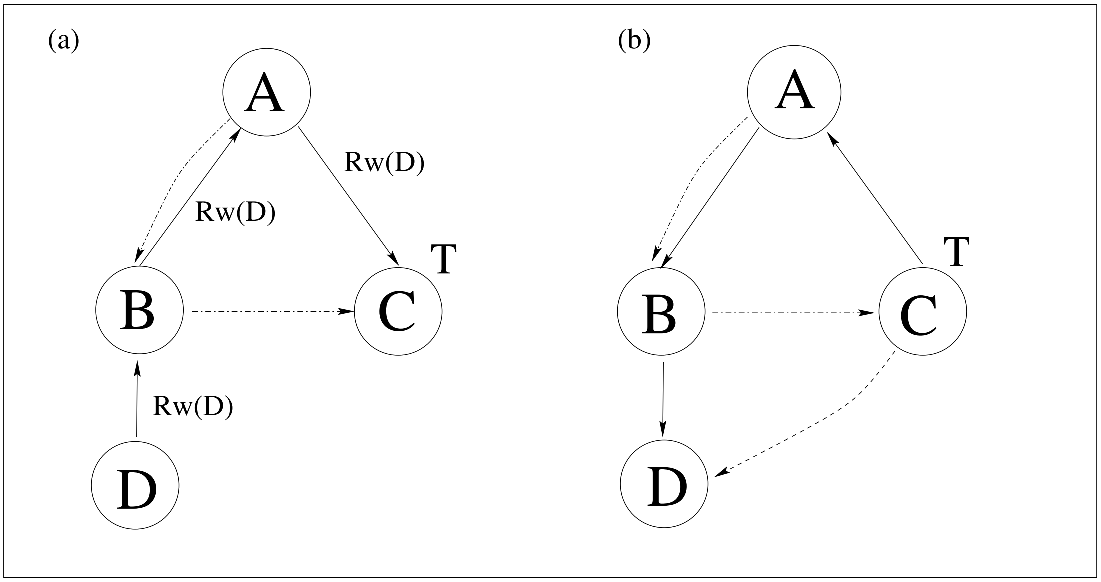


Abbildung 31: Beispielausführung beim SVARW.

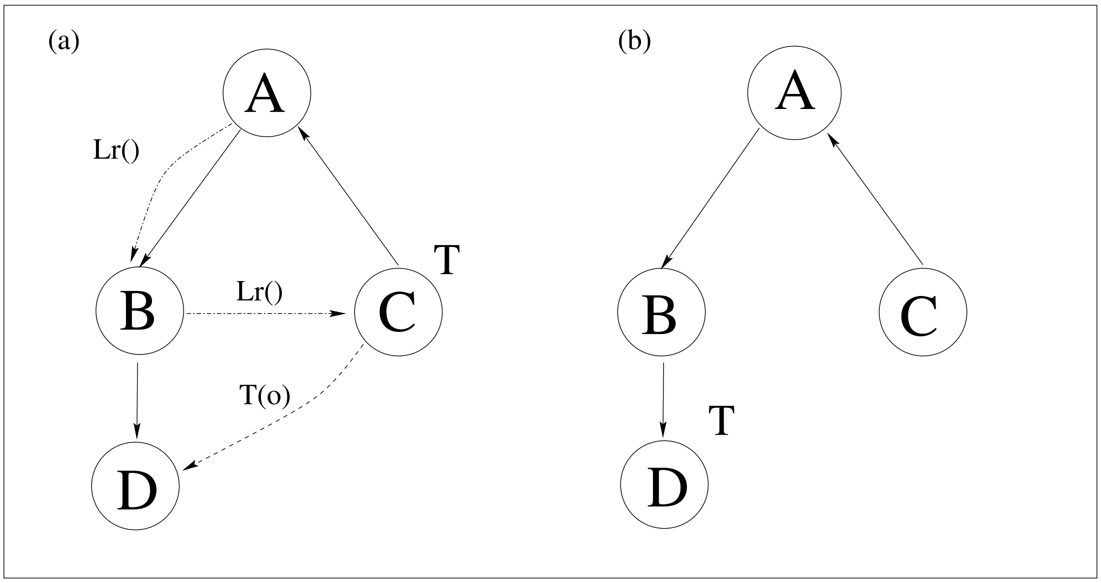


Abbildung 32: Beispielausführung beim SVARW.

6.5 Korrektheit

Der nun folgende Abschnitt soll die Korrektheit von Protokollen spezifizieren, die Lese- und Schreibsperren realisieren.

Gegenseitiger Ausschluß

Der gegenseitige Ausschluß ist erfüllt, wenn die folgenden Aussagen gelten:

1. Existiert in dem betrachteten System zu einem Zeitpunkt ein schreibender Knoten, so folgt, daß weder ein weiterer schreibender noch ein lesender Knoten existiert.

2. Existiert in dem betrachteten System zu einem Zeitpunkt ein lesender Knoten, so folgt, daß kein schreibender Knoten existiert.

Freiheit von Verklemmungen

Erfüllt der Algorithmus die folgenden zwei Bedingungen, so ist er frei von Verklemmungen.

1. Tokenanfragen erreichen immer den Tokenbesitzer bzw. den vermeintlichen Tokenbesitzer[5], d.h. der Weg der Tokenanfragen ist frei von Zyklen und nicht unterbrochen.

2. Der Anfragende Knoten erhält das Token letztendlich, d.h. der Tokenbesitzer sendet das Token, und der Weg des Tokens ist frei von Zyklen und nicht unterbrochen.

Vermeidung des Verhungerns von Knoten

Verhungern bezeichnet Szenarien, in denen sich eine Anzahl von Knoten abwechselnd das Token zusendet, und ein weiterer Knoten, der das Token angefragt hat, nie die Chance bekommt, seine kritische Sektion zu betreten. Lokale Fairness impliziert den Ausschluß des Verhungerns. Somit ist zu zeigen, daß folgende Aussage gilt:

1. Anfragen (zum Lesen oder Schreiben) werden auf den Knoten in der Reihenfolge verarbeitet, wie sie empfangen werden.

Dies gilt im speziellen für lokale Anfragen und für Schreibanfragen, während sich Knoten des Systems lesend in ihrer kritischen Sektion befinden.

[5] Vermeintlicher Tokenbesitzer bezeichnet hier, das Ende der durch die Zeiger *Next* aufgebauten, verteilten Warteschlange.

6.6 Korrektheit des DVARW und SVARW

Die Algorithmen DVARW und SVARW unterscheiden sich algorithmisch nur in einem Punkt: dem Verändern der Zeiger *Dir*, welche die Wege der Anfragen durch das System bestimmen. Aus diesem Grund erfolgt der Beweis für beide Algorithmen in einem Schritt. Lediglich im Punkt 4, Abschnitt Beweis, muß eine Fallunterscheidung erfolgen. Den vollständigen Beweis der Korrektheit der Basisalgorithmen findet der Leser in den Arbeiten von Chang et al. [7] und Neilsen und Mizuno [11].

6.6.1 Voraussetzungen

1. Ein Knoten generiert zu einem Zeitpunkt maximal eine Anfrage.

2. Es gehen keine Nachrichten verloren. Die Laufzeit von Nachrichten ist endlich, jedoch nicht voraussagbar.

3. Die Funktionen *mutex_read_lock()*, *mutex_read_unlock()*, *mutex_write_lock()* und *mutex_write_unlock()* sowie alle Nachrichtenhandler sind atomar verbunden. [6]

4. Im Speziellen sind folgende Anweisungen oder Operationen atomar verbunden:

   - Das Senden des Tokens und die Zuweisung *Token = FALSE*.
   - Das Empfangen des Tokens und die Zuweisung *Token = TRUE*.
   - Der Empfang einer Tokenanfrage und das Setzen des Zeigers *Dir*.

6.6.2 Beweis

1. Initial ist der Boolean *Token* nur bei einem Knoten auf *TRUE* gesetzt. Im Verlauf des Algorithmus sind nach Voraussetzung das Versenden des Tokens und das Setzen von *Token* auf *FALSE* sowie das Empfangen des Tokens und das Setzen von *Token* auf *TRUE* atomar verbunden. Daraus folgt, daß sich in dem System zu jedem Zeitpunkt genau ein Token befindet.

2. Ein Knoten, der seine kritische Sektion betreten will und nicht der Tokenbesitzer ist, blockiert auf der Anweisung *await(Token)*.

3. Befindet sich ein Knoten lesend in seiner kritischen Sektion, so ist er im Besitz des Tokens oder hat das Token zu einem Knoten, von dem er eine Leseanfrage erhalten hat, versendet. Da ein lesender Knoten das Token nur

[6] Der Ruf *await(Token)* verursacht einen Kontextwechsel, darum bedeutet atomar hier die Funktion kann nach dem Start nur an den definierten Stellen verlassen bzw. unterbrochen werden.

nach dem Verlassen der kritischen Sektion und unter den Bedingungen, daß der Boolean *Last_Reader TRUE* ist und *Token* gleich *TRUE* ist, versendet, folgt, daß das Token nur dann zu einem Schreiber gelangt, wenn keine Leser existieren. Ein Schreiber sendet das Token nur nach dem Verlassen der kritischen Sektion zu einem anderen Knoten.

4. Die Zeiger *Dir* bestimmen den Weg einer Tokenanfrage. Der Empfang einer Anfrage (Lese- oder Schreibanfrage) und das Ändern des Zeigers *Dir* auf einem Knoten sind nach Voraussetzung atomar verbunden.

- **DVARW:**

  Initial sind die Zeiger *Dir* der Knoten des Systems in Sternform angeordnet. Das Zentrum dieses Sterns bildet der initiale Tokenbesitzer. Diese Struktur ist frei von Zyklen und die Anfrage eines beliebigen Knotens erreicht den Tokenbesitzer. Das heißt, das System der Zeiger *Dir* befindet sich in einem konsistenten Zustand.

  Sei Knoten **t** das Ziel einer beliebigen Anfrage, d.h. **t** ist der Tokenbesitzer bzw. das Ende der verteilten Wartenschlange. Knoten **r** stellt eine Tokenanfrage. Zu zeigen ist, daß nachdem die Anfrage von Knoten **r** Knoten **t** erreicht hat, die Anfrage eines beliebigen Knotens des Systems Knoten **r** erreicht.
  Beweis:
  Sei Knoten **i** der Knoten, der die Anfrage stellt.
  Fall 1: (**i** = **t**)
  Knoten **t** hat nach Voraussetzung die Anfrage von **r** erreicht. Daraus folgt, daß der Zeiger *Dir* von **t** auf r gesetzt ist. Die Anfrage von **i** erreicht **r**.
  Fall 2:
  i ist Element der Menge der Knoten, welche die Anfrage von **r** weitergesendet haben. Da das Weitersenden einer Anfrage auf einem Knoten atomar mit dem Setzen des *Dir* Zeigers auf den Generator der Anfrage verbunden ist, folgt, daß der Zeiger *Dir* von Knoten **i** gleich **r** ist. Die Anfrage von **i** erreicht **r**.
  Fall 3:
  i ist Element der Menge der Knoten des Systems, welche die Anfrage nicht weitergesendet haben. Nach Voraussetzung erreicht die Anfrage von Knoten **i** den Knoten **t**. Knoten **t** hat nach Voraussetzung die Anfrage von **r** erreicht, d.h. der Zeiger *Dir* von Knoten **t** ist gleich **r**. Da der *Dir* Zeiger ungleich *NULL* ist, sendet t die Anfrage weiter in Richtung *dir*, d.h. zu Knoten **r**. Die Anfrage von **i** erreicht **r**. Es folgt, daß die Anfrage eines beliebigen Knotens ihr Ziel erreicht und

sich somit die Zeiger *Dir* des Systems in einem konsistenten Zustand befinden.

- **SVARW:**

  Beim SVARW bilden die Zeiger *Dir* einen binären Baum. Dieser Baum ändert im Verlauf des Algorithmus nur die Richtung der Kanten. Es kommen weder Kanten hinzu, noch werden Kanten entfernt. Ein binärer Baum ist frei von Zyklen und somit ist das System der Zeiger *Dir* frei von Zyklen.

  Initial bildet der Tokenbesitzer die Wurzel des binären Baums. Die Zeiger *Dir* aller Knoten verweisen initial auf ihren Vater im binären Baum. Lediglich der *Dir* Zeiger des initialen Tokenbesitzers ist *NULL*. Das heißt, die Anfrage eines beliebigen Knotens erreicht den Tokenbesitzer. Das System der Zeiger *Dir* ist konsistent.

  Ein Knoten, der eine Tokenanfrage generiert, setzt seinen Zeiger *Dir* auf *NULL*. Knoten, die eine Tokenanfrage weitersenden, setzen ihren Zeiger *Dir* auf den direkten Absender der Anfrage, einen Nachbarn im binären Baum. Der Mechanismus des Weitersendens endet bei dem Knoten, dessen Zeiger *Dir* gleich *NULL* ist. Sei wiederum Knoten **t** das Ziel einer beliebigen Anfrage, d.h. **t** ist der Tokenbesitzer bzw. das Ende der verteilten Wartenschlange. Knoten **r** stellt eine Tokenanfrage. Zu zeigen ist, daß nachdem die Anfrage von Knoten **r** Knoten **t** erreicht hat, die Anfrage eines beliebigen Knotens des Systems **r** erreicht. Die Anfrage von Knoten **r** beschreibt einen Pfad im binären Baum. Die Knoten in dem Pfad haben die Anfrage von **t** weitergesendet und ihren *Dir* Zeiger so verändert, daß er nun in die Richtung zeigt, aus der sie die Anfrage von Knoten **r** empfangen haben. Den Kopf dieses Pfades bildet Knoten **t**, und das Ende bildet Knoten **r**.
  Beweis:
  Sei Knoten **i** der Knoten, der die Anfrage stellt.
  Fall 1:
  i befand sich in der Menge, die die Anfrage von Knoten **r** weitergesendet haben und gelangt durch den Mechanismus des Weitersendens an das Ende des Pfades: Knoten **r**. Die Anfrage von Knoten **i** erreicht ihr Ziel.
  Fall 2:
  i befindet sich nicht in der Menge der Knoten, die die Anfrage von Knoten **r** weitergesendet haben. Da sich nach Voraussetzung das System in einem konsistenten Zustand befindet, erreicht die Anfrage von Knoten **i** den Knoten **t**. Knoten **t** hat nach Voraussetzung die Anfrage von Knoten **r** erreicht, daraus folgt, daß der Zeiger *Dir* von Knoten **t** nicht *NULL* ist und Knoten **t** die Anfrage weitersendet. Knoten **t**

befindet sich am Anfang der Menge, die die Anfrage von Knoten **r** erhalten haben. Die Anfrage von Knoten **i** erreicht ihr Ziel.

5. Die Kette der Zeiger *Next*, welche den Weg des Tokens durch das System bestimmen, ist eine einfache Liste. Da, wie in Punkt 4 gesehen, Tokenanfragen immer das Ende dieser Kette erreichen und ein Knoten, da er nur eine Anfrage zu einem Zeitpunkt hat, nie mehr als einmal in der Kette steht, ist die Kette frei von Zyklen. Somit erreicht das Token immer den nächsten Tokenbesitzer.

6. Nachrichten werden auf den Knoten in der Reihenfolge verarbeitet, in der sie die Knoten erreichen. Die Operationen der Nachrichtenhandler sind atomar verbunden.

7. Betrachtet wird, im Hinblick auf Schreib- oder Lesezugriffe, immer nur die Anforderung des letzten Knotens, der eine Anfrage generiert hat. Daher ist es unmöglich, daß immer wieder neue Leser einen auf einer Schreibanfrage blockierenden Knoten verhungern lassen.

Aus 1. 2. und 3. folgt der gegenseitige Ausschluß in einem Lese- und Schreibprotokoll.

Aus 4. folgt, daß Anfragen von Knoten immer das Ende der Warteschlange bzw. den Tokenbesitzer erreichen. Aus 5. folgt, daß das Token immer den Knoten, der eine Anfrage gestellt hat, erreicht. Der Algorithmus ist frei von Verklemmungen.

Aus 6. und 7. folgt, daß die Algorithmen das Verhungern von Knoten vermeiden.

Somit sichern die Algorithmen DVARW und SVARW den gegenseitigen Ausschluß, sind frei von Verklemmungen und vermeiden das Verhungern. Sie sind somit korrekte Protokolle für die Realisierung von Schreib- und Lesesperren.

6.7 Komplexität

Im folgenden Abschnitt werden die in Kapitel 6.1 und Kapitel 6.3 vorgestellten Algorithmen auf die Komplexität der Nachrichten in Abhängigkeit von der Größe des Systems N untersucht.

Bei der Verwendung tokenbasierende Algorithmen zum gegenseitigen Ausschluß in verteilten Systemen benötigen die Knoten im besten Fall keine Nachricht um ihre kritische Sektion zu betreten: Ein Knoten, der seine kritische Sektion betreten will, testet zunächst, ob er der Tokenbesitzer ist. Ist dies der Fall, betritt er seine kritische Sektion, ohne mit einem der anderen Knoten des Systems kommuniziert zu haben.

Beim Algorithmus von Chang et al. bzw. Naimi und Trehel (DVARW) beträgt die Laufzeit einer Anfrage im schlechtesten Fall N. Der Weg der Anfragen, beschrieben durch die Zeiger *Dir*, entartet zu einer Liste. Praktisch ist dieser Fall bei

$ncs \geq 2$ jedoch nicht zu erzeugen (siehe [12] Kapitel 7). Die Laufzeit des Tokens in Hops beträgt immer eins, das Token wird direkt zum nächsten Tokenbesitzer gesendet.

