Ausarbeitungen zum Seminar

# Konzepte und Methoden der Fehlertoleranz

**Sommersemester 2013**

**09. Juli 2013**


CARL VON OSSIETZKY universität OLDENBURG

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

System Software & verteilte Systeme

www.svs.informatik.uni-oldenburg.de

# Inhaltsverzeichnis

# Bewertung fehlertoleranter Systeme

Stephan Balduin
Carl von Ossietzky Universität Oldenburg

***Zusammenfassung*—Moderne Rechensysteme in kritischen Bereichen zeichnen sich durch hohe Fehlertoleranz aus. Zuverlässigkeit und Verfügbarkeit sind unerlässlich, wenn einem System ein Menschenleben anvertraut wird. Um diese Eigenschaften zu gewährleisten, werden Bewertungskriterien benötigt. Im Folgenden wird erläutert, wie die Fehlertoleranz eines Systems bewertet werden kann.**

## I. Einleitung

Rechensysteme in der heutigen Zeit finden in immer mehr Bereichen des Lebens Anwendung. Nicht selten hängen Menschenleben von deren Funktionstüchtigkeit ab. Deshalb wird es zunehmend wichtiger, dass ein System zuverlässig funktioniert. Dagegen wirken unterschiedliche Faktoren, wie Fehler von innerhalb oder außerhalb des Systems oder altersbedingte Ausfälle. Fehler können zum Beispiel durch beschädigte Hardware (Herstellungsfehler, Überhitzen), fehlerhafte Software (Bugs, Schadsoftware) oder falschen Umgang mit diesen Systemkomponenten (falsche Verkabelung der Komponenten, Löschen von Systemdateien) entstehen. Um den Ausfall durch solche Fehler zu verhindern, muss ein System *fehlertolerant* sein.

Systemfehler werden in drei Bereiche unterschieden. Als Anwender sieht man in der Regel den Ausfall oder das Versagen (*Failure*) des Systems. Dieser wird bedingt durch einen fehlerhaften Systemzustand (*Error*), welcher wiederum durch einen Mangel oder Defekt (*Fault*) zu Stande kommt. Diese Mängel können an verschiedenen Stellen auftreten. Die erste Möglichkeit im Lebenszyklus einer Systemkomponente sind Entwurfsfehler, die durch mangelhaften Entwurfsprozess entstehen, beispielsweise Spezifikations- oder Implementierungsfehler. In der nächsten Phase können Mängel durch einen mangelhaften Herstellungsprozess entstehen, z. B. fehlerhafte Materialien oder fehlerhafte Übersetzer. Die letzte Gruppe bilden Mängel, die während dem Betrieb eines Systems entstehen können, unter anderem Verschleissfehler oder Bedienungsfehler. [1]

Sinn und Zweck der Fehlertoleranz ist es, mit solchen Fehlern um zu gehen. Also Fehler fest stellen, Fehler analysieren, diese im Optimalfall zu beheben und das möglichst so, dass der Anwender davon nichts mit bekommt. Die Fehlertoleranz eines Systems wird durch zwei Eigenschaften beschrieben, Zuverlässigkeit und Verfügbarkeit. Soll die Fehlertoleranz eines gegebenen Systems erhöht werden, muss mindestens eine der beiden Eigenschaften verbessert werden. [2]

Um die Effizienz solcher Konzepte überprüfen zu können, muss es Verfahren geben, die die Fehlertoleranz eines gegebenen Systems bewerten. Dadurch lässt sich entscheiden, ob eine Maßnahmen zur Erhöhung der Fehlertoleranz erfolgreich war. Folglich werden Bewertungskriterien für solche Verfahren benötigt, die idealerweise quantifizierbar sind.

## II. Wahrscheinlichkeitstheoretischer Hintergrund

Jedes System kann sich in genau einem von zwei Fehlerzuständen befinden. Ein System ist *funktionstüchtig* oder *intakt*, wenn es seine Spezifikationen fehlerfrei erfüllt. Ein System ist *ausgefallen* oder *defekt*, wenn es nicht funktionstüchtig ist. [3] Ein System besteht in der Regel aus mehreren Komponenten, die wiederum ein eigenes System darstellen. Diese können ebenfalls intakt oder defekt sein. Der Zustand eines Systems hängt vom Zustand seiner Komponenten ab. Die Wahrscheinlichkeit, dass ein System funktionstüchtig ist, nennt man *Intaktwahrscheinlichkeit*. Die Intaktwahrscheinlichkeit des Systems hängt von der Intaktwahrscheinlichkeit der einzelnen Komponenten ab.

Wie und in welcher Weise hängt von der Art des Systems ab. Dazu betrachte die Intaktwahrscheinlichkeit $p$ eines gegebenen Systems $S$. Die einzelnen Komponenten von $S$ seien $K_1, \ldots, K_n$ und die Intaktwahrscheinlichkeit der Komponente

$K_i$ sei mit $p_i$ bezeichnet. Die Komponenten in $S$ können entweder ein Seriensystem $S_{Ser}$ oder ein Parallelsystem $S_{Par}$ bilden. Ein Seriensystem ist genau dann intakt, wenn alle Komponenten intakt sind. Ein Parallelsystem ist genau dann intakt, wenn mindestens eine Komponente intakt ist. [3] Für die Intaktwahrscheinlichkeit heißt das

$$p_{Ser} = p_1 * \cdots * p_n$$

und

$$p_{Par} = 1 - (1 - p_1) * \cdots * (1 - p_n),$$

wenn man von der Unabhängigkeit der einzelnen Komponenten ausgeht. Daraus lässt sich folgern, dass Seriensysteme eine geringere und Parallelsysteme eine größere Intaktwahrscheinlichkeit haben als jede einzelne ihrer Komponenten.

Der Zustand eines Systems lässt sich ermitteln, indem für jede Komponente geprüft wird, ob diese intakt ist. Beobachtet man ein System über einen bestimmten Zeitraum in regelmäßigen oder zufälligen Abständen, liefert jede Beobachtung ein unabhängiges Ergebnis, ob das System intakt ist oder nicht. Da dieses Ergebnis nicht vorher berechenbar ist, kann man diese Beobachtungen als Zufallsexperiment betrachten. Der Zustand des Systems wird als Zufallsvariable aufgefasst. Dadurch lassen sich verschiedene Konzepte der Wahrscheinlichkeitstheorie anwenden. Ein wichtiges ist die Verteilungsfunktion einer Zufallsvariable $X$, hier gegeben durch

$$F(t) = P(X \leq t),\ 0 \leq t < \infty,$$

da die Lebensdauer sinngemäß nur nicht-negative Werte annehmen kann. Diese ergibt sich aus der Integration der Wahrscheinlichkeitsdichte (*probability density function*, pdf) und liefert Informationen über das Verhalten der Zufallsvariable $X$. Für die Bewertung ist zunächst jedoch nur der arithmetische Mittelwert oder Erwartungswert $E(X)$ von Interesse. Dieser gibt eine grobe Auskunft über die durchschnittlich zu erwartende Lebenszeit des Systems. Konkret also, ob das System zum Zeitpunkt $t$ voraussichtlich funktionstüchtig sein wird. Neben der Verteilungsfunktion gibt es noch weitere spezielle Verteilungen, wie die Normalverteilung oder die Weibullverteilung [3]. Letztere ist die klassische Zuverlässigkeitsanalyse, gegeben durch [3]

$$F(t) = 1 - e^{-\lambda t^b},\ \text{für } t, \lambda, b \geq 0$$

Weibullverteilungen hängen von zwei Parametern ab, einem Formparameter $b$ und einem Skalierungsparameter $\lambda$. Mit dem Formparameter können Früh- bzw. Spätausfälle modelliert werden, je nachdem, ob $b < 1$ oder $b > 1$ gewählt wird. Der Skalierungsparameter wird bei der Zuverlässigkeitsanalyse auch durch seinen Kehrwert dargestellt. Dadurch kann die durchschnittliche Lebensdauer verändert werden. [4] Eine weitere Verteilung, die Exponentialverteilung, ergibt sich, wenn eine Weibullverteilung mit $b = 1$ gewählt wird.

$$F(t) = \begin{cases} 1 - e^{-\lambda t}, & \text{für } 0 < t \leq \infty \\ 0, & \text{sonst} \end{cases}.$$

Die Exponentialverteilung hat die Dichtefunktion

$$f(t) = \begin{cases} \lambda e^{-\lambda t}, & \text{für } t > 0 \\ 0, & \text{sonst} \end{cases}.$$

Der Parameter $\lambda$ steuert hierbei, wie schnell die Funktion für $t \to \infty$ gegen 0 konvergiert. [5] Die Exponentialverteilung (zu sehen in Abb. 1) wird meistens bei der allgemeinen Bewertung der Zuverlässigkeit eines gegebenen Systems verwendet. Ein Grund dafür ist ihre besondere Eigenschaft, die Gedächtnislosigkeit oder Markov Eigenschaft genannt wird. [2] Diese besagt, dass die Verteilung zu jedem Zeitpunkt exponentiell bleibt, unabhängig davon, wie viel Zeit zwischen zwei Betrachtungen vergangen ist. Dies hat den Vorteil, dass es bei der Bewertung irrelevant ist, wie lange das zu betrachtende System bereits am Laufen ist.

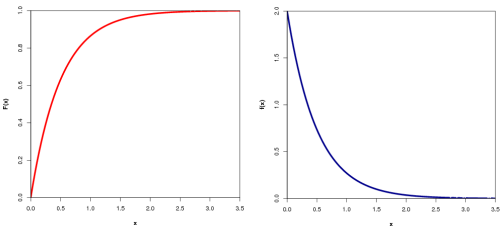


Abbildung 1. Allgemeine Dichte (l.) und Verteilungsfunktion (r.) der Exponentialverteilung [7]

## III. Zuverlässigkeit (Reliability)

Zuverlässigkeit ist ein bedeutendes Ziel von Fehlertoleranzkonzepten, welches sich jedoch nicht exakt berechnen lässt. Mit den oben genannten wahrscheinlichkeitstheoretischen Grundlagen lässt sich Zuverlässigkeit aber realistisch abschätzen. Dabei wird die Lebenszeit eines Systems durch eine Zufallsvariable *X* dargestellt. *X* steht für die Zeit, die vergeht, bis das System ausfällt (*time to failure*). Die Zuverlässigkeit eines Systems wird durch die Funktion *R(t)* definiert. Dies ist die Wahrscheinlichkeit, dass das System zum Zeitpunkt *t* noch intakt ist, also

$$R(t) = P(X > t) = 1 - F(t)$$

mit

$$R(t) = e^{-\lambda t},$$

falls *X* exponential-verteilt ist. [2] Diese Formel ermöglicht die Quantifizierung der Lebenszeit eines Systems schon bevor dieses ausgefallen ist. Man betrachtet jedoch weniger die reine Lebenszeit, sondern eher die erwartete Lebenszeit. Die erwartete Lebenszeit oder durchschnittliche Zeit bis zum Versagen (*mean time to failure,MTTF*) lässt sich aus der Lebenszeit eines Systems berechnen und hat die Formel

$$E[X] = \int_0^\infty t f(t) dt = -\int_0^\infty t R(t) dt$$

bzw.

$$MTTF = \frac{1}{\lambda},$$

wenn $X$ exponential verteilt ist. [2] Dieser Ausdruck stellt die erwartete Lebenszeit eines einzelnen isolierten Systems dar. In der Praxis bestehen Systeme aber fast immer aus einer Menge von Komponenten. Diese können seriell oder parallel miteinander verknüpft sein und jede dieser Komponenten hat eine eigene erwartete Lebenszeit. Sei die $i$-te Komponente eines Systems aus $n$ Komponenten exponentiell verteilt mit dem Parameter $\lambda_i$. In einem Seriensystem ist die Zuverlässigkeit eines Systems gegeben als Produkt der Zuverlässigkeit der einzelnen Komponenten. Für den Parameter $\lambda$ aus der MTTF-Formel heißt das $\lambda = \lambda_1 + \cdots + \lambda_n$. Setzt man dieses nun ein, ergibt sich: [2]

$$MTTF_{Ser} = \frac{1}{\sum_{i=1}^{n} \lambda_i}$$

Hier wird erneut deutlich, dass die Zuverlässigkeit eines Seriensystems kleiner als die seiner Komponenten ist. Genau umgekehrt verhält es sich bei einem Parallelsystem. Dieses fällt aus, wenn alle seine Komponenten ausfallen. Anders ausgedrückt hat ein Parallelsystem die gleiche Lebenszeit wie die Komponente mit der höchsten Lebenszeit. Für die erwartete Lebenszeit ergibt sich dann: [2]

$$MTTF_{Par} = \frac{1}{\lambda} * \sum_{i=1}^{n} \frac{1}{i}$$

wieder unter der Voraussetzung, das X exponential verteilt ist. Häufig wird ein Konzept eingesetzt, welches ähnlich wie ein Parallelsystem funktioniert. Hierbei geht man davon aus, dass für jede Komponente $K_i$ eine Menge von $n$ weiteren Komponenten vorhanden sind, die exakt die gleiche Funktion erfüllen und parallel zu $K_i$ arbeiten. Der Unterschied zu einem normalen Parallelsystem liegt darin, dass diese redundanten Komponenten nicht aktiv arbeiten. Sie befinden sich in einer Art Bereitschaftsmodus und kommen erst dann zum Einsatz, wenn $K_i$ ausfällt. Dieses Konzept nennt sich passive Redundanz (*standby redundancy*). Ein passiv redundantes System hat exponential verteilt eine erwartete Lebenszeit von

$$MTTF_{sr} = \frac{n}{\lambda}.$$

Daraus folgt, je größer die Anzahl der redundanten Komponenten, desto höher ist die Zuverlässigkeit eines Systems.

