Studienarbeit:

Bewertung von Algorithmen zum gegenseitigen Ausschluß.

Von Claus Wagner.

Betreuer: Dr. Frank Müller.

Fachbereich Informatik, Lehrstuhl für Softwaretechnik und
Theorie der Programmierung II.

Humboldt-Universität zu Berlin, im November 1998.

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

1 Einleitung

Gegenseitiger Ausschluß ist ein wichtiges Konzept, um den Zugriff auf 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 der Bewertung von verschiedenen verteilten tokenbasierenden Algorithmen, die den gegenseitigen Ausschluß in einer kritischen Sektion zusichern.
Das Privileg, die kritische Sektion zu betreten, wird bei den in dieser Arbeit vorgestellten Algorithmen durch das sogenannte Token repräsentiert. Der Prozeß, der das Token besitzt, und nur dieser, hat uneingeschränkten Zugriff auf die gemeinsamen Ressourcen. Die verteilten Algorithmen müssen, um korrekt zu sein, den gegenseitigen Ausschluß (mutual exclusion) garantieren, frei von Verklemmungen sein (deadlock free) und das Verhungern einzelner Prozesse ausschließen (absence of starvation).
Kapitel 2 motiviert die vorliegende Arbeit. In Kapitel 3 werden Hintergrundarbeiten vorgestellt und ein Überblick über den Stand der Forschung gegeben. Die verwendete Testumgebung, die Voraussetzung für eine komparative Bewertung ist, ist in Kapitel 6 beschrieben. In Kapitel 4 werden vier Algorithmen für den verteilten gegenseitigen Ausschluß formal, an Hand eines Beispiels und im Pseudocode vorgestellt. In diesem Kapitel befinden sich auch informelle Beweise des gegenseitigen Ausschlusses, der Vermeidung von Verklemmungen und des Ausschlusses von Verhungern eines Prozesses, die die Korrektheit der Algorithmen ausmachen.
Die Algorithmen wurden auf Anzahl der Nachrichten pro kritischer Sektion und die Zeit vom Anfragen des Tokens bis zum Eintritt in die kritische Sektion in der Testumgebung untersucht.
Kapitel 7 fast die Parameter und Ergebnisse grafisch und tabellarisch zusammen und bewertet die Ergebnisse.

2 Motivation

Ein verteiltes System ist eine Menge von Prozessen, die keinen gemeinsamen Speicher nutzen und untereinander durch das Senden und das Empfangen von Nachrichten kommunizieren. Das zugrundeliegende Netzwerk ist logisch abgeschlossen.
Neben dem Problem der Kommunikation spielt das Problem der Synchronisation in verteilten Systemen eine entscheidende Rolle. Hauptanforderung, neben Sicherheit und Fehlertoleranz, ist ein möglichst geringer Overhead, um den Gewinn der Rechenleistung durch den Einsatz mehrerer Prozessoren oder Clustern von Workstation nicht zu schmälern.
Dies erfordert eine Untersuchung der verschiedenen Algorithmen in einer homogenen Testumgebung.

[1] In der Literatur sind auch die Bezeichnungen Task, Node, Site und Knoten üblich.

Nimmt man einen konstanten Zeitverlust D durch das Senden und Empfangen einer Nachricht im Netz an, genügt es, die mittlere Anzahl der Nachrichten pro kritischer Sektion (nmE) in Abhängigkeit vom Algorithmus festzuhalten.
Um die Effizienz der einzelnen Algorithmen zu vergleichen, mißt man die mittlere Zeit zwischen dem Zeitpunkt des Anfragens des Tokens (tr) und dem Zeitpunkt des Eintretens (te) in die kritische Sektion. Diese Zeitspanne wird als Anfragezeit (Treq) bezeichnet.
Die Performance in Abhängigkeit vom Algorithmus P(algo) ergibt sich somit für die Algorithmen folgendermaßen:

$$P(\text{algo}) = \overline{nmcs} \cdot D + \overline{(te - tr)}.$$

3 Hintergrundarbeiten

Eine Vielzahl von Lösungen für das Problem des gegenseitigen Ausschlusses in verteilten Systemen wurden in den letzten Jahren vorgestellt. Man unterscheidet normalerweise zwei Ansätze für Lösungen: tokenbasierende und nicht tokenbasierende.
Bei den tokenbasierenden Algorithmen entscheidet der Besitz eines Tokens über das Privileg, exklusiven Zugriff auf gemeinsame Variablen innerhalb einer kritischen Sektion zu haben. In einem System existiert genau ein Token. Durch Weitergabe des Tokens, auf möglichst faire Weise, kann jeder Knoten irgendwann seine kritische Sektion betreten. Tokenbasierende Algorithmen werden in [NT87, NLM90, Ra89, RA83, SK85 und TN87] vorgestellt.
Der nicht tokenbasierende Ansatz verlangt von einem Knoten, der in die kritische Sektion eintreten will, die Erlaubnis dazu von einer Untermenge[2] der Knoten des Systems einzuholen. Zu den nicht tokenbasierenden Algorithmen zählen die in [CR83, Lam78, Ma85, RA81, San87 und Si92] vorgestellten Arbeiten.
Eine weitere Einteilung von Algorithmen für den gegenseitigen Ausschluß kann in dynamisch und nicht dynamisch vorgenommen werden. Hängen die Aktionen eines Knotens, um eine kritische Sektion zu betreten, vom momentanen Zustand des Systems ab, ist der Algorithmus dynamisch, anderenfalls ist er nicht dynamisch.
Tokenbasierende Algorithmen können wiederum in zwei Klassen eingeteilt werden: auf logische 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 logischen Strukturen basierenden Algorithmen unterteilen sich in zwei Gruppen. Es gibt Algorithmen, die auf statischen, und Algorithmen, die auf dynamischen, logischen Strukturen basieren. Bei den Erstgenannten ändern sich die Zeiger, die die Kommunikationswege zwischen den Knoten bestimmen, nicht. Bei den auf dynamischen logischen Strukturen basierenden Algorithmen können sich die Zeiger und die Richtungen der Zeiger ändern.

[2] Diese Untermengen werden oft als Quorums bezeichnet.

Einen Überblick über die Einteilung von Algorithmen für den gegenseitigen Ausschluß gibt der Graph in Abbildung 1. Die inneren Knoten sind die Algorithmenklassen. In den Blättern befinden sich die Verweise auf die Veröffentlichungen.
Die fett eingerahmten Felder zeigen die Klassen der in dieser Arbeit behandelten Algorithmen.

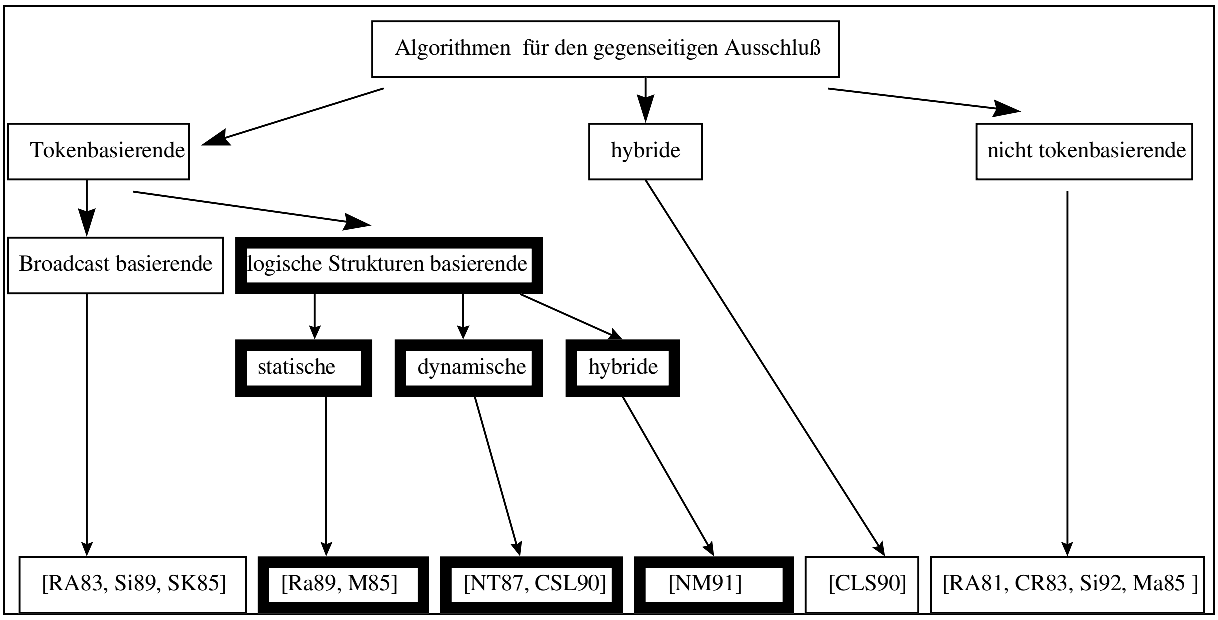


Abbildung 1: Einteilung von Algorithmen für den gegenseitigen Ausschluß.

Algorithmen mit O(n) Nachrichten für den Eintritt in eine kritische Sektion wurden von Lamport in [Lam78] vorgestellt. Maekawas [Ma85] Algorithmus braucht O(sqrt(n)), Kumars [Ku91] O(n^0.63) Nachrichten. Chang, Singhal und Liu präsentierten einen Algorithmus in [CSL90], der die Anzahl von Nachrichten auf O(log n) reduziert.
Der einfache Algorithmus von Raymond, vorgestellt in [Ray89] benötigt bei entsprechender Konfiguration O(log n) Nachrichten. Der Algorithmus von Neilsen und Mizuno [NM91], der eine verteilte Warteschlange der auf das Token wartenden Knoten benutzt, ist eine Weiterentwicklung von Raymonds Algorithmus. Die statische Struktur in Form eines Rings nutzt Martin [Ma85] für seinen Algorithmus, der O(n) Nachrichten benötigt.
Die Algorithmen von Chang, Singhal und Li, der Algorithmus von Neilsen und Mizuno, der Algorithmus von Raymond sowie der Ringalgorithmus von Martin werden in dieser Arbeit auf ihre praktische Performance untersucht.
Raymonds Algorithmus basiert auf einer statischen Baumstruktur und wird desweiteren als SBA (statischer Baumalgorithmus) abgekürzt.
Der Ringalgorithmus von A. J. Martin wird im Folgenden mit SRA (statischer Ringalgorithmus) bezeichnet, da er auf einem statischen logischen Ring basiert.
Der auf einer dynamische Struktur mit einer verteilten Warteschlange basierende Algorithmus von Chang, Singhal und Liu bzw. Naimi und Trehel wird im Folgenden mit DVA (dynamische Struktur mit verteilter Warteschlange) abgekürzt.

Der Algorithmus von Neilsen und Mizuno wird in den folgenden Kapiteln durch SVA (Algorithmus auf einer statischen Struktur basierend mit einer verteilten Warteschlange) abgekürzt.

4 Methoden zum gegenseitigen Ausschluß

Die behandelten Algorithmen sind dezentralisiert und tokenbasierend. Das heißt, es gibt keine ausgezeichnete Task, die spezielle, den gegenseitigen Ausschluß sichernde Aufgaben erfüllt. Das Recht, die kritische Sektion zu betreten, hängt vom Besitz des sogenannten Tokens ab. Das Token und die Anfragen werden entlang der Pfade zwischen den Tasks versendet. Physisch ist das Token ein Boolean Tj je Knoten.
Die Pfade sind im Falle von SBA, SVA sowie SRA statisch. Beim DVA handelt es sich um eine dynamische Struktur. Eine Übersicht über die Algorithmen gibt Tabelle 1. N bezeichnet hierbei die Anzahl der Tasks.
Der SBA und der SRA haben eine lokale Warteschlange, die Größe ist kleiner der Anzahl der Tasks N. Bei den Algorithmen mit verteilter Warteschlange hält jeder Knoten einen Teil der Warteschlange, dieser Teil ist repräsentiert durch einen Zeiger (im Folgenden durch „next" bezeichnet). Die Größe der Nachrichten ist bei allen Algorithmen konstant. Die Struktur bezeichnet den Weg der Tokenanfragen und des Tokens durch das System von Tasks.
Der DVA benutzt eine dynamische Struktur. Beim SBA und SVA wird ein binärer Baum benutzt. Dieser Baum ist in dem Sinn statisch, daß keine Zeiger gelöscht werden und keine neuen Zeiger hinzukommen. Er ändert lediglich die Richtungen der Zeiger. Der SRA arbeitet auf einem statischen, logischen Ring.

	Raymond (SBA)	Neilsen und Mizuno (SVA)	Chang, Liu und Singhal bzw. Naimi und Trehel (DVA)	Martin (SRA)
Warteschlange	lokal, < N	verteilt	verteilt	lokal, < N
Struktur	statisch	statisch	dynamisch	statisch

Tabelle 1: Die Methoden im Überblick.

Im Folgenden bezeichnet *dir* die Richtung in der Tokenanfragen gesendet werden bzw. beim SBA auch die Richtung des Tokens. Beim DVA und SVA bezeichnet *next* die Richtung des Tokens. Der eigene Knoten wird durchgängig durch *self* identifiziert. Der Absender einer Tokenanfrage wird desweiteren als *requester* bezeichnet. *UpTask* ist der Knoten der sich in dem statischen Baum über *self* befindet. Beim SBA und SVA wird *UpTask* zur Initialisierung der Mutexstruktur genutzt. Bei der Beschreibung des Ringalgorithmus SRA bezeichnet *pred* die Richtung in die das Token gesendet wird, *succ* steht für die Richtung der

Tokenanfragen. Die *id* ist ein eindeutiger Bezeichner des Knotens. *Sender* steht für den direkten Absender der Nachricht.
Queue bezeichnet beim SBA und SRA die lokalen Warteschlangen. Das Entnehmen eines Eintrags wird durch ***pop()***, das Hinzufügen eines Elements durch ***push()*** bezeichnet.

4.1 Der statische Baumalgorithmus (SBA)

Raymond stellte 1989 einen tokenbasierenden Algorithmus für den gegenseitigen Ausschluß [Ray89] vor. Dieser Algorithmus benutzt eine statische Baumstruktur. Jeder Knoten hält eine lokale Warteschlange, in der entweder er selbst oder einer seiner Nachbarn eingetragen ist. Weiterhin hat jeder Knoten einen Zeiger „dir", der auf die Wurzel des Unterbaums verweist, in dem sich das Token befindet.
Wenn ein Knoten, der nicht der Tokenbesitzer ist, in seine kritische Sektion eintreten will, sendet er eine Tokenanfrage in Richtung „dir" und trägt sich selber in die Warteschlange ein. Wenn ein Knoten, der nicht das Token hält, eine Tokenanfrage empfängt, trägt er den anfragenden Knoten in seine Warteschlange ein und sendet eine Tokenanfrage in Richtung des „dir"-Zeigers. Wenn ein Knoten, der das Token hält und nicht in seiner kritischen Sektion ist, eine Tokenanfrage empfängt, sendet er das Token sofort an den fragenden Knoten und setzt seinen „dir"-Zeiger auf ihn. Wenn ein Knoten der sich in seiner kritischen Sektion befindet, eine Anfrage erhält, trägt er den anfragenden Knoten in seine Warteschlange ein.
Verläßt ein Knoten die kritische Sektion, so nimmt er den Knoten, der an erster Stelle in der Warteschlange steht, aus der Warteschlange und sendet das Token zu diesem. Danach testet er ob die Warteschlange leer ist. Ist sie nicht leer, sendet der Knoten eine erneute Tokenanfrage in Richtung „dir". Empfängt ein Knoten das Token, so sendet er ihn an den ersten in der Warteschlange bzw. wenn er selber an erster Stelle der Warteschlange steht, betritt er seine kritische Sektion.
Bei dem statischen Baum handelt es sich bei den Tests um einen binären Baum. Der Algorithmus ist in Abbildung 2 im Pseudocode skizziert. 4.1.1 stellt den Algorithmus an Hand eines Beispiels vor.

```
Task_init(id,UpTask):
        if (id =0)
                Token := TRUE;
                Dir     := NULL;
        else
                Token := FALSE;
                Dir     := UpTask;
        fi
        Queue := NULL;
end Task_init.


Mutex_lock():
        if (Token)
```

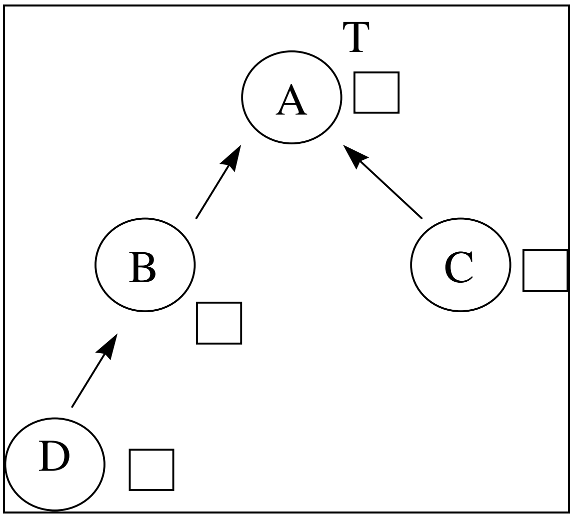
```
		Enter:= TRUE;
	else
		send_request(Dir);
		push(Queue,self);
		await (Token);
		Enter := TRUE;
	fi
end Mutex_lock.



Mutex_unlock(id):
	Enter := FALSE;
	if (Queue != NULL)
		Task := pop(Queue);
		send_token(Task);
		Dir := Task;
		Token := FALSE;
		if (Queue != NULL)
			send_request(Dir);
	fi;
end Mutex_unlock(id).



Beim Empfangen von Token_request(Task):
	if (Token = FALSE)
		send_request(Dir);
		push(Queue,Task);
	else
		if (Enter = FALSE)
			send_token(Task);
			Dir := Task;
			Token := FALSE;
		else
			pop(Queue,Task);
		fi
	fi
end.
```

```
Beim Empfangen von Token_Send:
        Task := pop(Queue);
        if (Task = id)
                Token := TRUE;
                Dir     := NULL;
        else
                send_token(Task);
                Dir := Task;
                if (Queue != NULL)
                        send_request(Dir);
        fi;
end.
```

Abbildung 2: Der statische Baumalgorithmus (SBA) im Pseudocode.