Beim SVARW ist der schlechteste Fall gegeben, wenn sich das Token bzw. das Ende der Warteschlange in einem Blatt des durch die Zeiger *Dir* aufgebauten Baumes befindet, der Knoten, der die Anfrage generiert, ebenfalls ein Blatt dieses Baumes ist und somit die Anfrage über die Wurzel versendet werden muß [7]. Die Anfrage gelangt über $2 * \log_2 N$ Hops an ihr Ziel. Das Token wird wiederum direkt zum nächsten Tokenbesitzer, gegeben durch *Next*, gesendet und hat somit eine Laufzeit von einem Hop.

Befinden sich zu einem Zeitpunkt k ($k \leq N$) lesende Knoten in dem betrachteten System, wird die Nachricht *LAST_READER* (k - 1)-mal versendet. Das bedeutet eine zusätzliche Nachricht je kritischer Sektion. Daraus folgt eine Komplexität der Nachrichten je kritischer Sektion für den Algorithmus DVARW von O(N) und für den Algorithmus SVARW von $O(\log_2 N)$ im schlechtesten Fall.

Tabelle 2 faßt die Komplexität der Algorithmen zusammen. Die Komplexität der Nachrichten in Abhängigkeit von der Größe des Systems im mittleren Fall (ac) ist aus [12] entnommen.

	Chang et al. bzw. Naimi und Trehel DVARW	Neilsen et al. SVARW
bc	0	0
wc	O(N)	$O(\log_2 N)$
ac	$\Theta(\ln N)$	$\Theta(\log_2 N)$

Tabelle 2: Komplexität der Nachrichten in Abhängigkeit von der Größe des Systems im besten Fall (bc), im schlechtesten Fall (wc) und im mittleren Fall (ac).

6.8 Performance

Die mittlere Anzahl der Nachrichten *nmE* und die mittleren Wartezeit *rt* bestimmen die Performance eines Algorithmus zum gegenseitigen Ausschluß. Eine ausführliche Analyse der Performance der Basisalgorithmen findet der Leser in [12].

Im folgenden gilt es, die Wirkung der Erweiterung um Lese- und Schreibsperren auf die Performance der Algorithmen zu untersuchen. Außerdem soll getestet werden, inwieweit der Einsatz von Schreib- und Lesesperren die mittlere Anfragezeit und die Gesamtausführungszeit beeinflußt, d. h. rechtfertigt der Gewinn den algorithmischen Mehraufwand der Erweiterung.

Weiterhin wird das Verhalten der Algorithmen bei verschiedenen Anfragefrequenzen untersucht. Von Interesse ist auch das Verhalten der Algorithmen bei Variation der zeitlichen Größe der kritischen Sektion bzw. des nichtkritischen

[7] Ausgegangen wird wiederum von einem binären Baum der Zeiger *Dir*.

Teils. Aus diesem Verhalten können Rückschlüsse auf die Anwendbarkeit bei verschiedenen parallelen Problemen gezogen werden.

Ergebnisse und Bewertung

Im ersten Schritt der Untersuchung wurde das Verhalten der Algorithmen bei Veränderung der Anteile Leseanfragen und bei verschiedenen Systemgrößen untersucht. Beide Algorithmen wurden mit den identischen Eingabeparametern, auf derselben Plattform und unter gleicher Last getestet. Genutzt wurde für die Tests ein mit 133MHz getakteter Pentium unter dem Betriebssystem Linux 2.1.

Die Größe des Systems N variierte zwischen 2 und 64 in Schritten von Potenzen der Basis 2. Die Anzahl der kritischen Sektionen *ncs* ist mit 30 festgesetzt. Verteilter Programmcode besteht aus kritischen und nichtkritischen Teilen. Die kritischen Teile enthalten normalerweise wenige Operationen, bei denen es sich meist um Schreib- und Leseoperationen handelt. Das Lesen oder Schreiben eines Datums in einem System mit verteilten Speicher benötigt wenigsten eine Nachricht. Die Zeit für eine Nachricht beträgt beispielsweise bei einer IBM SP2 110 μsec und bei einer iPSC 170μsec. Somit betrug die Verweildauer in der kritischen Sektion *Phi* 200 μsec und die Zeit zwischen dem Verlassen und dem erneuten Versuch, die kritische Sektion zu betreten, (*Gamma*) 2000 μsec.

Der Faktor 10 resultiert daraus, daß die kritischen Sektionen im allgemeinen kurzgehalten werden, sie enthalten nur wenige Operationen im Gegensatz zum nichtkritischen Teil. Die Anzahl der als Nurlesezugriffe ausgezeichneten Anfragen variierte zwischen 0 und 100% der Gesamtanzahl kritischer Sektionen.

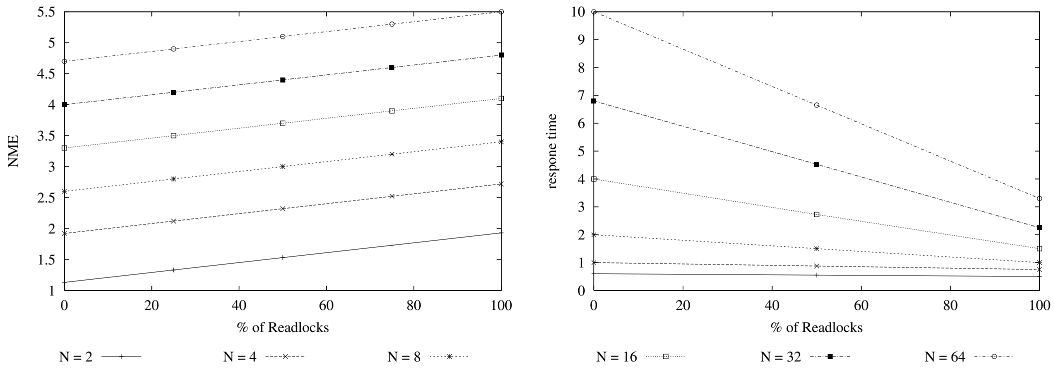


Abbildung 33: Erwartetes Verhalten bei Variation der Anzahl der Knoten und Variation der Anzahl Lesezugriffe.

Abbildung 33 zeigt den erwarteten Verlauf der Kurven in einer idealisierten Darstellung. Die Ergebnisse sind in den Abbildungen 34 (DVARW) und 35 (SVARW) grafisch dargestellt. Die Werte der Wartezeit *rt* ist die Summe der Wartezeit auf eine Lesesperre, gewichtet mit der Prozentzahl der Lesesperren, und die Wartezeit auf eine Schreibsperre, gewichtet mit der Prozentzahl der Schreibsperren.

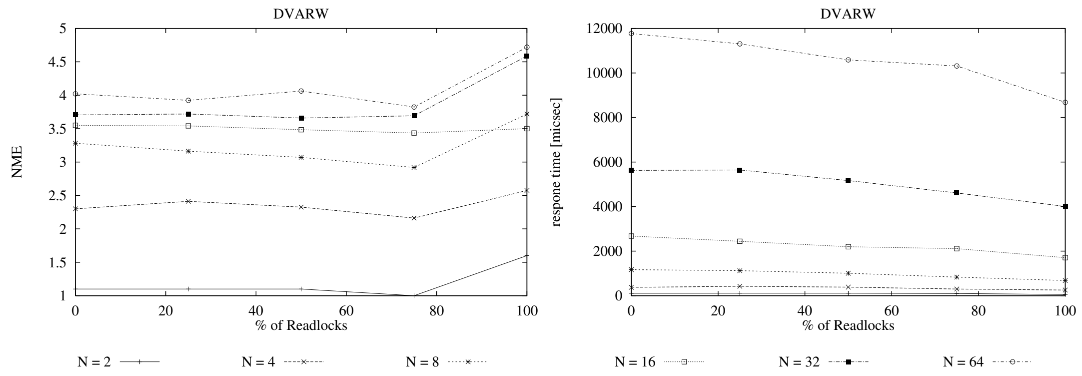


Abbildung 34: Verhalten des DVARW bei Variation der Anzahl der Knoten und Variation der Anzahl Lesezugriffe.

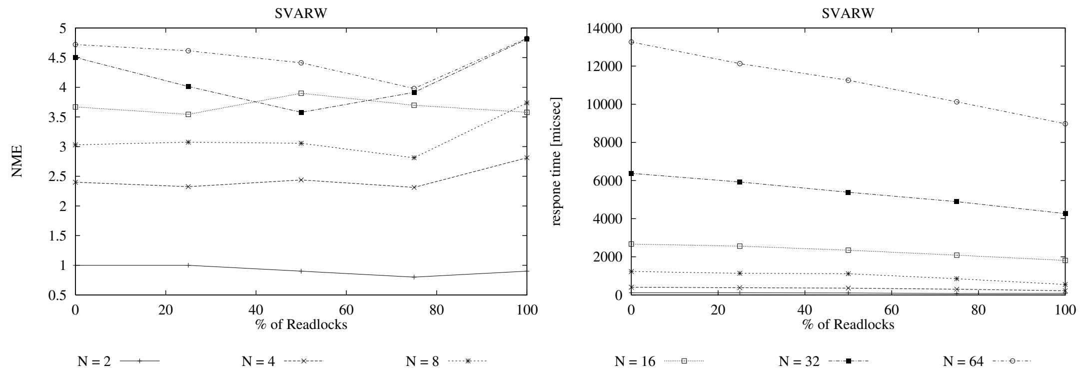


Abbildung 35: Verhalten des SVARW bei Variation der Anzahl der Knoten und Variation der Anzahl Lesezugriffe.

Die Anfragezeit nimmt mit Erhöhung der Anzahl der Leseanfragen für die gleiche Anzahl Knoten und kritischer Sektionen linear ab. Der Wert der negativen Steigung der Kurve der mittleren Anfragezeit wird stärker mit der Zunahme der Anzahl der Knoten im System. Ist die Kurve bei $N = 2$ annähernd konstant, fällt sie bei $N = 64$ um 30%. Dies resultiert ausschließlich aus der Parallelisierung der Lesezugriffe, da alle weiteren Parameter unverändert blieben. Das Token wird versendet, während sich der Knoten in seiner kritischen Sektion befindet.

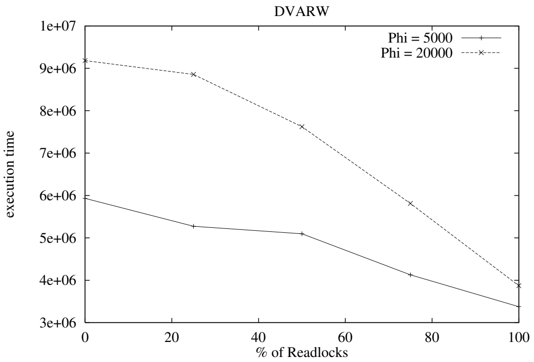
Die mittlere Anzahl der Nachrichten ist in Abhängigkeit von der Anzahl Nurlesezugriffe konstant und liegt ca. 1 über vergleichbaren Messungen mit den Basisalgorithmen (vergleiche [12]). Dies bestätigt die Komplexitätsanalyse des letzten Abschnitts. Die Mehrnachricht von 1 je kritischer Sektion, durch das Senden der

Nachricht *LAST_READER*, wirkt sich erst bei mehr als 75% Lesezugiffe aus. Bei weniger Lesesperren wird die Kette der Leser häufig durch Schreiber unterbrochen. In Abbildung 35 kreuzen sich die Kurven (16 und 32 Knoten) der Anzahl der Nachricht. Eine algorithmische Erklärung für dieses Phänomen kann nicht gefunden werden. Diese Anomalie ist auf die Testumgebung zurückzuführen. Denkbar ist, daß eine oder mehrere Tasks, ohne das Token abzugeben und somit ohne Kommunikation, mehrmals hintereinander ihre kritische Sektion abarbeiten. Betriebssystembedingte Lasten können diesen schlechten Schedule verursachen.

Das Verhalten der Algorithmen ist qualitativ annähernd identisch. Bei großer Anzahl der Knoten ist der Verlauf beim SVARW unregelmäßig. Die gleichzeitig lesenden Knoten können sich in der statischen Struktur gleichverteilt, weit entfernt oder in einem zusammenhängenden Teilbaum befinden. Das heißt, die Anfragewege sind verschieden, jedoch nie größer als $\log_2 N$. Die Anzahl der Nachrichten beim DVARW liegt leicht unter der Anzahl der Nachrichten des SVARW. In einem realen verteilten System würde sich dieser Unterschied stärker auf die mittlere Anfragezeit auswirken, da in der Testumgebung das Netzwerk nur simuliert wurde und das Netzwerk in verteilten Systemen meist den Flaschenhals bildet. Der Unterschied in der Anzahl der Nachrichten der Algorithmen resultiert aus der Benutzung der Pfadreduzierung beim DVARW: Die Wege der Anfragen werden kürzer, d.h. die Anzahl der Hops verringert sich. Dazu ist zu bemerken, daß die Erweiterung der Algorithmen keinen Enfluß auf die Nachrichtenwege hat.

Als nächster soll der Einfluß des Parameters Phi auf die Gesamtausführungszeit dargestellt werden.

Die Größe des Systems wurde bei den folgenden Testreihen mit 16 festgesetzt. Die Anzahl der kritischen Sektionen betrug 20 und der Parameter *Gamma* lag bei konstant 2000 μsec. *Phi* nahm die Werte von 5000 bis 20000 μsec an. Wiederum variierte der Anteil von Nurlesezugriffen zwischen 0 und 100% der Gesamtanzahl der kritischen Sektionen.



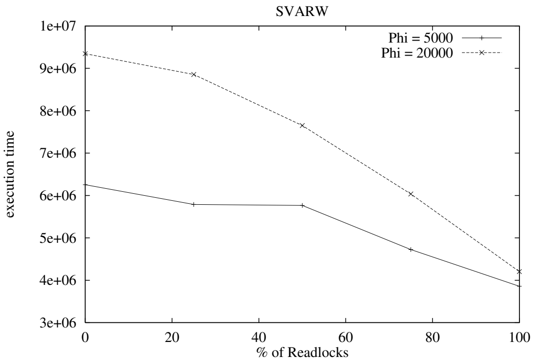


Abbildung 36: Verhalten der Gesamtausführungszeit bei Variation des Parameters Phi.

Die negative Steigung der Kurve der Ausführungszeit nimmt erwartungsgemäß mit dem Wachsen von *Phi* zu, d.h. die Ausführungszeit wird geringer. Erreicht wird ein Gewinn von bis zu 50%.

Von Interesse bei dem Verhalten von Algorithmen zum gegenseitigen Ausschluß ist immer auch die Reaktionen auf verschiedene Anfragefrequenzen. Mit diesen können Rückschlüsse auf die Anwendbarkeit auf verschiedene Probleme unternommen werden. Die Anzahl der Knoten betrug wiederum 32, die Anzahl kritischer Sektionen lag bei 20 und *Phi* wurde auf 200 μsec festgesetzt. *Gamma* nahm Werte zwischen 0 und 5000 μsec an. Die Ergebnisse sind in Abbildung 37 dargestellt.

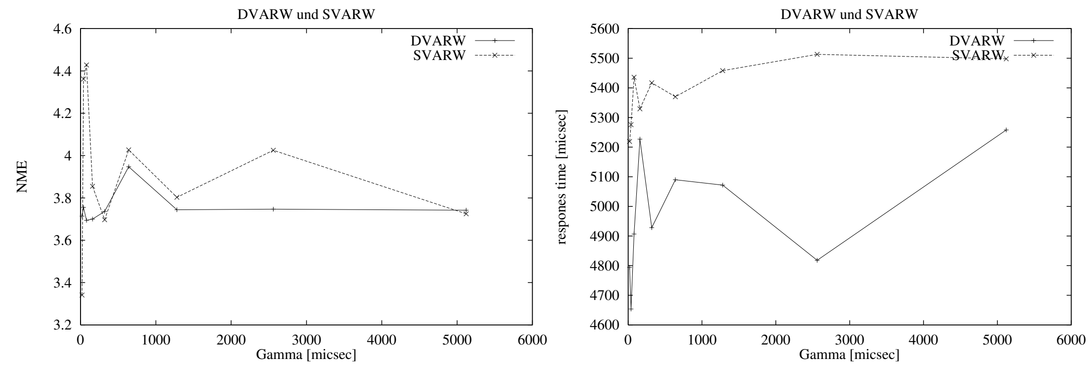


Abbildung 37: Ergebnisse des DVARW und SVARW unter verschiedenen Anfragefrequenzen.

Bei sehr hohen Anfragefrequenzen liegt der DVARW in der mittleren Anzahl der Nachrichten um 1 besser als der SVARW. Auch in der Wartezeit liegt der DVARW unter dem SVARW. Wiederum ist zu bemerken, daß sich der Unterschied der Anzahl der Nachrichten in einem verteilten System stärker auf die Wartezeit auswirken würde. Dieses Bild ändert sich bei sehr geringen Anfragefrequenzen: Der Graph des SVARW erreicht den des DVARW. Die Graphen kreuzen sich, und der SVARW erzielt bessere Ergebnisse. Dieses Verhalten resultiert daraus, daß sich die Anfragewege beim DVARW bei kleinen Anfragefrequenzen kaum noch verzweigen, der Trend zur Liste wird stärker. Somit ist irgendwann der binäre, statische Baum des SVARW der dynamischen Struktur des DVARW überlegen.

Es läßt sich die Schlußfolgerung ziehen, daß bei feingranularen Problemen, d.h. bei denen die Knoten oft auf gemeinsame Daten zugreifen müssen, der DVARW besser geeignet ist. Bei grobgranularen Problemen ist der Einsatz des SVARW angebrachter.