## IV. Verfügbarkeit (Availability)

Das zweite große Ziel der Fehlertoleranz ist die Verfügbarkeit eines Systems. Verfügbarkeit ist die Zeit, in der ein System richtig funktioniert oder funktionstüchtig ist. Allerdings wird hierbei nicht nur die Zeit bis zum ersten Ausfall betrachtet, sondern auch die Zeit, während das System nicht mehr funktionstüchtig ist. Die Verfügbarkeit $A$ gibt also das Verhältnis von der tatsächlichen Lebenszeit einschließlich Ausfallzeiten zur gesamten Lebenszeit an, also:

$$A = \frac{Gesamtlebenszeit - Gesamtausfallzeit}{Gesamtlebenszeit}$$

Eine Verfügbarkeit von 100% ist nur dann möglich ist, wenn das System zu keinem Zeitpunkt defekt

ist. Bei der Ausfallzeit werden nicht nur fehlerbedingte Ausfälle betrachtet, sondern ebenfalls geplante Ausfälle, etwa reguläre Wartungsarbeiten, die nicht bei laufendem Betrieb durch geführt werden können.

Neben der erwarteten Lebenszeit (MTTF) aus der Zuverlässigkeit wird ein weiterer Wert betrachtet: die durchschnittliche Zeit für die Reparatur (*mean time to repair, MTTR*). Dieser beschreibt die Zeit, die benötigt wird, um eine defekte Komponente zu reparieren oder durch eine neue zu ersetzen und kann als Ausfallzeit eines Systems betrachtet werden.

Die momentane Verfügbarkeit (*instantaneous availability*) $A(t)$ einer Komponente wird definiert als die Wahrscheinlichkeit, dass diese Komponente zum Zeitpunkt t funktionstüchtig ist. [2] Nach dieser Definition wären $R(t)$ und $A(t)$ äquivalent. Der Unterschied liegt in der MTTR, die bei $A(t)$ zusätzlich betrachtet wird. Hier bei können zwei Fälle eintreten. Der erste Fall geht davon aus, dass das gegebene System seit dem Zeitpunkt $t = 0$ funktionstüchtig ist. Die Verfügbarkeit entspricht hier der Zuverlässigkeit. Im zweiten Fall ist das System zum Zeitpunkt $x$, mit $0 < x < t$ bereits ausgefallen. Die Verfügbarkeit hat jetzt die Wahrscheinlichkeit [6]

$$A(t) = R(t) + \int_0^t R(t-x) * m(x)dx,$$

wobei $m(x)$ die Erneuerungsdichte (*renewal density*) [8] ist. Eine weitere Eigenschaft ist das Verhalten, wenn $t$ gegen unendlich läuft. Für die Zuverlässigkeit $R$ ergibt sich zwangsläufig

$$\lim_{t\to\infty} R(t) = 0,$$

da jedes System zu irgendeinem Zeitpunkt einmal ausfallen wird. Anders dagegen die Verfügbarkeit $A$, die im Normalfall ungleich null ist: [6]

$$\lim_{t\to\infty} A(t) = A.$$

Die Betrachtung des Grenzwertes wird stetige Verfügbarkeit (*steady state availability*) genannt und ist gegeben durch:

$$A = \frac{MTTF}{MTTF + MTTR}$$

## V. Zusammenfassung

Ein System besteht aus mehreren Komponenten, welche jeweils wieder ein eigenes System bilden. Diese können auf verschiedene Weise mit einander verknüpft sein. In einem Seriensystem müssen alle, in einem Parallelsystem mindestens eine Komponente funktionstüchtig sein, damit das System ebenfalls funktionstüchtig ist. Da Fehler immer auftreten können, bedarf es bestimmter Techniken, damit diese nicht zum Ausfall des gesamtem Systems führen. Fehlertoleranz soll dem System einen gewissen Grad an Fehlern gestatten, ohne einen Ausfall zu verursachen. Die Ziele von Fehlertoleranzkonzepten sind es, die Zuverlässigkeit und die Verfügbarkeit eines gegebenen Systems zu erhöhen. Zuverlässigkeit ist die Lebensdauer eines Systems von dessen Start an und beschreibt, wie lange das System funktionstüchtig ist, bis es ausfällt. Momentane Verfügbarkeit ist die Wahrscheinlichkeit, dass ein System zu einem beliebigen Zeitpunkt funktionstüchtig ist. Die Lebensdauer lässt sich mit Hilfe der Wahrscheinlichkeitstheorie als Zufallsvariable auffassen. Betrachtet man den Erwartungswert der Lebensdauer, erhält man die durchschnittliche Zeit bis zum Versagen (*MTTF*) des Systems. Die Verfügbarkeit eines Systems ergibt sich, wenn man die Möglichkeit betrachtet, dass ausgefallene Komponenten repariert oder ersetzt werden. Dies führt zum Wert der stetigen Verfügbarkeit, dem Grenzwert der momentanen Verfügbarkeit, wenn man die Zeit gegen unendlich laufen lässt.

## Literatur


[1] Oliver Theel, *Fehlertoleranz in verteilten Systemen* Vorlesungsfolien, Universität Oldenburg *http://dampf.svs.informatik.uni-oldenburg.de/lehre/ftvs/FVS-Folienkopien-02.pdf* (am 18. Juni 2013)

[2] Pankaj Jalote *Fault Tolerance in Distributed Systems* 1994: Prentice Hall, London

[3] *Buch Zuverlässigkeit* Universität Oldenburg

[4] Frederik Schaefer *http://www.exponentialverteilung.de/weibullverteilung.html* (am 18. Juni 2013)

[5] Fahrmeir, Ludwig / Pigeot, Iris / Tutz, Gerhard *Statistik. der Weg zur Datenanalyse ;* (2011): Heidelberg (Springer-Verlag GmbH).

[6] Kishor S. Trivedi, *http://amod.ee.duke.edu/definitions.htm* last update 2001 (am 2. Juni 2013)

[7] Humboldt Universität Berlin *http://mars.wiwi.hu-berlin.de/mediawiki/mmstat_de/index.php/Verteilungsmodelle_-_STAT-Exponentialverteilung* (am 2. Juni 2013)

[8] Universität Ulm *http://www.mathematik.uni-ulm.de/ stochastik/lehre/ss05/wt/skript/node12.html* (am 2. Juni 2013)

# Asynchronous Checkpointing and Rollback

Jan-Gerd Meß, Universität Oldenburg

***Zusammenfassung*—Verteilte Systeme sind ein wichtiger Bestandteil der modernen IT, doch ihre Zuverlässigkeit und Erreichbarkeit werfen neue Probleme auf. Nicht nur die einzelnen Prozesse innerhalb eines verteilten Systems, sondern auch der Status der Nachrichtenkanäle muss im Falle eines Ausfalls wiederhergestellt werden können. In dieser Arbeit werden einige Verfahren vorgestellt, die es ermöglichen, nach einem teilweisen Ausfall in einem verteilten System wieder einen konsistenten Zustand herzustellen, ohne dabei den normalen Betrieb zu stören oder zu verlangsamen.**

***Keywords*—*Distributed Systems, Fault Tolerance, Asynchronous Checkpointing, Rollback.***

## I. Einleitung

### A. Motivation

Rechnernetze und verteilte Systeme haben in der modernen IT einen wichtigen Stellenwert gewonnen. World Wide Web, Cloud Services oder Sensornetzwerke, etwa in Automobilen oder Flugzeugen, um nur einige zu nennen, haben den modernen Alltag durchdrungen. In einem verteilten System arbeiten dabei mehrere Prozesse kooperativ an der Lösung eines Problems und kommunizieren über Nachrichten. Als fester Bestandteil des täglichen Lebens ist die Zuverlässigkeit dieser Systeme - vor allem in kritischen Anwendungen - von großer Wichtigkeit [2].

Durch die Verteilung dieser Systeme ergeben sich im Fall von teilweisen Ausfällen jedoch Probleme, die nicht - wie im Fall von Ein-Prozess-Systemen - mit einfachen Backup-Mechanismen gelöst werden können, denn durch die Kommunikation zwischen den Prozessen entstehen Abhängigkeiten, die bei einer Wiederherstellung des verteilten Systems berücksichtigt werden müssen [1]. Fällt ein Prozess aus, muss außerdem nicht nur sein eigener Status wiederhergestellt werden, sondern das gesamte verteilte System muss wieder in einen konsistenten Zustand überführt werden. Dazu gehören nicht nur die Zustände der einzelnen Prozesse, sondern auch die der Kommunikationsleitungen. „Konsistent“ bedeutet in diesem Fall intuitiv, dass das Gesamtsystem in einen Zustand überführt wird, der während einer fehlerfreien Ausführung hätte auftreten können. Die genaue Definition eines konsistenten Zustands wird in Abschnitt II-A gegeben.

### B. Lösungsansatz

Ein *Checkpoint* ist ein Abbild des momentanen Zustands eines Prozesses, das ausreicht, um im Falle eines Ausfalls den gespeicherten Zustand wiederherzustellen. Zu einem Checkpoint gehören also beispielsweise Informationen über Variablenbelegung und die aktuell auszuführenden Anweisungen.

Während des normalen Betriebs werden solche Checkpoints von den einzelnen Prozessen auf einem stabilen Speicher abgelegt. Wird nun ein Ausfall registriert, müssen die verteilten Prozesse eine Menge dieser Checkpoints finden - einen pro Prozess -, sodass das verteilte System in einen Zustand überführt werden kann, der auch während einer ausfallfreien Ausführung hätte auftreten können. Dazu sollen in dieser Ausarbeitung Protokolle vorgestellt und untereinander verglichen werden, die jeweils unterschiedliche Ansätze verfolgen.

Dabei werden nur sogenannte asynchrone Verfahren betrachtet. Das bedeutet, dass die Prozesse ihre Checkpoints *unabhängig* voneinander und *ohne Kontrollinstanz* anlegen und im Falle eines Ausfalls wiederherstellen.

### C. Gliederung

Im Abschnitt II werden zunächst eine grundlegende Definitionen eingeführt. Danach wird eine Klassifizierung von Checkpointing- und Recovery-Protokollen vorgenommen, und es wird das Fehlermodell erklärt. Im anschließenden Abschnitt III werden dann verschiedene Protokolle für asynchrone Wiederherstellung in verteilten Systemen betrachtet. Zunächst werden checkpoint-basierte Verfahren vorgestellt, bevor auch ein log-basiertes Protokoll beschrieben wird. Schließlich bietet Abschnitt IV eine Zusammenfassung für die vorgestellte Thematik von asynchronen Checkpointing- und Recovery-Verfahren.

## II. Grundlagen

In diesem Abschnitt werden zunächst einige grundlegende Definitionen eingeführt, die für das weitere Verständnis dieser Arbeit von Bedeutung sind.

Anschließend erfolgt eine Klassifizierung von Checkpointing- und Recovery-Algorithmen allgemein vorgenommen. Im weiteren Verlauf wird dann nur noch die Klasse der asynchronen Verfahren näher betrachtet.

Im letzten Abschnitt dieser Sektion wird schließlich das zugrundeliegende Fehlermodell eingeführt.