4.1.1 Beispiel

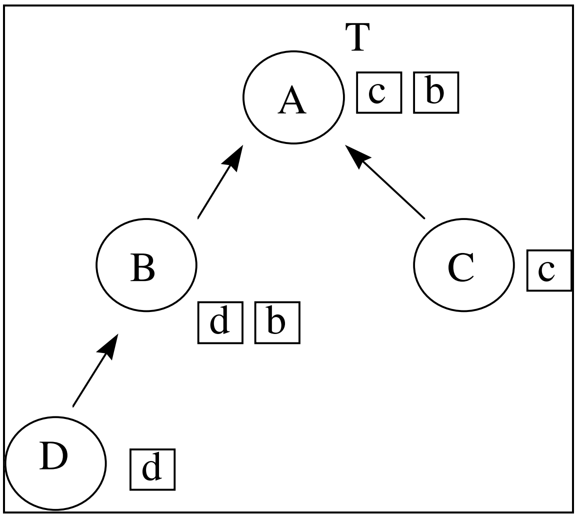
Sei ein Initialzustand für ein System von vier Knoten gegeben wie folgt:



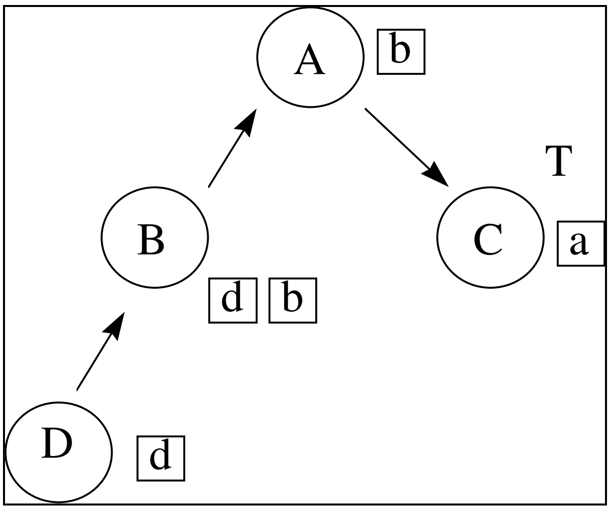
Die Kreise bezeichnen die Knoten (A,B,C und D), die Pfeile die Zeiger „dir“. Die Vierecke an jedem Knoten repräsentieren die Warteschlange und das „T“ steht für das Token.
Initial ist A der Besitzer des Tokens, die Zeiger „dir“ der Knoten zeigen in Richtung des Tokenbesitzers.
Knoten A führt ein ***mutex_lock()***[3] aus, testet das Tokenflag und betritt, da dieser Test positiv ausfällt, die kritische Sektion.
Nacheinander fragen C, D und B das Token an. C sendet eine Tokenanfrage zu A und trägt sich in seine Warteschlange ein. A erhält die Anfrage von C und trägt C in die Warteschlange ein. Als nächstes erhält B die Tokenanfrage von D. B sendet

[3] Die Bezeichnung mutex_lock() und mutex_unlock() ist in Anlehnung an die POSIX-Terminologie gewählt. In der Literatur werden auch die Bezeichnungen release_token() und request_token() sowie enter_cs() und release_cs() verwendet.

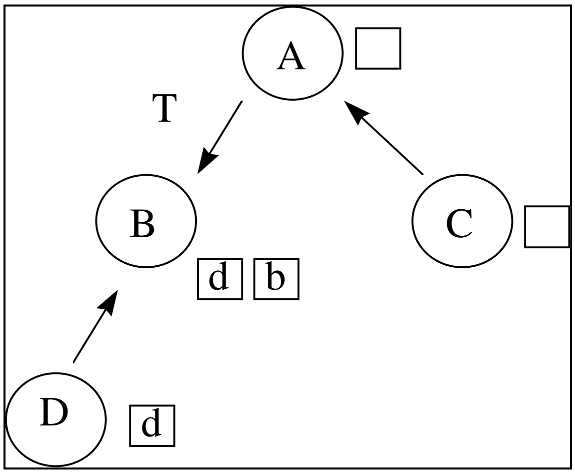
diese Anfrage weiter zu A und trägt D in seine Warteschlange ein. A empfängt die Anfrage von B und trägt B in die Warteschlange ein.
Nun will B selber seine kritische Sektion betreten, sendet eine erneute Anfrage an A. A trägt wiederum B in seine Warteschlange ein. Folgende Konfiguration des Systems entsteht:



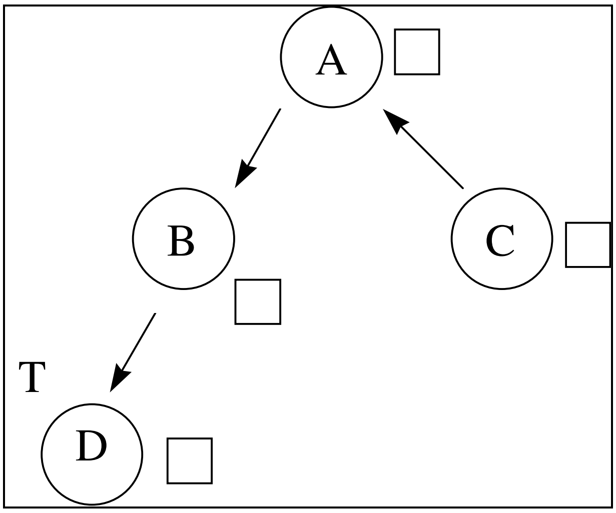
Nun führt A ein ***mutex_unlock()*** aus, nimmt C aus der Warteschlange und sendet das Token zu C. Danach setzt A seinen Zeiger „dir“ auf C. Nach dem Senden des Tokens testet A die Warteschlange, stellt fest, daß diese nicht leer ist, und sendet eine Tokenanfrage in Richtung „dir“.
C erhält das Token, entfernt sich selber aus der lokalen Warteschlange und betritt damit die kritische Sektion. C erhält nun die Anfrage von A, trägt damit A in die Warteschlange ein:



Verläßt nun C seine kritische Sektion, testet er zunächst die Warteschlange und entfernt A aus der Warteschlange. Daraufhin sendet C das Token zu A und setzt den Zeiger „dir“ auf A. A erhält das Token, entfernt B aus der Warteschlange setzt „dir“ auf B und sendet das Token zu B. A stellt jetzt fest, daß die Warteschlange leer ist und sendet somit keine erneute Anfrage:



Das Token gelangt über B, der D aus seiner Warteschlange holt, das Token weitersendet und seinen Zeiger „dir" auf D setzt, zum nächsten Tokenbesitzer (D). D betritt, nachdem er sich selbst in der Warteschlange gefunden hat, seine kritische Sektion. Nach dem Verlassen der kritischen Sektion behält D das Token, ohne es zu nutzen.



Das Token und die Tokenanfrage wandern auf dem Weg zwischen zwei Knoten entlang der statischen Struktur. Die Zeiger, die die Kommunikationsrichtung innerhalb der Struktur vorgeben, ändern lediglich ihre Richtung.

4.2 Der statische Baumalgorithmus mit verteilter Warteschlange (SVA)

Der Algorithmus, vorgestellt von M. L. Neilson und M. Mizuno in [NM91], benutzt im Gegensatz zum SBA eine verteilte Warteschlange. Jeder Knoten hält einen Zeiger „next" der auf den zukünftigen Tokenbesitzer zeigt bzw. gleich null ist, wenn dieser nicht existiert. Die Gesamtheit der „next"- Zeiger baut eine verteilte Warteschlange auf. Das Token wird nicht entlang des Pfades „dir", sondern direkt zum nächsten Tokenbesitzer gesendet. Im Zeiger „dir" wird die Richtung in der sich das Ende der verteilten „next"- Warteschlange befindet gespeichert, d.h. „dir" zeigt auf die Wurzel des Unterbaums in der sich das Ende der verteilten Warteschlange befindet. Tokenanfragen werden wiederum in Richtung „dir“ gesendet und gelangen so an das Ende der „next“-Warteschlange. Die Zeiger „dir“ bilden eine statische Baumstruktur. Bei den Tests wurde ein binärer Baum verwendet. 4.2.1 zeigt eine Beispielausführung. Der Algorithmus ist in Abbildung 3 im Pseudocode skizziert.

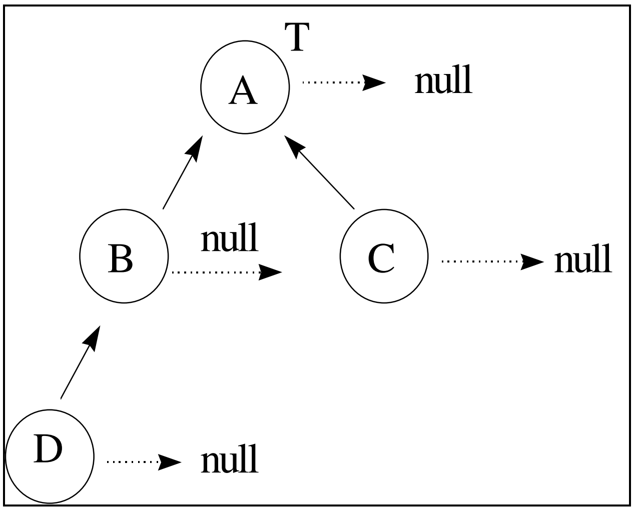
```
Task_Init(id,UpTask):
        if (id = 0)
                Token  := TRUE;
                Dir    := NULL;
        else
                Token  := FALSE;
                Dir    := UpTask;
        fi;
        Enter  := FALSE;
        Next   := NULL;
end Task_Init.


Mutex_lock:
        if (Token = TRUE)
                Enter := TRUE;
        else
                send_request(Sender,Requester) to Dir;
                Dir := NULL;
                await(Token);
                Enter := TRUE;
        fi;
end Mutex_lock.


Mutex_unlock:
        Enter := FALSE;
        if (next != NULL)
                send_token() to next;
                next := NULL;
                Token := FALSE;
        fi;
end Mutex_lock.


Beim Empfang von TOKEN_SEND:
        Token := TRUE;
end.


Beim Empfangen von TOKEN_REQUEST(Sender,Requeser):
        if (Token = TRUE)
                if  (Enter = TRUE)
                        Next   := Requester;
                        Dir    := Sender;
                else
                        Dir    := Sender;
```

```
                Token := FALSE;
                send_token() to Requester;
            fi;
    else
            send_request(Sender,Requester) to Dir;
            Dir := Sender;
    fi;
end.
```

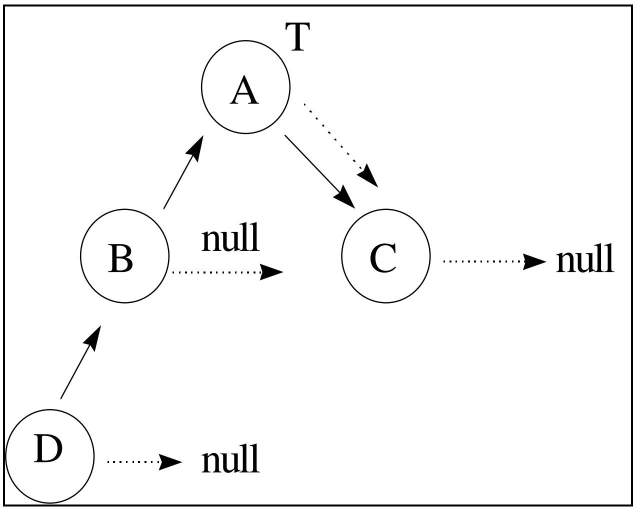
Abbildung 3: Der Algorithmus von Neilsen und Mizuno (SVA) im Pseudocode.

4.2.1 Beispiel

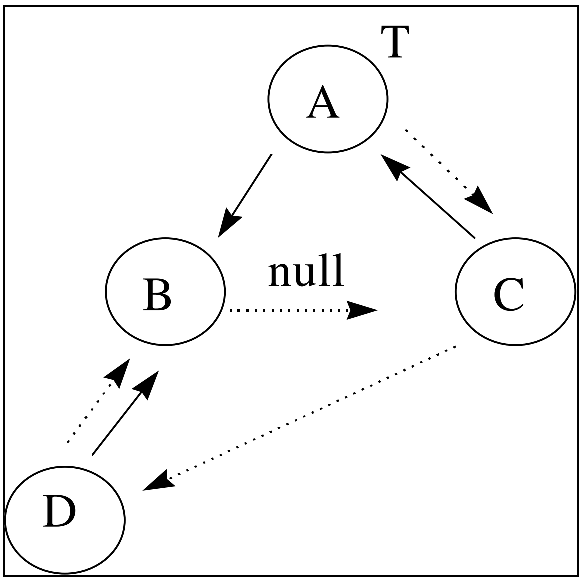
Die Knoten sind wiederum in einem binären Baum angeordnet und der initiale Tokenbesitzer ist A. Durchgehend gezeichnete Pfeile bezeichnen die Zeiger „dir", die auf den Tokenbesitzer bzw. auf das Ende der Warteschlange zeigen. Die gestrichelten Pfeile repräsentieren die Warteschlange, d.h. den Zeiger „next". Dieser ist initial auf NULL gesetzt. Das „T" bezeichnet das Token.
Die Ausgangssituation für vier Knoten, A,B,C und D zeigt sich wie folgt :



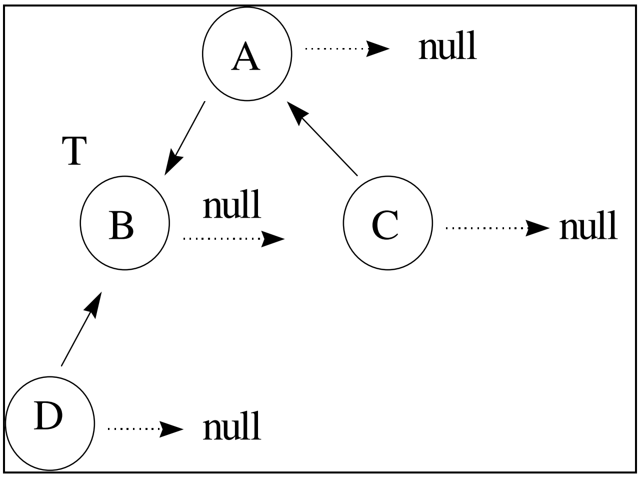
Knoten A führt als erstes die Funktion ***mutex_lock()*** aus, testet den Besitz des Tokens und betritt, nachdem dieser Test positiv ist, seine kritische Sektion.
Nacheinander führen nun C, D und B ein ***mutex_lock()*** aus. C sendet nach dem negativen Test des Tokens eine Tokenanfrage in Richtung „dir". Die Anfrage gelangt zu A. A ist der Tokenbesitzer und setzt somit „next" und „dir" auf C:



Die Tokenanfrage von D erreicht über B und A das Ende der verteilten Warteschlange (C). B setzt seinen Zeiger „dir“ auf D und A setzt „dir“ auf B. C setzt seinen Zeiger „next“ auf D und den Zeiger „dir“ auf A. „next“ zeigt auf den nächsten Tokenbesitzer. Die Zeiger „dir“ zeigen somit weiterhin auf den vermeintlich nächsten Tokenbesitzer bzw. auf die Wurzel des Unterbaums in dem sich das Ende der Warteschlange befindet.
Die Anfrage von B geht somit zu D. D setzt seine Zeiger „dir“ und „next“ auf B :



A führt nun die Funktion ***mutex_unlock()*** aus und testet den Zeiger „next“. Dieser ist nicht null. Daher sendet A das Token nach „next“, d. h. zu C, und setzt danach „next“ auf null. C erhält das Token und betritt seine kritische Sektion. Nacheinander gelangt das Token nun entlang der verteilten „next“- Warteschlange zu D und B. Verläßt B seine kritische Sektion, testet er seinen Zeiger „next“, stellt fest, daß dieser null ist, und behält somit das Token, ohne ihn zu nutzen:



Das Token wandert nicht, wie beim SBA, durch die statische Ordnung der Knoten, gegeben durch „dir“, sondern gelangt mit einer Nachricht direkt zum nächsten Tokenbesitzer.