6.9 Zusammenfassung

Algorithmen zum gegenseitigen Ausschluß können ein Protokoll unterstützen, welches gleichzeitig entweder mehreren Lesern oder einem Schreiber den Zugriff auf Ressourcen in einer kritischen Sektion gestattet (These 1 der Arbeit). Algorithmen, die solche Protokolle implementieren, müssen effizient arbeiten. Der Gewinn durch die Parallelisierung der Leseoperationen muß den algorithmischen Mehraufwand der Protokolle wettmachen.

Die Algorithmen DVARW und SVARW sind tokenbasierend. Der Besitz des Tokens sichert im Fall des Schreibens einen exklusiven Zugriff und im Fall des Lesens den Ausschluß von weiteren Schreibern. Die Algorithmen arbeiten nach einem leicht verständlichen Protokoll.

Die Komplexität der Nachrichten beträgt im besten Fall 0, um eine kritische Sektion schreibend oder lesend zu betreten. Die Komplexität der Nachrichten, in Abhängigkeit von der Größe des Systems N, beträgt im schlechtesten Fall je kritischer Sektion beim DVARW $\mathrm{O}(N)$ und beim SVARW $\mathrm{O}(log_2 N)$. Die Komplexität der Nachrichten verändert sich nicht im Vergleich zu den Basisalgorithmen. Die Algorithmen sind korrekte Algorithmen zum gegenseitigen Ausschluß in verteilten Systemen, d.h. durch die Erweiterung um die Möglichkeit von Schreib- und Lesesperren ist weder der gegenseitige Ausschluß verletzt, noch treten Verklemmungen oder das Verhungern auf. Damit wird die These 2 dieser Arbeit bestätigt.

Die Messungen in der Testumgebung, vorgestellt in Abschnitt 5, bestätigen die These 3: Die Wartezeit je kritischer Sektion nimmt mit der als Lesezugriffe ausgezeichneten Anzahl der kritischen Sektionen linear ab. Die Gesamtausführungszeit nimmt mit der Zunahme von Phi ebenfalls linear ab. Die Anzahl der Nachrichten nimmt, im Vergleich zu den Basisalgorithmen, um einen konstanten Wert zu. Die negative Steigung der Wartezeit nimmt mit der Erhöhung der Größe des Systems zu (bis zu 30%) und bleibt gleich bei variierender Anzahl der kritischen Sektionen. Die Zunahme des Parameters *Phi* erhöht die negative Steigung der mittleren Wartezeit. Die mittlere Wartezeit liegt bei 10% Lesesperren doppelt so hoch wie bei 90% Lesesperren. Die Anzahl der Nachrichten pro kritischer Sektion steigt um maximal eins, was die Untersuchungen bezüglich der Komplexität bestätigt. Der DVARW erzielt im allgemeinen bessere Ergebnisse bei den relevanten Parametern. Lediglich bei sehr geringen Anfragefrequenzen ist der SVARW dem DVARW leicht überlegen. Dazu muß bemerkt werden, daß bei geringen Anfragefrequenzen der Einsatz von Schreib- und Lesesperren keinen Effekt hat: Die Wahrscheinlichkeit, Zugriffe zu parallelisieren, ist gering.

7 Fehler und verteilte Systeme

Dieser Abschnitt gibt in Bezugnahme auf verteilte Systeme eine Einführung zu den Begriffen Fehler und Fehlertoleranz.

Fehler und Fehlermodelle

Die Spezifikationen dieses Abschnitts gehen auf die Arbeit von V. Hübsch [45] zurück.

Es existieren drei verschiedene Bedeutungen des Begriffs Fehler.

- Fehlerursache (fault),
- interner Fehler (error) und
- Fehlverhalten (failure).

Fehlverhalten liegt dann vor, wenn das System in seinem externen Verhalten von dem erwarteten Verhalten abweicht. Ein Fehlverhalten ist auf einen internen Fehler zurückzuführen. Dieser interne Fehler kann wiederum auf eine Fehlerursache zurückgeführt werden.

Zu bemerken ist hier, daß ein fault nicht notwendig einen internen Fehler und ein Fehlverhalten nach sich ziehen muß.

Das folgende Beispiel soll die Ausführungen illustrieren und ist ebenfalls [45] entnommen: Eine 1-Bit Speicherzelle gibt aufgrund einer defekten Transistorschaltung stets den Wert 0 zurück. Die Fehlerursache ist der defekte Transistor. Dieser fault führt erst dann zu einem internen Fehler (error), wenn der Speicherzelle 1 zugewiesen wird. Ein Fehlverhalten (failure) tritt auf, wenn die Speicherzelle gelesen wird.

Da ein verteiltes System insbesondere aus autonomen Knoten besteht, muß beispielsweise der Ausfall eines Knotens (fault) nicht notwendig zu einem internen Fehler (error) führen, der ein Fehlverhalten (failure) des Systems verursacht.

Aus Sicht der Systemnutzung sind außerdem Fehlermodelle, die spezifizieren, welche Fehler beim Betrieb eines Systems auftreten können, notwendig. Das folgende Modell wurde von Cristian [46] entwickelt und hat sich inzwischen als Quasistandard etabliert. Das Modell enthält vier Fehlerklassen:

- Ausfall (crash failure),
- Unterlassung (omission failure),
- Zeitfehler (timing/performance failure) und

- beliebiger Fehler (arbitrary/byzantine [8] failure).

Die Fehlerklassen unterliegen der strikten Klassenhirachie:

Ausfall ⊂ Unterlassung ⊂ Zeitfehler ⊂ beliebiger Fehler.

Diese Klassenhirachie ergibt sich daraus, daß nach Definition ein Ausfall ein beliebiger Fehler ist, jede Unterlassung ein Spezialfall eines Zeitfehlers ist (unendliche Verzögerung) und ein Ausfall ein Spezialfall einer Unterlassung (permanentes Ausbleiben) ist.

Ein Merkmal eines verteilten Systems ist es, daß die Knoten nur über ein Netzwerk miteinander kommunizieren. Daraus folgt die folgende Abbildung der oben genannten Fehlerklassen auf ein verteiltes System: Netzzusammenbruch läßt sich der Klasse Ausfall zuordnen. Verlust von Paketen des zugrundeliegenden Netzwerks gilt als Unterlassung (omission). Zeitliche Verzögerung von Paketen kann einem Zeitfehler zugeordnet werden (z. B. kann ein timeout-Intervall auslaufen). Semantische Veränderung schließlich wird durch beliebiger Fehler identifiziert.

Abschließend sei eine Definition für ein *korrektes System* gegeben. Diese definition geht auf P. Jalote [48] zurück:

Ein System wird als fehlertolerant (fault tolerant) bezeichnet, wenn es potentielle Fehlerursachen (faults) maskiert, so daß weder ein fehlerhafter Zustand (error) auftritt, noch ein Fehlverhalten (failure) seitens der Umgebung beobachtet werden kann.

Fehlerfreiheit - korrektes Systemverhalten

Im Allgemeinen läßt sich ein korrektes Systemverhalten auf zwei Arten realisieren:

- Durch Fehlervermeidung (fault avoidance/fault prevention) oder

- durch Fehlertoleranz (fault tolerance).

Fehlervermeidung findet in der Designphase statt und nutzt Verifikation, Debugging und Tests. Tritt allerdings während der Laufzeit des Systems ein Hardwarefehler auf, sind die Mechanismen der Fehlervermeidung machtlos. Der Ansatz der Fehlertoleranz versucht während der Laufzeit auftretene Fehler (error und fault) zu maskieren und so ein Fehlverhalten (failure) zu vermeiden. Die Mechanismen der Fehlertoleranz gehen von einem logisch fehlerfreien System aus, d.h. das System ist korrekt entworfen und hat keine logischen Fehler.

[8]Dieser Begriff geht auf Lamport [47] zurück, der die Analogie kooperierender byzantinischer Generäle verwendete.

Die vorliegende Arbeit beschäftigt sich ausschließlich mit der Realisierung von Fehlertoleranz.

7.1 Fehlertoleranz

Im allgemeine ist Fehlertoleranz nur durch Redundanzen möglich. Redundanzen sind zusätzliche Informationen, die für das System nur im Fehlerfall relevant sind. P. Jalote definiert Redundanzen wie folgt [48]:

Redundanzen sind definiert als solche Teile eines Systems, welche nicht für die Korrektheit eines Systems notwendig sind, solange das System frei von Fehlern ist.

In der Literatur ist allerdings die Einteilungen in *statische* und *dynamische* redundante Mittel üblich. Statisch redundante Mittel übernehmen schon im fehlerfreien Fall Funktionalität des Systems. Dynamische redundante Mittel sind passiv, bis ein Fehler auftritt. Systeme, die mit statisch redundanten Mitteln arbeiten, sind beispielsweise Votingsysteme. Systeme mit Primär - und Ersatzkomponenten sind typische Vertreter für Systeme mit dynamisch redundanten Mitteln.

Belli et al. führen eine zusätzliche Einteilung von redundanten Mitteln ein [49]. Diese Einteilung umfaßt die Begriffe strukturelle, funktionelle und zeitliche Redundanzen. Zu den strukturelle Redundanzen (auch Replikationen genannt) gehören Systemkomponenten, die zusätzlich in das System integriert wurden, um im Fehlerfall defekte Komponenten zu ersetzen bzw. den Defekt zu maskieren. Funktionelle Erweiterungen wie z. B. Paritätsbits und Kontrollsummen werden unter funktionelle Redundanzen geführt. Zeitliche Redundanzen bezeichnet zusätzliche Rechenzeit, die genutzt wird, um Fehler zu entdecken und zu beheben.

Die Bewertung von Fehlertoleranz ist durch ihre Ziele und durch Kenngrößen möglich. Die formalen Ziele der Fehlertoleranz sind *Zuverlässigkeit* und *Verfügbarkeit*. Zuverlässigkeit ist die Zeitspanne, in der das System fehlerfrei arbeitet. Verfügbarkeit beschreibt die Zeit, in der das System verfügbar ist unter Einbeziehung möglicher Reperaturmaßnahmen. Eine ausführliche formale Beschreibung dieser Größen gibt die Arbeit von Trivedi [50].

Weitere Kenngrößen nach Hübsch [45] sind der Grad der Redundanz und der Grad der Zuverlässigkeit. Diese Kenngrößen sind im Zusammenhang mit den oben eingeführten strukturell redundanten Mitteln von Interesse. Der Grad der Redundanz beschreibt die Anzahl redundanter Komponenten. Der Grad der Zuverlässigkeit beschreibt die Maximalanzahl der tolerierten Fehler in einem bestimmten Zeitintervall.

7.2 Voraussetzungen

Beschäftigt man sich mit Möglichkeiten, Fehlertoleranz zu realisieren, sind idealisierte Annahmen notwendig. Diese Annahmen bilden eine Menge von Voraussetzungen, auf denen sich die Methoden der Fehlertoleranz stützen.

Zuverlässige Kommunikation

Cristian definiert eine zuverlässige Kommunikation in einem verteilten System wie folgt [46]:

Aus Sicht der Anwendung besteht dieser Dienst im Transport von Nachrichten von einem Sender zu einem oder mehreren Empfängern, wobei typischerweise davon ausgegangen wird, daß eine vom Sender initiierte Nachricht innerhalb einer festgelegten Zeitspanne unverfälscht bei dem Empfänger (bzw. bei der Empfängergruppe) eintrifft.[9]

Zuverlässige Kommunikation wird durch Hardwarekomponenten und durch ein geeignetes Protokoll realisiert. Ein Möglichkeit der Realisierung befindet sich im Abschnitt *Fehlertolerante Kommunikation*.

In dieser Architektur lassen sich alle Fehlerarten des im Abschnitt *Fehler und Fehlermodelle* vorgestellten Fehlermodells ausmachen. Ausfall liegt vor beim Verlust einer Nachrichtenverbindung, Unterlassung tritt auf, wenn ein Paket verloren geht, ein Zeitfehler liegt vor, wenn eine Nachricht verzögert wird und ein beliebiger Fehler subsumiert semantische Veränderungen.

Stabiler Speicher

Das persistente Speichern von Information, so daß diese Informationen nach einem Rechnerausfall wieder verfügbar sind, ist eine weitere Grundannahme. Viele fehlertolerante Systeme nutzen den stabilen Speicher zur Rekonstruktion eines Systemzustands.

In einem verteilten System sind diese Informationen im allgemeinen nicht persistent sondern durch Replikation auf anderen Knoten verfügbar. Die Architektur eines verteilten Systems, im Besonderen das Vorhandensein von autonomen Knoten, macht dieses Verfahren plausibel.

Prozessoren mit angenehmen Fehlverhalten

Eine weitere Voraussetzung betrifft den Ausfall eines Knotens. Man geht davon aus, daß, wenn ein Knoten ausfällt, keine weiteren systemrelevanten Effekte auftreten. Knoten mit diesem Verhalten werden als *Prozessoren mit angenehmen*

[9] Diese Übersetzung ist ebenfalls der Arbeit von V. Hübsch entnommen [45].

Fehlverhalten (fail stop processors) bezeichnet. Eine genaue Charakterisierung wurde von Schlichting und Schneider entwickelt ([51] und [52]). Im besonderen beinhaltet diese Charakterisierung, daß der Ausfall eines Knotens von jedem anderen Knoten festgestellt werden kann.

7.3 Basismechanismen

Ein verteiltes System kann als Gruppe von unzuverlässigen Komponenten charakterisiert werden. Die Komponenten der Gruppe bilden eine Einheit mit dem Ziel, einen Dienst zu erbringen. Um die gewünschte Fehlertoleranz zu erreichen und damit einen Dienst zu garantieren, müssen die Komponenten der Gruppe miteinander kooperieren. Die Kooperation der Gruppenkomponenten basiert auf einer *zuverlässigen Kommunikation* und der *gegenseitigen Überwachung*.

Zuverlässige Kommunikation

Das Konzept der zuverlässigen Kommunikation sieht ein Protokoll vor, welches Nachrichten zwischen den Gruppenmitgliedern (Knoten) versendet. Die Nachrichten erreichen immer ihren Empfänger und sind unverfälscht. Duplizierungen werden ausgeschlossen und die Laufzeit ist beschränkt. Zuverlässige Kommunikation ist existenziell für ein fehlertolerantes verteiltes System. Eine Vielzahl von Protokollen wurde bisher entwickelt und eingesetzt. Der Abschnitt *Fehlertolerante Kommunikation* dieser Arbeit beschäftigt sich mit einer Realisierungsmöglichkeit.

Gegenseitige Überwachung

Um Maßnahmen zu ergreifen, die den Ausfall eines Knotens maskieren, ist es als erstes notwendig, den Ausfall des Knotens festzustellen. Die Komponenten der Gruppe (Knoten eines verteilten Systems) müssen sich schon zur fehlerfreien Laufzeit gegenseitig überwachen. Ausgespart wird das freiwillige Ausscheiden aus der Gruppe, da dieses vor dem Verlassen angekündigt werden kann.

Von Zeit zu Zeit muß ein Mechanismus greifen, der beispielsweise eine "Bist Du am Leben." Nachricht versendet und ein "Ich bin am Leben." als Antwort erwartet (siehe [53] und [1]). Solche Mechanismen werden allgemein als *callbacks* bezeichnet.

In dem fehlertoleranten Algorithmus zum gegenseitigen Ausschluß (vorgestellt in Kapitel *DVAFT - ein fehlertoleranter Mutexalgorithmus*) werden solche *callbacks* in den normalen Ablauf integriert. Dies hat den Vorteil, daß keine zusätzlichen Nachrichten benötigt werden und daß dem Fakt, daß ein *fault* nicht notwendig einen *error* und *failure* nach sich zieht, Tribut gezollt wird.

8 Fehlertoleranz und gegenseitiger Ausschluß

Die Architektur verteilter Systeme birgt in sich Quellen für Fehler. Knoten können ausfallen, Nachrichtenverbindungen können dauerhaft unterbrochen werden und Nachrichten können verloren gehen oder semantisch verändert werden. Die Folge dieser Fehler kann die Verletzung der Korrektheit des Algorithmus zum gegenseitigen Ausschluß sein. Das verteilte System ist somit in einem inkonsistenten Zustand bzw. in einem Zustand der Verklemmung.

8.1 Fehler in verteilten Systemen

Im allgemeinen werden zwei Arten von Fehlern in verteilten Systemen unterschieden: Knotenfehler und Netzwerkfehler. Beide Arten sind relevant für tokenbasierende Algorithmen zum verteilten gegenseitigen Ausschluß. Je nach dem Zustand des Systems können sie Verklemmungen und Verhungern verursachen und die Verletzung des gegenseitigen Ausschlusses zur Folge haben.

Knotenfehler

Fällt ein Knoten des verteilten Systems aus, spricht man von einem Knotenfehler. Betrachtet wird hier nur der Fall, daß der Knoten plötzlich nicht mehr verfügbar ist, den Kontrollfluß abbricht und weder Nachrichten empfangen noch senden kann. Das Betriebssystem sendet in diesem Fall auch kein Signal an laufende Prozesse, wie z.B. *SIGHUP* (für hangup) oder *SIGPWR* (für power off).

Ein Knotenfehler ist z. B. beim plötzlichen Ausschalten eines Computers gegeben. Wenn ein Knoten des Systems durch eine physische Störung der Netzwerkverbindung nicht mehr erreichbar ist, spricht man ebenfalls von einem Knotenfehler. Der Knoten kann weder Nachrichten empfangen noch senden. Der Ausfall ist nicht temporär (transient), d.h. nicht zeitlich begrenzt. Die Wirkung von Knotenfehlern auf das verteilte System können

- Verhungern einzelner Knoten des Systems und
- Verklemmung des Systems

sein.