### A. Definitionen

*1) Transienter Ausfall:* Ein *Ausfall* eines Prozesses bedeutet, dass im Prozess oder in der Hardware, auf der der Prozess läuft, ein kritischer Fehler aufgetreten ist, sodass der Prozess seinen aktuellen Zustand, also unter anderem Berechnungen und Variablenbelegungen, verliert. *Transient* bedeutet, dass der Ausfall zeitlich begrenzt ist und der Prozess beispielsweise nach einem Neustart in endlicher Zeit wieder zur Verfügung steht.

*2) Stabiler Speicher: Stabiler Speicher* ist Speicher, der - im Gegensatz zu flüchtigem Speicher - auch Ausfälle von Prozessen ohne Datenverlust übersteht, und sich nach einem Neustart des Prozesses im gleichen Zustand wie vor dem Ausfall befindet [2]. Für diese Arbeit wird angenommen, dass jedem Prozess ein solcher stabiler Speicher zur Verfügung steht, ohne näher auf die Implementierung einzugehen.

*3) Checkpoint:* Ein *Checkpoint* (Sicherungspunkt) eines Prozesses zu einem bestimmten Zeitpunkt ist eine Sammlung aller Informationen, die benötigt werden, um den Zustand des Prozesses zu diesem Zeitpunkt wiederherstellen zu können, falls der Prozess ausfällt. Checkpoints werden in regelmäßigen Abständen von den Prozessen angelegt und auf stabilem Speicher gesichert, damit sie nach einem Ausfall zur Verfügung stehen [2].

*4) Rollback:* Ein *Rollback* bezeichnet die Wiederherstellung eines Zustands, der zeitlich vor dem aktuellen Zustand aufgetreten ist. Ein Prozess unternimmt einen Rollback, indem er einen Checkpoint aus dem stabilen Speicher lädt und aus ihm seinen neuen Zustand ableitet. Informationen und Berechnung, die zeitlich vor diesem Zustand liegen, werden vergessen. Beispielsweise *rollt* ein Prozess während seines Neustarts nach einem Ausfall auf den aktuellsten Checkpoint auf dem stabilen Speicher *zurück* [2].

*5) Waisen-Nachricht:* Eine *Waisen-Nachricht* bezeichnet eine Nachricht, die zwar vom Empfänger empfangen, vom Sender aber im aktuellen Zustand aber nie abgeschickt wurde [1]. Dies kann auftreten, wenn beispielsweise der Prozess $P_i$ an den Prozess $P_j$ eine Nachricht $m$ schickt. $P_j$ erhält diese Nachricht, doch kurz darauf fällt $P_i$ aus und muss auf einen Zustand zurückrollen, der zeitlich vor dem Senden von $m$ liegt. Somit befindet sich das System in einem Zustand, in dem $m$ von $P_j$ zwar empfangen, von $P_i$ im aktuellen Zustand aber (noch) nicht versendet wurde. $m$ ist eine Waisen-Nachricht.

Waisennachrichten müssen vermieden werden, weil $P_i$ wie im Beispiel nicht garantieren kann, dass bei einer erneuten Ausführung nach dem Ausfall $m$ noch einmal und mit gleichem Inhalt verschickt wird. $P_j$ würde also mit Informationen weiterarbeiten, deren Korrektheit nicht gewährleistet werden kann.

*6) Konsistenter Zustand:* In der Literatur finden sich unterschiedliche Auffassungen eines konsistenten Zustands [2], [3], [1], je nachdem, ob *verloren Nachrichten* - also Nachrichten, die zwar abgeschickt, aber noch nicht empfangen wurden - erlaubt sind oder nicht. In dieser Arbeit wird jedoch die Definition aus [3], [1] verwendet: Ein System befindet sich genau dann in einem konsistenten Zustand, wenn es keine Waisen-Nachrichten gibt. In diesem Fall ist es Aufgabe des Kommunikationsprotokolls, für das erneute Versenden der verlorenen Nachricht zu sorgen.

*7) Konsistente Menge von Checkpoints* und *Recovery Line:* Eine *konsistente Menge von Checkpoints* ist eine Menge, die pro Prozess aus einem Checkpoint dieses Prozesses besteht. Sie ist genau dann konsistent, wenn der Zustand des verteilten Systems, der entsteht, wenn alle Prozesse auf die Checkpoints der Menge zurückgerollt werden, konsistent ist.

Die *Recovery Line* ist die jüngst-mögliche dieser Mengen, das heißt, dass für jeden Prozess der aktuellst-mögliche Checkpoint gewählt wurde, sodass noch eine konsistente Menge entsteht.

*8) Domino Effekt:* Der *Domino Effekt* beschreibt ein Problem, dass bei ungünstiger Verteilung der versendeten Nachrichten auftreten kann [2]: Durch vielfältige Abhängigkeiten der Prozesse untereinander kann keine konsistente Menge von Checkpoints gefunden werden, sodass das verteilte System insgesamt sehr weit, schlimmstenfalls bis zu seinem Initialzustand zurückrollen muss.

Checkpointing- und Rollbackverfahren versuchen stets, eine möglichst aktuelle konsistente Menge von Checkpoints zu finden und insbesondere den Domino Effekt zu vermeiden.

### *B. Klassifizierung*

Es gibt viele verschiedene Möglichkeiten, wie ein Prozess die Zeitpunkte festlegen kann, zu denen er Checkpoints anlegt. [1] unterscheidet dabei drei grundlegende Klassen von Verfahren beziehungsweise Protokollen:

- Synchrone (koordinierte) Verfahren
- Kommunikations-induzierte Verfahren
- Asynchrone (unkoordinierte) Verfahren

In *synchronen Verfahren* gibt es entweder Kontrollinstanzen, die bestimmen, wann die einzelnen Prozesse Checkpoints anlegen müssen, oder die Prozesse einigen sich auf Zeitpunkte für das Anlegen von Checkpoints. Der Vorteil ist die schnelle Wiederherstellung eines konsistenten Zustands, da die Menge der konsistenten Checkpoints bekannt ist und nicht zunächst die Informationen der einzelnen Prozesse über ihre Checkpoints und Abhängigkeiten ausgetauscht werden müssen. Die Nachteile sind jedoch 1) der höhrere Aufwand während des normalen Betriebs, weil gemeinsame Zeitpunkte für das Anlegen von Checkpoints gefunden werden müssen, und 2) der Geschwindigkeitsverlust durch die Synchronisierung.

Die *kommunikations-induzierten Verfahren* nutzen keine expliziten Verwaltungs-Nachrichten oder Kontrollinstanzen, sondern hängen Informationen an die Datennachrichten an (sog. *Piggybacking*), um zu bestimmen, wann das Anlegen eines Checkpoints nötig ist. Das vermindert die Anzahl der Nachrichten während des Betriebs, verhindert jedoch trotzdem den Domino Effekt.

Bei *asynchronen Verfahren* legen die Prozesse ihre Checkpoints völlig unabhängig voneinander an. Wenn ein Prozess ausfällt, dann müssen aus der Menge dieser Checkpoints eine konsistente Menge berechnet und die Prozesse entsprechend wiederhergestellt werden. Die Vorteile sind, dass diese Verfahren im normalen Betrieb nahezu ohne Kommunikations-Overhead auskommen und die Prozesse nicht durch Synchronisation gebremst werden. Ein Nachteil ist jedoch, dass im Falle eines Ausfalls von ein oder mehreren Prozessen ein deutlich größerer Aufwand betrieben werden muss, um eine konsistente Menge von Checkpoints zu berechnen. Dieser ist nicht zu unterschätzen, da Informationen über die Checkpoints der einzelnen Prozesse und der Abhängigkeiten meist nur lokal vorliegen und erst zusammengetragen werden müssen. Außerdem besteht immer die Gefahr, dass durch ungünstige Abhängigkeiten die Recovery Line gar gleich dem Initialzustand ist [2].

Diese Arbeit wird sich nun nur den asynchronen Verfahren widmen, die besonders dann zum Einsatz kommen, wenn die unterschiedlichen Komponenten ohnehin schon hohe Verfügbarkeiten aufweisen und der höhere Aufwand während des normalen Betriebs im Vergleich zu der selten auftretenden Wiederherstellung nicht gerechtfertigt wäre.

Die asynchronen Verfahren lassen sich wiederum in zwei Kategorien einteilen [1]:

- checkpoint-basierte Verfahren
- optimistische log-basierte Verfahren

Bei checkpoint-basierten Verfahren legen die Prozesse unabhängig voneinander in regelmäßigen Abständen Checkpoints auf einem stabilen Speicher ab. Im Fehlerfall sorgt ein verteilter Algorithmus dafür, dass aus den gespeicherten Checkpoints ein konsistenter Systemzustand wiederhergestellt wird (sog. *Rollback*).

Log-basierte Verfahren im Allgemeinen schreiben dagegen nicht nur Checkpoints, sondern auch Informationen über eingehende Nachrichten auf den stabilen Speicher, wie etwa den Sender, den Inhalt und den Zeitpunkt des Eintreffens der Nachricht. Diese Informationen werden genutzt, um das Systemverhalten bis zu einem gewissen Punkt nachspielen zu können, sodass sich ein aktuellerer Zustand als bei den checkpoint-basierten Verfahren wiederherstellen lässt. Allerdings benötigt diese Methode deutlich mehr Speicherplatz und erzeugt größeren Verwaltungsaufwand.

### C. Fehlermodell

Bevor die einzelnen Protokolle betrachtet werden können, muss ein einheitliches Fehlermodell geschaffen werden, um die Verfahren vergleichen zu können. Daher werden folgende Annahmen getroffen [2], [3]:

- Das betrachtete System besteht aus $n$ Prozessen $P_0, ..., P_{n-1}$
- Die beteiligten Prozesse $P_i$ verfügen über keinen gemeinsamen Speicher, jedoch werden die Nachrichtenkanäle als verlustfrei angesehen und die Kanäle stellen sicher, dass die Nachrichten in der Reihenfolge empfangen werden, in der sie auch abgeschickt wurden.
- Die Prozesse verfügen jeweils über einen stabilen Speicher, der auch im Falle eines Ausfalls erhalten bleibt und auf dem die Checkpoints abgelegt werden können.
- Der Ausfälle von Prozessen $P_i$ sind immer transient, das heißt, die Topologie des Netzwerkes nach Wiederherstellung von $P_i$ ist die gleiche wie vor dem Ausfall von $P_i$.
- Alle Prozesse sind komplett vernetzt, das heißt, jeder Prozess kann jedem anderen Prozess Nachrichten schicken.
- Während der Wiederherstellung tritt kein weiterer Ausfall auf. Angenommen, es könnte doch ein weiterer Ausfall auftreten, so müsste der jeweilige Algorithmus so lange neu gestartet werden, bis er ohne Prozessausfall zum Halten kommt.

Basierend auf diesem Fehlermodell werden im nächsten Abschnitte nun einige Rollbackverfahren besprochen.

## III. Recovery-Protokolle

In diesem Abschnitt werden nun einige Verfahren zum asynchronen Erstellen von Checkpoints und Wiederherstellen eines konsistenten Zustands in verteilten Systemen vorgestellt. Zunächst wird dazu das grundsätzliche Vorgehen beschrieben, bevor auf das konkrete Protokoll nach Bidyut Gupta und Shahram Rahimi (siehe III-B1) eingegangen wird. Da jedoch gezeigt wurde, dass dieser Algorithmus zu inkonsistenten Zuständen führen kann [4], wird gleich auch die Verbesserung durch Monika Kapus-Kolar eingeführt [4], die die Vorteile des Algorithmus erhält, aber gleichzeitig seine Korrektheit herstellt. Danach folgt ein Abschnitt über log-basierte Verfahren, in dem ein Algorithmus nach David B. Johnson und Willy Zwaenepoel beschrieben wird.

### A. Grundsätzliches Vorgehen

Ziel der Recovery-Algorithmen ist das Auffinden der Recovery Line. Im Gegensatz zu Ein-Prozess-Systemen reicht es dabei jedoch nicht, die einzelnen Prozesse unabhängig voneinander zurückzusetzen, sondern es muss auch der Status des Netzwerks betrachtet werden. Durch die Kommunikation zwischen den Prozessen entstehen Abhängigkeiten, die bei der Suche nach einer Recovery Line beachtet werden müssen [2].

Dazu kann ein *Recovery-Dependency-Graph* genutzt werden. Dieser Graph ist wie folgt aufgebaut: Als Knoten des Graphen dienen die Checkpoints $c_{i,r}$, wobei $c_{i,r}$ den r-ten Checkpoint des Prozesses $P_i$ darstellt.