4.3 Der dynamische Algorithmus mit verteilter Warteschlange (DVA)

Einen Algorithmus mit verteilter Warteschlange und dynamischer Struktur, der die Anzahl der Nachrichten je kritische Sektion auf O(log n) reduziert, wurde von Y. I. Chang, M. Singhal und M. T. Liu in [CLS90,CLS90/1] vorgestellt.
Unabhängig von Chang et al. wurde der gleiche Algorithmus von M. Naimi und M. Trehel entwickelt. M. Naimi und M. Trehel veröffentlichten ihre Ergebnisse in [NT87,NT96].
Jede Task hält einen Zeiger „dir“, einen Zeiger „next“, sowie ein Boolean Token. „dir“ ist die Richtung, in der Tokenanfragen gesendet werden. Der Zeiger „next“ bildet eine verteilte Warteschlange, der nächsten Tokenbesitzer. Der Knoten, der das Token hält, bildet den Kopf der verteilten Warteschlange. Der letzte Knoten in dieser Warteschlange ist der Knoten, der als letztes eine Tokenanfrage gesendet hat. Wenn eine Anfrage das Ende der verteilten Warteschlange, erreicht gibt es drei Möglichkeiten und Reaktionen:

1. Der Knoten am Ende der Warteschlange hat das Token und ist in seiner kritischen Sektion. Der anfragende Knoten wird dann das neue Ende der verteilten Warteschlange.
2. Der Knoten am Ende der Warteschlange hat das Token und ist nicht in seiner kritischen Sektion. Dann wird das Token sofort an den anfragenden Knoten gesendet.
3. Der Knoten wartet auf das Token. Dann wird der anfragende Knoten das neue Ende der Warteschlange.

Wenn der Kopf der Warteschlange seine kritische Sektion verläßt, sendet er das Token an „next“ und setzt danach „next“ auf NULL.
Erhält ein Knoten eine Tokenanfrage, setzt er den Zeiger „dir“ auf den Absender der Anfrage.
Die Wurzel der Struktur aus den „dir“- Zeigern ist das Ende der Warteschlange. Initial hat diese Struktur die Form eines Sterns. Der Algorithmus ist in Abbildung 4 im

Pseudocode skizziert. In 4.3.1 befindet sich ein Beispielszenario, das den Algorithmus illustriert.

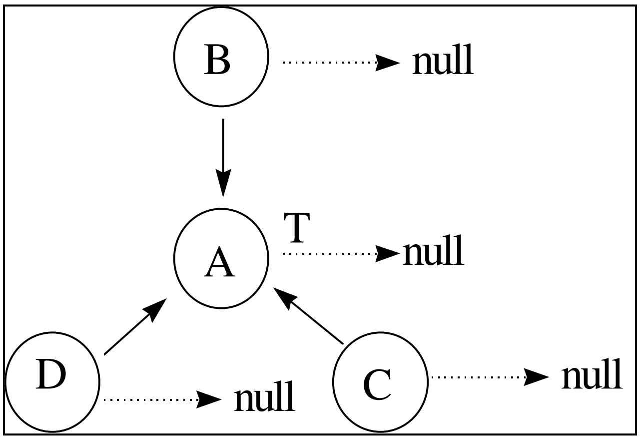
```
Task_init(id):
        Requested    := FALSE;
        Next         := NULL;
        if (id = 0)
                Token  := TRUE;
                Dir    := NULL;
        else
                Token  := FALSE;
                Dir    := 0;
        fi;
end Task_init.


Mutex_lock:
        if (Token = FALSE)
                requested := TRUE;
                if (Dir != NULL)
                        send_request(Dir);
                        Dir := NULL;
                fi;
                await(Token);
        fi;
end Mutex_lock.


Mutex_unlock:
        Requesting := FALSE;
        if (Next != NULL)
                send_token(Next);
                Token  := FASLE;
                Next   := NULL;
        fi;
end Mutex_unlock.


Beim Empfangen von Token_Request(Task):
        if (Dir = NULL)
                if (Requesting = TRUE)
                        next := Task;
                else
                        send_token(Task);
                fi;
        else
                send_request(Dir);
        fi;
        Dir := Task;
end.


Beim Empfangen von Token_Send
```

```
        Token := TRUE;
end.
```

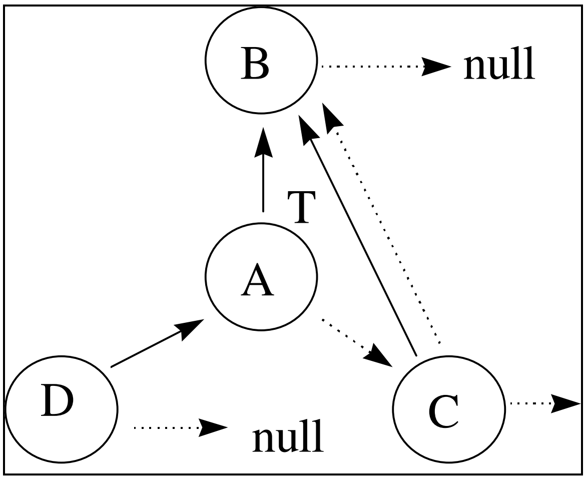
Abbildung 4: Der Algorithmus von Chang, Singhal und Liu bzw. von Naimi und Trehel (DVA) im Pseudocode.

4.3.1 Beispiel

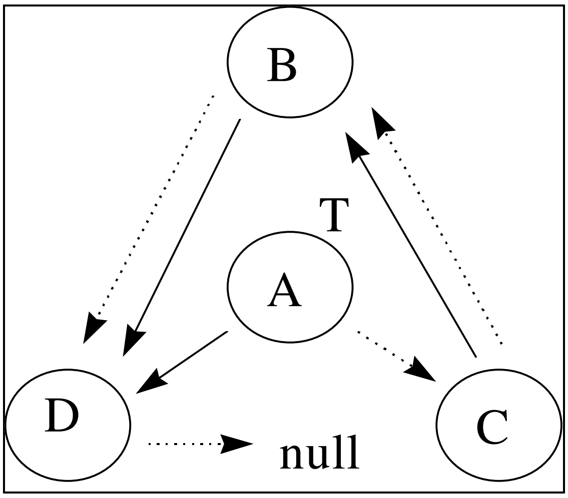
Der Zeiger „dir“ wird in den folgenden Abbildungen durch durchgehend gezeichnete Pfeile und der Zeiger „next“ durch gestrichelte Pfeile dargestellt. Das „T“ repräsentiert das Token.
Wiederum besteht das System aus vier Knoten, die nacheinander das Token anfordern. Der Initialzustand zeigt sich wie folgt: A ist der initiale Tokenbesitzer, sein Zeiger „dir“ ist null. Die „dir“- Zeiger der anderen Knoten zeigen auf A. Die Zeiger „next“ sämtlicher Knoten sind ebenfalls null:



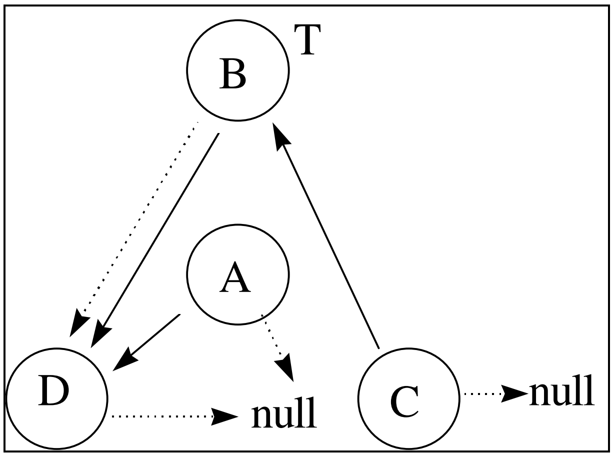
Knoten A ruft die Funktion ***mutex_lock()*** auf, testet das Token und betritt als Besitzer des Tokens die kritische Sektion. Als zweiter ruft Knoten C ***mutex_lock()***, testet das Token und schickt daraufhin eine Tokenanfrage zu „dir“, d.h. zu A. C setzt danach den Zeiger „next“ auf null. A empfängt die Nachricht von C, stellt fest, daß er sich in seiner kritischen Sektion befindet und setzt somit den Zeiger „dir“ und den Zeiger „next“ auf C.
Nun führt Knoten B die Funktion ***mutex_lock()*** aus. B testet das Token, der Test ist negativ, somit sendet B eine Tokenanfrage an A und setzt „dir“ auf null. A erhält die Anfrage des Tokens von B, sendet diesen weiter in Richtung „dir“, d. h. zu C und setzt dann den Zeiger „dir“ auf B. C erhält nun die Anfrage von B und testet „dir“. „dir“ ist gleich null und somit setzt C „next“ und „dir“ auf B:



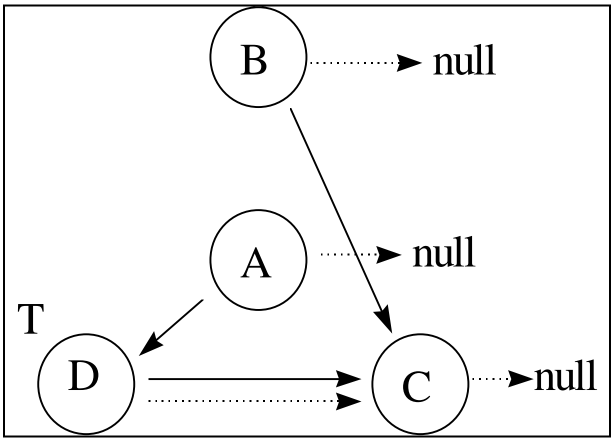
Nun ruft Knoten D die Funktion ***mutex_lock()*** und testet das Token. Nachdem dieser Test negativ ist, sendet Knoten D eine Tokenanfrage in Richtung des Zeigers „dir“, also zu A. Knoten D setzt danach „dir“ auf null. A sendet diese Anfrage weiter in Richtung „dir“, und setzt dann „dir“ auf den Sender der Anfrage, d. h. auf D.
B empfängt die Tokenanfrage und testet den Zeiger „dir“. Nachdem B festgestellt hat, daß „dir“ gleich null ist, setzt er „dir“ und „next“ auf D:



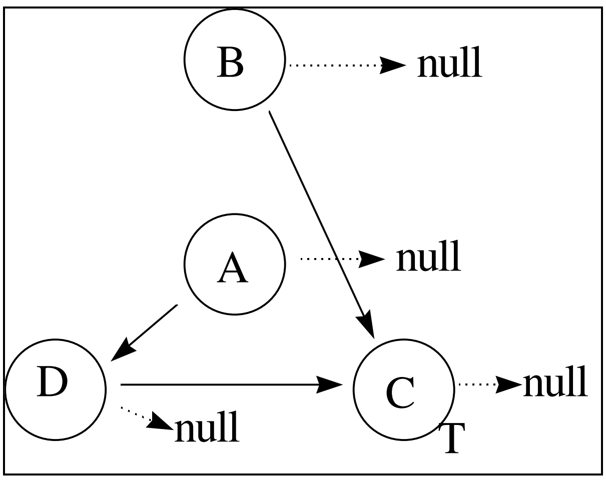
A verläßt nun seine kritische Sektion, testet den Zeiger „next“, stellt fest, daß dieser nicht null ist und sendet somit das Token in Richtung „next“, d. h. zu C. Nach dem Senden weist A „next“ null zu.
C empfängt das Token und betritt seine kritische Sektion, arbeitet diese ab und ruft die Funktion ***mutex_unlock().*** C stellt fest, daß „next“ ungleich null ist, sendet somit das Token zu B und setzt dann „next“ auf null:



Nun führt C erneut ein ***mutex_lock()*** aus, setzt „dir“ auf null und sendet eine Tokenanfrage an B. B sendet diese Anfrage weiter zu D und setzt seinen Zeiger „dir“ auf C. D empfängt die Tokenanfrage von C, stellt fest, daß „next“ gleich null ist und setzt somit den Zeiger „dir“ und den Zeiger „next“ auf Knoten C.
B verläßt seine kritische Sektion, sendet das Token zu D und setzt „next“ auf null. D betritt seine kritische Sektion:



Nun verläßt D seine kritische Sektion, sendet das Token zu C und setzt den Zeiger „next“ auf null. C erhält das Token, arbeitet seine kritische Sektion ab und testet mit dem Verlassen „next“, stellt fest das „next“ gleich null ist und behält so das Token ohne es zu benutzen:



Das Token wird direkt zum nächsten Tokenbesitzer gesendet. Die Pfade der Tokenanfragen verändern sich mit den möglichen Anfragereihenfolgen.

4.4 Der statische Ringalgorithmus (SRA)

Beim SRA, vorgestellt in [Ma85], bilden die Knoten einen logischen Ring. Jeder Knoten n aus N kennt seinen Vorgänger ((n-1) mod N) und seinen Nachfolger ((n+1) mod N). Weiterhin hält jeder Knoten eine lokale Warteschlange mit Werten vom Typ TStatus = {ST_SUBS, ST_WAIT}.
Der Nachfolger eines Knotens gibt die Richtung, in der eine Tokenanfrage gesendet wird, an. Die Richtung des Tokens wird von den Vorgängern bestimmt.
Wenn ein Knoten seine kritische Sektion betreten will und nicht der Tokenbesitzer ist, schickt er eine Tokenanfrage an seinen Nachfolger und schreibt ST_WAIT in seine Warteschlange.
Wenn ein Knoten eine Tokenanfrage empfängt, ohne der Tokenbesitzer zu sein, sendet er ebenfalls eine Tokenanfrage an seinen Nachfolger und trägt ST_SUBS in seine Warteschlange.
Wenn der Tokenbesitzer eine Tokenanfrage erhält und sich nicht in seiner kritischen Sektion befindet, sendet er das Token in Richtung seines Vorgängers.
Befindet sich der Knoten in seiner kritischen Sektion und empfängt eine Tokenanfragen, trägt er ST_SUBS in seine Warteschlange ein.
Beim Verlassen der kritischen Sektion prüft der Knoten die Warteschlange. Ist die Warteschlange leer, so behält er das Token, ohne es zu nutzen. Ist die Warteschlange nicht leer, entfernt er den ältesten Status und sendet das Token an seinen Vorgänger.
Erhält ein Knoten das Token, so testet er den Status im Kopf der Warteschlange. Ist der Status ST_WAIT, so betritt er seine kritische Sektion. Andernfalls sendet er das Token an seinen Vorgänger.
Der Algorithmus ist in Abbildung 1 im Pseudocode skizziert. In Kapitel 4.4.1 befindet sich eine Beispielausführung.