Ist zum Beispiel in dem Algorithmus SVARW die Kette der *Dir*-Zeiger unterbrochen, erreichen Tokenanfragen nicht ihr Ziel, der Knoten, der eine Anfrage generiert hat, verhungert. Ging durch das Ausfallen eines Knotens das Token verloren, ist der Fall der Verklemmung gegeben. Kein Knoten des Systems bekommt die Möglichkeit, seine kritische Sektion zu betreten, das System befindet sich im dauerhaften Zustand der Verklemmung. Der Ausfall eines Knotens kann das System in Partitionen zerlegen, d.h. in logisch oder physisch unabhängig Teile. Diese Teile haben keine Möglichkeit, über die Grenzen der Partition hinaus zu kommunizieren. Der Teil, in dem sich das Token nicht befindet, verhungert.

Netzwerkfehler

Ein verteiltes System nutzt für die Kommunikation ausschließlich das zugrundeliegende Netzwerk. Das Netzwerk beeinflußt die Performance einer Anwendung erheblich. Es ist meist der Flaschenhals eines verteilten Systems. Weiterhin ist das Senden und Empfangen von Nachrichten übers Netzwerk eine Hauptquelle für Fehler: Nachrichten können verlorengehen oder semantisch Verändert werden, die Verbindung kann physisch oder logisch dauerhaft getrennt werden. Die Laufzeit von Nachrichten ist nicht vorhersagbar, das Netzwerk steht selten dem verteilten System exklusiv zur Verfügung. Netzlasten sind ebenfalls nicht vorhersagbar.

Die Korrektheit eines Protokolls zum gegenseitigen Ausschluß und damit die Konsistenz der Daten eines verteilten Systems ist von einem fehlerfreien Nachrichtenaustausch abhängig. Tokenbasierende Algorithmen zum gegenseitigen Ausschluß arbeiten nach dem Prinzip, daß Anfragen und daß das Token von einem Knoten über mehrere Zwischenstationen ihr Ziel erreichen. Auf den Zwischenstationen werden Operationen ausgeführt, die die Korrektheit des Algorithmus sichern.

Der Verlust von Nachrichten kann Verklemmungen und das Verhungern bewirken. Geht z. B. eine Tokennachricht verloren, erhält kein Knoten jemals die Möglichkeit seine kritische Sektion zu betreten - das System befindet sich im Zustand der Verklemmung. Geht eine Tokenanfrage verloren, verhungert der Knoten, der die Anfrage generiert hat. Semantische Veränderung können die Verletzung des gegenseitigen Ausschlusses nach sich ziehen. Es können sich mehrere Token im System befinden. Duplizierung von Nachrichten kann ebenfalls die Verletzung des gegenseitigen Ausschlusses zur Folge haben bzw. zu Inkonsistenzen führen, die die Korrektheit des Algorithmus verletzen. Die dauerhafte Unterbrechung einer Verbindung kann das System in Partitionen zerlegen. Der Teil der Partitionen, in dem sich nicht das Token befindet, verhungert.

Abbildung 38 gibt einen Überblick über Fehler und deren Wirkung auf einen Algorithmus zum gegenseitigen Ausschluß in verteilten Systemen.

Um Fehlertoleranz in Algorithmen zum verteilten gegenseitigen Ausschluß zu realisieren, ist an erster Stelle eine fehlertolerante Kommunikation notwendig. Das Netzwerk wird bei einer Fehlerbehebung benötigt. Zu beachten ist dabei, daß, wie oben erwähnt, der Nachrichtenverkehr in Hinblick auf die Performance weitgehend minimiert wird, da das Netzwerk meist der Flaschenhals eines verteilten Systems ist.

8.2 Fehlertolerante Kommunikation

Als Alternative zu einem zuverlässigen Netzwerk soll in dem nun folgenden Kapitel die Realisierung einer fehlertoleranten Kommunikation diskutiert werden.

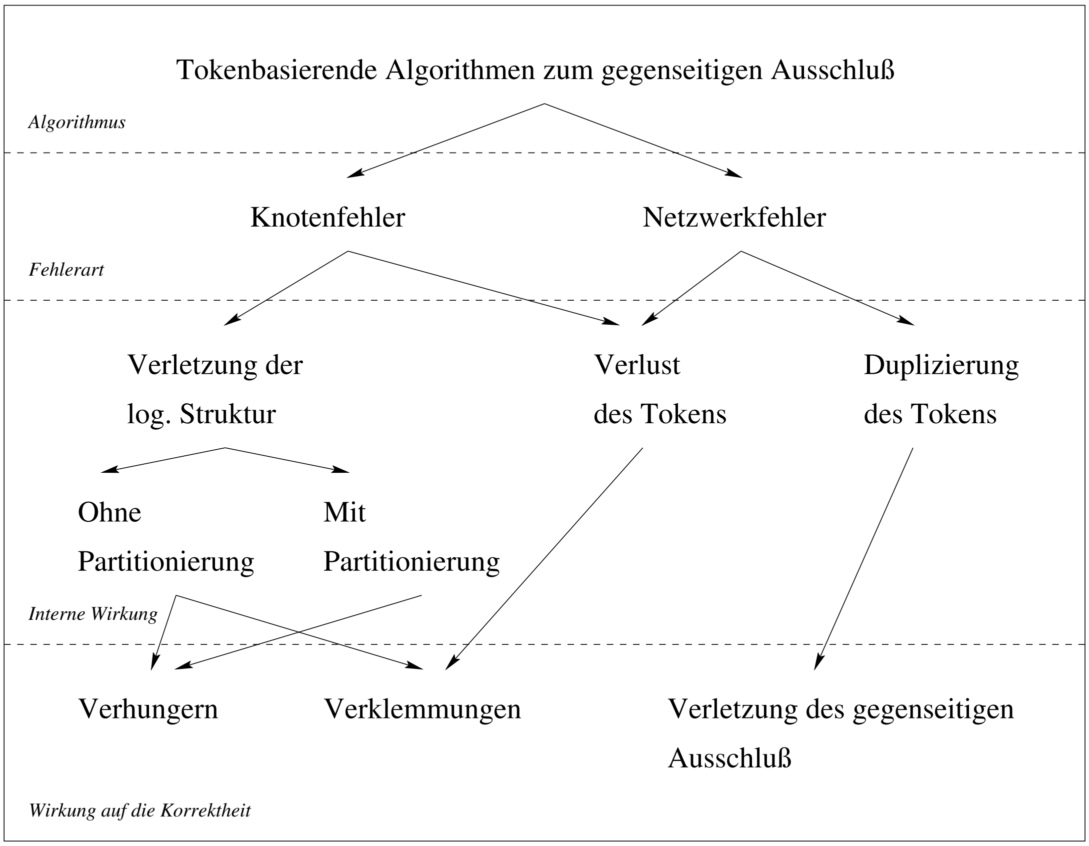


Abbildung 38: Schematische Darstellung der Fehlerarten und ihrer Wirkung auf einen Algorithmus zum gegenseitigen Ausschluß.

8.2.1 Allgemeines

Zwei Aspekte spielen bei der Fehlertoleranz im Hinblick auf Algorithmen zum gegenseitigen Ausschluß eine Rolle:

- Die Fehlererkennung und
- die Fehlerbehebung.

Fehlertolerante Kommunikation ist für beide Aspekte notwendig: Sie behebt Fehler, die aus dem Verlorengehen von Nachrichten oder deren semantischer Veränderung resultieren, durch erneutes Senden der Nachricht bzw. durch ignorieren der Nachricht (kein Acknowledgement). Sollte die Nachricht wiederum verlorengehen, kann sie einen Knotenfehler bzw. eine dauerhafte Netzstörung erkennen und somit eine Fehlerbehebung einleiten. Zu realisieren ist ein Protokoll, das

- für jede Nachricht eine Quittung verlangt und
- Nachrichten eindeutig identifiziert.

Eindeutige Identifikation bedeutet hier: Eindeutig für den Sender und den Empfänger der Nachricht. Beim Betrachten der Reihenfolge von Nachrichten ist die lokale Reihenfolge und, beschränkt, die Reihenfolge im gesamten System interessant.

8.2.2 Realisierung

Die fehlertolerante Kommunikation besteht aus den Primitiven

- *send_msg()*,
- *send_ack()* und
- *resend_msg()*,

sowie aus den Servern

- *comm_server()* und
- *ack_server()*.

Die Funktion *send_msg()* veranlaßt das Senden einer Nachricht und signalisiert dem *ack_server()*, daß eine Quittung erwartet wird. Der *comm_server()* muß in einem eigenen Thread gestartet werden und behandelt die an einen Knoten gesendeten Nachrichten. Im speziellen sendet er Quittungen für empfangene Nachrichten. Der *ack_server()* ist für die Überwachung der Quittungen verantwortlich, d. h. für das erneute Senden der Nachricht nach dem Verstreichen eines bestimmten Zeitintervalls (Timeout), falls die Nachricht nicht quittiert wurde. Sollte die Nachricht wiederum unquittiert bleiben, leitet der *ack_server()* eine Fehlerbehandlung ein. Der *ack_server()* muß ebenfalls in einem eigenen Thread gestartet werden. Die Funktion *resend_msg()* sendet eine Nachricht, die nach einem bestimmten Timeout nicht quittiert wurde, erneut. Sie wird vom *ack_server()* gerufen. Wie oft das Senden einer Nachricht wiederholt wird, bestimmt ein Parameter, der durch eine Heuristik unter Einbeziehung der Timeout-Intervalle festgelegt wird. Dieser Schritt gehört zum Feintuning des Systems und wird stark durch die Hardware beeinflußt. Die Funktionen *send_msg()*,*comm_server()* und *ack_server()* haben Zugriff auf eine Struktur, in der Informationen über gesendete Nachrichten gespeichert werden. Die Funktion *send_ack()* wird im *comm_server()* gerufen und setzt die Quittung für eine empfangene Nachricht ab, ohne eine Quittung zu erwarten. Die Nachrichten werden durch Versionsnummern eindeutig identifiziert. Erhält ein Knoten eine Nachricht vom gleichen Absender mit der gleichen Versionsnummer, wird diese verworfen. Damit sind Duplizierungen ausgeschlossen.

8.2.3 Acknowledgment-Server

Die gesendeten Nachrichten werden sortiert nach ihrem Timeout in einer verketteten Liste gespeichert. Der Acknowledgement-Server besteht aus einem Thread, der auf dem kleinsten Timeout-Wert der Liste bedingt wartet, beziehungsweise, wenn die Liste leer ist, auf einer zweiten Bedingungsvariable blockiert, die immer beim Senden einer Nachricht signalisiert wird. Wird das Timeout ausgelöst, kontrolliert der Thread, ob für diese Nachricht schon eine Quittung eingegangen ist. Ist dies der Fall, löscht er die Nachricht und wartet, sofern das Timeout der nächsten Nachricht noch nicht erreicht ist, auf das Timeout der nächsten Nachricht. Blieb die Nachricht unquittiert, sendet er sie erneut, versieht sie mit einem neuen Timeout und sortiert sie in die Liste ein. Bei jedem erneuten Senden einer Nachricht wird ein Zähler inkrementiert. Erreicht dieser Zähler eine bestimmte Grenze, leitet der Acknowledgement-Server eine Fehlerbehandlung ein.

8.2.4 Die Testumgebung unter Benutzung der fehlertoleranten Kommunikation

Um den fehlertoleranten Algorithmus zu testen, ist es nötig, eine Testumgebung zu schaffen, die auf eine fehlertolerante Kommunikation aufbaut.

Die Struktur, der in Kapitel 5 vorgestellten Testumgebung, erweitert durch die fehlertoleranten Kommunikation, zeigt Abbildung 39. Die Rechtecke sind als Threads implementiert. Die Rechtecke mit abgerundeten Ecken sind Datenstrukturen, auf denen die Threads arbeiten. Durchgehend gezeichnete Pfeile bezeichnen Lese- bzw. Schreiboperationen der Threads auf den Datenstrukturen. Gestrichelte Pfeile stellen das Signalisieren einer Bedingungsvariable dar. Der Kommunikationsserver und der Acknowledgement-Server haben eine eigene Portadresse. Der Mutexserver führt Operationen der Methode zum gegenseitigen Ausschluß aus, d.h. durch ihn werden die zum gegenseitigen Ausschluß gehörenden Nachrichten verarbeitet. Der Worker ruft die Funktionen *mutex_lock()* und *mutex_unlock()*. Der Kommunikationsserver empfängt Nachrichten und beantwortet diese. Er läuft mit höchster Priorität. Der Acknowledgement-Server läuft mit zweithöchster Priorität.

Die Timeout-Intervalle sind mit Hilfe von *pthread_cond_timedwait()* und *clock_gettime()* implementiert. *pthread_cond_timedwait()* und *clock_gettime()* sind Funktionen des Posix-Standards [18] und [21].

Das im Abschnitt 5 vorgestellte Interface behält seine Gültigkeit, die für die fehlertolerante Kommunikation notwendigen Komponenten werden implizit gestartet.

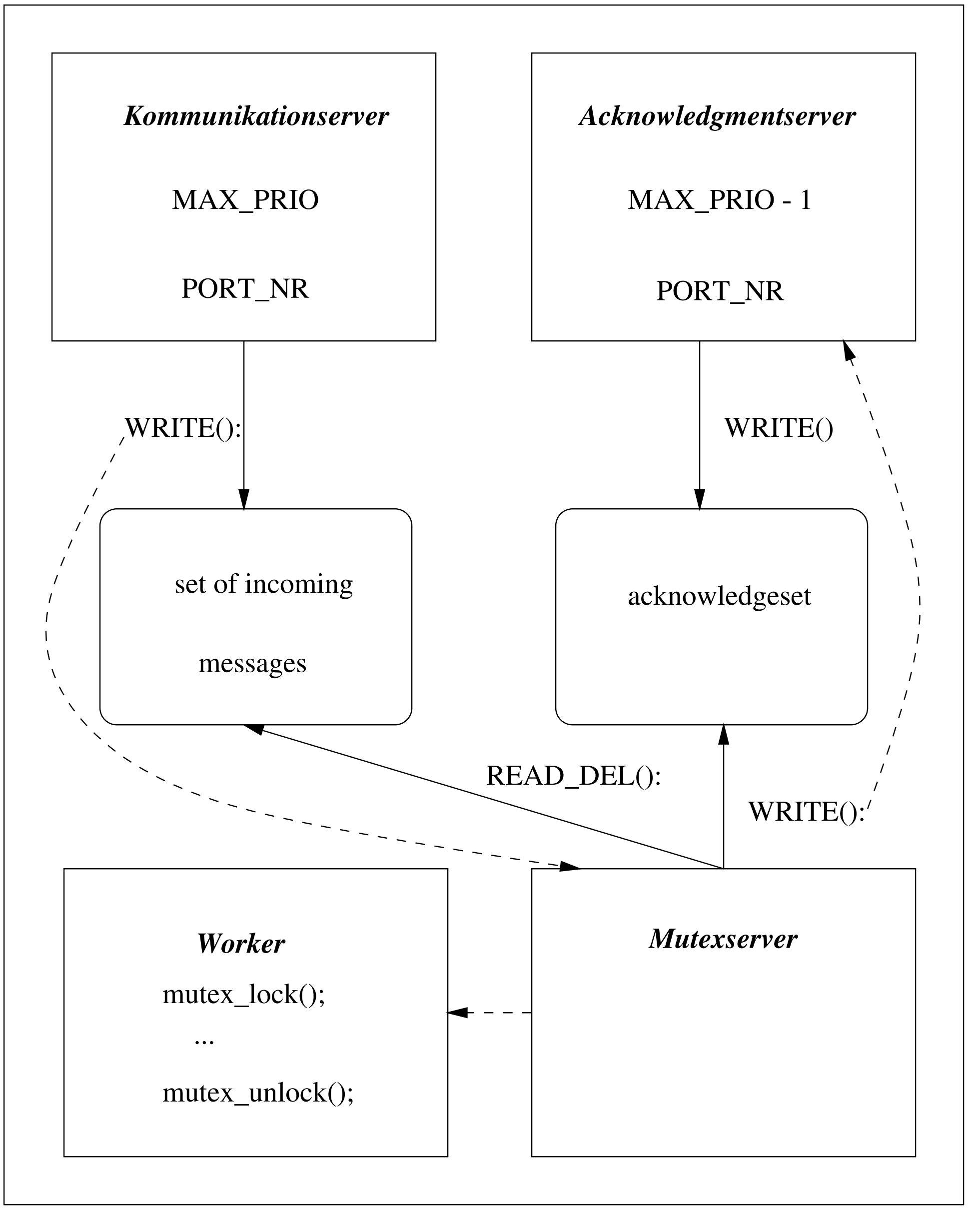


Abbildung 39: Struktur der Testumgebung unter Benutzung der fehlertoleranten Kommunikation.

8.3 DVAFT - Ein fehlertoleranter Mutexalgorithmus

Der Algorithmus basiert auf dem DVA von Chang et al. bzw. Naimi und Trehel. Der Algorithmus ist ein tokenbasierender Algorithmus zum gegenseitigen Ausschluß und realisiert keine Lese- und Schreibsperren. Behandelt werden Knotenfehler, wie in Abschnitt 8.1 spezifiziert. Der Algorithmus toleriert den Ausfall eines Knotens zu einem Zeitpunkt, d.h. der Ausfall eines weiteren Knotens während der fehlerbehandlung wird nicht behandelt.

Erweitert wurde die Datenstruktur des Algorithmus durch einen logisch doppelt verketteten Ring der Knoten.