Das Zeitintervall zwischen $c_{i,r}$ und $c_{i,r+1}$ sei mit $I_{i,r+1}$ bezeichnet. Dann existiert eine gerichtete Kante von $c_{i,r}$ nach $c_{j,p}$, falls eine der beiden Bedingungen erfüllt ist [1]:

- $i = j$ und $p = r + 1$
- $i \neq j$ und in $I_{i,r}$ wird eine Nachricht von $P_i$ an $P_j$ geschickt und von diesem in $I_{j,p}$ empfangen.

Die Abbildungen 1 und 2 zeigen eine beispielhafte Kommunikation zwischen drei Prozessen beziehungsweise den zugehörigen Recovery-Dependency-Graphen. Der rote Punkt in Abbildung 1 zeigt den Ausfall von Prozess $P_0$ an. Die Wolken in Abbildung 2 zeigen den momentanen Status der Prozesse, der noch nicht durch einen Checkpoint gesichert wurde. Die Nachrichten werden folgendermaßen benannt: $m_{1,0,3}$ ist beispielsweise die 3. Nachricht, die von Prozess $P_1$ an Prozess $P_0$ geschickt wurde. Die Farbe der Nachrichtenpfeile spielt in Abschnitt III-C eine Rolle: Nachrichten mit grünen Pfeilen wurden bereits auf stabilem Speicher gesichert, während orange-farbene Pfeile noch im flüchtigen Speicher gehalten werden und im Falle eines Ausfalls verloren wären.

Um auf einem solchen Recovery-Dependency-Graphen nun eine Recovery Line zu finden, werden ausgehend vom letzten Checkpoint vor dem Ausfall (hier: $c_{0,2}$) alle über gerichtete Kanten erreichbaren Knoten markiert. Die aktuellsten nichtmarkierten Knoten eines jeden Prozesses bilden zusammen die Recovery Line.

Dieses Verfahren basiert allerdings auf vollständiger Information über die Kommunikation zwischen den Prozessen, die im Falle asynchron arbeitender Prozesse nicht gegeben ist, da es explizit keine Kontrollinstanzen gibt. Es muss also ein

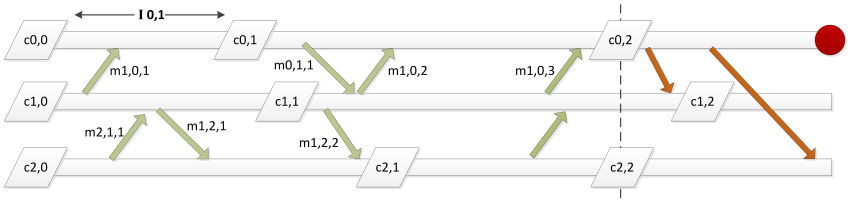



Abbildung 1. Beispielhafte Kommunikation mit drei Prozessen

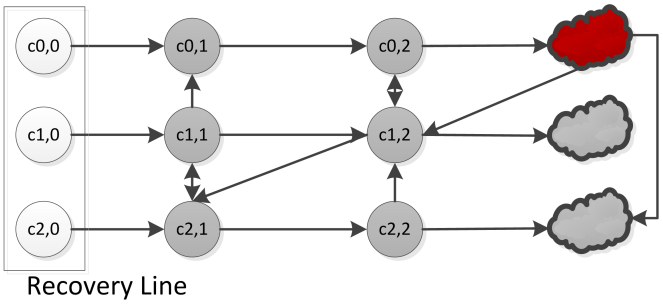



Abbildung 2. Rollback-Dependency Graph

verteilter Algorithmus gefunden werden, der das Problem löst. Dazu werden im Folgenden zwei Klassen von Algorithmen, jeweils mit einem konkreten Vertreter, vorgestellt.

### B. *Checkpoint-basierte Protokolle*

In checkpoint-basierten Verfahren legen die Prozesse in regelmäßigen Abständen Checkpoints auf einen stabilen Speicher. Im Falle eines Prozessausfalls wird dann mithilfe eines verteilten Algorithmus eine Recovery Line gesucht. Ein konkreter Vertreter der Klasse der asynchronen Verfahren, der Algorithmus nach Gupta und Rahimi, wird im Folgenden vorgestellt.

*1) Algorithmus nach Gupta und Rahimi mit Verbesserungen nach Kapus-Kolar [3], [4]:* Grundsätzlich müssten im Falle eines Ausfalls sämtliche Checkpoints miteinander verglichen werden, um die konsistenten Mengen zu bestimmen. Dieser Algorithmus zielt nun darauf ab, die Anzahl der Vergleiche zu reduzieren, indem die empfangenen und versandten Nachrichten gezählt werden, um so den Aufwand für eine Wiederherstellung zu verkleinern. Die Checkpoints selbst werden von jedem Prozess willkürlich angelegt.

Dazu speichert jeder Prozess $P_i$ bei jedem Checkpoint einen n-dimensionalen Vektor $V_i = (v_{i,0}, ..., v_{i,n-1})$ mit $v_{i,j} = V_i(j)$ und $v_{i,j}$ speichert die Anzahl der Nachrichten von Prozess $P_i$ zu Prozess $P_j$ bis zu diesem Checkpoint. Dabei gilt

$$\forall i \in \{1, ..., n\} : v_{i,i} = 0$$

.

Außerdem speichert jeder Prozess eine Liste von n-dimensionalen Vektoren $R_i =< r_{i,0}, ..., r_{i,r}, ... >$ wobei

$$r_{i,r} = (r_{i,r,0}, ..., r_{i,r,n-1})$$

und $R_i(r, j) = r_{i,r,j}$. $R_i(r, j)$ speichert dabei alle bis zum Checkpoint $r$ von Prozess $P_j$ an Prozess $P_i$ versandten Nachrichten. Die Erweiterung von Kapus-Kolar liegt hier darin, dass in der Liste nicht nur die Summe aller erhaltenen Nachrichten festgehalten wird, sondern die empfangenen Nachrichten jedes einzelnen Senders gespeichert werden [3], [4].

Zuletzt gibt es noch ein Merker vom Typ $boolean$, der für die Abbruchbedingung des Algorithmus benötigt wird.

Angenommen, Prozess $P_i$ fällt nun aus. Nachdem er wiederhergestellt wurde, initiiert er den Recovery-Prozess. Dazu verlangt er zunächst von jedem anderen Prozess $P_j, j \neq i$ den Vektor $V_j$. $P_i$ berechnet daraus den Vektor $V_c^j = (V_0(j), ..., V_{n-1}(j))$ für jeden Prozess $P_j$ und verschickt diesen an $P_j$.

Angenommen $r_j$ sei der aktuellste Checkpoint des Prozesses $P_j$. Jeder Prozess $P_j$ prüft nun, ob folgende Formel erfüllt ist:

$$\forall m \in \{0, ..., n-1\} \setminus \{j\} : R_j(r_j, m) \leq V_c^j(m)$$

Die Differenz von $R_j(r_j, m) - V_c^j(m)$ bezeichnet dabei die Anzahl der Waisen-Nachrichten von Prozess $P_m$ an Prozess $P_j$. Ist diese Differenz für einen Prozess größer 0, so wählt der Prozess den ersten Checkpoint $r_c$ aus der gespeicherten Liste $R_j$ mit

$$\forall m \in \{0, ..., n-1\} \setminus \{j\} : R_j(r_c, m) \leq V_c^j(m)$$

Um alle Waisen-Nachrichten aus dem System zu entfernen, muss bis zu diesem Checkpoint zurückgesetzt werden. Also wird $r_c$ der aktive Checkpoint von $P_j$: Ein aktiver Checkpoint ist der Checkpoint eines Prozesses, der bei Beenden der Suche nach der Recovery Line wiederhergestellt wurde. $P_j$ schickt nun den Vektor $V_j$ dieses Checkpoints an $P_i$, zusammen mit einem $boolean$ Merker, der auf $true$ gesetzt wird. Er zeigt an, dass $P_j$ auf einen älteren Checkpoint zurückgesetzt werden musste und daher eine weitere Iteration nötig ist. Ist die Test-Bedingung erfüllt, schickt $P_j$ den Merker mit $false$ zurück um anzuzeigen, dass es momentan keine Waisen-Nachrichten gibt und $P_j$ daher nicht zurückgesetzt werden musste.

$P_i$ sammelt nun die Antworten der Prozesse ein. Sendet einer der Prozesse ein $true$ zurück, so beginnt eine neue Iteration und $P_i$ berechnet wiederum $V_c^j$ für die gerade aktiven Checkpoints. Kommen hingegen nur $false$-Antworten von allen Prozessen, so ist ein konsistenter Systemzustand erreicht und $P_i$ informiert die übrigen Prozesse, dass der normale Betrieb fortgesetzt werden kann.

*2) Beispiel:* Es wird die beispielhafte Kommunikation zwischen drei Prozessen aus Abbildung 1 betrachtet. Zu Beginn gilt

$$V_0 = V_1 = V_2 = (0, 0, 0)$$

und für die Listen $R_1$, $R_2$ und $R_3$ gilt

$$R_i =< r_{i,0} > \text{mit } r_{i,0} = (0, 0, 0)$$

Nun schickt $P_1$ die Nachricht $m_{1,0,1}$ an $P_0$. $P_1$ aktualisiert seinen Vektor $V_1$ auf

$$V_1 = (1, 0, 0)$$

und $P_0$ fügt seiner Liste $R_0$ ein neues Element $r_{0,1} = (0, 1, 0)$ hinzu, da es noch keinen Vektor für seinen nächsten Checkpoint $c_{0,1}$ gibt. Analog verfahren $P_2$ und $P_1$ mit der Nachricht $m_{2,1,1}$. Es gilt dann:

$$V_2 = (0, 1, 0) \text{ und } R_1 =< r_{1,0}, r_{1,1} > \text{ mit } r_{1,1} = (0, 0, 1)$$

Legt ein Prozess $P_i$ einen Checkpoint $c_{i,j}$ an, wird das entsprechende Element $r_{i,j}$ aus der Liste $R_i$ mit auf den stabilen Speicher geschrieben und bei der nächsten eintreffenden Nachricht wird ein neuer Vektor $r_{i,j+1}$ an die Liste angehängt (Wenn zuvor ein weiterer Checkpoint angelegt wird, dann gilt $r_{i,j+1} = r_{i,j}$ und es wird noch ein weiterer Vektor $r_{i,j+2}$ angehängt usw.). Die Situation kurz vor Ausfall von Prozess $P_0$ ist also die folgende:

$$V_0 = (0, 2, 1)\ V_1 = (3, 0, 2)\ V_2 = (0, 2, 0)$$
$$R_0 =< (0, 0, 0), (0, 1, 0), (0, 3, 0) >$$
$$R_1 =< (0, 0, 0), (0, 0, 1), (0, 2, 2) >$$
$$R_2 =< (0, 0, 0), (0, 2, 0), (0, 2, 0), (1, 2, 0) >$$

Nun fällt $P_0$ aus und wird bei $c_{0,2}$ wiederhergestellt. Zu diesem Zeitpunkt war $V_0 = (0, 1, 0)$ außerdem lässt er sich nun die aktuellen Vektoren $V_1$ und $V_2$ von den Prozessen $P_1$ und $P_2$ zuschicken und erstellt daraus die Matrix

$$V_c = \begin{pmatrix} 0 & 1 & 0 \\ 3 & 0 & 2 \\ 0 & 2 & 0 \end{pmatrix}$$

Weiterhin stellt er fest, dass

$$V_c^0 = (0, 3, 0) = r_{0,3}$$

gilt, er also für den Moment nicht weiter zurückgesetzt werden muss. Also schickt er $V_c^1 = (1, 0, 2)$ an $P_1$ und $V_c^2 = (0, 2, 0)$ an $P_2$.

$P_2$ und $P_3$ vergleichen nun diese Vektoren mit den aktuellsten Vektoren aus ihren Listen $R_i$. Beide stellen fest, dass sie Waisen-Nachrichten empfangen haben:

$$R_1(0, 2) = 2 > 1 = V_c^1(0)$$
$$R_2(0, 3) = 1 > 0 = V_c^2(0)$$

Sie rollen zum jeweils ersten Checkpoint zurück, an dem es keine Waisen-Nachrichten mehr gibt (in diesem Fall $c_{1,1}$ beziehungsweise $c_{2,2}$) und schicken die Vektoren $V_1 = (1, 0, 1)$ und $V_2 = (0, 2, 0)$ dieser Checkpoints an $P_0$, zusammen mit einem Merker, dass sie zurückgesetzt wurden.