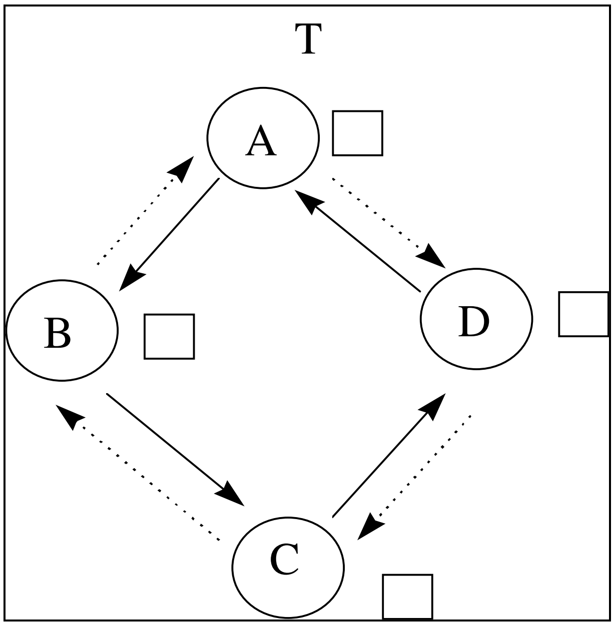
```
Task_init(id,number_of_tasks):
        Queue       : = NULL;
        Token       := FALSE;
        Enter       := FALSE;
        Succ        := (id+1) mod number_of_tasks;
        Pred        := (id -1) mod number_of_tasks;
        if (id = 0)
                Token := TRUE;
        fi;
end Task_init.


Mutex_lock:
        if (Token = TRUE)
                Enter := TRUE;
```

```
        else
                send_request(Succ);
                push(Queue,ST_WAIT);
                await(Token);
                Enter := TRUE;
        fi;
end Mutex_lock.


Mutex_unlock:
        Enter := FALSE;
        if (Queue != NULL)
                pop(Queue),
                send_token(Pred);
                Token := FALSE;
        fi;
end Mutex_unlock.


Beim empfangen von Token_Request:
        if (Token = FALSE)
                send_request(Succ);
                push(Queue,ST_SUBS);
        else
                if (Enter = FALSE)
                        send_token(Pred);
                        Token := FALSE;
                else
                        push(Queue,ST_SUBS);
                fi;
        fi;
end.


Beim empfangen vom Token:
        status := pop(Queue);
        if (status = ST_SUBS)
                send_token(Pred);
        else
                Token := TRUE;
        fi;
end.
```

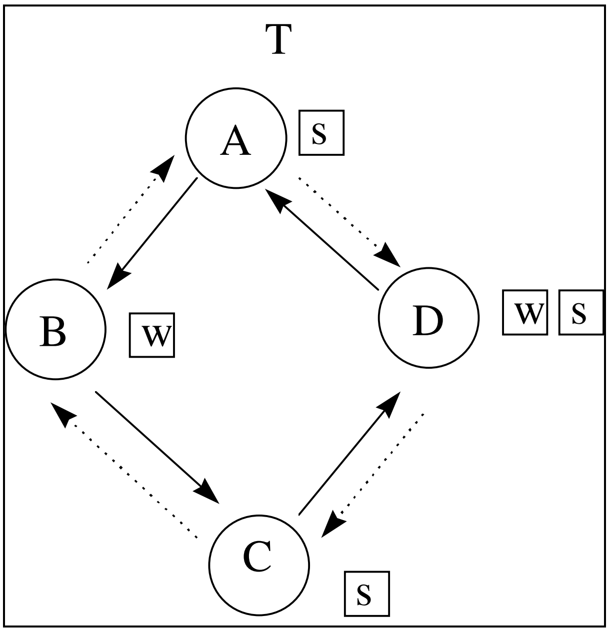
Abbildung 5: Der Algorithmus von A. J. Martin (SRA) im Pseudocode.

4.4.1 Beispiel

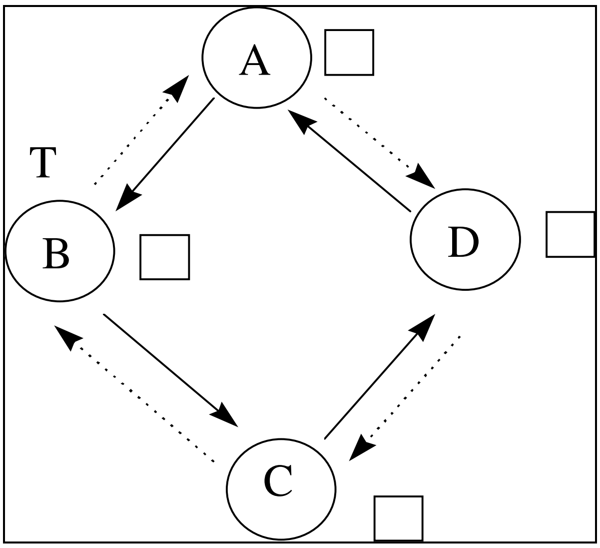
In den folgenden Abbildungen repräsentieren die gestrichelten Pfeile die Richtung einer Tokennachricht und die durchgehend gezeichneten Pfeile die Richtung einer Tokenanfrage. Die Rechtecke an den durch Kreisen dargestellten Knoten stellen die Warteschlange für den Status dar. Die möglichen Einträge in der Warteschlange sind „w“ für den Status ST_WAIT und „s“ für den Status ST_SUBS. Das „T“ repräsentiert das Token.
Betrachtet wird wiederum ein System mit vier Knoten, A, B, C und D. A ist der initiale Tokenbesitzer. Initial zeigt sich folgendes Bild:



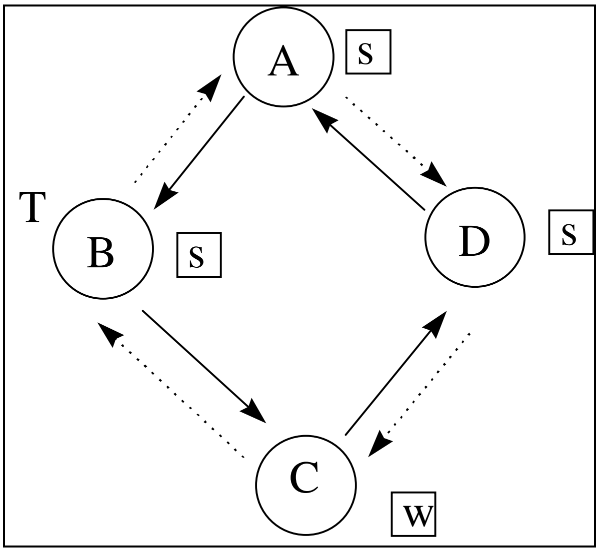
A führt als initialer Tokenbesitzer ein ***mutex_lock()*** aus. Der Test des Tokens ist positiv, daher betritt A seine kritische Sektion. Nun rufen nacheinander D, B, und C die Funktion ***mutex_lock()***.
D sendet eine Tokenanfrage an A und trägt ST_WAIT in seine Warteschlange ein. A empfängt die Anfrage von D und schreibt ST_SUBS in seine Warteschlange. Als nächstes sendet B eine Tokenanfrage und schreibt ST_WAIT in seine Warteschlange. C, als Nachfolger von B, empfängt die Tokenanfrage und sendet, nachdem er ST_SUBS in seine Warteschlange geschrieben hat, weiter an D. D schreibt ebenfalls den Status ST_SUBS in seine Warteschlange und sendet die Anfrage weiter in Richtung seines Nachfolgers (A):



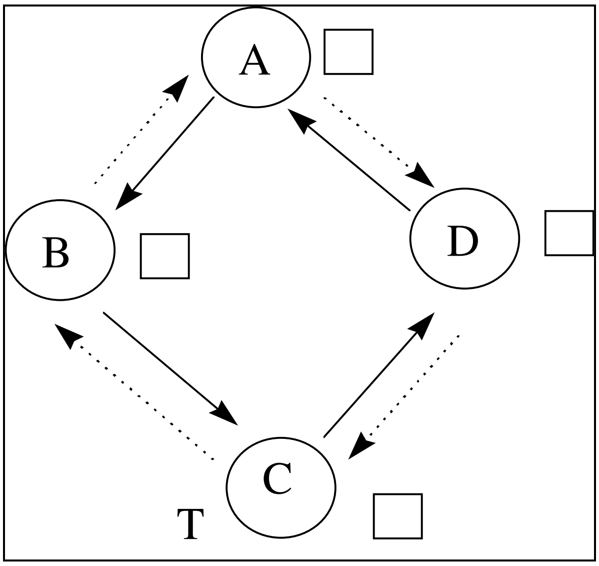
Nun führt A die Funktion ***mutex_unlock()*** aus. A nimmt den ältesten Status aus der Warteschlange und sendet somit das Token zu D.
D empfängt die Tokennachricht, nimmt ein ST_WAIT aus der Warteschlange und beginnt, seine kritische Sektion abzuarbeiten. Hat er diese beendet, testet er die Warteschlange, nimmt ST_SUBS aus seiner Warteschlange und sendet das Token an C.
Knoten C nimmt wiederum ST_SUBS aus der Warteschlange und sendet das Token somit weiter an B. B entfernt ein ST_WAIT aus der Warteschlange und betritt seine kritische Sektion:



Nun führt C ein ***mutex_lock()*** aus, testet das Token und sendet, nachdem dieser Test negativ ist, eine Tokenanfrage an seinen Nachfolger (D).
D trägt daraufhin ein ST_SUBS in seine Warteschlange ein. Die Anfrage gelangt über A zu B, die jeweils ST_SUBS in ihre Warteschlange schreiben:



B verläßt mit einem ***mutex_unlock()*** seine kritische Sektion. Über den Knoten A und D, die jeweils ST_SUBS aus ihrer Warteschlange holen, erreicht das Token Knoten C.
C arbeitet seine kritische Sektion ab, und behält das Token danach, ohne es zu nutzen:



Das Token und die Anfragen werden durch die statisch vorgegebene Struktur des Systems versendet.

5 Beweise

Algorithmen für den gegenseitigen Ausschluß müssen, um korrekt zu sein, frei von Verklemmungen sein (freedom of deadlocks), müssen das Verhungern von Knoten vermeiden (absence of starvation) und schließlich den gegenseitigen Ausschluß garantieren. Es folgt ein informeller Beweis der drei Anforderungen. Die vollständigen formalen Beweise befinden sich für den SBA in [Ra89], für den SRA in [Ma85], für den SVA in [NM91] und für den DVA in [CSL90] bzw. [NT96].

5.1 Gegenseitiger Ausschluß

In allen vier Algorithmen kann ein Knoten nur, nachdem er das Token erhalten hat, seine kritische Sektion betreten. Die Funktion ***mutex_lock()*** kehrt nur dann zurück, wenn das Tokenflag auf TRUE gesetzt wurde. Bis zu diesem Zeitpunkt blockiert der Knoten auf ***await(Token)***.
Es bleibt zu zeigen, daß in dem System immer nur genau ein Token existiert. Initial wird der Token nur beim Knoten mit der id=0 auf TRUE gesetzt. Das Setzen des Tokens auf TRUE ist im weiteren Verlauf der Algorithmen atomar mit dem Empfang des Tokens verbunden. Weiterhin ist das Senden des Tokens immer atomar mit der Anweisung Token := FALSE verbunden.

Aus der Existenz genau eines Tokens und dem blockierenden Ruf ***await(Token)***, folgt der gegenseitige Ausschluß.

5.2 Vermeidung von Verklemmung

Ein System von Knoten befindet sich in einem Zustand der Verklemmung, wenn sich keiner der Knoten in seiner kritischen Sektion befindet und kein Knoten, der das Token anfragt, jemals die Möglichkeit bekommt, seine kritische Sektion zu betreten. Um zu beweisen, daß diese Situation ausgeschlossen ist, muß man zeigen, daß die folgenden zwei Bedingungen gelten:

1. Ein Anfrage erreicht immer den Tokenbesitzer bzw. den vermeintlichen Tokenbesitzer.
2. Der Knoten, der das Token anfragt, erhält dieses letztendlich.

Die erste Bedingung wird verletzt, wenn die Zeiger der Knoten, die den Weg einer Tokenanfrage bestimmen, Zyklen enthalten oder nicht beim Tokenbesitzer bzw. beim vermeintlichen Tokenbesitzer enden. Enthält der Weg des Tokens Zyklen oder ist unterbrochen, ist die zweite Bedingung verletzt.

SBA:
Die Wege der Anfrage werden durch das System der „dir"-Zeiger bestimmt. Initial befindet sich das Token bei Knoten 0 und die „dir"-Zeiger aller anderen Knoten zeigen auf 0 bzw. auf die Wurzel des Unterbaums, in dem sich Knoten 0 befindet. Die Anfragen werden entlang der Zeiger „dir" gesendet. Knoten, die nicht Tokenbesitzer sind, senden die Anfragen in Richtung „dir" weiter. Dieser Mechanismus des Weitersendens endet beim Tokenbesitzer. Im Verlauf des Algorithmus wird der Zeiger „dir" eines Knotens immer nur genau dann verändert, wenn der Knoten das Token verschickt. „dir" wird auf den Knoten gesetzt, zu dem das Token gesendet wurde. Das Senden des Tokens und das Setzen des „dir"-Zeigers sind atomar verbunden. Somit zeigt „dir" immer auf den Tokenbesitzer bzw.

auf die Wurzel des Unterbaums, in dem sich das Token befindet, und die Anfrage wird immer den Tokenbesitzer erreichen.
Erhält der Tokenbesitzer eine Anfrage, sind zwei Fälle zu unterscheiden:

1. Der Tokenbesitzer ist in seiner kritischen Sektion. Daraus folgt, daß er die Anfrage in seine Warteschlange einträgt. Nach dem Verlassen testet er die Warteschlange und sendet das Token an den ersten Eintrag in der Warteschlange.
2. Der Tokenbesitzer hält das Token, ohne es zu nutzen. Daraus folgt, daß er das Token sofort an den Nachbarknoten, von dem er die Anfrage erhalten hat sendet.

Ist der anfragende Knoten ein Nachbar des Tokenbesitzers, so erhält er das Token sofort. Ist er kein Nachbar, hat sich durch den Mechanismus des Weitersendens der Anfrage eine Kette aufgebaut, d.h. auf allen Knoten, die die Anfrage weitergesendet haben, erfolgte ein Eintrag in die lokale Warteschlange. Diesen Weg beschreibt das Token bis zum anfragenden Knoten. Der Weg ist durch die „dir"-Zeiger vorgeschrieben.
Es bleibt zu zeigen, daß der Graph der „dir"-Zeiger zyklenfrei ist.
Initialisiert wird das System der Knoten mit einem binären Baum. Der binäre Baum ist zyklenfrei. Zyklen können nur dann entstehen, wenn weitere Kanten hinzukommen. Die „dir"-Zeiger jedoch ändern im Verlauf des Algorithmus nur ihre Richtung. Daraus folgt, daß keine Zyklen in der Struktur der Zeiger „dir" entstehen können.

SRA:
Der Weg der Anfragen ist durch den Nachfolger der Knoten vorgeschrieben. Die Ringstruktur ist statisch vorgegeben. Nach spätestens (N-1) mal Weitersenden des Knotens hat die Anfrage den Tokenbesitzer erreicht (Bedingung 1).
Das Weitersenden der Anfrage bei den Knoten auf dem Weg der Anfrage ist mit Schreiben des Status ST_SUBS in die Warteschlange verbunden. Der Tokenbesitzer schreibt, wenn er in seiner kritischen Sektion ist ein ST_SUBS in seine Warteschlange, was sicherstellt, daß er das Token beim Verlassen der kritischen Sektion sofort verschickt. Ist der Tokenbesitzer nicht in seiner kritischen Sektion, verschickt er das Token sofort. Die Richtung des Verschickens des Tokens ist durch den Vorgänger eines Knotens statisch vorgeschrieben. Ist der anfragende Knoten der direkte Vorgänger des Tokenbesitzers, erreicht ihn das Token sofort. Beim Weitersenden der Anfrage hat sich durch das mit dem Weitersenden atomar verbundene Eintragen des Status ST_SUBS in die lokale Warteschlange eine Kette, die das Weitersenden des Tokens auf jedem Knoten veranlaßt, gebildet. Somit erreicht das Token den anfragenden Knoten. Die Kette wird beim Weitersenden des Tokens wieder abgebaut.
Da jeder Knoten maximal eine Anfrage zu einem Zeitpunkt hat, ist die Kette, die sich beim Mechanismus des Weitersendens aufbaut, eine einfache Liste und somit zyklenfrei.

SVA:
Den Weg der Tokenanfragen bestimmt das System der Zeiger „dir“ der Knoten. Initial zeigen die „dir“-Zeiger auf den initialen Tokenbesitzer bzw. auf die Wurzel des Unterbaums, in dem sich das Token befindet. Im Verlauf des Algorithmus verändern sich die Zeiger „dir“ mit dem Senden bzw. Weitersenden von Tokenanfragen. Die Operationen „Weitersenden einer Anfrage“ und „ändern von ‘dir’ “ sind atomar verbunden. Wenn aktuelle Anfragen existieren zeigen die „dir“-Zeiger auf die Wurzel des Unterbaums, in der sich der Knoten befindet, der die letzte Anfrage gestellt hat. Existieren keine Anfragen, zeigen die Zeiger „dir“ auf die Wurzel des Unterbaums, in dem sich das Token befindet. Somit erreicht die Anfrage eines Knotens immer entweder den letztmöglichen Tokenbesitzer bzw. den aktuellen Tokenbesitzer.
Die auf das Token wartenden Knoten bauen durch ihre „next“-Zeiger eine verteilte Warteschlange auf. Da jeder Knoten nur maximal eine Anfrage zu einem Zeitpunkt hat, hat diese Warteschlange eine Länge von maximal (N-1) Hops, und jeder Knoten befindet sich zu jedem Zeitpunkt maximal einmal in dieser Warteschlange. Befindet sich der Tokenbesitzer nicht in seiner kritischen Sektion, wird das Token sofort zum anfragenden Knoten gesendet. Anderenfalls setzt der letztmögliche Tokenbesitzer seinen „next“-Zeiger auf den anfragenden Knoten, und sendet beim Verlassen der kritischen Sektion das Token in Richtung „next“. Somit erreicht das Token den anfragenden Knoten.
Das System der „dir“-Zeiger ist identisch dem des SBA und somit zyklenfrei. Die Anfragen enden auf Grund der Operationen beim oben beschriebenen Mechanismus des Weitersendens immer bei dem letzten Knoten der „next“-Kette. Jeder Knoten hat maximal eine aktuelle Anfrage an das Token, und die Anfragen erreichen immer das Ende der durch die „next“-Zeiger aufgebauten verteilten Warteschlange.
Daraus folgt, daß die Kette der „next“-Zeiger frei von Zyklen ist.