8.3.1 Der Basisalgorithmus

Der Basisalgorithmus ist tokenbasierend und arbeitet mit Pfadreduzierung, d.h. die Wege der Anfragen sind nicht konstant und werden weitestgehend reduziert. Jeder Knoten hat durch die Variable *Dir* die Richtung, in der Anfragen versendet werden, vorgegeben. Die Variable *Self* identifiziert den eigenen Knoten und *Next* bestimmt den Weg des Tokens durch das System. Der Algorithmus ist in den Abbildungen 40 ff. im Pseudocode skizziert. Die durch Fettdruck gekennzeichneten Teile gehören zur fehlertoleranten Erweiterung. Initial weisen alle Zeiger *Dir* auf den Tokenbesitzer (siehe Abbildung 40, Zeile 8). Lediglich der Zeiger *Dir* des Tokenbesitzer ist auf *NULL* gesetzt. Man spricht von der initialen Sternform der Zeiger *Dir*.

Will ein Knoten seine kritische Sektion betreten, testet er zunächst, ob er der Tokenbesitzer ist. Ist er im Besitz des Tokens, betritt er seine kritische Sektion, anderenfalls sendet er eine Anfrage in Richtung *Dir* (Abbildung 41, Zeile 21 bis Zeile 28).

Ein Knoten verläßt mit dem Ruf *mutex_unlock()* seine kritische Sektion. Mit dem Verlassen testet er den Zeiger *Next*, der angibt, ob ein weiterer Knoten eine Tokenanfrage generiert hat. Ist *Next* nicht *NULL*, sendet er das Token in Richtung *Next* (Abbildung 42, Zeile 37 ff.). Ist *Next* gleich *NULL*, behält der Knoten das Token ohne es zu nutzen.

Erhält ein Knoten die Nachricht *REQUEST*, sendet er sie, falls sein *Dir* Zeiger ungleich *NULL* ist, in Richtung *Dir* weiter (Abbildung 43 Zeile 43). Ist *Dir* gleich *NULL*, hat der Empfänger entweder das Token oder wartet auf das Token. Im ersten Fall sendet er das Token sofort an den Knoten, der die Anfrage generiert hat, sofern der Empfänger der Anfrage nicht in seiner kritischen Sektion ist. (Abbildung 43, Zeile 47 und Zeile 48.) Wartet er auf das Token bzw. nutzt er das Token, weist er seinem *Next* Zeiger den Knoten zu, welcher die Anfrage generiert hat (Abbildung 43, Zeile 50 bzw. Zeile 53.).

In jedem Fall setzt er *Dir* auf den anfragenden Knoten (siehe Abbildung 43,

```
mutex_init(root,self,pred,succ) begin

   if (root == Self)
      Token = TRUE;
      Dir = NULL;
   else
      Token = FALSE;
      Dir = root;
   fi
   cs = FALSE;
   Next = NULL;
   requested = FALSE;
   Recover_Node = NULL;
   Pred = pred;
   Succ = succ;
   cnt_status_container = 0;
   local_status_container = NULL;
end mutex_init.
```

Abbildung 40: Initialisierung der Struktur.

Zeile 58).

Beim Empfang des Tokens wird die lokale Variable *Token* auf TRUE gesetzt (siehe Abbildung 44 Zeile 61).

8.3.2 Fehlererkennung

Ein aufgetretener Fehler muß zunächst erkannt werden. Kommunikationsfehler werden, bis auf den dauerhaften Ausfall einer physischen Verbindung, von den Komponenten der fehlertoleranten Kommunikationseinheit erkannt und behoben. Knotenausfälle und der dauerhafte Ausfall einer Nachrichtenverbindung sind äquivalent und werden unter Knotenfehler subsumiert. Die Knoten, die durch den Verlust der Nachrichtenverbindung unerreichbar sind, können als ausgefallen betrachtet werden. Der Fall, in dem einer dieser Knoten in das System zurückkehrt, wird unten betrachtet und hier ausgespart. Die Fehlererkennung eines so definierten Knotenfehlers kann auf zwei Arten erfolgen:

1. Durch die fehlertolerante Kommunikation bzw. Nutzung eines zuverlässigen Netzwerks: Das dem verteilten System zugrundeliegende Netzwerk kann, falls es sich um kein zuverlässiges Netzwerk handelt, durch die Benutzung der fehlertoleranten Kommunikation zuverlässig gemacht werden. Werden

```
mutex_lock() begin
   if (Token == FALSE)
      send_request(self) to Dir;
      Dir = NULL;
      Requested = TRUE;
      await(Token == TRUE) or
            (TIME_OUT);
      if (Token == FALSE)
         mutex_error();
         await((Token == TRUE) and
               (recover_node == NULL))
      fi
   fi
   Requested = FALSE;
   cs = TRUE;
end mutex_lock.
```

Abbildung 41: Die Funktion *mutex_lock()*.

```
mutex_unlock() begin
   cs = FALSE;
   if (next != NULL)
      send_token() to Next;
      Token = FALSE;
      Next = NULL;
   fi
end mutex_unlock.
```

Abbildung 42: Die Funktion *mutex_unlock()*.

```
Beim Empafang von request(requester) begin
   if (Dir != NULL)
      send_request(requester) to Dir
   else
      if (Token == TRUE)
         if (cs == FALSE)
            send_token() to requester;
            Token = FALSE;
         else
            Next = requester;
         fi
      else
         Next = requester;
      fi
   fi
   Dir = requester;
end.
```

Abbildung 43: Der Nachrichtenhandler für *REQUEST*.

```
Beim Empafang von token begin
   Token = TRUE;
end.
```

Abbildung 44: Der Nachrichtenhandler für *TOKEN*.

```
type status begin
  node self;
  node next;
  node dir;                  type status_container begin
  node pred;                   node   initiator;
  node succ;                   node   crashed_node;
  bool Token;                  status state[];
end.                         end.
```

Abbildung 45: Datenstruktur für Informationen zur Fehlerbehebung.

Nachrichten an einen Knoten nicht quittiert, versucht die fehlertolerante Kommunikationseinheit diese vermeintlichen Netzwerkfehler durch erneutes Senden zu korrigieren. Ist das nicht möglich - das erneute Senden bleibt ebenfalls unquittiert - erkennt die fehlertolerante Kommunikationseinheit einen wie oben definierten Knotenfehler und leitet eine Fehlerbehandlung ein.

2. Durch den Ablauf eines Zeitintervalls (Timeouts): Ein Knoten, der eine Tokenanfrage gesendet hat, erwartet, daß er das Token nach einer bestimmten Zeit erhält. Ist dies nicht der Fall, geht er davon aus, daß ein Fehler im System aufgetreten ist und leitet eine Fehlerbehandlung ein.

8.3.3 Einleitung der Fehlerbehebung

Ein Knoten, der meint, einen Fehler erkannt zu haben, sendet die Nachricht *ERROR_DETECTED* in beide Richtungen des logisch doppelt verketteten Rings, d.h. in Richtung *pred* und *succ* (siehe Abbildung 46, Zeile 65 und 66). Sichergestellt ist damit, daß bei einem Knotenfehler jeder Knoten, der sich im System befindet und auf dem kein Knotenfehler aufgetreten ist, die Nachricht *ERROR_DETECTED* erhält. Diese Nachricht führt einen Container mit sich, in dem die Knoten, die die Nachricht erhalten und weitersenden, ihren Status eintragen. Die Datenstrukturen zeigt Abbildung 45. Der *status_container* speichert in *initiator* den Knoten, der den Fehler erkannt hat, das Feld *state* sammelt die Knotenstati, und in *crashed_node* wird der vermeintlich fehlerhafte Knoten gespeichert. Die *status* Struktur enthält alle Informationen, die benötigt werden, um den momentanen Zustand des gesamten Systems auf einem Knoten lokal zu rekonstruieren. Abbildung 46, Zeile 74 behandelt den Spezialfall, wenn die Größe des Systems gleich 2 ist und ein Knotenfehler auftritt: der Knoten, der den Fehler feststellt, ist allein im System, kann *Token* auf *TRUE* setzen und die Fehlerbehandlung beenden. In Abbildung 46, Zeile 70 und Zeile 73, wird der Fall, daß der ausgefallene Knoten ein Nachbar des Knotens ist, der den Fehler erkannt hat, behandelt.

Haben mehrere Knoten den Fehler erkannt, muß sichergestellt sein, daß nur ein Knoten die Fehlerbehebung durchführt. Dieser Knoten wird im folgenden mit *recover_node* bezeichnet. Notwendig ist dies aus Effizienzgründen und um zu

```
mutex_error begin
   if (recover_node != NULL)
      return;
   recover_node = self;
   cs.initiator = self;
   send_error_detect(cs) to pred;
   await (ACK) or (TIMEOUT)
   if (ACK == FALSE) cnt_sc++
   send_error_detect(cs) to succ;
   await (ACK) or (TIMEOUT)
   if (ACK == FALSE) cnt_sc++
   if (cnt_sc == 2)
      Token = TRUE
      recover_node = NULL
      Dir = NULL
      Next = NULL
   fi
end mutex_error.
```

Abbildung 46: Die Funktion *mutex_error()*.

vermeiden, daß bei einem Tokenverlust das Token mehrfach regeneriert wird.

Der Fall, daß mehrere Knoten ein- und denselben Fehler erkennen, tritt beispielsweise dann auf, wenn die *Dir* Zeiger mehrerer Knoten auf den fehlerhaften Knoten verweisen.

Die Wahl eines Knotens als *recover_node* erfolgt an Hand der eindeutigen IDs der Knoten: Der Knoten mit der kleinsten ID, der Menge der Knoten, welche den Fehler erkannt haben, wird als *recover_node* bestimmt(siehe Abbildung 47, Zeile 72).

Da die Nachricht *ERROR_DETECTED* von einem Knoten maximal einmal gesendet wird, kann sich dadurch, daß mehrere Knoten den Fehler erkannt haben, eine Kette bilden. Das Ende dieser Kette ist der eigentliche *recover_node*. Mit dem Empfangen und dem Weitersenden der Nachricht *STATUS_CONTAINER* und dem Mischen des erhaltenen Statuskontainers mit dem lokalen Statuscontainer baut sich diese Kette sukzessive ab (Abbildung 48, Zeile 4 und 5).

Der Mechanismus des Weitersendens der Nachricht *ERROR_DETECTED* endet bei den Knoten, deren Vorgänger bzw. Nachfolger der ausgefallene Knoten ist, durch den Ablauf eines Timeouts (Abbildung 47, Zeile 20).

Diese Knoten senden daraufhin die Nachricht *STATUS_CONTAINER* an den *recover_node*.

Erreicht den *recover_node* die zweite *STATUS_CONTAINER* Nachricht, hat er alle Informationen erhalten die notwendig sind, das System in einen konsistenten

Zustand zurückzuführen (Abbildung 48, Zeile 8 und 9).

Nachdem ein Knoten seinen Status verschickt hat, sind alle seine Aktionen eingefroren. Die Knoten blockieren auf *await(recover_node == NULL)*.

8.3.4 "Falscher Alarm"

Der Fall "Falscher Alarm" ist gegeben, wenn ein Knoten einen Fehler erkannt hat, in dem System jedoch kein Fehler aufgetreten ist. Dies ist z. B. der Fall, wenn durch nicht vorhersagbare Lasten ein Zeitintervall abgelaufen ist.

Im DVAFT wird ein "Falscher Alarm" wie folgt festgestellt: Der Knoten, der den vermeintlichen Fehler festgestellt hat, sendet, wie oben beschrieben, die Nachricht *ERROR_DETECTED* in beide Richtungen des logischen Rings. Ist ein Fehler aufgetreten, ist der Ring nicht geschlossen. Der Mechanismus des Weitersendens endet bei den Knoten, dessen Nachfolger bzw. Vorgänger der ausgefallene Knoten ist.

Im Falle des "Falschen Alarm" erreicht die Nachricht nach N Schritten den Initiatorknoten der Nachricht, der so feststellt, daß es sich um einen "Falschen Alarm" handelt (siehe Abbildung 47, Zeile 82). Alle anderen Knoten erhalten die Nachricht *ERROR_DETECTED* zweimal von dem gleichen Absender und setzen somit *recover_node* auf *NULL* (Abbildung 47, Zeile 87 bis Zeile 93).

8.3.5 Fehlerbehebung - *mutex_recover()*

Hat der *recover_node* die Statuscontainer erhalten, hat er alle notwendigen Informationen, um eine Fehlerbehandlung durchzuführen. Nun rekonstruiert der Knoten das System, d. h. konstruiert lokal die Kette der Zeiger *Next* (Abbildung 49, Zeile 2) und überprüft, ob das Token verlorengegangen ist durch Iteration über den Statuscontainer. Die *Next*-Kette wird geschlossen (Abbildung 49, Zeile 9), *Dir*-Zeiger des Systems, die auf den ausgefallenen Knoten verweisen, müssen auf den Knoten, der das Ende der Nextkette bildet, gesetzt werden (Abbildung 49, Zeile 7). Das Token muß, falls es verlorengegangen ist, regeneriert und an den Knoten, der das Ende der *Next-Kette* bildet, gesendet werden (Abbildung 49, Zeile 23 bis 28). Abschließend muß der Ring geschlossen werden (Abbildung 49, Zeile 12 bis 15).

Erhält ein Knoten die Nachricht *NEW_STATUS*, setzt er zunächst *recover_node* auf *NULL* um anzuzeigen, daß die Fehlerbehebung abgeschlossen ist (Abbildung 50, Zeile 2). Nun trägt er die neue Variablenbelegung in seine Struktur ein, falls sich diese in den Komponenten nicht verändert hat, d. h. *Dir*, *Next*, *pred* und *succ* erhalten gegebenenfalls neue Werte. Die Fehlerbehandlung ist somit abgeschlossen.

Tabelle 3 zeigt die von dem Algorithmus benötigten Variablen und Konstanten. Die fettgedruckten Teile der Tabelle heben die Teile hervor, die für die fehlertolerante Erweiterung dem Basisalgorithmus hinzugefügt wurden.

```
Beim Empafang von error_detected(container sc) begin
   if (cs.initiator = self)
      reconer_node = NULL
      return;
   fi
   if (recover_node != NULL)
      if (cs.initiator == recover_node)
         recover_node = NULL
         if (sender == succ)
            send_error_detected(sc) to pred
         else
            send_error_detected(sc) to succ
         return
      fi
      if (cs.initiator < recover_node)
         recover_node = cs.initiator;
      else
         return;
      fi
   else
      recover_node = cs.initiator;
      cs += SelfStatus;
      if (recover_node < self)
         send_error_detected(cs) to succ;
         prob_crashed_node = succ;
      else
         send_error_detected(cs) to pred;
         prob_crashed_node = pred;
      fi
      await (ACK) or (TIMEOUT)
      if (ACK = FALSE)
         cs.crashed_node = prob_crashed_node;
         send_status_container (cs) to cs.initiator;
      fi
   fi
end.
```

Abbildung 47: Der Nachrichtenhandler für die Nachricht *ERROR_DETECTED*.

```
Beim Empafang von status_container(cs) begin
   local_cs += cs;
   cnt_cs++;
   if (recover_node != self)
      send_status_container(local_cs) to recover_node;
      cnt_cs = 0;
      return;
   if (cnt_cs = 2)
      mutex_recover(local_cs);
end.
```

Abbildung 48: Der Nachrichtenhandler für die Nachricht *STATUS_CONTAINER*.

```
mutex_recover(cs) begin
   nc = reconstruct(cs);
   token_lost = TRUE;
   new_status = NULL;
   foreach nc in cs do
      if (nc.dir = cs.crashed_node)
         new_status.dir = head(next_chain);
      if (nc.next = cs.crashed_node)
         new_status.next = hole(next_chain);
      if (nc.Token = TRUE)
         token_lost = FALSE;
      if (nc.pred = cs.crashed_node)
         new_status.pred = pred - 1;
      if (nc.succ = cs.crashed_node)
         new_status.succ = succ + 1;
      send_new_status(new_status) to ns.self;
      new_status = NULL;
   od
   if (dir = cs.crashed_node)
      dir = head(next_chain);
   if (next = cs.crashed_node)
      next = hole(next_chain);
   if (token_lost)
      if (tail(next_chain) = self)
         Token = TRUE;
      else
         send_token() to tail(next_chain);
      fi
   fi
end.
```

Abbildung 49: Die Funktion *mutex_recover()*.

```
Beim Empafang von new_status(s) begin
   recover_node = NULL
   if (s.dir)
      if (dir)
         dir = s.dir;
   if (s.next)
      if(next)
         next = s.next;
   if (s.pred)
      pred = s.pred;
   if (s.succ)
      succ = s.succ;
end.
```

Abbildung 50: Der Nachrichtenhandler für *NEW_STATUS*.

Name	Type	Beschreibung
Token	Boolean	Repräsentiert das Token auf einem Knoten.
cs	Boolean	Zeigt an, ob der Knoten sich in seiner kritischen Sektion befindet.
requested	Boolean	Zeigt an, ob der Knoten eine aktuelle Anfrage hat.
Self	Knoten	Identifiziert den eigenen Knoten.
Dir	Knoten	Bestimmt die Richtung einer Tokenanfrage.
Next	Knoten	Speichert den Knoten, der als nächstes das Token erhält.
pred	Knoten	Speichert den Vorgänger eines Knotens. Die Gesamtheit dieser Zeiger bildet eine Hälfte des logischen Rings.
succ	Knoten	Speichert den Nachfolger eines Knotens. Die Gesamtheit dieser Zeiger bildet die andere Hälfte des logischen Rings.
cnt_sc	integer	Zeigt an, wieviele Container der *recover_node* bisher erhalten hat.
local_sc	Feld	Enthält den Status von Knoten.
TOKEN	Nachricht	Das Token wird versendet.
REQUEST	Nachricht	Eine Tokenanfrage, mitgeführt wird in ihr der Absender der Anfrage.
ERROR_DETECT	Nachricht	Wird von einem Knoten in Richtung pred und succ verschickt, wenn der Knoten meint, einen Fehler erkannt zu haben.
STATUS_CONTAINER	Nachricht	Die Knotenstati werden an den *recover_node* versendet.
NEW_STATUS	Nachricht	Nachricht, die veranlaßt, eine neue Variablenbelegung einzutragen.