$P_0$ empfängt nun diese Nachrichten und bestimmt anhand des Merkers, dass eine weitere Iteration nötig ist. Er stellt eine neue Matrix $V_c$ auf mit

$$V_c = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 2 & 0 \end{pmatrix}$$

und stellt anhand von

$$R_0(1, 2) = 3 > 1 = V_c^1(0)$$

fest, dass er ebenfalls zurückgesetzt werden muss, und zwar auf $c_{0,1}$. Nun ist $V_0 = (0, 0, 0)$ (in $c_{0,1}$ hat $P_1$ noch keine Nachrichten verschickt) und damit

$$V_c = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 1 \\ 0 & 2 & 0 \end{pmatrix}$$

Eine weitere Iteration beginnt, indem $V_c^1 = (0, 0, 2)$ an $P_1$ und $V_c^2 = (0, 1, 0)$ an $P_2$ geschickt werden.

Nach insgesamt vier Iterationen kommt der Algorithmus zum Halten. Alle Prozesse mussten auf ihren Initialzustand zurückgesetzt werden, ein Domino Effekt ist aufgetreten, doch das Verfahren hat die korrekte Recovery Line gefunden.

Für eine genauere Beschreibung dieses Verfahrens, den Korrektheitsbeweis und eine Laufzeit-Analyse, siehe [3], [4].

### *C. Optimistische log-basierte Protokolle*

Log-basierte Protokolle nutzen nicht nur die Checkpoints der einzelnen Prozesse, sondern auch Protokolle von empfangenen Nachrichten. Schlägt ein Prozess fehl, wird er zwar zu seinem letzten Checkpoint zurückgesetzt, kann aber alle Nachrichten, die auf stabilem Speicher gesichert wurden nutzen, um damit seine Ausführung bis zu einem gewissen Punkt getrieben von den Nachrichten noch einmal durchzuspielen und so einen aktuelleren Systemzustand wiederherzustellen, als es nur auf Basis von Checkpoints möglich wäre [5].

Meist nutzen diese Verfahren dazu nicht nur einen stabilen, sondern auch flüchtigen Speicher. Dieser steht zwar im Falle eines Ausfalls nicht mehr zur Verfügung, beschleunigt die fehlerfreie Ausführung aber deutlich, da zum einen schneller darauf zugegriffen werden kann, wenn ein anderer Prozess ausfällt und zum anderen das Ablegen der Nachrichten-Protokolle auf den stabilen Speicher nun asynchron erfolgen kann und im operativen Betrieb nicht auf langsame Speicherzugriffe gewartet werden muss.

Weiterhin werden den Nachrichten häufig noch Abhängigkeits-Informationen angehängt, um die Suche nach einer maximalen Recovery Line zu beschleunigen. Daher gehören die meisten optimistischen Verfahren ebenfalls zur Klasse der kommunikations-induzierten Protokolle.

Ein großes Problem der log-basierten Verfahren sind Nachrichten an die oder aus der Umwelt, die sich nicht wiederholen lassen [5]. Beispiele dafür wären etwa die Ausgabe von Geld an einem Geldautomaten oder eine Nutzereingabe zu einem bestimmten Zeitpunkt. Eine Lösung sieht vor, Eingaben aus der Umwelt direkt zu speichern, sodass sie sich im Falle eines Ausfalls wiederholen lassen. Mit Ausgaben an die Umwelt ist dies nicht so einfach möglich. Bevor eine solche Nachricht ausgegeben werden kann, muss zunächst sichergestellt werden, dass diese Nachricht im Falle eines Ausfalls nicht wiederholt werden muss. Nachrichten, die auch im Falle von Rollbacks nicht wiederholt werden müssen, heißen auch $committable$.

Im Folgenden wird nun ein Verfahren von Robert E. Strom and Shaula Yemini als vorgestellt.

*1) Verfahren von Strom und Yemini [5]:* Das Verfahren von Strom und Yemini beruht - wie etwa auch [6] - auf sogenannten Status-Intervallen und deren Abhängigkeiten. Die Zeit der Ausführung wird dabei anhand der eingegangenen Nachrichten in Intervalle $[I, M]$ unterteilt. $I$ steht dabei für die Inkarnation, das heißt, die Anzahl der Rollbacks eines Prozesses und $M$ ist ein Zähler für eingegangene Nachrichten. Die Inkarnation wird benötigt, um Intervalle auch im Falle von Rollbacks eindeutig unterscheiden zu können.

Um das Verfahren anwenden zu können, müssen die Prozesse über eine Reihe von Datenstrukturen verfügen [5]:

Der *Abhängigkeitsvektor* $DV_i$ eines Prozesses $P_i$ ist ein n-stelliger Vektor von Intervallen $[I_j, M_j]$, der die kausalen Abhängigkeiten des aktuellen Status-Intervalls zu den Zustandsintervallen aller anderen Prozesse speichert. Die $j$-te Komponente des Vektors - $DV_i(j)$ - gibt dabei das bekannte aktuellste Zustandsintervalls von $P_j$ an. $DV_i(i)$ gibt das eigene Zustandsintervall an und ist daher immer aktuell. Der aktuelle Abhängigkeitsvektor wird an jede Nachricht angehängt, um so transitive Abhängigkeiten aufzeichnen zu können.

Eine *Inkarnations-Tabelle* $IST_i$ speichert für jede Inkarnation eines jeden Prozesses die erste Nachricht dieser Inkarnation. Wenn beispielsweise $P_i$ mit Nachricht $[1, 5]$ eine neue Inkarnation beginnt, dann gilt: $IST_i(1, i) = 5$. Die Tabelle wird benötigt, um zurückgerollte Zustände und wiederholte Nachrichten erkennen zu können.

Ein *Logvektor* $LV_i$ speichert den aktuellen Zustand des Nachrichten-Logs aller Prozesse. Es wird benötigt, um festzustellen, ob eine Nachricht $committable$ ist und an Prozesse außerhalb der Systemgrenze ausgegeben werden kann. Der Logvektor ist wie der Abhängigkeitsvektor ein n-stelliger Vektor aus Intervallen $[I_j, M_j]$. Ein Intervall wird genau dann Teil eines Logvektors, wenn alle Nachrichten bis zu der Nachricht, die das Intervall startet, im stabilen Speicher abgelegt wurden. Falls für eine empfangene Nachricht $[I_j, M_j]$ mit Abhängigkeitsvektor $(d_1, ..., d_m)$ am Empfänger $P_i$ gilt:

$$d_k < LV_i(k)\ \forall k$$

, dann ist $[I_j, M_j]$ $committable$. Grundsätzlich kann der Logvektor einen Prozesses asynchron über einen Broadcast verteilt werden. Häufig wird er jedoch auch an alle Nachrichten angehängt, um die Zahl der Nachrichten zu reduzieren. In diesem Fall müssen jedoch auch solche Prozesse, die lange keine Nachricht mehr erhalten haben, regelmäßig einen aktuellen Logvektor zugeschickt bekommen.

Eine *Sitzungssequenznummer* $SSN_j$ für jeden Prozess, mit dem kommuniziert wird, ermöglicht es, eingehende Nachrichten beim Empfänger zu filtern. $SSN_j$ wird mit jeder gesendeten Nachricht an Prozess $P_j$ inkrementiert.

Wenn eine Nachricht $m$ mit Abhängigkeitsvektor $([\iota_1, \mu_1], ..., [\iota_n, \mu_n])$ von $P_i$ empfangen wird, dann ruft dieser Prozess den Algorithmus 1 auf. Dieser prüft zunächst, ob die Nachricht gültig ist: Sie muss in der richtigen Reihenfolge mit den bisherigen Nachrichten stehen und mindestens die jeweils aktuellse Inkarnationsnummern aufweisen. So können Duplikate aussortiert und die richtige Reihenfolge der Nachrichten sichergestellt werden, indem entweder das Nachrichtenlog durchsucht, oder auf die Übertragung des Nachrichtenkanals gewartet wird.

**Algorithm 1** EmpfangeNachricht $m$

**Require:** $erwarteteSSN \geq 0$
**Require:** $erwarteteInkarnation \geq 0$
  **if** $erwarteteSSN = SSN_m$ **then**
    $akzeptiere\ m$
  **else if** $erwarteteSSN > SSN_m$ **then**
    **if** $erwarteteInkarnation \geq I_m$ **then**
      [m ist Duplikat]
      $ignoriere\ m$
    **else** $\{erwarteteInkarnation < I_m\}$
      $erwarteteSSN \leftarrow SSN_m + 1$
      $erwarteteInkarnation \leftarrow I_m$
      $akzeptiere\ m$
    **end if**
  **else** $\{erwarteteSSN < SSN_m\}$
    **while** $erwarteteSSN < SSN_m$ **do**
      $loggedM \leftarrow Log.hole(erwarteteSSN)$
      **if** $loggedM == null$ **then**
        $Warte\ auf\ Nachricht$
      **else**
        [erwartete Nachricht befindet sich im Log]
        $Wiederhole(loggedM)$
        $erwarteteSSN \leftarrow erwarteteSSN + 1$
      **end if**
    **end while**
  **end if**

Hält $m$ diese Kriterien ein, kann es akzeptiert und verarbeitet werden. Im nächsten Schritt ruft der Prozess den Algorithmus 2 auf. Dieser prüft im ersten Schritt, ob es sich bei $m$ bekanntermaßen um eine Waisen-Nachricht handelt: Falls $m$ von ungültigen Nachrichten abhängt, wird es ignoriert, weil davon ausgegangen wird, dass der Sender ohnehin noch zurückgesetzt werden muss. Im zweiten Schritt wird geprüft, ob $m$ von Inkarnationen abhängt, die $P_i$ noch nicht bekannt sind. In diesem Fall muss gewartet werden, bis $P_i$ eine entsprechende Ankündigung erhält und mit Sicherheit prüfen kann, ob $m$ sicher weiterverarbeitet werden kann. Falls $m$ auch von diesem Algorithmus akzeptiert wird, wird der Abhängigkeitsvektor schließlich angepasst und $m$ kann an die übergeordnete Schicht beziehungsweise Anwendung weitergegeben werden.

**Algorithm 2** VerarbeiteNachricht $m$

**Require:** akzeptierte Nachricht $m$
  **if** $\exists k \in \{1, .., n\}, j \leq \iota_k : IST_i(j, k) > \mu_k$ **then**
    $ignoriere\ m$ [m ist Waisen-Nachricht]
  **else if** $\exists k \in \{1, ..., n\} : \forall j \leq \iota_k : IST_i(j, k) < \mu_k$ **then**
    [Noch keine Information über neue Inkarnation von $k$]
    $Warte\ auf\ k$
  **else**
    [akzeptiere $m$]
    $DV_i(k) \leftarrow \begin{cases} [I_i, M_i + 1] & \text{falls } i = k \\ max(DV_i(k), [\iota_k, \mu_k]) & \text{sonst} \end{cases}$
  **end if**

Empfängt ein Prozess $P_i$ einen Logvektor $LV_k$, so passt er seinen eigenen Logvektor $LV_i$ folgendermaßen an:

$$LV_i(j) = max(LV_k(j), LV_i(j))\ \forall j$$

Angenommen ein Prozess $P_i$ fällt nun aus und wird auf

einen Checkpoint $C_{i,j}$ zurückgesetzt. Er wiederholt dann seine geloggten Nachrichten bis entweder alle Nachrichten im Log wiederholt wurden, oder er auf eine Waisen-Nachricht trifft. Dann erhöht der Prozess seine Inkarnationsnummer und sendet eine Recovery-Nachricht mit dem aktuellen Zustandsintervall an alle Prozesse. Diese erklärt alle späteren Zustandsintervalle für verloren, sodass andere Prozesse gegebenenfalls zurückgesetzt werden müssen.

Wenn ein Prozess $P_i$ eine Recovery-Nachricht mit Inkarnation $\iota$ von Prozess $P_j$ empfängt, dann prüft er den Abhängigkeitsvektor seines aktuellen Zustandsintervalls: Falls

$$DV_i(j) = [I_j, M_j] \ \wedge \ IST(\iota, j) \leq M_j$$

erfüllt ist, muss auch $P_i$ bis zu einem Checkpoint zurückrollen, der diese Abhängigkeit nicht hat. Er führt die gleichen Schritte aus, als wäre er ebenfalls ausgefallen: Er spielt den aktuellstmöglichen Checkpoint auf und wiederholt alle Nachrichten, bis er auf eine Waisen-Nachricht trifft und sendet eine Recovery-Nachricht an alle anderen Prozesse [5].