DVA:
Das System der „dir“-Zeiger bestimmt wiederum den Weg von Anfragen an das Token. Initial zeigen alle Zeiger „dir“ auf den initialen Tokenbesitzer, Knoten 0. Anfragen erreichen somit immer den Tokenbesitzer in einem Schritt. Im Verlauf des Algorithmus ändern sich die „dir“-Zeiger mit dem Senden bzw. Weitersenden von Nachrichten. Sie werden auf den mit der Anfrage verschickten Absender der Anfrage gesetzt. Beim Tokenbesitzer bzw. beim letztmöglichen Tokenbesitzer, der seinen „next“-Zeiger auf den fragenden Knoten setzt, endet dieser Mechanismus. Das Senden einer Anfrage ist atomar mit der Zuweisung „dir := self“ verbunden. Das Weitersenden ist atomar mit der Zuweisung „dir := Requester“ verbunden. Daraus folgt, daß die Tokenanfragen immer den Tokenbesitzer bzw. den letztmöglichen Tokenbesitzer erreichen.
Das Setzen der „next“-Zeiger baut eine verteilte Warteschlange auf. Aus der Tatsache, daß jeder Knoten maximal eine Anfrage hat, folgt, daß die Größe der Kette kleiner N ist und, daß sich jeder Knoten maximal einmal zu einem Zeitpunkt in der Kette befindet. Da die Anfragen immer das Ende der „next“-Kette erreichen, hat die Kette die Form einer einfachen Liste. Darausfolgt, daß die Kette zyklenfrei ist.

Die Algorithmen erfüllen somit die obengenannten Bedingungen und sind frei von Verklemmungen.

5.3 Vermeidung von Verhungern eines Knotens

Zu zeigen ist, daß, wenn ein Knoten der seine kritische Sektion betreten will und eine Tokenanfrage absetzt, das Token letztendlich bekommt. Szenarien, in denen sich zwei oder mehrere Knoten das Token immer wieder gegenseitig zusenden, und ein dritter Knoten so nie die Möglichkeit bekommt seine kritische Sektion zu betreten, dürfen nicht auftreten.
Kann man zeigen, daß die Algorithmen fair sind, also daß Knoten das Token in der Reihenfolge, erhalten in der sie das Token angefragt haben, so beinhaltet dies, daß der Algorithmus auch das Verhungern vermeidet. Betrachten der Reihenfolge bedeutet hier das Betrachten einer lokalen Reihenfolgen. Das heißt, wenn ein Knoten eine Anfrage erhält und danach selber das Token anfragt, wird er immer zunächst die erste Anfrage bearbeiten.

SBA:
Knoten i will seine kritische Sektion betreten.
Fall 1: Knoten i ist der Tokenbesitzer. Nach dem Test des Tokens betritt Knoten i seine kritische Sektion.
Fall 2: Knoten i ist nicht der Tokenbesitzer. Darausfolgt, daß Knoten i eine Tokenanfrage sendet. Allgemein gilt: Die Anfragen werden entlang den „dir"-Zeigern transportiert. Die Anfragen werden beim Empfang auf jedem Knoten in eine Warteschlange geschrieben. Beim Tokenbesitzer endet dieser Mechanismus des Weitersendens in dem Sinne, daß der Tokenbesitzer, Knoten j, keine weitere Anfrage versendet. Ist Knoten j in seiner kritischen Sektion, speichert er lediglich die Anfrage in seiner Warteschlange. Hält Knoten j das Token, ohne es zu nutzen, sendet er das Token sofort in Richtung der Anfrage. Ist er in seiner kritischen Sektion, sind zwei Fälle beim Verlassen der kritischen Sektion zu unterscheiden:

1. Die Anfrage von i steht an erster Stelle in der Warteschlange des Tokenbesitzers, d. h. das Token wird, nachdem Knoten j seine kritische Sektion verlassen hat, in Richtung Knoten i gesendet.
2. Die Anfrage von Knoten k steht an erster Stelle, Knoten j sendet das Token in Richtung k und setzt „dir" in die Richtung in der er das Token gesendet hat. Nun muß sich aber in der Warteschlange des nun ehemaligen Tokenbesitzers j mindestens ein weiterer Eintrag befinden, namentlich die Anfrage von i. Die Warteschlange ist nicht leer. Daraus folgt, daß Knoten j eine erneute Tokenanfrage sendet. Erhält Knoten j daraufhin das Token, entfernt er Knoten i aus der Warteschlange und sendet das Token in Richtung Knoten i.

In beiden Fällen erhält i das Token, und zwar in der Reihenfolge, wie die Anfragen bei j eintrafen. Aus 1., 2. und den strikten FIFO-Warteschlangen folgt die Vermeidung des Verhungerns.

SRA:
Knoten i will seine kritische Sektion betreten.
Fall 1: Knoten i ist der Tokenbesitzer. Darausfolgt, daß der Test des Tokens positiv ist und Knoten i seine kritische Sektion betritt.
Fall 2: Knoten i ist nicht der Tokenbesitzer, der Test des Tokens ist negativ, somit sendet Knoten i eine Tokenanfrage.
Die Anfrage wird in einer Richtung von Knoten ((i+1)mod N) bis Knoten ((i-1)mod N) weitergeleitet. Die Knoten {((i+1)mod N), .. ,((i-1) mod N)} schreiben jeweils ein ST_SUBS in ihre Warteschlange. Die ST_SUBS in den Warteschlangen sind entsprechend der Reihenfolge der Anfragen geordnet. Erst wenn die Knoten Anzahl ST_SUBS oft das Token gesendet haben, finden sie ein ST_WAIT in der Warteschlange und betreten die kritische Sektion. Der Knoten rotiert durch dieses Protokoll öfter als nötig, aber der Algorithmus ist somit fair und vermeidet das Verhungern von Knoten.

DVA und SVA:
Knoten i will seine kritische Sektion betreten.
Fall 1: Knoten i ist Tokenbesitzer, darausfolgt das Knoten i sofort seine kritische Sektion betritt.
Fall 2: Knoten i ist nicht der Tokenbesitzer und sendet somit, nachdem der Test des Tokens negativ ist, eine Tokenanfrage.
Hier erreicht die Tokenanfrage von Knoten i entweder den Tokenbesitzer oder das Ende der „next“-Warteschlange (siehe Vermeidung von Verklemmung).
Im Falle, daß die Anfrage den Tokenbesitzer erreicht, erhält Knoten i das Token sofort, falls der Tokenbesitzer nicht in seiner kritischen Sektion ist. Ist der Tokenbesitzer in seiner kritischen Sektion, erhält Knoten i das Token, nachdem der Tokenbesitzer seine kritische Sektion verlassen hat. Knoten i verhungert nicht. Gelangt die Anfrage zum Ende der Warteschlange, wird Knoten i das neue Ende der Warteschlange. Knoten i reiht sich somit in der Reihenfolge der Anfragen ein und hat, spätestens nach (n-1) mal Versenden des Tokens (siehe Vermeidung von Verklemmung), das Token erhalten und damit die Möglichkeit, seine kritische Sektion abzuarbeiten.

Somit sind die Algorithmen fair und das Verhungern wird vermieden.

6 Die Testumgebung

Die Algorithmen wurden in einer homogenen Umgebung getestet. Diese Testumgebung ist in C unter LINUX 2.0 und SunOS 4.1 implementiert. Die N Knoten sind als Unixprozesse implementiert und kommunizieren untereinander über Sockets.
Der Aufruf erhält als Parameter die gewünschte Anzahl der Knoten (N), die Anzahl der kritischen Sektionen pro Knoten (ncs), die Verweildauer in einer kritischen Sektion (Phi) und die Zeit zwischen dem Verlassen der kritischen Sektion und des erneuten Versuchs, die kritische Sektion zu betreten (Gamma).

Hier der Aufruf für den Algorithmus von Chang, Singhal und Liu bzw. Naimi und Trehel (DVA) mit acht Knoten, jeweils acht kritischen Sektionen, Phi = 200 µsec und Gamma = 2000 µsec:

% mt_cls 8 8 2000 200

Der Startknoten (ein LINUX- bzw. UNIX-Prozess) startet N Knoten und terminiert bzw. wartet auf das Terminieren der gestarteten Knoten und gibt Meßergebnisse aus. Die Knoten laufen ab diesem Zeitpunkt unabhängig.
Die Kommunikation zwischen den Knoten erfolgt über Sockets mit dem TCP-Protokoll. Ein Knoten besteht aus zwei Threads, einem Masterthread der die Schleife mit den kritischen Sektionen abarbeitet und einem Kommunikationsthread. Diese Threads sind als POSIX-Threads [Posix.1c] implementiert und nutzen die POSIX-Threads-Bibliotheksfunktionen zur Synchronisation. Verwendet wurde die Implementation der Florida State University [Mu93].
Den strukturellen Aufbau eines Knotens (Task) der Testumgebung zeigt Abbildung 6. Die Pfeile beschreiben die Kommunikationswege zwischen den Knoten und zwischen den Komponenten eines Knotens.
Um das ordnungsgemäße Terminieren der Knoten sicherzustellen, benötigt man ein zusätzliches Werkzeug zur Synchronisation der Knoten, eine Barriere. Diese Barriere ist als binäre Baumbarriere implementiert, d.h. wenn einer der Knoten die Schleife mit den kritischen Sektionen abgearbeitet hat, wartet sie auf die noch arbeitenden Knoten.
Dies ist notwendig, weil die Knoten, auch wenn sie selber ihre kritischen Sektionen abgearbeitet haben, weiter Operationen wie das Weitersenden von Nachrichten ausführen müssen, bis der letzte Knoten terminiert.
Die Kommunikation der Barrierensynchronisation wird ebenfalls über den Kommunikationsthread abgewickelt, hat jedoch keinen Einfluß auf die Messungen bezüglich der Algorithmen für den gegenseitigen Ausschluß. Die Baumbarriere arbeitet auf einem binären Baum. Die Knoten melden sich mit ***Barriere_att()*** bei der Barriere an und warten dann mit dem Ruf ***Barrier_wait()*** auf das Signal zum Weiterlaufen bzw. in diesem Fall auf das Signal zum Terminieren.
Die benötigte ***await(Token)*** Anweisung ist mit Hilfe der Funktionen ***pthread_cond_wait()*** und ***pthread_cond_signal()*** der Pthreadbibliothek implementiert.
Für das Verzögern innerhalb der kritischen Sektion und zwischen dem Verlassen und erneutem Anfragen wurde die Funktion ***nanosleep()*** benutzt. Sämtliche Zeitmessungen wurden mit der Funktion ***clock_gettime()*** realisiert. ***nanosleep()*** und ***clock_gettime()*** sind Funktionen des POSIX-Standards [Posix.1b].

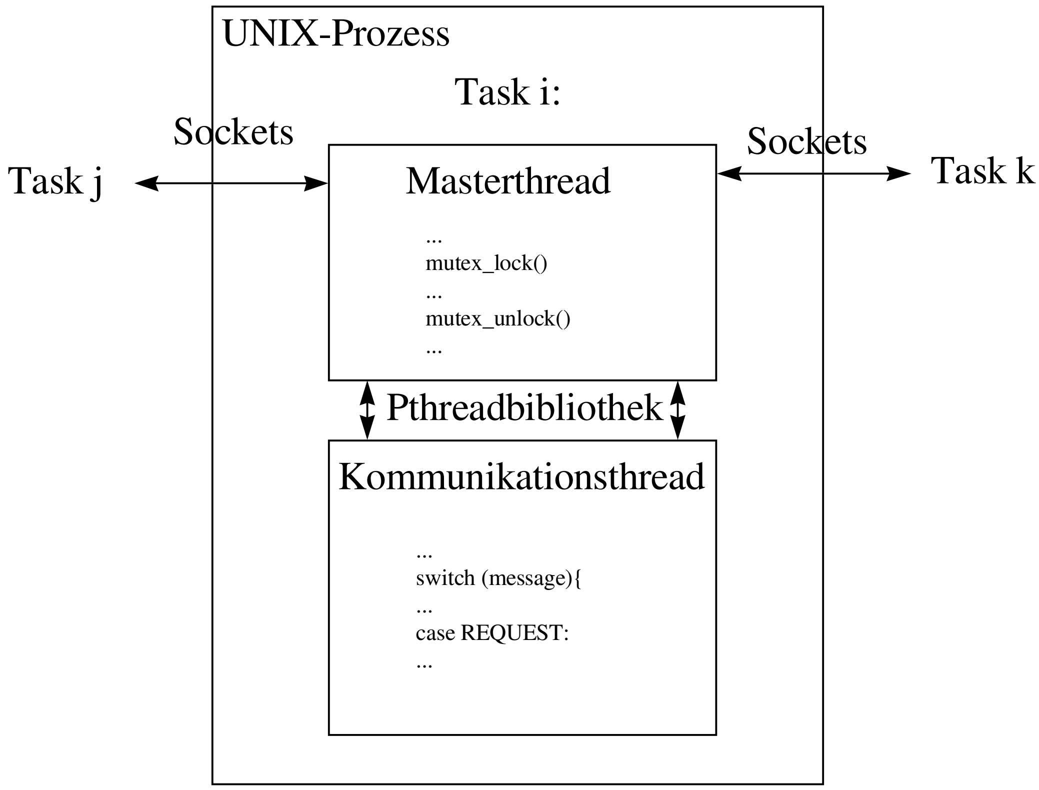


Abbildung 6: Struktureller Aufbau einer Task der Testumgebung.

7 Performance

Tokenbasierende Algorithmen für den gegenseitigen Ausschluß benötigen im besten Fall keine Nachricht für das Betreten der kritischen Sektion: Der 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 Knoten des Systems kommuniziert zu haben.
Betrachten wir nun die Komplexität der Nachrichten in Abhängigkeit von der Anzahl der Knoten N im schlechtesten Fall.
Beim SBA sind die Knoten des Systems in einem binären Baum geordnet. Die Laufzeit einer Anfrage ist im schlechtesten Fall (2 * logN). Ein Blatt des binären Baums stellt eine Tokenanfrage, das Token befindet sich ebenfalls in einem Blatt, und die Tokenanfrage muß über die Wurzel gesendet werden. Die Laufzeit des Tokens ergibt sich ebenso. Daraus folgt eine Komplexität von O(logN) im schlechtesten Fall.
Beim SVA ist die Anfrageentfernung im schlechtesten Fall gleich der beim SBA, (2 * logN). Der Weg des Tokens jedoch ist immer konstant eins, das Token wird direkt zum nächsten Tokenbesitzer gesendet. Somit ergibt sich ebenfalls ein Komplexität der Nachrichten von O(logN).
Beim Ringalgorithmus SRA ist der schlechteste Fall gegeben, wenn der Vorgänger des anfragenden Knotens der Tokenbesitzer ist. Die Anfrage gelangt über (N-1) Hops zum Vorgänger. Das Token muß ebenfalls (N-1) mal weitergesendet werden

bis es den anfragenden Knoten erreicht, daraus folgt eine Komplexität der Nachrichten von O(N).
Der Weg des Tokens beim DVA im schlechtesten Fall ist, wie beim SVA, gleich 1, das Token wird direkt zum nächsten Tokenbesitzer gesendet. Die Kette der „dir“-Zeiger, die den Weg der Tokenanfrage bestimmt kann zu einer (N-1) Hops langen Liste entarten. Der Weg der Tokenanfragen ist somit (N-1). Es ergibt sich eine Komplexität der Nachrichten von O(n) im schlechtesten Fall. Der Fall, daß die Kette der „dir“-Zeiger eine (N-1) Hops lange Liste ist, ist experimentell jedoch schwer zu erzeugen bzw. bei der initialen Konfiguration der Knoten in einem Stern (siehe Beispiel in Kapitel 4.3) unmöglich, man erreicht so maximal eine Komplexität der Nachrichten von (N/2). Das folgende Beispiel illustriert diese Aussage. Die Pfeile repräsentieren das System der „dir“-Zeiger, das „T“ bezeichnet wo das Token sich befindet und „R(Knoten)“ steht für eine Tokenanfrage.
Ein System von vier Knoten (A, B, C und D) ist initial als Stern geordnet, d. h. „dir“ ist beim initialen Tokenbesitzer Knoten A gleich NULL und bei allen anderen Knoten gleich A. Gamma (die Zeit vom Verlassen der kritischen Sektion zum erneuten Anfragen des Tokens) und Phi (die Zeit innerhalb der kritischen Sektion) sind sehr groß gewählt. Damit wird erreicht , daß die Anfragen, von den Knoten B, C, und D, Knoten A erreichen, bevor A seine kritische Sektion verläßt. Der initiale Tokenbesitzer A, betritt sofort seine kritische Sektion (Abbildung 7a).
Unter diesen Voraussetzungen wird bei einer Anfragereihenfolge (B - C - D - A) die für den schlechtesten Fall notwendige Liste der Zeiger „dir“ sukzessive aufgebaut (Abbildung 7b,c,d).