Tabelle 3: Struktur für die fehlertolerante Erweiterung des Algorithmus DVA.

8.3.6 Beispiele

Der folgende Abschnitt soll die Fehlererkennung und die Fehlerbehebung anhand von Beispielen illustrieren. Es wird ein System von acht Knoten (A, B, C, D, E, F, G und H) betrachtet. Knoten sind durch Kreise, in denen der eindeutige Bezeichner des Knotens steht, gekennzeichnet. Der für die Behandlung eines Fehlers notwendige doppeltverkettete Ring wird in den Abbildungen durch eine gestrichelte Ellipse dargestellt. Die Knoten haben die Möglichkeit, in beide Richtungen des Rings zu kommunizieren. Durchgehend gezeichnete Pfeile stehen für das Senden einer Nachricht, wobei die Pfeilspitze die Kommunikationsrichtung angibt. An diesen Pfeilen befinden sich die jeweiligen Nachrichtennamen. In einer Klammer hinter dem Namen befindet sich der Initiator der Nachricht, falls dieser nicht der Absender ist. Die Zahl vor einer Nachricht bezieht sich auf die Reihenfolge des Sendens im gesamten System. Ein ausgefallener Knoten, der so genannte *crashed_node*, ist durch ein Kreuz gekennzeichnet. Die schraffierten Knoten sind Knoten, die einen Fehler erkannt haben und somit Kandidaten für den *recover_node* sind. Die Nachricht *ERROR_DETECTED* wird durch *ERROR()* abgekürzt. Bei der Nachricht *STATUS_CONTAINER* wurde auf die Parameterliste verzichtet. Ein Kommunikationspfeil, der auf den Absender verweist, illustriert, daß der Adressat die Nachricht nicht quittiert hat und somit ein Timeout eingetreten ist.

Fehlererkennung

Zunächst soll der Fall des "Falschen Alarms" betrachtet werden (siehe Abbildung 51). Knoten E hat einen Fehler erkannt und sendet somit die Nachricht *ERROR_DETECTED* an Knoten F und Knoten D. Diese Nachrichten werden entlang des Rings weitergesendet. Beim Empfang dieser Nachricht setzen die einzelnen Knoten ihre lokale Variable *recover_node* auf Knoten E. Da es sich um den Fall des "Falschen Alarms" handelt, umrundet die Nachricht den gesamten Ring. Empfangen die Knoten des Systems die Nachricht *ERROR_DETECTED* zum zweiten Mal von dem gleichen Initiator erkennen sie einen "Falschen Alarm" und setzen somit *recover_node* auf *NULL*. Dies ist notwendig, weil ein Wert von *recover_node* anzeigt, daß eine Fehlerbehandlung läuft. Wenn die Nachricht Knoten E erreicht, erkennt auch dieser den "Falschen Alarm". Beim Initiator endet der Mechanismus des Weitersendens.

Abbildung 52 zeigt die Operationen und die Nachrichtenfolge im Fall, daß ein Knoten ausgefallen ist und ein Knoten den Ausfall erkannt hat. Der ausgefallene Knoten ist Knoten A, somit existiert ein Fehler im System. Knoten E hat den Fehler erkannt und sendet die Nachricht *ERROR_DETECTED* an Knoten F und Knoten D. Nacheinander erreicht die Nachricht die anderen Knoten des Systems.

Wenn ein Knoten des Systems die Nachricht *ERROR_DETECTED* erhal-

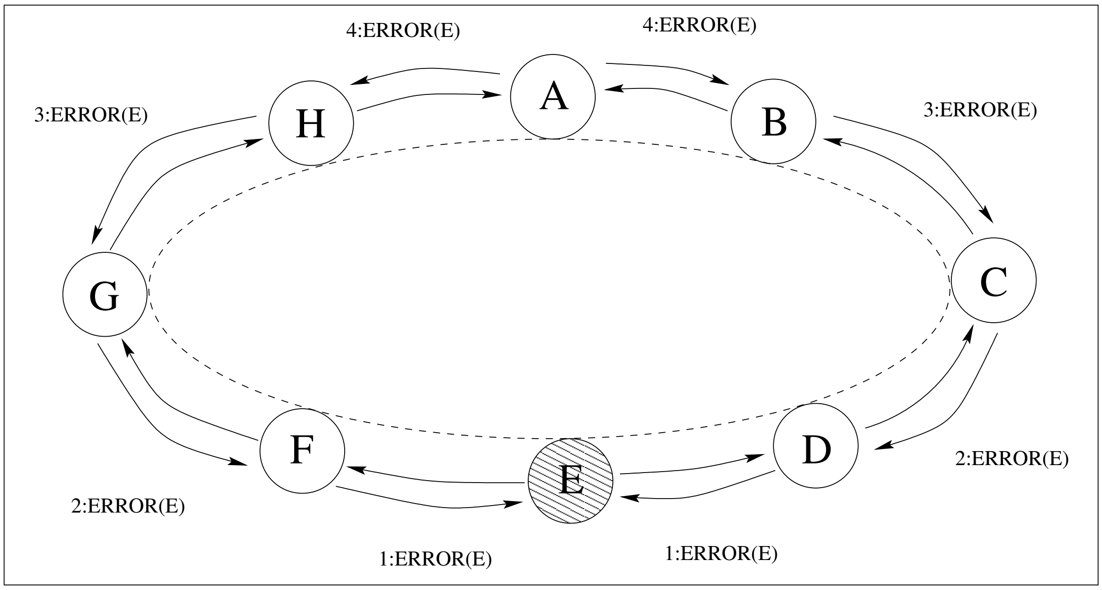


Abbildung 51: Beispiel für die Erkennung eines "Falschen Alarms".

ten hat und einer seiner Komponenten ebenfalls den Ausfall von Knoten A erkennt, bleibt diese Fehlererkennung ohne Wirkung: Mit dem Empfang von *ERROR_DETECTED* wird der Variablen *recover_node* der Initiator zugewiesen. Ein Wert von *recover_node* ungleich *NULL* zeigt an, daß schon eine Fehlerbehandlung läuft.

Der Versuch, die Nachricht *ERROR_DETECTED* an den Knoten A zu senden, schlägt für den Knoten H und den Knoten B fehl. Die Nachrichten bleiben nach dem Ablauf des Timeouts unquittiert. Daraus folgt, daß Knoten H und Knoten B den nun vollständigen Statuscontainer an den Initiator der *ERROR_DETECTED* Nachricht senden. Knoten E hat somit alle Informationen, um eine Fehlerbehebung zu starten.

Erkennen mehrere Knoten einen Fehler, d.h. bevor die Nachricht *ERROR_DETECTED* den Knoten erreicht und somit *recover_node* gesetzt wurde, erkennt eine Komponente des Knotens ebenfalls einen Fehler, folgt, daß mehrere Knoten eine Fehlerbehandlung starten. In diesem Fall muß unter den Knoten, die den Fehler erkannt haben, einer ausgewählt werden, der die Fehlerbehandlung durchführt. Wie oben beschrieben, wird der Knoten gewählt, der die kleinste ID hat.

Im Beispiel der Abbildung 53 erkennen die Knoten C, E und F den Ausfall von Knoten A. Nacheinander senden sie die Nachricht *ERROR_DETECTED* an ihre Ringnachbarn. Erhält nun Knoten F die von Knoten E initiierte Nachricht, weist er der Variablen *recover_node* Knoten E zu, da E kleiner ist als seine eigene ID. Da die Variable *recover_node* des Knotens F nicht *NULL* war, sendet

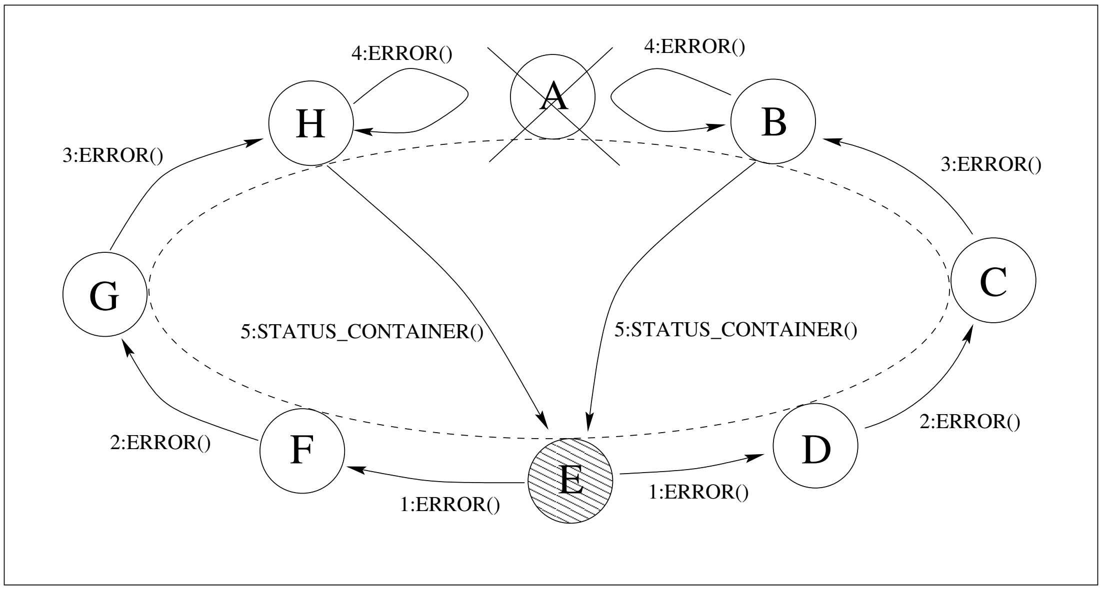


Abbildung 52: Beispiel 1 für die Fehlererkennung.

er die Nachricht nicht weiter. Dasselbe geschieht auf Knoten E mit dem Erhalt der *ERROR_DETECTED* Nachricht von Knoten C. Das Umsetzen des Wertes von *recover_node* baut somit eine Kette auf (siehe Abbildung 53) [10]. Knoten H und Knoten B senden, nachdem das Timeoutintervall des Sendens an Knoten A ausgelaufen ist, die Nachricht *STATUS_CONTAINER* an den Initiator der *ERROR_DETECTED* Nachricht. Wenn Knoten F den Statuscontainer von Knoten H erhält, signalisiert ihm ein Wert von *recover_node* ungleich *Self*, daß er die Nachricht in Richtung *recover_node* weitersenden muß. Über Knoten E gelangt der Statuscontainer letztendlich zu Knoten C, dem *recover_node*. Die Nachricht *STATUS_CONTAINER* von Knoten B erreicht den *recover_node* auf direktem Weg. Knoten C hat somit alle Informationen, um eine Fehlerbehebung durchzuführen.

Fehlerbehebung

In den folgenden Abbildungen geben die gestrichelten Pfeile die *Next* Zeiger an, durchgehend gezeichnete Pfeile kennzeichnen die Zeiger *Dir*. Das **T** bezeichnet die Position des Tokens im System.

In Abbildung 54 ist durch den Ausfall von Knoten A die *Next*-Kette unterbrochen. Weiterhin verweist der *Dir* Zeiger von Knoten B auf den ausgefallenen Knoten. Das Token ist in diesem Falle nicht verlorengegangen. Knoten E rekon-

[10] Je nachdem, ob Knoten D erst die *ERROR_DETECTED* Nachricht von Knoten C oder von Knoten E empfängt, sendet er *ERROR_DETECTED* an Knoten C.

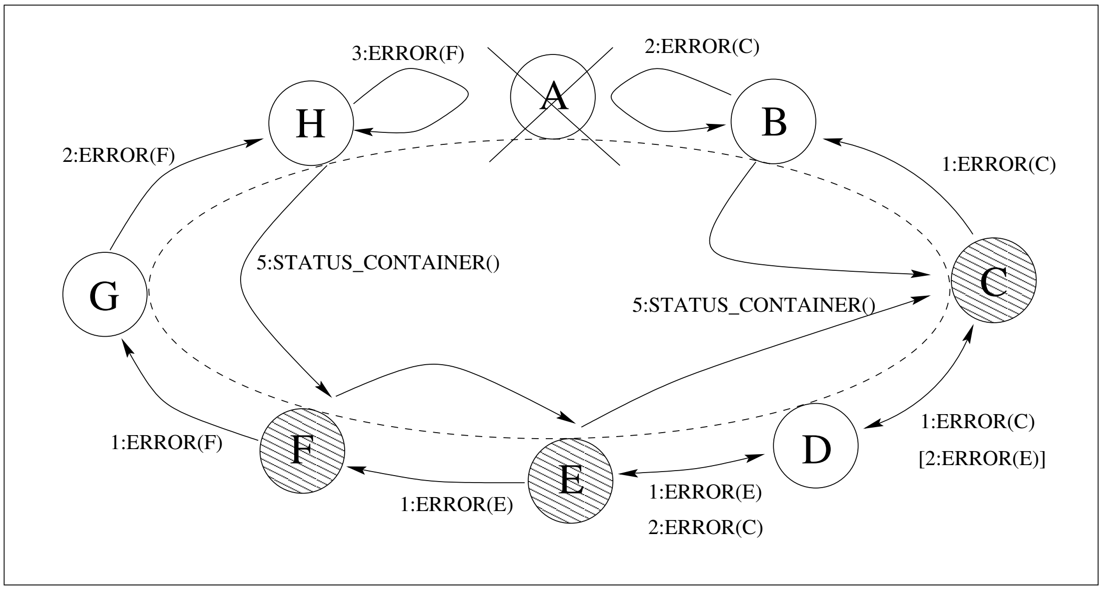


Abbildung 53: Beispiel 2 für die Fehlererkennung.

struiert, nachdem er die Statuscontainer erhalten hat, den Zustand des Systems. Er veranlaßt, daß Knoten B seine Zeiger *Next* und *Dir* auf Knoten H setzt. Nachdem Knoten H und Knoten B den Ring geschlossen haben, ist das System in einen konsistenten Zustand zurückgekehrt. Knoten E sendet an alle Knoten die Nachricht *NEW_STATUS*. Diese Nachricht veranlaßt bei Knoten C, D, F, G und H lediglich, daß die Knoten ihrer Variablen *recover_node* den Wert *NULL* zuweisen. Bei Knoten B führt er den neuen Wert für *Next*, *Dir* und *pred* mit. Die Nachricht *NEW_STATUS*, die Knoten H erreicht, führt den neuen Wert für *succ* mit.

Abbildung 55 zeigt das System nach der Fehlerbehebung: Der Ring ist geschlossen, die Kette der Zeiger *Next* ist wieder hergestellt und alle Zeiger *Dir* des Systems haben definierte Werte. Da mit dem Empfang von *STATUS_CONTAINER* auf allen Knoten des Systems der Variablen *recover_node* der Wert *NULL* zugewiesen wurde, arbeitet das System nach dem Basisalgorithmus.

Das Beispiel in Abbildung 56 soll folgendes Szenario beschreiben: Kurz nach dem Start des Systems tritt bei Knoten A ein Fehler auf: Knoten A fällt aus. Knoten H und Knoten G haben eine Anfrage an das Token generiert. Knoten A konsumiert das Token. Alle anderen Knoten des Systems befinden sich im initialen Zustand. Knoten E hat mit einer Tokenanfrage den Fehler von A festgestellt und die Fehlerbehandlung eingeleitet.

Knoten E rekonstruiert das System und stellt dabei fest, daß das Token verlorengegangen ist. Knoten, deren Zeiger *Dir* auf Knoten A zeigten, werden veran-

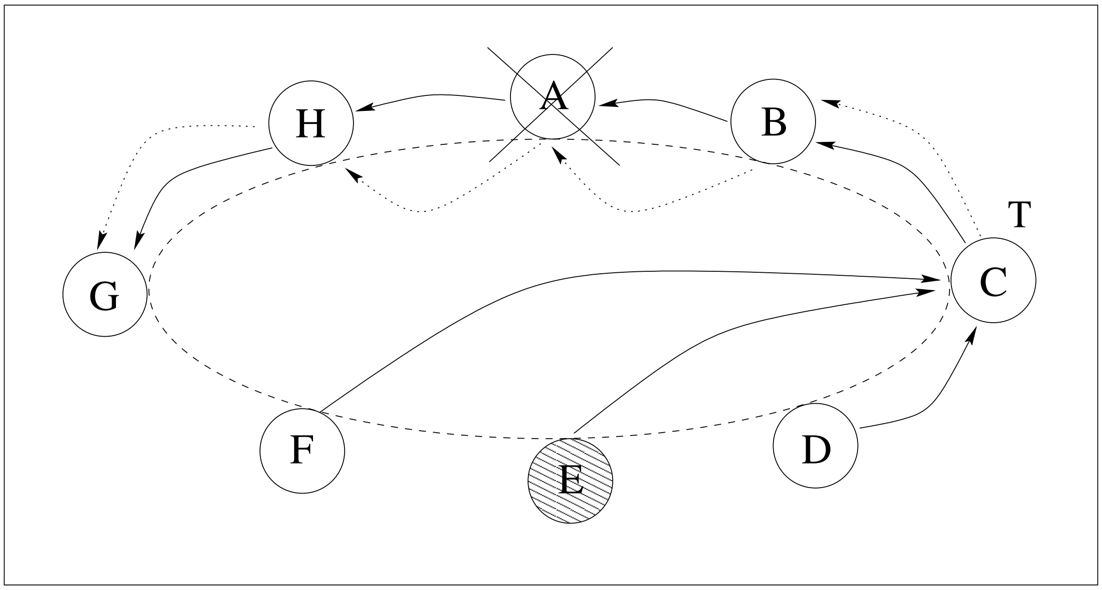


Abbildung 54: Beispiel für einen Zustand der Knoten im Fehlerfall.

laßt, *Dir* auf das Ende der *Next*-Kette zu setzen. An den Anfang der *Next*-Kette wird das Token gesendet. Das Token ist somit regeneriert und das System befindet sich in einem konsistenten Zustand. Abschließend sendet Knoten E die fehlgeschlagene Anfrage erneut. Das Setzen des Systems in einen neuen Zustand erfolgt mit dem Senden der Nachricht *NEW_STATUS* und somit in einem Schritt. Der Empfang von *NEW_STATUS* bewirkt wiederum, daß die Variable *recover_node* aller Knoten auf *NULL* gesetzt wird.