Iterativ wird so unter Ausnutzung von Checkpoints und Nachrichtenlogs ein neuer konsistenter Systemzustand gefunden.

*2) Beispiel:* Grundlage dieses Beispiels sei wieder die Kommunikation zwischen drei Prozessen aus Abbildung 1. Zu Beginn gilt

$$DV_0 = DV_1 = DV_2 = ([0,0],[0,0],[0,0])$$
$$\forall i,j \in 0,1,2 : IST_i(0,j) = 0$$

In diesem Beispiel spielt es keine Rolle, ob eine Nachricht *committable* ist oder nicht, da keine Interaktion mit der Außenwelt stattfindet. Auch die Sitzungssequenznummer wird hier nicht betrachtet, da wir von Nachrichten Kanälen ausgehen, die einen *Atomic Broadcast* garantieren, also sicherstellen, dass alle Nachrichten in der richtigen Reihenfolge eintreffen. Außerdem werden Statusintervalle nicht nur mit Empfang einer Nachricht gestartet, sondern auch beim Anlegen eines Checkpoints. Dies erlaubt eine feinere Gliederung der Rechenzeit in Intervalle und kann so helfen, Abhängigkeiten von langen Intervallen abzumildern.

Empfängt nun Prozess $P_0$ die Nachricht $m_{1,0,1}$, so setzt er sein Statusintervall $DV_0(0) = [0,1]$. Sein Abhängigkeitsvektor ändert sich nicht, da dort bereits das aktuelle Statusintervall von $P_1$ vermerkt ist. Nun empfängt $P_1$ Nachricht $m_{2,1,1}$ und setzt entsprechend sein Statusintervall $DV_1(1) = [1,1]$. Verschickt $P_1$ dann die Nachricht $m_{1,2,1}$, so hängt er seinen aktuellen Abhängigkeitsvektor $DV_1 = ([0,0],[0,1],[0,0])$ an. $P_2$ wird darauf beim Empfang der Nachricht sein eigenes Statusintervall $DV_2(2) = [1,1]$ setzen und seinen Abhängigkeitsvektor ebenfalls anpassen: $DV_2 = ([1,0],[1,1],[1,1])$.

Kurz vor Ausfall von Prozess $P_0$ gilt dann:

$$DV_0 = ([0,5],[0,3],[0,0])$$
$$DV_1 = ([0,5],[0,6],[0,3])$$
$$DV_2 = ([0,5],[0,3],[0,5])$$
$$\forall i,j \in 0,1,2 : IST_i(0,j) = 0$$

Wenn $P_0$ nun ausfällt, dann wird er auf $c_{0,2}$ mit Statusintervall $[0,5]$ zurückgesetzt. Der Prozess hat keine Nachrichten im Log, die er wiederholen könnte und beginnt daher eine neue Inkarnation bei seinem aktuellen Statusintervall. Er schickt eine Recovery-Nachricht $[1,5]$ an die Prozesse $P_1$ und $P_2$. $P_2$ erweitert seine Inkarnationstabelle um $IST_2(1,0) = 5$ und stellt fest, dass sein Abhängigkeitsvektor $DV_1$ vom Statusintervall $[0,5]$ von $P_0$ abhängig war. Dieses ist nun nicht mehr gültig. Daher wird er auf $c_{1,1}$ zurückgesetzt und wiederholt nun seine Nachrichten ab $m_{0,1,1}$. Diese werden von $P_0$ ignoriert, doch $P_1$ kann seine Berechnungen bis zur gestrichelten Linie in Abbildung 1 wiederholen. Dort trifft er auf Nachricht $m_{0,1,2}$ mit Abhängigkeitsvektor $([0,5],[0,3],[0,0])$. Hierbei handelt es sich wegen $[0,5]$ um eine Waisen-Nachricht. Damit kann $P_1$ keine Berechnungen wiederholen und er sendet eine Recovery-Nachricht $[1,4]$ an $P_0$ und $P_2$ (die jeweils nur ihre Inkarnationstabelle aktualisiert, da sie beide nur von $[0,3]$ abhängen).

Gleiches gilt für $P_2$ beim Empfang der Recovery-Nachricht von $P_0$. Er aktualisiert seine Inkarnationstabelle und stellt fest, dass er auf $c_{2,2}$ mit $DV_2 = ([0,0],[0,2],[0,4])$ zurücksetzen muss. Der Prozess wiederholt seine Berechnungen bis $m_0, 2, 1$, erkennt diese als Waisennachricht und sendet eine Recovery-Nachricht mit dem Statusintervall $[1,4]$ an $P_0$ und $P_1$. Beide aktualisieren wieder ihre Inkanrationstabelle $IST_0(1,2) = 4$ und ignorieren die Nachricht ansonsten. Damit ist der Algorithmus abgeschlossen und es gilt:

$$DV_0 = ([1,5],[0,3],[0,0])$$
$$DV_1 = ([0,2],[1,4],[0,3])$$
$$DV_2 = ([0,0],[0,2],[1,4])$$

Die Recovery Line liegt nun insgesamt bei der gestrichelten Linie in Abbildung 1 und ist damit deutlich aktueller als die erzeugte Recovery Line aus dem letzten Verfahren (siehe Abschnitt III-B1). Allerdings benötigen die Nachrichtenlogs der Prozesse auch deutlich mehr Speicherplatz.

In [5] wird ein Korrektheitsbeweis dieses Verfahrens erbracht und es wird auf weiterführende Themen wie ein effizienteres Speichermanagement eingegangen.

Ein wesentliches Ergebnis dieser Beweises sei hier noch angemerkt: Unter der Annahme, dass 1) kein Prozess für unendlich lange Verzögerungen sorgt, 2) irgendwann einmal seinen Logvektor verschickt und 3) irgendwann einen Checkpoint anlegt, tritt mit diesem Verfahren der Domino Effekt nicht auf [5].

### *D. Evaluation*

Offensichtlich haben beide Verfahren Vor- und Nachteile. Während beim checkpoint-basierten Vorgehen der Verwaltungsaufwand minimiert und die fehlerfreie Ausführung nicht behindert wird, verhindern Verfahren mit Message-Logging erfolgreich den Domino Effekt und finden in der Regel eine weiter vorangeschrittene Recovery Line. Sie benötigen dazu allerdings auch deutlich mehr Speicherplatz und Rechenzeit für die einzelnen Log-Einträge. Schließlich hängt von der Häufigkeit und Geschwindigkeit der Inter-Prozesskommunikation ab, ob das Wiederholen der Nachrichten in einem log-basierten

Verfahren wirklich einen deutlichen Geschwindigkeitsvorteil gegenüber der erneuten Ausführung bei einem rein checkpoint-basierten Verfahren bietet.

Eine Bewertung beziehungsweise Entscheidung für die eine oder andere, oder gar eine hybride Methode, hängt daher von den Parametern des betreffenden Systems (beispielsweise Zuverlässigkeit, Geschwindigkeit und Ausprägung der Kommunikation, ...) ab. *Ausprägung der Kommunikation* meint hier beispielsweise die Häufigkeit von Nachrichten im Vergleich zur reinen Rechenzeit, die Zahl der Prozesse, die aktiv an der Kommunikation beteiligt ist und eventuelle Topologien in der Prozessstruktur (Baum-, Ring-, vollstände Netzstruktur usw.).

## IV. Zusammenfassung

In diesem Artikel wurde beschrieben, was in verteilten Systemen als konsistenter Zustand betrachtet werden kann und warum es im Gegensatz zu Ein-Prozess-Systemen deutlich aufwändiger ist, diesen nach einem Fehler wiederherzustellen. Mit Checkpointing und Rollback wurde anschließend eine Gruppe von Verfahren vorgestellt, die dieses Problem lösen. Die Gruppe wurde in drei Klassen aufgeteilt: (1) Synchrone Verfahren, (2) Kommunikations-Induzierte Verfahren und (3) asynchrone Verfahren, die sich wiederum in checkpoint- und log-basierte Verfahren aufteilen lassen. Anschließend wurden jeweils ein Vertreter der checkpoint- und log-basierten asynchronen Verfahren genauer betrachtet, um die grundsätzliche Funktionsweise dieser Algorithmen zu verstehen. Zum einen einen Algorithmus nach Gupta (und Korrektur von Kapus), der die Anzahl der versendeten Nachrichten von Prozess zu Prozess zählt und so eine Recovery Line bildet und zum anderen ein Verfahren von Strom und Yemini, das über Nachrichten-Logs und Abhängigkeitsvektoren zu einem Ergebnis kommt.

Ein Problem, das in dieser Arbeit nicht betrachtet wurde, betrifft den Speicher für Checkpoints und Logs. Offensichtlich ist dieser begrenzt, sodass nicht benötigte Checkpoints und Logs schließlich gelöscht werden müssen. Beide Verfahren verfügen jedoch über Möglichkeiten, die Einträge (Checkpoints) zu erkennen, die niemals Teil der Recovery Line werden können und somit nicht mehr benötigt werden [3], [5].

Eine Möglichkeit, den Daten-Overhead aus dem Verfahren von Strom und Yemini, der linear mit der Zahl der Prozesse wächst, zu verkleinern, zeigt ein ähnliches Verfahren von David B. Johnson Willy Zwaenepoel [6]. Es hängt nur jeweils das aktuelle Zustandsintervall des Senders an Nachrichten an, muss im Gegenzug aber einen zentralisierten Algorithmus für seine Recovery nutzen.

## Literatur


[1] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput. Surv.*, vol. 34, no. 3, pp. 375–408, Sep. 2002.

[2] P. Jalote, *Fault Tolerance in Distributed Systems*. Prentice Hall, 1994.

[3] B. Gupta, S. Rahimi, and Y. Yang, "A novel roll-back mechanism for performance enhancement of asynchronous checkpointing and recovery," 2007.

[4] M. Kapus-Kolar, "Improvements to a roll-back mechanism for asynchronous checkpointing and recovery." *Informatica (Slovenia)*, vol. 33, no. 4, pp. 511–519, 2009.

[5] R. E. Strom and S. Yemini, "Optimistic recovery in distributed systems," *ACM Transactions on Computer Systems*, vol. 3, pp. 204–226, 1985.

[6] D. B. Johnson and W. Zwaenepoel, "Recovery in distributed systems using optimistic message logging and check-pointing," *J. Algorithms*, vol. 11, no. 3, pp. 462–491, Sep. 1990.

# Coordinated Checkpointing and Rollbacks in Distributed Systems

Uli Schlachter
Department of Computer Science
University of Oldenburg
Oldenburg, Germany
uli.schlachter@informatik.uni-oldenburg.de

***Abstract*—A distributed system consists of individual processes working towards a common goal. In such systems a process runs on a – often physical – node which is subject to failures. If a process fails and corrupts its local state, a consistent system state must be restored. One possible way to do this is to create checkpoints of the local state of each individual process in advance and restore the latest valid checkpoint after a process has failed. Such an operation is called a rollback. This paper presents an algorithm for recording checkpoints so that a consistent system state can be reached by rolling back all processes to their most recent checkpoint. Also, the algorithm creates checkpoints only for the processes where this is necessary. No process needs to store more than two checkpoints at any given time.**

## I. Introduction

When a distributed system consists of multiple components, the probability for every component to be available is low. For example, a system made of 100 components, each of which is available with a probability of $0.99$ at any given point in time, has an overall probability of $0.99^{100} \approx 0.366$ of having all components available. Thus, this system becomes unreliable due to its many subsystems.

This naïve calculation assumes that, as soon as a component fails, the system as a whole is compromised and needs to be restarted. With such an approach, all computations up to this point are lost. This means that long-term calculations are very unlikely to complete successfully.

If it is assumed that failures are transient, i.e. they will eventually be fixed again, one possible approach to handle this problem is through *checkpointing and rollbacks*. Each component regularly creates a checkpoint of its current state on a *stable storage* which is assumed not to fail. After a transient failure is repaired, all affected components can be brought back into consistent states by *roll-back* to their last checkpoint which restores the local state to the time when the checkpoint was created.

Algorithms for this kind of *backward error-recovery* can be put into two different categories. In *uncoordinated* checkpointing and rollbacks, processes locally decide to create new checkpoints. During a rollback the algorithm makes sure that a consistent system state is reached, which is not always possible.

Alternatively, in *coordinated* checkpointing and rollbacks it is made sure that all checkpoints together describe a consistent system state. This means that only the most recent process state need be saved while older checkpoints can be deleted. This also avoids the so-called *domino effect*.