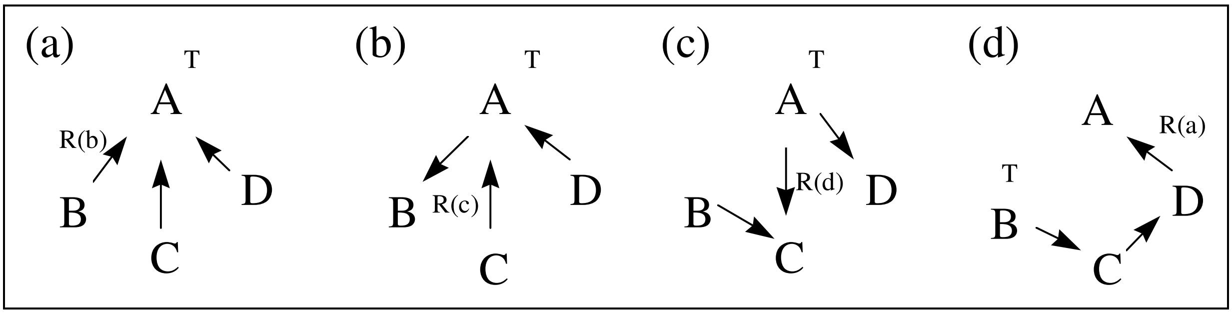


Abbildung 7: Erzeugung des schlechtesten Falls für den Anfrageweg beim DVA.

Nun sendet B eine Tokenanfrage, diese benötigt in der Tat (N-1) Hops bis sie das Ende der Warteschlange erreicht (Abbildung 8a). Die Struktur verändert sich allerdings mit der Anfrage von B so, daß die nächsten Tokenanfrage, egal von welchem Knoten, einen Weg von nur einem Hop haben (Abbildung 8d). Dieser Ablauf wiederholt sich im weiteren Verlauf.

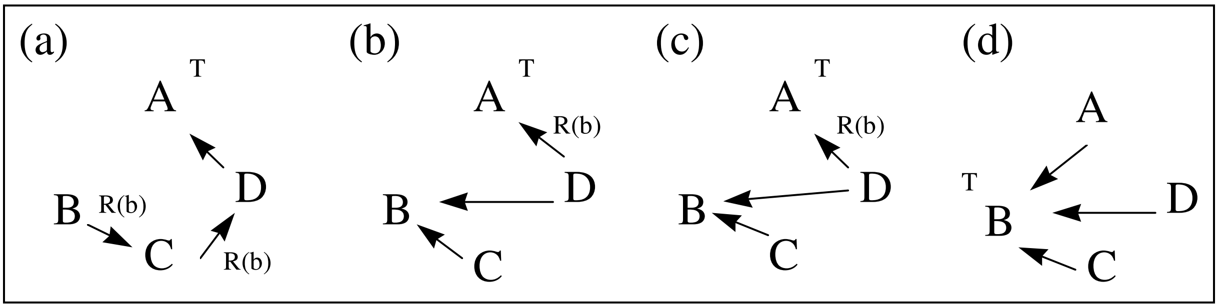


Abbildung 8: Veränderung beim zweiten Anfragen des Tokens.

Für den Weg der Anfragen gibt es somit in dem konstruierten Beispiel zwei Konfigurationen: Die in Abbildung 7a und Abbildung 8d gezeigte Sternform, mit einer Weglänge von einem Hop, und die in Abbildung 7d und Abbildung 8a dargestellte Listenform mit einer Weglänge von (N-1) Hops. Diese Konfigurationen wechseln sich in Experimenten mit höherer Anzahl der kritischen Sektionen als eins gegenseitig ab.
Somit wird in Experimenten nie mehr als (N/2) Nachrichten je kritischer Sektion erreicht werden. Mehr noch, (N/2) setzt eine gleichmäßige Anfragereihenfolge der Knoten voraus, die praktisch jedoch schwer zu erzeugen ist.

Die folgende Tabelle zeigt die Komplexität der Nachrichten im besten Fall bzw. beim schlechtesten Fall die Größe der Wege einer Tokenanfrage und die Größe der Wege des Tokens in Abhängigkeit von der Anzahl der Knoten des Systems gemessen in Hops.

	SBA	SRA	SVA	DVA
bc	0	0	0	0
wc Token	O(logN)	O(N)	1	1
wc Anfrage	O(logN)	O(N)	O(logN)	O(N)

Tabelle 2: Komplexität der Nachrichtenwege im schlechtesten Fall (wc) und im besten Fall (bc).

Die Untersuchung der tatsächlichen Performance wird in drei Schritten durchgeführt. Ziel ist es den mittleren Fall zu bestimmen und die Algorithmen komparativ zu bewerten. Die Voraussetzung einer solchen Bewertung ist die Schaffung einer homogenen Testumgebung, wie in Kapitel 6 vorgestellt.

Der erste Schritt der Untersuchung der tatsächlichen Performance bestand aus Testläufen auf verschiedenen Betriebssystemen und Prozessorarchitekturen. Die Algorithmen wurden nacheinander mit gleichen Eingabeparametern auf einem Intel Pentium mit dem Betriebssystem Linux 2.0, auf einer SPARCStation-20 und auf einer SPARCclassic, jeweils mit dem Betriebssystem SunOS 4.1, getestet.
In diesem Schritt wurde gleichzeitig die Wirkung der Anzahl der Knoten auf die Anfragezeit und die Anzahl der Nachrichten untersucht. Nachdem im ersten Schritt nur quantitative, von der Rechenleistung abhängige, jedoch keine qualitativen

Unterschiede ermittelt wurden, wurden die folgenden Schritte nur auf einem System durchgeführt.
Im zweiten Schritt stand das Verhalten der für die Performance relevanten Parameter in Abhängigkeit von der Anzahl der kritischen Sektionen bei konstanter Anzahl der Tasks auf dem Prüfstand.
Im letzten Schritt wurde durch die Veränderung der Parameter Gamma (Zeit zwischen Verlassen und des Wiederanfragens der kritische Sektion) und Phi (Verweildauer in der kritischen Sektion) die Anfragefrequenz variiert.
In allen Schritten wurde der Parameter Gamma randomisiert durch +- 10% verändert. Dies hat eine realistisch gestreute Anfrage des Tokens zur Folge und verhindert eine zeitlich identische Abfolge des Kontrollflusses in den einzelnen Knoten.

7.1 Schritt 1: Anzahl der Knoten

7.1.1 Parameter

Der verteilte Programmcode besteht aus kritischen Sektionen und nichtkritischen Teilen. Die kritischen Sektionen sind kurz und enthalten erheblich weniger Instruktionen (meist Zuweisungen und Leseoperationen von Daten) als der nichtkritische Teil. Aus diesem Grund wurde im Schritt 1 Phi, die Verweildauer in der kritischen Sektion, um Faktor 10 kleiner als Gamma gewählt. Die Abhängigkeit der Anzahl der Nachrichten je kritischer Sektion (nmE) und der mittleren Anfragezeit (Treq) von der Anzahl der Nachrichten wurde im Intervall N=2 bis N=32, in Schritten von Potenzen der Basis 2, untersucht. Die Anzahl der kritischen Sektionen pro Knoten betrug jeweils 20. Die konkreten Parameter zeigt Tabelle 3.
Um die Algorithmen auf verschiedenen Plattformen und Prozessorarchitekturen zu testen, wurden die Läufe mit den Parametern aus Tabelle 3 nacheinander auf einer SPARCStation-20 mit dem Betriebssystem SunOS 4.1, einer SPARCclassic, ebenfalls unter SunOS 4.1, und einem Intel Pentium mit dem Betriebssystem Linux 2.0 durchgeführt.
Die SPARCStation-20 ist eine Zweiprozessormaschine, die Taktfrequenz der Prozessoren beträgt 60 MHz. Die SPARCclassic hat einen mit 50 Mhz getakteten Prozessor. Die Taktfrequenz des Einprozessor Intel Pentiums beträgt 133 MHz.

N	ncs	Gamma in µsec	Phi in µsec
2,4,8,16,32	20	2000	200

Tabelle 3: Parameter der Testläufe im Schritt 1.

7.1.2 Ergebnisse und Bewertung

Bei den folgenden Grafiken wurde auf Werte des SRA, besonders bei der Anzahl der Nachrichten (nmE) verzichtet, um die Skalierung der Y- Achse der Grafiken zu vergrößern und damit eine bessere Betrachtung zu ermöglichen. Bei der Wirkung der Anzahl der kritischen Sektionen wurde aus dem selben Grund die Anzahl der Tasks bei Tests des SRA reduziert.

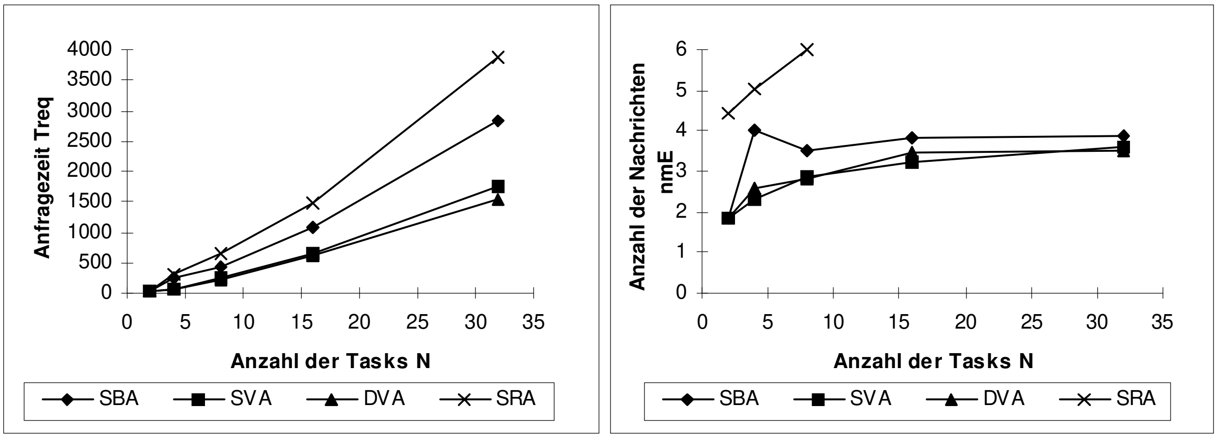


Abbildung 9: Ergebnisse der Testläufe auf der SPARCclassic bei 20 kritischen Sektionen.

Die Anfragezeit steigt mit Zunahme der Anzahl der Tasks in Form einer Parabel. Jede Task hat maximal eine Anfrage des Tokens zu einem Zeitpunkt. Die Abhängigkeit müßte demnach linear sein. Eine lineare Abhängigkeit ist jedoch durch verlängerte Nachrichtenwege (auspacken, bearbeiten und einpacken auf den Zwischenstationen) nicht zu erreichen. Bei der Simulation kommt noch hinzu, daß der nichtkritische Teil ebenfalls sequentiell und nicht parallel abläuft. Die Annäherung der Kurve der Anfragezeit in Abhängigkeit von der Anzahl der Tasks an einen linearen Graphen kann als Kriterium für die Güte eines Algorithmus benutzt werden.

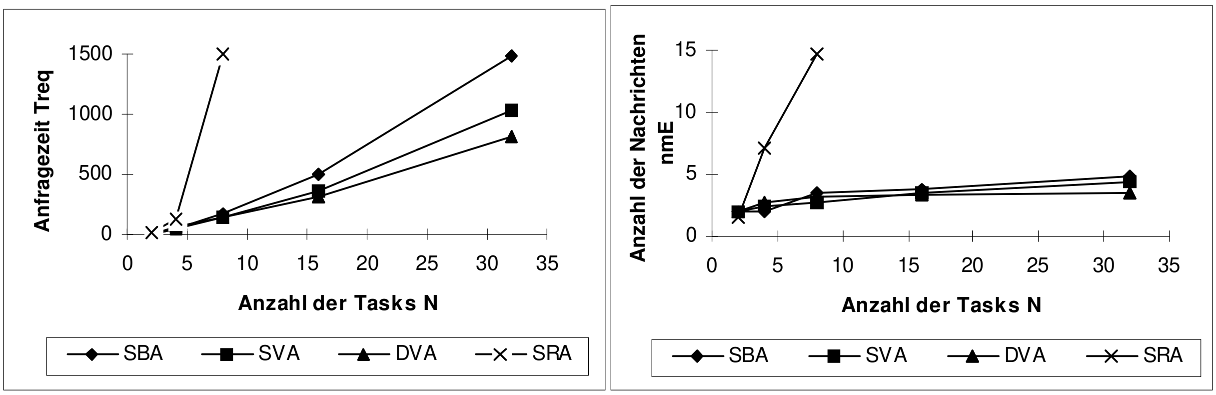


Abbildung 10: Ergebnisse der Testläufe auf der SPARCStation-20 bei 20 kritischen Sektionen.

Die Anzahl der Nachrichten nimmt beim DVA, beim SVA und beim SBA logarithmisch zu. Beim SRA ist eine lineare Zunahme zu verzeichnen. Betrachtet man die Komplexität im schlechtesten Fall (Tabelle 2), ist dies plausibel. Der Anstieg der Kurve beim SRA ist größer als 1. Begründet ist dies darin, daß bei der fairen Version des SRA das Token noch einige Male nutzlos rotiert.

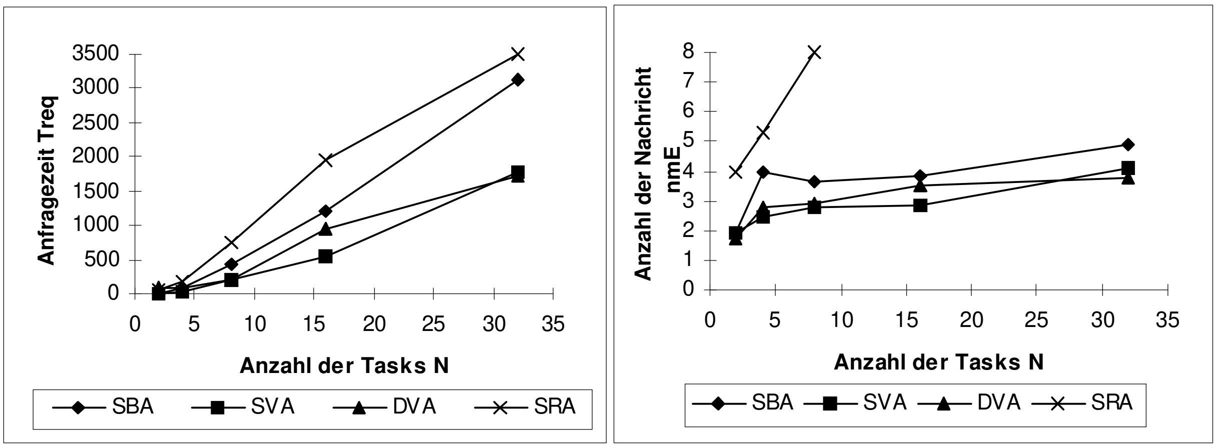


Abbildung 11: Ergebnisse der Testläufe auf dem Intel Pentium bei 20 kritischen Sektionen.

Betrachtet man die Prozessortaktfrequenzen und die Prozessorarchitekturen, sind die Ergebnisse ebenfalls qualitativ plausibel.
Die Ergebnisse bei der Anfragezeit liegen auf der SPARCclassic mehr als Faktor 2 über den Ergebnissen auf der mit zwei 60 MHz getakteten Prozessoren ausgerüsteten SPARCStation-20. Die Anfragezeiten bei dem Intel Pentium, unter dem Betriebssystem Linux liegen zwischen den Ergebnissen der SPARC-Computer. Der große Unterschied zwischen den Ergebnissen der SPARC-Rechner ist auf die obengenannten Taktfrequenzen und auf die superskalare Architektur (mehr als eine Instruktion je Zyklus) der SPARCStation-20 zurückzuführen. Die SPARCClassic ist ein RISC-Prozessor mit ca. einer Instruktion je Zyklus. Der Intel Pentium ist ein CISC-Prozessor. Die Ergebnisse des Intel Pentium liegen somit trotz der höheren Taktfrequenz unter denen der SPARCStation-20.