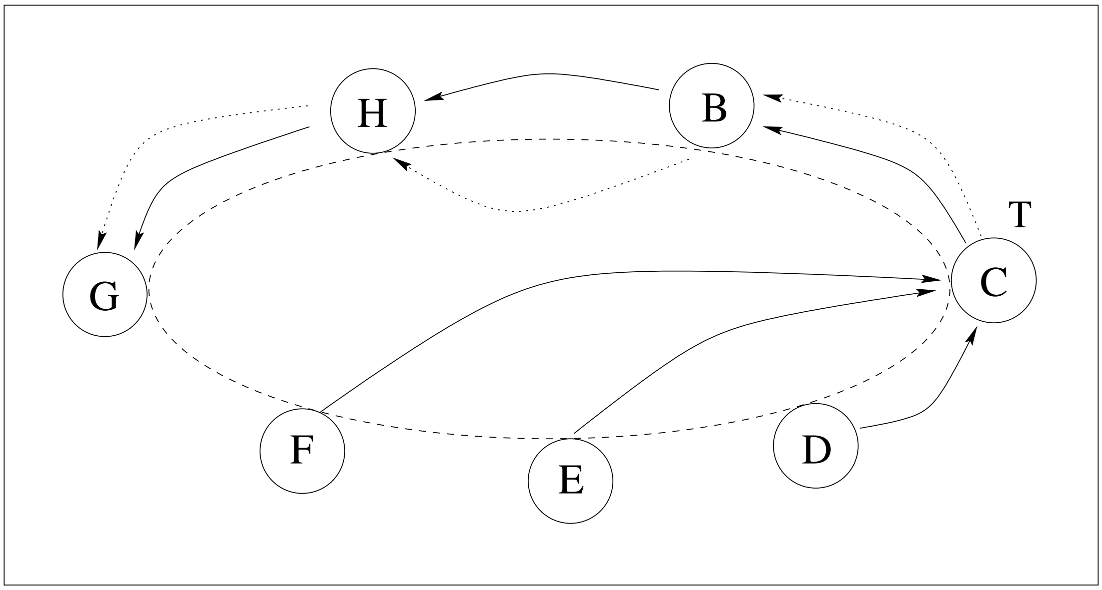


Abbildung 55: Der Zustand des Systems nach erfolgter Fehlerbehandlung.

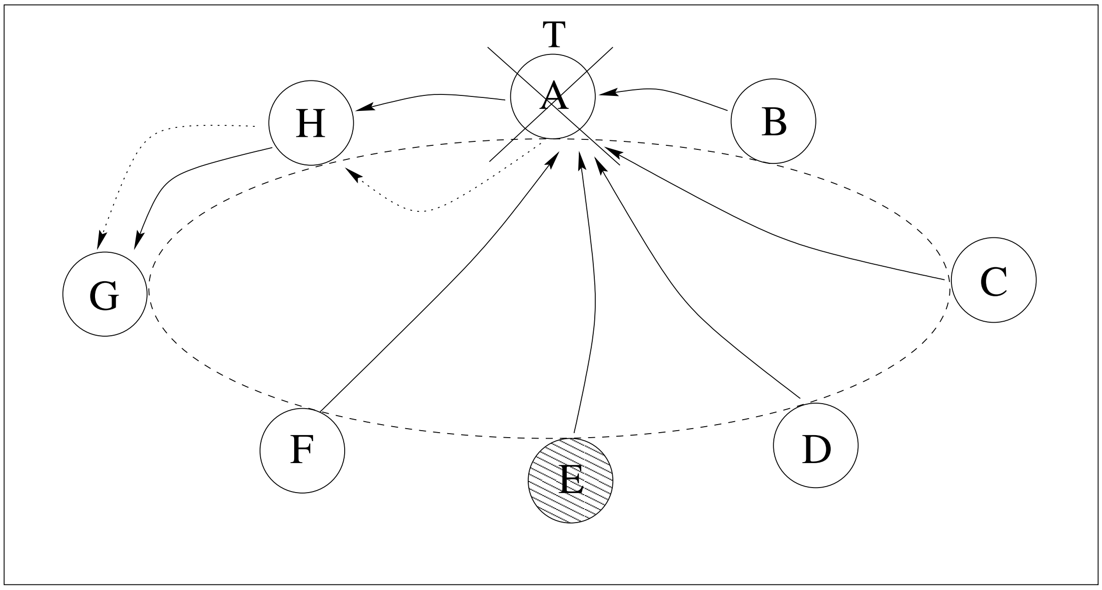


Abbildung 56: Der Zustand des Systems nach dem Ausfall von Knoten A.

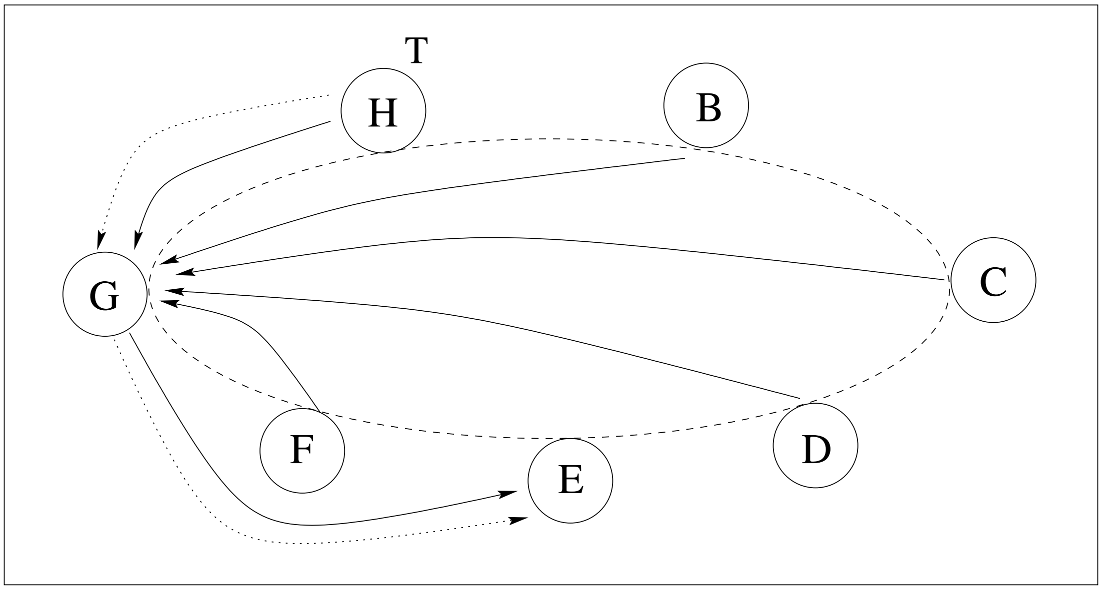


Abbildung 57: Der Zustand des Systems nach der Fehlerbehebung.

8.3.7 Korrektheit

Der Basisalgorithmus ist ein korrekter Algorithmus zum gegenseitigen Ausschluß in verteilten Systemen [12]. Zu zeigen ist, daß sich das System nach einer erfolgten Fehlerbehandlung in einem konsistenten Zustand befindet, welcher den gegenseitigen Ausschluß garantiert, frei von Verklemmungen ist und das Verhungern vermeidet.

Gegenseitiger Ausschluß

Zu zeigen ist, daß sich nach der Fehlerbehebung nur ein Token im System befindet.
Fall 1: Das Token ging bei dem Fehler nicht verloren.
Es wird kein neues Token generiert, da der *recover_node* anhand der Stati feststellt, daß ein Token im System existiert.
Fall 2: Das Token ist durch den Fehler verlorengegangen.
Durch die Wahl genau eines Knotens zum *recover_node* ist sichergestellt, daß nur ein Knoten die Fehlerbehebung durchführt. Das Token wird auf dem Knoten regeneriert, der das Ende der verteilten Warteschlange bildet, falls dieser existiert. Anderenfalls wird das Token auf dem *recover_node* regeneriert. Dies ist der Fall, wenn sich im System keine Knoten befinden, die eine aktuelle Anfrage haben. Daraus folgt, daß, falls das Token verlorengegangen ist, nur ein neues Token generiert wird. Somit ist die Situation des Basisalgorithmus erreicht. Es folgt, daß der Algorithmus weiterhin den gegenseitigen Ausschluß garantiert.

Vermeidung von Verklemmungen

1. Anfragen erreichen weiterhin den Tokenbesitzer bzw. den vermeintlichen Tokenbesitzer: [11]
Das System der Zeiger *Dir* wird nur insofern verändert, daß Knoten deren Zeiger *Dir* auf den ausgefallenen Knoten verweisen, veranlaßt werden, ihren *Dir* Zeiger auf das Ende der *Next* Warteschlange zu setzen. Damit ist sichergestellt, daß alle Anfragen ihr Ziel erreichen.
2. Das Token erreicht den nächsten Knoten in der verteilten Warteschlange:
Das Token wird entlang der *Next* Zeiger des System versendet. Dazu sind zwei Fälle zu unterscheiden:
Fall 1: Die Kette ist durch den Ausfall des Knotens nicht verletzt.
Der fehlertolerante Algorithmus läßt die Kette unverändert. Der Algorithmus befindet sich im Zustand des Basisalgorithmus.
Fall 2: Die Kette ist durch den Ausfall des Knotens verletzt .
Die Kette wird lediglich geschlossen und bleibt somit strikt FIFO und frei von

[11] Ein vermeintlicher Tokenbesitzer bezeichnet hier das Ende der verteilten Warteschlange, aufgebaut durch die Zeiger *Next*.

Zyklen. Auch hier ist die Situation des Basisalgorithmus hergestellt.

Durch die Wiederherstellung des konsistenten Zustandes des Basisalgorithmus ist der Algorithmus nach einer Fehlerbehebung frei von Verklemmungen.

Vermeidung des Verhungerns

Nachrichten werden weiterhin in der Reihenfolge verarbeitet, wie sie die Knoten erreichen.

Die Reihenfolge in der verteilten Warteschlange der Zeiger *Next* wird nicht verändert. Daraus folgt, daß wenn ein Knoten in die Warteschlange eingereiht wurde, der Knoten, selbst wenn immer wieder neue Fehler auftreten, das Token in der Reihenfolge bekommt, die die Warteschlange vorgibt.

Nach dem Regenerieren des Tokens wird das Token an das Ende der Warteschlange versendet. Somit bleibt die Reihenfolge bestehen. Der Algorithmus vermeidet so weiterhin das Verhungern von Knoten.

Der Algorithmus DVAFT ist trotz der fehlertoleranten Erweiterung ein korrekter Algorithmus zum gegenseitigen Ausschluß, d.h. er garantiert den gegenseitigen Ausschluß, ist frei von Verklemmungen und vermeidet das Verhungern.

8.3.8 Komplexität

Im Falle des "Falschen Alarms" umrundet die Nachricht *ERROR_DETECTED* einmal in Richtung der Vorgänger und einmal in Richtung der Nachfolger den gesamten Ring. Die Anzahl der Nachrichten im System ist somit 2 * N. Bezüglich eines Knotens ist sie konstant 2. Es folgt somit eine Komplexität der Nachrichten im Gesamtsystem von O(N) und bezüglich eines Knotens von O(1).

Ist tatsächlich ein Fehler aufgetreten, werden im besten Fall, d.h. nur ein Knoten erkennt den Fehler, $N + 2$ Nachrichten benötigt: Die Nachricht *ERROR_DETECTED* wird über den gesamten Ring versendet, N / 2 in Richtung der Vorgänger und N/2 in Richtung der Nachfolger. Mit einbezogen in die Betrachtung wurden hier die Nachrichten, die an den ausgefallenen Knoten gesendet werden. Somit folgt eine Komplexität der Nachrichten für die Fehlererkennung von O(N) im gesamten System. Pro Knoten ergibt sich ein konstanter Aufwand von O(1).

Bei der Fehlerbehandlung sendet der *recover_node* die Nachricht *NEW_STATUS* an alle Knoten des Systems, d.h. die Nachricht *NEW_STATUS* wird N - 2 mal versendet. Es folgt eine Komplexität der Nachrichten von O(N).

Die Komplexität der Nachrichten im schlechtesten Fall beträgt somit für den gesamten Algorithmus (2 * N - 2), d.h. O(N).

8.4 Performance

Im ersten Schritt wurde das Systemverhalten beim Auftreten von "Falschen Alarms" getestet. Die Anzahl der Knoten wurde mit 8 festgesetzt, Phi betrug 200 μsec, Gamma 2000μsec und die Anzahl der kritischen Sektionen betrug 64. Abbildung 58 zeigt das Verhalten der Antwortzeit beim Auftreten von "Falschen Alarms". Die Zunahme ist linear mit Trend zum logarithmischen Verlauf. Es werden 2 * N Nachrichten je "Falschen Alarm" benötigt. Im unteren Bereich (Anzahl der "Falschen Alarms" kleiner 4) ist der Verlauf unregelmäßig. Die Abweichung sind jedoch minimal und sind auf betriebssystembedingte Lasten zurückzuführen.

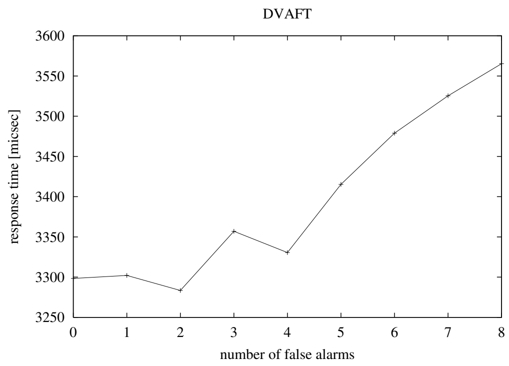


Abbildung 58: Verhalten des DVAFT beim Auftreten vom "Falschen Alarm".

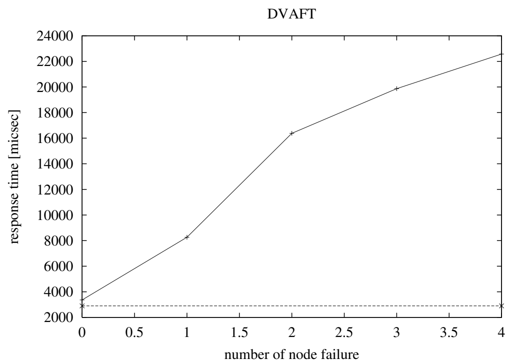


Abbildung 59: Verhalten der Antwortzeit beim Auftreten von Knotenfehlern.

Die zweite Meßreihe stellt die mittlere Antwortzeit in Abhängigkeit von der Anzahl aufgetretener Knotenfehler dar. Die Anzahl der Knoten liegt wiederum bei 8, die Anzahl kritischer Sektion lag bei ca. 40, Phi betrug 200 μsec und Gamma 2000μsec. Abbildung 59 stellt die Ergebnisse grafisch dar. Die Antwortzeit steigt linear um einen starken Faktor. Dazu ist zu sagen, daß mit der Zunahme der

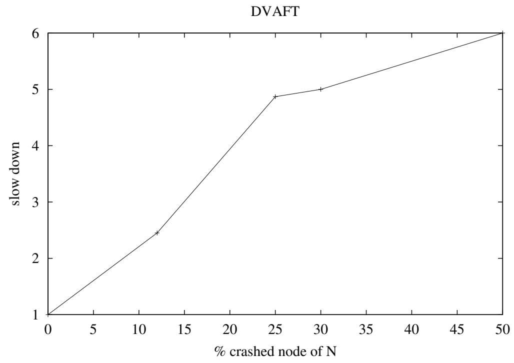


Abbildung 60: Verhalten der Antwortzeit bei Knotenfehlern.

Anzahl ausgefallener Knoten die Systemgröße abnimmt. Somit nimmt die Anzahl der Nachrichten ab und die Antwortzeit wird dementsprechend geringer. Der konstante Graph in Abbildung 59 zeigt Ergebnisse des Algorithmus DVARW bei einem Testlauf mit vergleichberan Parametern . [12]

Die größere Antwortzeit im fehlerfreien Fall ist auf die zusätzlichen Einheiten der fehlertoleranten Kommunikation zurückzuführen. Diese sind statisch redundante Mittel und beeinflussen die Performance schon im fehlerfreien Zustand des Systems.

Abbildung 60 zeigt den Faktor der Zunahme bei der mittleren Wartezeit in Abhängigkeit von der Prozentzahl ausgefallener Knoten. Schon bei dem Ausfall von 25% der Knoten des Systems ist eine Zunahme um den Faktor 5 abzulesen, was für bestimmte zeitkritische Anwendung nicht zumutbar ist. Während der Behandlung des Fehlers sind alle Aktionen eingefroren. Die Wartezeit erhöht sich für die Knoten, die eine aktuelle Anfrage habe, enorm. Bei einer Applikation mit hoher Anfragefrequenz (siehe die Testläufe), habe alle Knoten des Systems eine aktuelle Anfrage. Die Anfragezeit vervielfacht sich. Da, nachdem die Knotenfehler behandelt wurden, keine weiteren Verzögerungen eintreten, relativiert sich diese Vervielfachung.