The following section describes the assumptions made about the underlying distributed system. In section III, problems with uncoordinated checkpointing are described in-depth to motivate section IV which presents an algorithm for coordinated checkpointing and rollback operations introduced in [1]. This section starts by presenting the most basic approach which is then refined so that as few processes as possible save their local states. Afterwards, the rollback algorithm is presented. This algorithm only rolls back processes when otherwise the system is in an inconsistent state.

## II. System Model

In this paper, a *distributed system* consists of *nodes* which are connected through *(communication) channels*. Each node executes a single *process* which can send and receive arbitrary *messages* on every channel that its node is connected to. This means that channels are bi-directional with one process at each end. When a message is sent on a channel, the node at the other end of the channel will receive this message within finite time and deliver it to the local process. Also, channels deliver messages in the same order in which they were sent (FIFO). There are no requirements for the order of messages on different channels. These messages are the only form of inter-process communication available.

Every process has a local *stable storage* for saving checkpoints. A *checkpoint* fully describes the state of the process at the time that the checkpoint was created. A process can decide to *roll-back* to any of its checkpoints. This rollback operation is a saved process state being restored. Checkpoints can also be deleted which means it can no longer be used for roll-back. These operations are done without affecting other processes.

The processes can be modeled as the vertices on a graph and the channels as its edges. This graph is strongly connected, which means that every process can be reached from every other process via intermediate processes that forward messages. If this assumption is not satisfied, the set of processes become partitioned and different partitions cannot influence others. Thus, the system effectively is divided into multiple separate distributed systems which can be examined on their own.

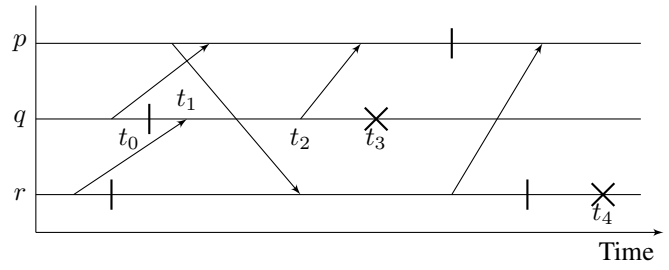



Fig. 1. An example of the behavior of three processes $p$, $q$ and $r$. Arrows represent the transit of a message (in time) through a channel. A vertical bar marks the creation of a checkpoint and an 'X' indicates the failure of a process. In the example shown, $r$ starts by sending a message to $q$ and then records a checkpoint. Before $q$ receives this message, it sends a message to $p$ and also creates a checkpoint of its current state.

By performing a rollback, a process can be restored to an earlier execution state. If this process sends a message to another process between creating a checkpoint and failing, the process state cannot contain a record about sending this message. This message is then called an *orphan message*. Likewise, if a process receives a message before rolling-back its local state, there cannot be a record about receiving this message. This is called a *lost message*. The system state is understood to be *consistent* if no messages are orphans. The next section contains an example for these definitions.

Finally, the system is assumed to function perfectly. The exception are processes which, when they fail, stop completely. This means that the process will no longer send any messages nor is its stable storage compromised. Also, process failures can be detected by other processes within finite time. A process failure is transient, which means that eventually a failing process will be repaired and it can continue execution.

These assumptions are optimistic, but there are approaches and algorithms for approximating such a system [2]. For example, channel failures can be detected when receiving processes fail to respond with an acknowledgement. If a sender does not get an acknowledgement within a given time, it then knows that the channel failed and it may attempt to retransmit the message.

## III. Uncoordinated Checkpointing

To better understand the problem to be solved, let us assume a simple checkpointing algorithm where each process creates checkpoints at random times. After a transient failure is repaired, a failing process will rollback to its latest checkpoint.

Fig. 1 shows a sample execution of three processes which exchange messages, create checkpoints and are subject to failures. Process $r$ fails at time $t_4$ and must be repaired. Afterwards, it must restart execution from one of its checkpoints. Since processes only communicate by sending messages on channels and $r$ did not send any messages since its latest checkpoint, this process can roll back to its last checkpoint and continue execution without giving rise to an inconsistent system state. Any calculations that are lost did not influence other processes and thus can be recomputed without causing any side effects.

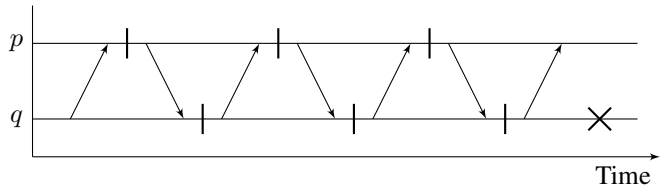



Fig. 2. An example scenario eliciting the domino effect. After process $q$ fails, process $p$ has to roll-back to avoid an orphan message. This introduces another orphan message and $q$ has to roll back to an earlier checkpoint. This repeats until the initial system state is reached.

The failure that *does* cause problems occurs at time $t_3$. Process $q$ fails and must revert to its latest checkpoint which was made at time $t_0$. This rollback does not restore a consistent system state, because previously $q$ received a message from $r$ at time $t_1$ and at time $t_2$ it sent a message to $p$. The first message is lost and the second message becomes an orphan, as explained in section II. This means the resulting system state is inconsistent.

This example illustrates that after a process has been rolled-back, more actions might be needed to restore a consistent system state. This shows that other processes must be rolled-back as well. For example, in the situation above, after $q$ has been restored, $p$ and $r$ would have to roll-back to avoid the orphan and lost messages. These operations could in turn make further rollbacks necessary.

Since processes might need to roll back multiple times, this can go on until the initial system state is reached and all computations have been lost. This is referred to as the *domino effect*. This is illustrated in Fig. 2. Every combination of the available checkpoints would result in an orphan message which makes the initial system state the only viable 'checkpoint' to return to.

Algorithms which do not require any communication with other processes when creating checkpoints implement so-called *uncoordinated checkpointing*. These algorithms only calculate a consistent system state when a process needs to restore itself. Knowing the problems that arise from these algorithms helps in understanding *coordinated checkpointing*, which will be explained next.

## IV. Coordinated Checkpointing

In coordinated checkpointing, processes communicate with each other while creating checkpoints in order to make sure the latest set of checkpoints is always consistent.

The following algorithm allows a process to store two checkpoints at most at any given time. Every process always has a *permanent checkpoint* and the permanent checkpoints of all processes together are guaranteed to describe a consistent system state. This means that there will be no resulting orphaned messages if all processes roll-back to their permanent checkpoints. Lost messages are negligible and would be the focus for another protocol which deals with unreliable communication channels, as mentioned in section II. Permanent checkpoints can only be deleted after a newer checkpoint becomes permanent.

The second kind of checkpoints are *tentative checkpoints*. Permanent checkpoints are created during a 2-phase commit protocol. In the first phase each process creates a tentative checkpoint of its current state. If this finishes without issue, the next phase commits these checkpoints and makes them permanent. Otherwise, the tentative checkpoints are deleted.

In its simplest variant, a coordinated checkpointing algorithm works by making sure all processes create a consistent checkpoint when any process goes to make a checkpoint. Thus, when process $q$ creates a (tentative) checkpoint, it broadcasts a message to all other processes to do the same. Once every process confirms that a checkpoint was created successfully, these tentative checkpoints are made permanent by soliciting another message to processes involved. To make sure that no orphan messages can occur, a process is not allowed to send any messages while waiting for a tentative checkpoint to become permanent. Thus, all messages received during the run of the algorithm are part of a checkpoint of their sender. Lost messages are irrelevant, as explained above.

However, this simple approach requires all processes to create checkpoints together which interrupts normal execution. In other words, this approach has a large overhead and can be improved upon.

### A. Creating fewer Checkpoints

When a process $q$, called the *initiator*, wants to save a checkpoint it has to make sure that the resulting checkpoints describe a consistent system state. This means that there may not be any orphan messages. Thus, any process that received a message from $q$ also needs a current checkpoint. This is done by sending checkpoint request messages to these other processes.

To implement the above, each process tracks its communications with other processes and assigns a strongly monotonically increasing identifier $m.l$ to each message $m$ that is sent and not generated by the checkpointing algorithm. These identifiers are used to define $\mathrm{last\_rmsg}_q(p)$. This is the identifier of the *last* message that process $q$ *received* from $p$ after $q$ recorded its last checkpoint (tentative or permanent). Analogously, $\mathrm{first\_smsg}_q(p)$ is the identifier of *first* message that $q$ *sent* to $p$ after $q$ recorded its latest checkpoint (tentative or permanent).

Both of these functions assume a special value $\perp$ if no suitable message exists. The value $\perp$ is defined to be less than any valid message identifier.

Furthermore, both functions can only be calculated by $q$, because other processes cannot reliably know which messages $q$ already sent or received. Thus, if another process $p$ needs to know such a value, $q$ has to send this value to $p$.

As was mentioned, it might be necessary that a process $q$ tells process $p$ to create a checkpoint. The algorithm must avoid orphan messages, so all messages that $q$ has received when it creates a checkpoint must also be included in the checkpoint of the sending process. The set $\mathrm{chkpt\_cohort}_q$ is the set of other processes that need to record a checkpoint. It contains all processes from which $q$ received a message since its last checkpoint:

$$\mathrm{chkpt\_cohort}_q := \{p \mid \mathrm{last\_rmsg}_q(p) > \perp\}$$

A checkpoint request message together with the information $\mathrm{last\_rmsg}_q(p)$ needs to be sent to each process $p \in \mathrm{chkpt\_cohort}_q$. This request means that $p$ should establish a checkpoint which records all messages up to $\mathrm{last\_rmsg}_q(p)$, if no such checkpoint already exists.

When $p$ receives such a checkpoint request, it will then determine if it needs to create a checkpoint. If $\mathrm{last\_rmsg}_q(p) \geq \mathrm{first\_smsg}_p(q) > \perp$, $p$ is said to *inherit* a checkpoint request from $q$. This condition checks to see if the latest message that $q$ received from $p$ was sent after $p$'s latest checkpoint and thus could become an orphan.

If $p$ inherits a checkpoint request, it creates a tentative checkpoint and sends a checkpoint request to any process in $\mathrm{chkpt\_cohort}_p$, in the same way as $q$ does. This could result in some processes receiving multiple checkpoint requests. Section IV-B will explain why this does not cause problems.

All processes which participate in the checkpointing protocol must not send any messages not specifically created by the protocol until the protocol finishes.

When all processes reply to the checkpoint request or if $p$ does not inherit the checkpoint request, a reply is sent to $q$ which acknowledges the checkpoint request and informs $q$ if the tentative checkpoint was created successfully. Thus, eventually the initiator of the protocol receives such a reply from all processes that it sent a checkpoint request to.

Once this completes, the second part of the 2-phase commit protocol starts. In this step, every process is informed of the success (or failure) of the first phase. A process sends this information to every process that it sent a checkpoint request to in the first phase. If the first phase finished successfully, all tentative checkpoints are made permanent and the previous permanent checkpoints are forgotten. Otherwise, the tentative checkpoints are deleted. Normal execution resumes from this point.

An example for this algorithm can be found in Fig. 3, where after some simple communication, the checkpointing protocol is invoked by process $q$. Because this process received messages from $p$ and $r$, it must notify them with a checkpoint request including $\mathrm{last\_rmsg}_q(p)$ and $\mathrm{last\_rmsg}_q(r)$, respectively. Both processes inherit the checkpoint request and $r$ has to send another request to $q$. Afterwards, all processes reply with an acknowledgement. Once $q$ received all acknowledgements, the second phase of the algorithm starts and the tentative checkpoints are made permanent. Then, normal execution can resume.

In the next invocation of the algorithm, $p$ can record a checkpoint without asking other processes, because it did not receive any messages since its last save.

Next $q$ wants to create a checkpoint and so has to send a checkpoint request to $p$. However, $p$ does not inherit this request, because it already established a checkpoint after its latest message to $p$. Thus, it sends an acknowledgement to $q$ without creating a new checkpoint.