Die Anzahl der Nachrichten nimmt mit zunehmender Anzahl der Tasks N beim DVA, SVA und SBA logarithmisch zu. Beim SRA steigt sie linear durch den linear zunehmenden Abstand in Knoten (Hops) in Abhängigkeit von der Anzahl der Tasks N.
Bei ($N < 20$) liegt der SVA noch besser als der DVA, für ($N > 20$) braucht der DVA die geringste Anzahl Nachrichten. Der SBA liegt gering, aber doch deutlich über dem DVA und dem SVA. Das Token muß beim SBA immer entlang der statischen Struktur gesendet werden, bei SVA und DVA wird er direkt zum nächsten Tokenbesitzer gesendet.
Der DVA reduziert die Wege der Anfragenachrichten durch eine dynamische Struktur. Tabelle 2 zeigt die Nachrichtenwege in Abhängigkeit von der Anzahl der Knoten N im schlechtesten Fall. Im schlechtesten Fall hat DVA eine längere Laufzeit der Anfragen als der SVA, da die Knoten in einer Liste angeordnet sind. Bei einer annähernd gleich verteilten Anfrage tritt dieser Fall jedoch nicht auf, die Struktur verzweigt sich ständig (siehe oben).
Wenn von N Knoten nur drei Knoten ihre kritische Sektion betreten, und das wiederholt, reduzieren sich die Nachrichtenwege beim DVA. Die anderen (N-3)

Knoten haben keine Operationen auszuführen und sind keine Zwischenstation für Nachrichten.
Da das Senden und Empfangen von Nachrichten (im Gegensatz zum algorithmischen Aufwand) eine größere Bedeutung für die Performance hat, ist zu erwarten, daß die Unterschiede der Wartezeit in einem tatsächlich verteilten System noch größer sind.

Teilweise vergleichbar sind die Ergebnisse mit denen von Johnson [Jo95] und Fu et al. [FTL97], die jedoch auf anderen Plattformen durchgeführt wurden. Johnson betrachtet die Algorithmen nur unter Last. Die Untersuchungen von Fu et al. überschneiden sich mit den Algorithmen dieser Arbeit nur für den Algorithmus von Chang et al. bzw. Naimi und Trehel (DVA). Bei dem von Fu et al. betrachteten Ringalgorithmus handelt es sich nicht um den ursprünglich von A. J. Martin vorgestellten und in dieser Arbeit betrachteten Algorithmus. Fu et al. untersuchen eine unfaire, jedoch performantere Version des ursprünglichen Ringalgorithmus.

7.2 Schritt 2: Anzahl der kritischen Sektionen

Da Schritt 1 zeigt, daß die Plattform, das Betriebssystem und die Prozessorarchitektur nur quantitativen, jedoch keinen qualitativen Einfluß auf die Performance der Algorithmen hat, wurden die Testläufe für die folgenden Schritte auf einer Plattform durchgeführt. Gewählt wurde der Intel Pentium, der unter dem Betriebssystem Linux 2.0 betrieben wird.

7.2.1 Parameter

In diesem Schritt wurde die Wirkung der Anzahl der kritischen Sektionen auf die Performance untersucht. Die Anzahl der kritischen Sektionen pro Knoten wurde in Zehnerschritten von 10 bis 100 variiert. Hier sind nur die Ergebnisse bis zu 60 kritischen Sektionen aufgeführt, da es sich gezeigt hat, daß dies für eine Charakterisierung genügt. Die Anzahl der Knoten wurde mit 32 festgesetzt.
Die Ergebnisse des Schrittes 1 zeigen, daß sich ab 16 Knoten auf allen Plattform eine Regelmäßigkeit der Kurven zeigt. Somit sind bei 32 Knoten die Abweichungen durch Anomalien (wie mehrmaliges Abarbeiten der kritischen Sektion hintereinander durch einen Knoten oder das abwechselnde Eintreten zweier Knoten) nicht mehr gegeben. Beziehungsweise beeinflussen die Anomalien die relevanten Meßwerte, durch die hohe Anzahl der Knoten, nicht mehr entscheidend.
Gamma ist wiederum um Faktor 10 größer als Phi gewählt und randomisiert um +- 10% verändert, um zufällige und damit variierende Anfragesequenzen zu erzeugen. Die Parameter sind in Tabelle 4 zusammengefaßt.

N	ncs	Gamma in microsec	Phi in µsec
32	10,20,30,40,50,60	2000	200

Tabelle 4 Parameter bei der Untersuchung der Auswirkung der Anzahl der kritischen Sektionen auf die Performance.

7.2.2 Ergebnisse und Bewertung

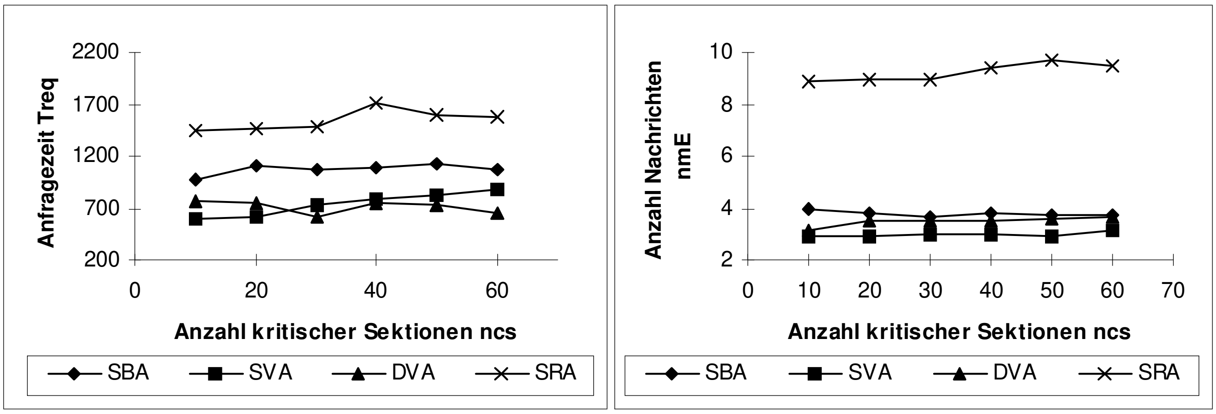


Abbildung 12: Die Wirkung der Anzahl kritischer Sektionen auf die Anzahl der Nachrichten und die mittlere Anfragezeit.

Die Kurven der Abbildung 12 zeigen, daß die Anzahl der kritischen Sektionen keinen Einfluß auf die für die Performance relevanten Meßwerte haben. Bei der mittleren Anzahl der Nachrichten sind die Kurven faktisch konstant. Der nichtregelmäßige Verlauf der Kurven bei der Anfragezeit ist auf temporäre Prozessorlasten zurückzuführen. Diese Prozessorlasten sind betriebssystembedingt, da auf dem System keine zusätzlichen Lasten erzeugt wurden.
Da jeder Knoten nur eine Anfrage zu einem Zeitpunkt hat, laufen bei unterschiedlicher Anzahl der kritischen Sektionen die gleichen Szenarien ab. Lediglich bei kleinen Werten (ncs = 2) kommt es zu Anomalien. Zum Beispiel erreichen die Tokenanfragen den initialen Tokenbesitzer nicht, bevor er seine gesamten kritischen Sektionen abgearbeitet hat. Dies reduziert die Anzahl der Nachrichten je kritischer Sektion.
Der DVA hat auf Grund der Pfadreduzierung die geringste Wartezeit und die geringste Anzahl der Nachrichten. Der SVA liegt nur wenig über dem DVA, aber deutlich unter dem SBA. Das Token wandert beim SVA direkt zum nächsten Tokenbesitzer. Beim SBA und SRA wandert das Token durch die statische Struktur.

Fu et al. erzielten, im Hinblick auf die sich in den Arbeiten überschneidenden Algorithmen ähnliche Ergebnisse (vergleiche Abb. 3. in [FTL97]).

7.3 Schritt 3: Anfragefrequenz

Im letzten Schritt galt es nun, die Parameter so zu verändern, daß sich die Frequenz der Versuche der Knoten, ihre kritische Sektion zu betreten, und somit die Anfragefrequenz „an“ das Token verändert. Die Anfragefrequenz kann durch die Wahl der Parameter Gamma und Phi beeinflußt werden.

7.3.1 Parameter

Angefangen von sehr kleinen Werten (100 µsec) und keinem Unterschied zwischen Phi und Gamma bis zu sehr großen Gamma (16 000 µsec) und Faktor 160 Unterschied zu Phi wurden nacheinander Testrechnungen durchgeführt.
Die Anzahl der Knoten wurde mit 20 festgesetzt, die Anzahl der kritischen Sektionen betrug ebenfalls 20.
Die konkreten Parameter sind in Tabelle 5 aufgeführt.

N	ncs	Gamma in µsec	Phi in µsec
20	20	100, 2000, 4000,8000, 16000	100, 200, 400

Tabelle 5: Parameter der Testläufe im Schritt 3 der Performancetests.

7.3.2 Ergebnisse und Bewertung

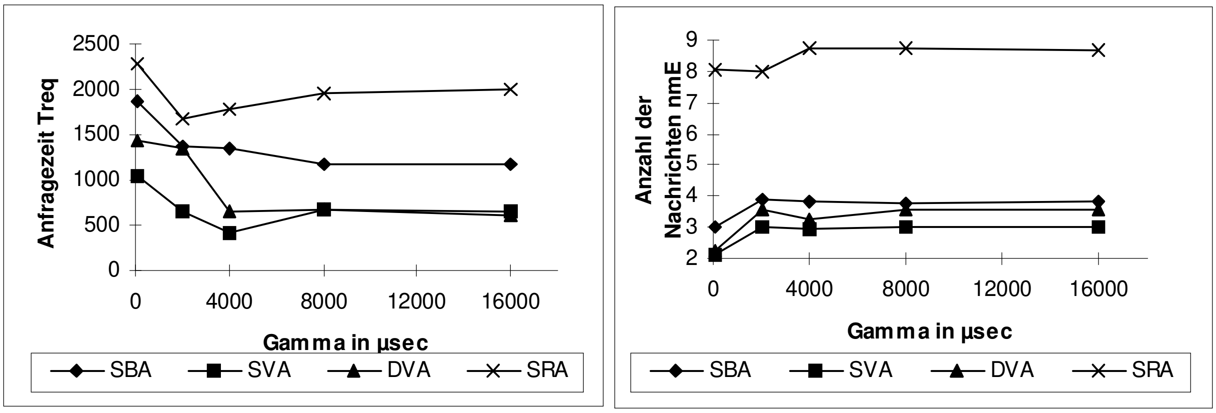


Abbildung 13: Die Wirkung der Anfragefrequenz auf die Anzahl der Nachrichten und der mittleren Anfragezeit bei Phi = 100 µsec.

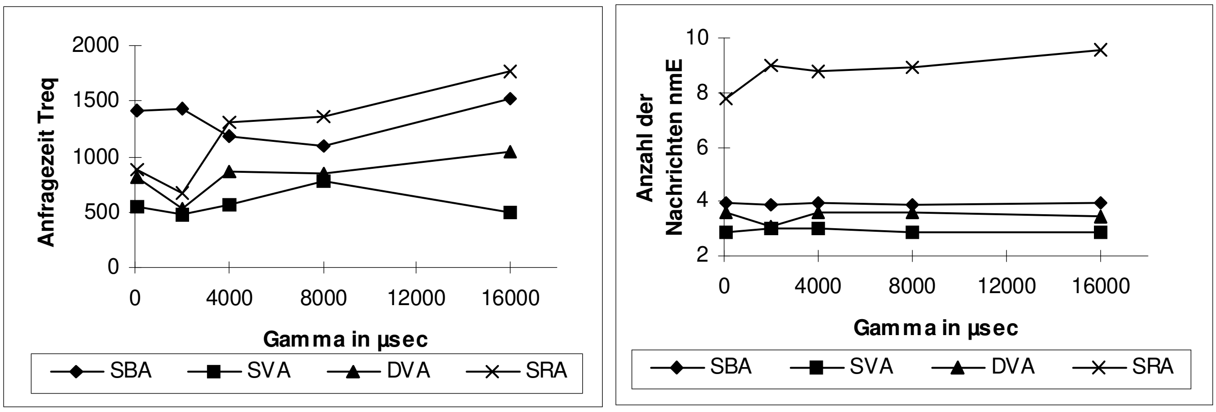


Abbildung 14: Die Wirkung der Anfragefrequenz auf die Anzahl der Nachrichten und der mittleren Anfragezeit bei Phi = 200 µsec.

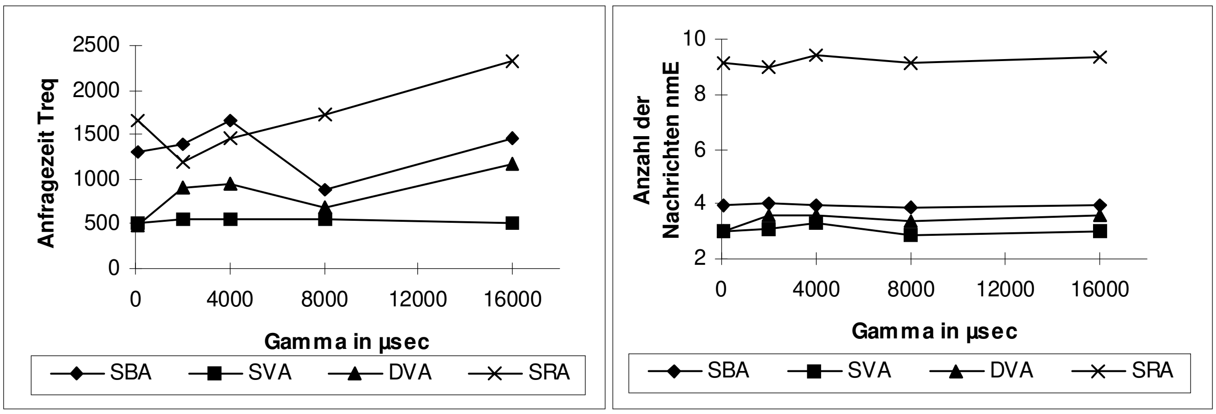


Abbildung 15: Die Wirkung der Anfragefrequenz auf die Anzahl der Nachrichten und der mittleren Anfragezeit bei Phi = 400 µsec.

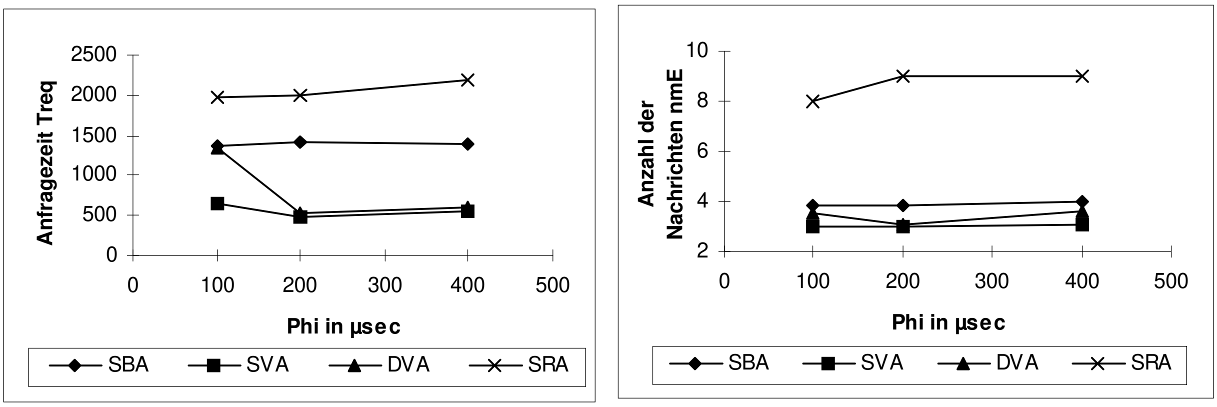


Abbildung 16: Die Wirkung der Anfragefrequenz auf die Anzahl der Nachrichten und der mittleren Anfragezeit bei Gamma = 2000 µsec.