8.5 Zusammenfassung

In einem verteilten System können Fehler auftreten, die die Verletzung der Korrektheit eines Algorithmus zum gegenseitigen Ausschluß nach sich ziehen. Unterschieden werden Knoten- und Netzwerkfehler. Knotenfehler bedeutet den plötzlichen Ausfall eines Knotens des Systems. Der Verlust von Nachrichten und die dauerhafte Störung von Nachrichtenverbindungen sowie semantische Veränderungen von Nachrichten werden unter Netzwerkfehler zusammengefaßt. Netzwerkfehler

[12] Da der Algorithmus DVAFT nur exklusive Sperren unterstützt, wurde die Anzahl Lesesperren bei dem Vergleich auf 0% festgesetzt.

können durch ein zuverlässiges Netzwerk oder durch eine fehlertolerante Kommunikationseinheit erkannt und behoben werden. Diese Kommunikationeinheit nutzt Quittungen, Versionsnummern und Timeouts. Die fehlertolerante Kommunikationseinheit ist ein statisch redundantes Mittel und wirkt sich somit schon zur fehlerfreien Laufzeit negativ auf die Performance aus. In dem vorgestellten fehlertoleranten Algorithmus zum gegenseitigen Ausschluß hat die fehlertolerante Kommunikationseinheit eine weitere Aufgabe: Sie ist für das Erkennen der meisten Knotenfehler verantwortlich. Nach einer Anzahl von Versuchen, eine nicht quittierte Nachricht erneut zu senden, wird kapituliert und die Behandlung eines Knotenfehlers ausgelöst. Knotenfehler, die nicht durch die Kommunikationseinheit erkannt werden, werden wiederum durch Timeouts erkannt. Die Evaluierung der Timeoutintervalle gehört zum Feintuning eines verteilten Systems und hängt stark von der Größe des Systems und der zugrundeliegenden Architektur ab. Durch kurze Timeout-Intervalle werden einerseits Fehler schnell erkannt und behoben, andererseits tritt der Fall des "Falschen Alarms" häufiger auf. Der Fall des "Falschen Alarms" bewirkt, daß $2 * N$ Nachrichten versendet werden. Dieser Fakt ist bei der Evaluierung der Timeout-Intervalle unbedingt zu berücksichtigen.

Der fehlertolerante Algorithmus ist tokenbasierend und arbeitetet im fehlerfreien Fall mit einer Komplexität der Nachrichten im mittleren Fall (ac) von $O(\log N)$. Er hat einen konstanten Speichermehraufwand pro Knoten und benötigt für die Behebung eines Knotenfehlers oder eines dauerhaften Netzwerkfehlers $O(N)$ Nachrichten.

Um Fehlertoleranz zu realisieren werden die Knoten des Systems, zusätzliche zu den von dem Basisalgorithmus benötigten Struktur, in einem doppelt verketteten Ring organisiert. Jeder Knoten erhält einen Vorgänger und einen Nachfolger. Im Fehlerfall wird der Ring als Nachrichtentopologie genutzt und nach erfolgter Fehlerbehebung wird der, durch den Fehler möglicherweise verletzte, Ring wieder geschlossen.

Die Wirkung des "Falschen Alarms" auf die mittlere Wartezeit ist eine geringe Zunahme mit linearen Verlauf. Mit Zunahme von Knotenfehlern steigt die mittlere Wartezeit ebenfalls linear, jedoch um einen starken Faktor. Die Ausführungszeit erhöht sich schon bei 25% Knotenausfällen um den Faktor 4. Diese Zahl relativiert sich dadurch, daß, nachdem die Knotenfehler behoben sind, das System ohne weitere Verzögerungen weiterarbeitet und sich die Größe des System verkleinert. Ein Extrembeispiel: Ist die Systemgröße gleich 2 und 50% der Knoten fallen aus, geht die mittlere Anfragezeit mit Zunahme der kritischen Sektionen gegen 0. Bei diesem Beispiel bleibt natürlich die Frage nach der Gesamtausführungszeit unbeantwortet.

9 Ausblick

Die Ergebnisse der Lese- und Schreibsperren sind befriedigend, wie aus den Testreihen ablesbar ist. Das Protokoll ist leicht verständlich und läßt sich nach dem gegebenen Pseudocode leicht in beliebige Programmiersprachen übertragen. Basierend auf dem Algorithmus DVARW wurde die Konsistenz eines Prototypen für einen verteilten Server in der Firma Tech@Spree Software GmbH realisiert. Der Prototyp erzielte in der Performance, trotz der Implementierung in Java, sehr gute Ergebnisse. Der Einsatz von Lese- und Schreibsperren ist in allen Fällen sinnvoll und hat besondere Stärken in der Umgebung von Datenbanken.

Die Ergebnisse des fehlertoleranten Algorithmus sind nicht befriedigend: Das Protokoll arbeitet nach einer komplizierten Vorschrift, die die Behandlung vieler Ausnahmen bedingt. Eine Implementierung ist dadurch komplex und mit großem Aufwand verbunden. Weitere Probleme sind die zusätzliche Threads und die Bereitstellung einer fehlertoleranten Knommunikation. Diese Komponenten sind statisch redundante Mittel und beeinflussen die Performance schon im fehlerfreien Fall.

Allgemein als schweres Problem bekannt ist die Evaluierung der Timeoutintervalle. Diese Evaluierung ist von vielen Faktoren abhängig, wie z. B. der Plattform, der Topologie und dem Netzwerk. Heuristiken und Korrekturen zur Laufzeit sind notwendig.

Offen bleibt auch das Problem der Partitionierung des Systems: Um Partitionierungen des Systems bei N gleichzeitigen Fehlern zu tolerieren, sind mehrere Ringe notwendig. Mit N Ringen kann die Partitionierung des Systems in N Partitionen verhindert werden.[13] Diese Erweiterung benötigt einen linearen Speichermehraufwand je Knoten.

Die schlechten Ergebnisse der praktischen Untersuchung führen zu folgenden Überlegungen: Ziel ist es, ein Protokoll zu entwickeln, welches das System nicht sofort beim Auftreten eines Fehlers einfriert. Dazu ist eine weitere Einheit, ein weiterer Thread nötig, der die Nachrichten der Fehlerbehandlung verarbeiten. Eine weitere Einheit bringt einen erhöhten Synchronisationsaufwand mit sich, trägt aber zur besseren Strukturierung des Protokolls bei. Die Anzahl der Kontextwechsel nimmt zu, die Funktion, welche das System lokal rekonstruiert, verliert allerdings an Komplexität.

Unter Betrachtung der obengenannten Fakten ist zu überlegen, ob ein anderer Ansatz als der tokenbasierende sinnvoller ist. Denkbar sind beispielsweise zentralisierte Ansätze: Ein Knoten fungiert als Mutex- und Recover-Server. Dieser Server wird gespiegelt, um einen "single point of failure" zu vermeiden, ist aber zentraler Ansprechpartner für alle anderen Knoten. Dieser Ansatz hat Vorteile bei der Wartung und ist leicht zu implementieren, erzeugt jedoch einen Flaschenhals.

[13] Die Partitionierung des Systems ist in maximal $N/2$ Teile möglich. Es ist allerdings nicht vorhersagbar, welche Knoten sich in den einzelnen Partitionen befinden. Somit sind N disjunkte Ringe nötig.

In dem bei Tech@Spree entwickelten verteilten Server wird dieser Ansatz favorisiert: Der gegenseitige Ausschluß wird mit dem Algorithmus DVARW realisiert und es existiert ein zentraler Recover-Server.

Literatur

[1] J. Nehmer und T. Becker. A Fault Tolerance Approache for Distributed ISDN Control Systems. ACM Operating Systems Review 25(2), p. 126-129, 1991.

[2] E. W. Dijkstra. Solution of a problem in concurrent control. ACM Communication 8(9), p. 569, 1965.

[3] M. Naimi und M. Trehel. How to detect a failure and regenerate the token in the log(n) distributed algorithm for mutual exclusion. Proceedings of the second internatial workshop on distributed algorithms, Vol. 312, 155-166, 1987.

[4] M. Naimi und M. Trehel. An improvement of the log(n) distributed algoritm for mutual exclusion. In Distributed Computing Systems, 1987.

[5] M. Naimi, M. Trehel und A. Arnold. A log(n) distributed mutual exclusion algoritm based on path reversal. JPDC: Journal of Parallel and Distributed Computing, 34(1): 1-13, Apr. 1996.

[6] Y. I. Chang, M. Singhal und M. T. Liu. A fault-tolerant algorithm for distributed mutual exclusion. In Proceedings of the ninth Symposium on Reliable Distributed Systems, Oct. 1990.

[7] Y. I. Chang, M. Singhal und M. T. Liu. An improved O(log(n)) mutual exclusion algorithm for distributed systems. In Int'l Conf. on parallel Processing, p. III295-302, 1990.

[8] D. Agrawal und A. El Abbadi. A Token-based Fault-Tolerant Distributed mutual Exclusion algorithm. Journal of parallel and distributed Computing, 24: 164-176, 1995.

[9] L. Lamport. Time, Clocks, and the Ordering of Events in a distributed System. Comm. ACM, Vol. 7, p. 558-564, July 1978.

[10] S. Nishio, K. F. Li und e. G. Manning. A resilient mutual exclusion algorithm for computer networks. IEEE Trans. Parallel Distrib. Inform. Systems Vol. 1, 3, p. 344-355, July 1990.

[11] M. L. Neilsen und M. Mizuno. A dag-based algorithm for mutual exclusion. In International Conference on Distributed Computer Systems, p. 354-360, 1991.

[12] C. Wagner. Bewertung von Algorithm zum gegenseitigen Ausschluß. Studienarbeit an der Humboldt-Universität zu Berlin, Institut für Informatik, 1998.

[13] G. Ricart und A. K. Agrawal. An optimal algorithm for mutual exclusion in computer networks. Comm. ACM Vol. 24 (1), p. 9-17, 1981.

[14] K. Raymond. A distributed algorithm for multiple entries to a critical Section. Information Processing Letters, p. 189-193, Feb. 1989.

[15] P. K. Srimani und R. L. N. Reddy. Another Distributed Algorithm for Multiple Entries to a critical Section. InformationProcessing Letters, p. 51-57., Jan. 1992.

[16] I. Suzuki und T. Kasami. A distributed mutual exclusion algorithm. ACM Trans. Comput. Syst. Vol. 3, p. 344-349, 1985.

[17] S. Mishra und K. Srimani. Fault-Tolerant Mutual Exclusion Algorithms. Journal of Systems Software, Vol. 11 (2),p. 111-129, Feb.1990.

[18] Portable Operating System Interface (POSIX) part 1: System Applikation Program Interface (API). Technical Commitee on Operating Systems and Application Enviroments of the IEEE. ANSI/IEEE Std 1003.1, 1995 Edition including 1003.1c: Amendment2: ThreadsExtension [C Languages], 1996.

[19] F. Müller. A Library Implementation of POSIX Threads under UNIX. Proceedings of the USENIX Conference, p. 29-41,1993.

[20] Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API) Amendment 1: Realtime Extension [C Language]. Technical Committee on Operating Systems and Application Environments of the IEEE, P1003.1b-1993, 1993.

[21] M. Raynal und M. Singhal. Logical Time: Capturing Causality in Distributed Systems. IEEE Transaction on Computer, Vol. 18, p. 49-56, 1996.

[22] K. Raymond. A tree-based algorithm for distributed mutual exclusion. ACM Transaction on Computer Systems. Vol 7 (1), p. 61-77, 1989.

[23] Y. I. Chang. Design of mutual exclusion algorithms for realtime distributed systems. Journal of Information Science and Engeneering, Vol. 10, p 527-548, 1994.

[24] Y. I. Chang. Comments on "Two Algorithms for Mutual Exclusion in Real-Time Distributed Computer Systems". Journal of Parallel and Distributed Computing, Vol. 23, p. 449-454, 1994.

[25] Y. I. Chang. A simulation study on distributed mutual exclusion. Journal of parallel and distributed computing, Vol 33, p. 107-121, 1996.

[26] M. Choy und A. K. Singh. Efficient Fault-Tolerant Algorithms for Distributed Resource Allocation. ACM Transaction on Programming Languages ans Systems, Vol. 17(3), p. 535-559, 1995.

[27] Y. I. Chang, M. Singhal und M. T. Liu. An improved O(log n) mutual exclusion algorithm for distributed Systems. In Int. Conf. On Parallel Processing, p III295-302, 1990.

[28] Y. I. Chang, M. Singhal und M. T. Liu. A hybrid approach to mutual exclusion for distributed systems. Proceedings of the Annual International computer Software and Application Conference. IEEE Comput. Soc., Chicago, p. 289-294, 1990.

[29] O. S. F. Carvalho und G. Roucairol. On mutual exclusion in computer networks, technical correspondence. Comm. ACM, Vol. 26 (2), p. 146-148, Feb. 1983.

[30] Shiwa S. Fu, Nian-Feng Tzeng and Zhiyuan Li. Empirical Evaluation of Distributed Mutual Exclusion Algorithms. IEEE Computer 1997.

[31] S. S. Fu, N.-F.Tzeng und Z. Li. Empirical Evaluation of Distributed Mutual Exclusion Algorithms. In International Parallel Processing Symposium, p. 255-259, 1997.

[32] Th. Johnson. A performance Comparison of Distributed Mutual Exclusion Algorithms. In International Confernece on Parallel Processing, p. 258-264, 1995.

[33] A. Kumar. Hierachical quorum consensus: A new algorithm for managing replicated data. IEEE Trans. On Computers, Vol. 40 (9), p. 994-1004, 1991.

[34] A. J. Martin. Distributed Mutual Exclusion on a Ring of Processes. Science of Computer Programming, Vol. 5, p. 265-276, Feb. 1985.

[35] M. Maekawa. A sqrt(n) algorithm for mutual exclusion in decentralized systems. ACM Trans. on Computer Systems, Vol. 3 (2), p. 145-159, 1985.

[36] F. Müller. Prioritized Token-Based Mutual Exclusion for Distributed Systems. International Parallel Processing Symposium, p. 791-795, Mar. 1996.

[37] G. Ricart und A. K. Agrawala. Authoris response to on mutual exclusion in computer networks, technical correspondence. Comm. ACM, Vol. 26 (29, p. 146-148, Feb. 1983.

[38] B. A. Sanders. The information structure of distributed mutual exclusion algorithms. ACM trans. Computer Systems, Vol. 5 (3), p. 284-299, Aug. 1987.

[39] M. Singhal. A heuristically-aided algorithm for mutual exclusion in distributed systems. IEEE Trans. on Computer, Vol. 38 (5), p. 651-662, May 1989.

[40] M. Singhal. A dynamic information structure mutual exclusion algorithm for distributed systems. IEEE Trans. on Parallel ans distributed Systems, Vol. 3 (1), p. 121-125, Jan. 1992.

[41] M. Trehel und M. Naimi. A distributed algorithm for mutual exclusion based on data structures and fault tolerance. Proc of the Conference on Computer and Communication. IEEE Comput. Soc, Phoenix, p. 35-39, 1987.

[42] A. Goscinski. Two algorithms for mutual exclusion in realtime distributed computer systems. Journal of Parallel and Distributed Computing, p. 77-82, 1990.

[43] H. F. Korth und A. Silberschatz. Database Systems concepts. Mc Crower Vill,inc., 1990.

[44] K. Scholtyssik. Laufzeitbeobachtung und Evaluierung verteilter Synchronisationsprotokolle. Diplomarbeit an der Rheinisch-Westphälischen technischen Hochschule Aachen, Lehrstuhl für Betriebssysteme, 1998.

[45] V. Hübsch. Transaktionensbasierter globaler gemeinsamer Speicher - ein Modell zur verteilten fehlertoleranten Programmierung.Shaker Verlag.1997.

[46] F. Cristian. Understanding Fault-Tolerant Distributed Systems. IBM Research Report RJ 6980, Juni 1990.

[47] L. Lamport, R. Shostak und M. Pease. The Byzantine Generals Problem. ACM Transaction on Programming Languages ans Systems 4(3), p. 382-401. 1982.

[48] P. Jalote. Fault Tolerance in Distributed Systems. Prentice-Hall, Englewood Cliffs, New Jersey, 1994.

[49] F. Belli, K. Echtle und W. Görke. Methoden und Modelle der Fehlertoleranz. Informatik Spektrum 9, p. 68-81, 1986.

[50] K. S. Trivedi. Probability and Statistics with Reliability, Queuing and Computer Science Applications. Printice Hall, Eglewood Cliffs, New Jersey, 1982.

[51] R. D. Schlichting und F. B. Schneider. Fail-stop processors: An Approach to Designing Fault-Tolerant Computing Systems. ACM Transaction on Computing Systems 1(3), p. 222-238, 1983.

[52] F. B. Schneider. Byzantine Generals in Action: Implementing Fail-Stop Processors. ACM Transaction on Computing Systems 2(2), p. 145-154, 1984.

[53] A. Tanenbaum und R. van Renesse. Reliability Issues in Distributed Operating Systems. Proc. 6th Symposium on Teliability of Distributed Software and database Systems, p. 3-11, 1987.

Eigenständigkeitserklärung:

Ich versichere, daß ich diese Arbeit selbstständig erstellt und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Einverständniserklärung:

Hiermit erkläre ich mein Einverständnis, daß diese Arbeit in der Bibliothek des Institutes für Informatik der Humboldt-Universität zu Berlin öffentlich ausgelegt wird.

13. Dezember 1999 (Claus Wagner)