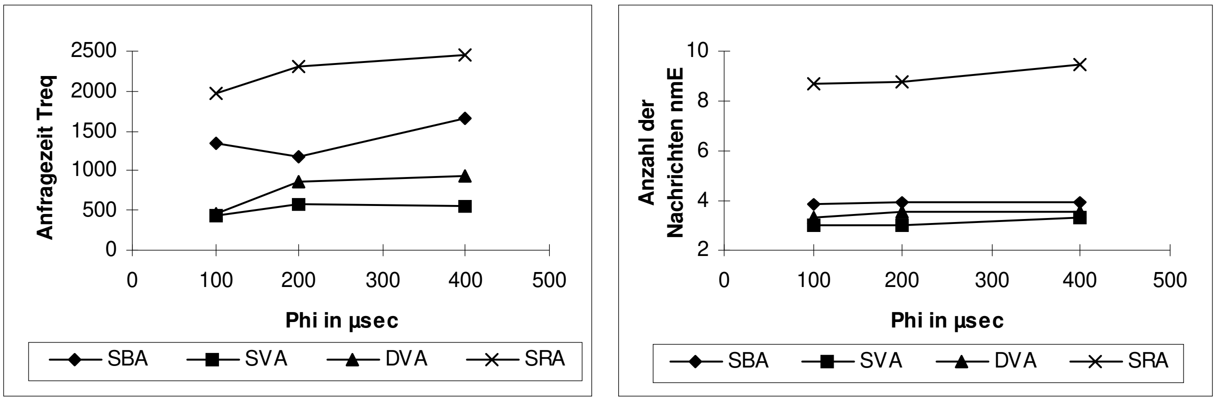


Abbildung 17: Die Wirkung der Anfragefrequenz auf die Anzahl der Nachrichten und der mittleren Anfragezeit bei Gamma = 4000 µsec.

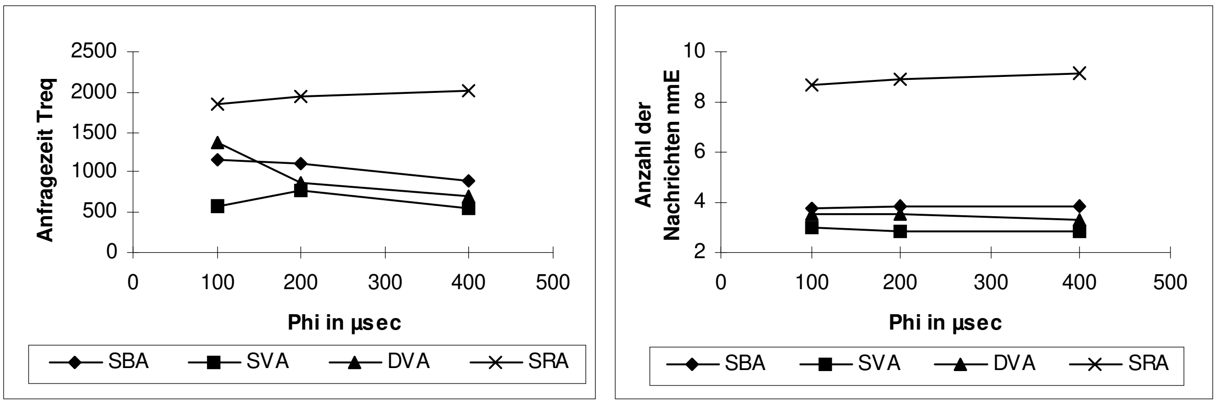


Abbildung 18: Die Wirkung der Anfragefrequenz auf die Anzahl der Nachrichten und der mittleren Anfragezeit bei Gamma = 8000 µsec.

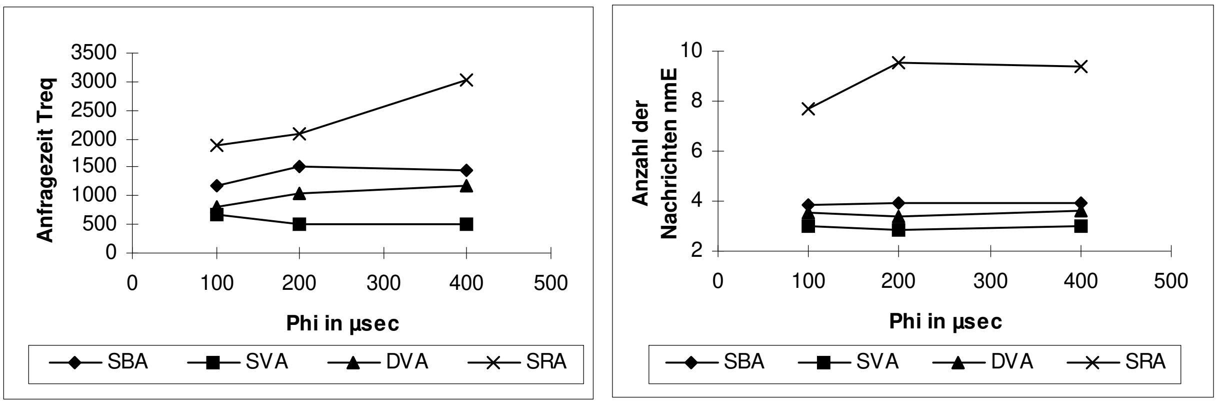


Abbildung 19: Die Wirkung der Anfragefrequenz auf die Anzahl der Nachrichten und der mittleren Anfragezeit bei Gamma = 16000 µsec.

Unter einer großen Anfragefrequenz (insbesondere bei Gamma = 100 und Phi = 100) ist die Anzahl der Nachrichten geringer als bei geringeren Anfragefrequenz. Ab einem (Gamma > 4000) ist der Graph konstant, die Anfragefrequenz hat keinen Einfluß mehr auf die Anzahl der Nachrichten.
Die Wartezeit verhält sich antiproportional zu der Anzahl der Nachrichten. Dies liegt einerseits daran, daß bei sehr hoher Anfragefrequenz die Knoten mehrmals hintereinander, ohne das Token abzugeben, ihre kritische Sektion betreten.
Andererseits ist das nicht ausschließlich die Begründung, betrachtet man die mit 20 gewählte Anzahl der kritischen Sektionen. Die Wege (auspacken der Nachricht, bearbeiten und einpacken der Nachricht auf den Zwischenstationen), die das Token und die Tokenanfragen zurückzulegen haben, werden geringer, insbesondere beim DVA durch die Pfadreduzierung. Es befindet sich immer ein auf das Token wartender Knoten in unmittelbarer „Nähe“ des Tokenbesitzers. Anfragen weiterentfernter Knoten müssen somit länger auf das Token warten, es wird unterwegs von anderen Knoten benutzt. Dies resultiert aus der nur lokalen Fairneß.

Quantitativ und qualitativ sind die erzielten Ergebnisse mit den in [FTL97] Abb. 1 und Abb. 2, und [Jo95] zu vergleichen, mit der Einschränkung, daß es sich nur zum Teil um identische Algorithmen handelt und die Plattformen unterschiedlich sind.

7.4 Fazit

Die besten Ergebnisse in allen Testrechnungen weisen die Algorithmen von Chang et al. bzw. Naimi und Trehel (DVA) und Neilsen und Mizuno (SVA) auf. Sie reduzieren durch ihre teilweise bzw. komplett dynamische Struktur die Nachrichten und die Anfragezeit. Die Anzahl der Nachrichten in Abhängigkeit von der Anzahl der Knoten N liegt unter (log N).
Der SBA liegt deutlich über den Ergebnissen der Erstgenannten, die Anzahl der Nachrichten liegt bei ca. (log N), ist jedoch besser als der SRA.
Der SRA zeigt insbesondere bei der Anzahl der Nachrichten um ein Vielfaches schlechtere Ergebnisse als die anderen Algorithmen. Dies ist plausibel, denn der

Abstand vom Sender zum Empfänger beträgt beim Ringalgorithmus, für das Token und die Tokenanfragen, im schlechtesten Fall (N-1) Hops (siehe Tabelle 2).
Tabelle 6 faßt die Ergebnisse zusammen.

	SBA	SRA	SVA	DVA
bc	0	0	0	0
wc	O(logN)	O(N)	O(logN)	O(N)
ac	logN	>N	logN	< logN

Tabelle 6: Die Komplexität der Nachrichten in Abhängigkeit von der Anzahl der Knoten im besten (bc), schlechtesten(wc) und mittleren (ac) Fall.

8 Zusammenfassung und Ausblick

Diese Arbeit gibt einen Überblick über tokenbasierende Algorithmen für den gegenseitigen Ausschluß in verteilten Systemen.
Die Literaturrecherche ergab eine Vielzahl von Arbeiten zu diesem Thema. Es erfolgte eine Einteilung dieser Arbeiten nach Art der zugrundeliegenden Prinzipien der Algorithmen.
Einen Zweig dieser Einteilung verfolgend, wurden zwei Algorithmen mit statisch logischer Struktur, ein Algorithmus mit dynamischer logischer Struktur und ein hybrider Algorithmus (im Sinne der Einteilung nach der Struktur) vorgestellt, informell bewiesen und empirisch untersucht.
Der operationalen Beschreibung der Algorithmen folgte ein einfaches Beispiel zum besseren Verständnis. Der Einheitliche Pseudocode kann als Grundlage für eine problemlose Implementierung genutzt werden.
Ein Überblick zur Korrektheit eines Algorithmus zum gegenseitigen Ausschluß in einem verteilten System und wie diese Korrektheit nachgewiesen werden kann, wurde gegeben.
Für die empirischen Untersuchungen wurde eine einfache Simulationsumgebung, auf herkömmlichen Plattformen geschaffen. Diese Testumgebung liefert mit anderen Arbeiten vergleichbare und allgemein plausible Ergebnisse. Damit ist ein Fazit dieser Arbeit, daß die Simulationsumgebung für weitere Forschungen nutzbar ist. Diese Arbeit bildet somit die Grundlage für weitere Studien, die sich in erster Linie mit Schreib- und Lesesperren und der Fehlertoleranz von Algorithmen für den gegenseitigen Ausschluß in verteilten Systemen befassen sollen.
Die Untersuchung der Performance wurde schrittweise durchgeführt. In Anschluß an einen Schritt folgte eine Plausibilitätskontrolle einschließlich Vergleichen mit anderen Arbeiten. Johnson [Jo96] untersuchte ein Teil der Algorithmen dieser Arbeit unter Last. Chang legt in [Ch96] mehr Aufmerksamkeit auf die theoretische Performance.
In der vorliegenden Arbeit ging es, im Gegensatz zu den oben erwähnten, um das Verhalten der Algorithmen auf verschieden Plattformen, verschiedenen Systemgrößen und unter verschiedener Anfragefrequenzen.

Die Reihe der Arbeiten über das Thema des gegenseitigen Ausschlusses in verteilten Systemen reißt seit 1961[4] nicht ab. In den letzten Jahren wurden Arbeiten veröffentlicht, die sich mit Algorithmen, die den Mehrfacheintritt in die kritische Sektion zulassen [Ray89, SR92], beschäftigen.
Ein weiterer Schwerpunkt der Forschung auf diesem Gebiet betrifft Echtzeitanforderungen. In diesem Zusammenhang sind Arbeiten über fehlertolerante Algorithmen [NT87/1, CSL90/2, Ch94/1, MS90, AA95, CS95, DH98] und priorisierte Algorithmen [Mu96, Ch94, Go90] für den gegenseitigen Ausschluß interessant.

[4] Dijkstra brachte 1961 zum ersten Mal das Thema in Zusammenhang mit Zugriffen von mehreren Hardwarekomponenten auf einen Speicher auf.

LITERATURVERZEICHNIS

[AA95] D. Agrawal und A. ElAbadi. A Token-Based Fault-Tolerant Distributed Mutual Exclusion Algorithm. Journal of Parallel and Distributed computing, Vol. 24, p. 164-176, 1995.

[Ch94] Y. I. Chang. Design of mutual exclusion algorithms for realtime distributed systems. Journal of Information Science and Engeneering, Vol. 10, p 527-548, 1994.

[Ch94/1] 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.

[Ch96] Y. I. Chang. A simulation study on distributed mutual exclusion. Journal of parallel and distributed computing, Vol 33, p. 107-121, 1996.

[CS95] 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.

[CSL90] 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.

[CSL90/1]. 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.

[CSL90/2] 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, p. 146-154, 1990.

[CR83] 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.

[DH98] P. K. Dash und R. C. Hansdah. A Fault-Tolerant Token-Based Algorithm for Distributed Mutual Exclusion Using Roughly Synchronized Clocks. IPPS, 1998.

[FTL97] Shiwa S. Fu, Nian-Feng Tzeng and Zhiyuan Li. Empirical Evaluation of Distributed Mutual Exclusion Algorithms. IEEE Computer 1997.

[Fu97] S. S. Fu, N.-F.Tzeng und Z. Li. Empirical Evaluation of Distributed Mutual Exclusion Algorithms. Center for Advanced Computer Studies, University of Southwestern Louisiana, Lafayette,1997.

[Go90] A. Goscinski. Two algorithms for mutual exclusion in realtime distributed computer systems. Journal of Parallel and Distributed Computing, p. 77-82, 1990.

[Jo95] Th. Johnson. A performance Comparison of Distributed Mtual Exclusiion Algorithms. Department of CIS, University of Florida, Gainesville. IEEE Computer 1995.

[Ku91] A. Kumar. Hierachical quorum consensus: A new algorithm for managing replicated data. IEEE Trans. On Computers, Vol. 40 (9), p. 994-1004, 1991.

[Lam78] L. Lamport. Time, clocks and the the ordering of events in a distributed System. Communication of the ACM, Vol. 21 (7), p. 558-564, 1978.

[M85] A. J. Martin. Distributed Mutual Exclusion on a Ring of Processes. Science of Computer Programming, Vol. 5, p. 265-276, Feb. 1985.

[Ma85]. M. Maekawa. A sqrt(n) algorithm for mutual exclusion in decentralized systems. ACM Trans. on Computer Systems, Vol. 3 (2), p. 145-159, 1985.

[Mu96] F. Mueller. Prioritized Token-Based Mutual Exclusion for Distributed Systems. International Parallel Processing Symposium, p. 791-795, Mar. 1996.

[Mu93] F. Müller. A Library Implementation of POSIX Threads under UNIX. Proceedings of the USENIX Conference, p. 29-41, 1993.

[MS90] Sh. Mishra und P. K. Srimani. Fault-Tolerant Mutual exclusion Algorithms. Journal of System Software, Vol. 11(2), p. 111-129, 1990.

[NLM90] S. Nishio, K. F. Li und E. G. Manning. A resilient mutual exclusion algorithm for computer network. IEEE trans. on Parallel and Distributed Systems, Vol. 1 (3), p. 344-355, 1990.

[NM91] M. L. Neilson und M. Mizuno. A dag-based algorithm for distributed mutual exclusion. In International Conf. On Distributed Computer Systems, p. 354-360, 1991.

[NT87] M. Maimi und M. Trehel. An improvement of the log (N) distributed algorithm for mutual exclusion. Proceedings of the International Conference on Distributed Computing Systems. IEEE Comput. Soc., Columbus Ohio, p. 371-375, 1987.

[NT87/1] 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 International Worhshop on Distributed Algorithms, Vol. 312, p. 155166, 1987.

[NT96] M. Naimi, M. Trehel und A. Arnold. A log(N) distributed exclusion algorithm based on path reversal. Journal of Parallel and Distributed Computing, Vol. 34, p. 1-13, 1996.

[Posix.1b] 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.

[Posix.1c] Portable Operating System Interface (POSIX)---Part 1: System Application Program Interface (API). Technical Committee on Operating Systems and Application Environments of the IEEE. ANSI/IEEE Std 1003.1, 1995 Edition, including 1003.1c: Amendment 2: Threads Extension [C Language], 1996.

[RA81] G. Ricart und A. K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. Comm. Of the ACM, Vol. 24 (1), p. 9-17, 1981.

[RA83] G. Ricart und A. K. Agrawala. Author´s response to on mutual exclusion in computer networks, technical correspondence. Comm. ACM, Vol. 26 (29, p. 146-148, Feb. 1983.

[Ra89] K. Raymond. A tree-based algorithm for distributed mutual exclusion. ACM Trans. on Computer Systems, Vol. 7 (1), p. 61-77, 1989.

[Ray89] K. Raymond. A Distributed Algorithm for Multiple Entries to a critical Section. Information Processing Letters, p. 189-193, Feb. 1989.

[San87] B. A. Sanders. The information structure of distributed mutual exclusion algorithms. ACM trans. Computer Systems, Vol. 5 (3), p. 284-299, Aug. 1987.

[Si89] M. Singhal. A heuristically-aided algorithm for mutual exclusion in distributed systems. IEEE Trans. on Computer, Vol. 38 (5), p. 651-662, May 1989.

[Si92] 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.

[SK85] I. Suzuki und T. Kasami. A distributed mutual exclusion algorithm. ACM Trans. on Computer Systems, Vol. 3 (4), p. 344-349, 1985.

[SR92] P. K. Srimani und R. L. N. Reddy. Another Distributed Algorithm for Multiple Entries to a Critical Section, Information Processing Letters, p. 51-57, Jan. 1992.

[St92] W. R. Stevens. Advanced Programming in the UNIX Enviroment. Addison-Wesley Publishing Company, Reading, 1992.

[TN87] 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.