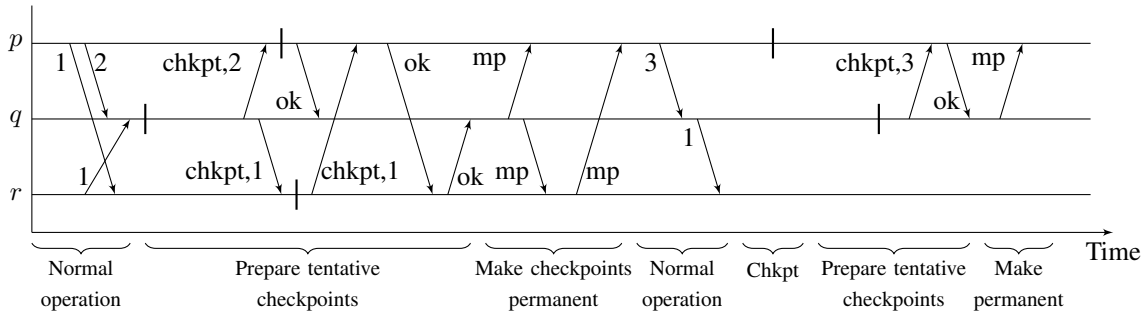



Fig. 3. Example execution portraying a coordinated checkpointing algorithm. The labels next to the arrows identify the message. Different phases of executions are shown. Initially processes communicate normally. Then the 2-phase commit protocol for establishing new permanent checkpoints is initiated by process $q$. After the protocol finishes, normal communication continues. Then $p$ invokes the checkpointing algorithm without needing any communication for this. In the last two phases, $q$ establishes a new checkpoint.

### *B. All Processes Create at most one Tentative Checkpoint*

In the above protocol, a process could receive checkpoint requests from two different processes. The first one is handled normally. Do further checkpoint requests cause problems?

No, because a process $p$ may not send any messages until the procedure finishes. Because $p$ just recorded a tentative checkpoint, it follows that $\mathrm{first\_smsg}_p(q) = \perp$ for any other process $q$. This means the condition for inheriting another checkpoint request is always false after the first checkpoint request is inherited; $p$ replies that it established the requested checkpoint without creating another checkpoint.

Thus, during execution of the checkpointing algorithm, a process is limited to creating a single checkpoint. This prevents checkpoint requests from triggering indefinitely.

### *C. Roll-Back*

When a process fails and its execution state is restored, it has to make sure that the resulting global state is consistent. A process' only option is to restart computation from its last permanent checkpoint. However, between the time that this checkpoint was recorded and the time a process failed, it could have communicated. This potential for orphaned messages could result in an inconsistent global state.

The simplest way to handle this problem is to make all processes roll-back to their latest checkpoint after any process is restored. However, this would again involve more processes than needed, because processes which did not participate in any communication have no reason to perform a rollback.

The following algorithm only rolls back processes if an orphan message is detected, as lost messages are explicitly allowed. This means a restored process has to make sure it has not send messages to other processes since its checkpoint. Otherwise, a process receiving such a message must roll back to its last checkpoint as well, and repeat this procedure[1].

To do this, the initiator $q$ sends a rollback request to all processes $p$ that it may have communicated with[2]. This request includes $\mathrm{last\_smsg}_q(p)$ which is the identifier of the last message that $q$ sent to $p$ before constructing its checkpoint. A rollback request is inherited by $p$ if $\mathrm{last\_rmsg}_p(q) > \mathrm{last\_smsg}_q(p)$. This means that $p$ inherits the request if it received a message from $q$, but $q$ rolled-back to a point before sending this message. This is the condition for orphan messages. When $p$ inherits a rollback request, it has to roll-back and forward this request to other processes in the same manner as $q$.

When each of these processes has replied to the request or if $p$ did not inherit the request, $p$ can respond to $q$ saying that the request was successful. When the initiator receives all the responses from the related processes, the second phase begins and every process is informed that the rollback is completed.

Similar to the checkpointing protocol, processes may not communicate after inheriting a rollback request until this protocol completes. This is to make sure that no messages from before the rollback are received after the protocol finishes, because rollback requests are sent on all channels and because channels deliver messages in the same order that they are sent (FIFO).

An example for this algorithm is shown in Fig. 4. In this scenario, all processes begin with permanent checkpoints.

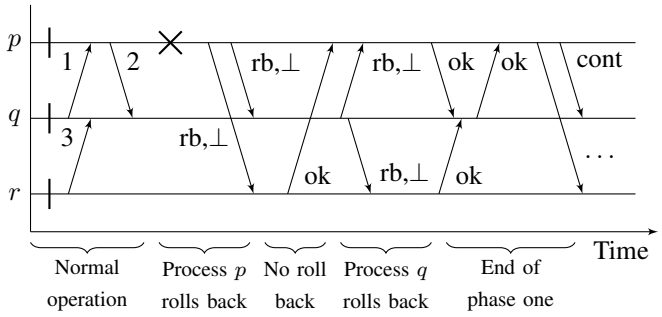



Fig. 4. Example execution of the roll-back algorithm. All processes begin with permanent checkpoints. Eventually, $p$ fails and initiates the algorithm. Please note that not all messages from the algorithm are shown. The second phase continues by sending a message on each channel which previously carried a rollback request. Also note that messages during normal operation here have globally unique identifiers which would not be possible in a real execution.

[1] This would need to be done again with a 2-phase commit protocol to make sure that no messages from before the rollback interfere with the new system state.

[2] If it is assumed that all processes can communicate with each other directly, $q$ must broadcast a rollback request to all other processes.

| | $p$ | $q$ | $r$ | | $p$ | $q$ | $r$ |
|---|---|---|---|---|---|---|---|
| $\text{last\_smsg}_p$ | | $\perp$ | $\perp$ | $\text{last\_rmsg}_p$ | | $\perp$ | $\perp$ |
| $\text{last\_smsg}_q$ | 1 | | $\perp$ | $\text{last\_rmsg}_q$ | 2 | | 3 |
| $\text{last\_smsg}_r$ | $\perp$ | 3 | | $\text{last\_rmsg}_r$ | $\perp$ | $\perp$ | |

TABLE I
VALUES WHEN $p$ INITIATES THE ROLLBACK IN FIG. 4. THE LEFT HALF SHOWS E.G. $\text{last\_smsg}_q(p) = 1$ WHICH MEANS THAT THE LAST MESSAGE THAT $q$ SENT TO $p$ HAD IDENTIFIER 1. LIKEWISE, THE RIGHT SIDE SHOWS THE IDENTIFIER OF THE LAST MESSAGE THAT A PROCESS RECEIVED. UNDEFINED VALUES ARE LEFT BLANK.

Eventually, $p$ fails and initiates the algorithm. Table I shows the values of the $\text{last\_smsg}$ and $\text{last\_rmsg}$ array in this situation.

When $p$ broadcasts its rollback request, $q$ has to rollback, because $\perp = \text{last\_smsg}_p(q) < \text{last\_rmsg}_q(p) = 2$, while $r$ continues its execution without inheriting the request, because no orphan messages occur.

After the rollback algorithm finishes, both processes which were involved with messages 1 and 2 rolled back to before sending or receiving this message. Message 3 became lost.

### D. Handling Process Failures while running the Algorithms

The presented algorithm is designed to protect against process failures. However, processes may also fail during execution of one of these algorithms.

Our system model assumes that processes can detect when another process fails. Thus, when a process fails after a checkpoint request was sent to it, it must be assumed that the creation of the requested checkpoint failed. Both algorithms already allow processes to signal failures to the initiator, in this scenario the algorithm is aborted and the tentative checkpoints are removed. This method can be used in the event of failure.

When a process fails after recording a tentative checkpoint, it is possible that the initiator later promoted this checkpoint to permanent status. A process which failed during the run of the algorithm must then ask the other processes about the result of the checkpoint algorithm and decide if the tentative checkpoint should be discarded or made permanent. This can be done by recording the process from which the checkpoint request was inherited within the checkpoint.

Another possible failure scenario is that the initiator of the protocol crashes. This stalls the whole system until the initiator is restored. [1] suggests replacing the 2-phase commit protocol with an nonblocking 3-phase commit protocol, so that the rest of the system can continue while the initiator is repaired.

### E. Multiple processes initiate the algorithm

Another possible problem-case is that two processes start the algorithm simultaneously. This can be mitigated by including the identity of the initiator in all messages the algorithm generates.

When a process that already executes one of these algorithms receives a checkpointing- or rollback-related messages for a different initiator, this newer execution of the algorithm is aborted by sending a negative reply. This may make all concurrent executions fail, but still guarantees a consistent system state.

## V. CONCLUSION AND OUTLOOK

This paper introduced uncoordinated checkpointing as a motivation for coordinated checkpointing. The core issue with uncoordinated checkpointing is that it cannot guarantee a successful rollback after a failure, because it may elicit a domino effect. In coordinated checkpointing, this is avoided by only recording checkpoints which describe a consistent system state.

Another difference is that in coordinated checkpointing, stable storage for two checkpoints is needed while no such upper limit is defined for uncoordinated checkpointing.

The algorithm for coordinated checkpoints and rollbacks was explained in detail with reasons presented in favor of its correctness. This began with the most straight-forward implementation where a process creating a checkpoint asks all other processes to do the same. This approach was then refined so that as few checkpoints as possible are recorded for efficiency. The same was then done for the associated rollback algorithm.

There are more interesting points relating to this subject. For example, the number of messages transmitted can be analyzed. It can easily be seen that creating checkpoints becomes more expensive if many channels are used during normal operation.

Obviously, uncoordinated checkpointing performs better in this case, because checkpoints can be created without message-passing. Thus, when implementing backward error-recovery, factors like the required reliability and the possible overhead become important. It can be seen that these are conflicting aims with both variants of checkpointing and rollbacks having their uses.

The algorithms themselves can be further improved. One example of this is replacing the 2-phase commit protocol with an asynchronous 3-phase procedure, so that the system does not hang when the initiator fails.

A suggestion from [1] handles the case where a process fails and related processes cannot create new checkpoints, because they would have to send a checkpoint request to the failed process. The suggested approach is to roll-back to the last checkpoint. Afterwards, new checkpoints can be constructed because a failed process cannot send new messages. Thus, orphaned messages would be an impossibility and could not prevent this.

Finally, [3] explains a combination of the two approaches called *communication-induced checkpointing* where information about uncoordinated checkpoints is distributed during normal operation to preemptively detect the domino effect. Creation of new checkpoints is forced if needed.

## APPENDIX

This section presents the algorithm from this paper as pseudo-code. Normal messages have to be sent via the send() function, while the algorithm itself bypasses this function. This function updates the last_smsg and first_smsg arrays. Likewise, the code for updating first_rmsg only handles normal system messages.

```
send(msg, q):
  last_smsg[q] += 1;
  if first_smsg[q] = ⊥ then
    first_smsg[q] = last_smsg[q]
  fi
  msg.l = last_smsg[q];
  send(msg) to q;

On receipt of msg from process q:
  last_rmsg[q] = msg.l;
```

When a process decides to create a new permanent checkpoint, it is assumed that it will send a checkpoint request to itself. This means that it is its own initiator and it will send a message to itself at the end of phase one to start the second phase. This simplifies the description of the algorithm, because otherwise the initiator needs to be handled specially.

```
On receipt of chkpt_req(initiator, last_rmsg) from q:
  if own_initiator != null and initiator != own_initiator then
    -- We are already participating in another execution
    -- of the checkpointing/rollback algorithm.
    send(false) to q;
    return;
  fi
  own_initiator = initiator;
  if not (last_rmsg >= first_smsg[q] > ⊥) then
    -- Checkpoint request not inherited,
    -- wait until protocol finishes
    send(true) to q;
    await(result(success)) from q;
    own_initiator = null;
    return;
  end
  -- Checkpoint request inherited
  success = true;
  for r in chkpt_cohort:
    send(chkpt_req(initiator, last_rmsg[r])) to r;
    last_smsg[r] = first_smsg[r] = ⊥;
  end
  create_tentative_checkpoint;
  -- Note: If failure of some process from chkpt_cohort
  -- is detected, handle as negative reply
  await all replies:
    if checkpoint request failed:
      success = false;
  fi
  send(success) to q;
  -- Wait until first phase is finished and
  -- DO NOT continue computation until then
  await(result(success)) from q;
  send(result(success)) to each r in chkpt_cohort
  if success then
    make_checkpoint_permanent;
  else
    delete_tentative_checkpoint;
  fi
  own_initiator = null;
```

The above trick is also applied to the rollback algorithm. However, the algorithm also has to handle the situation where a process failed after creating a tentative checkpoint and before the checkpointing algorithm finished. If the failing process was the initiator, this means aborting the algorithm.

```
After a failed process is restored:
  own_initiator = me
  if have_tentative_checkpoint then
    -- Figure out result of checkpointing algorithm,
    -- make checkpoint permanent or delete it.
    -- (Also handle the case when we were the initiator)
  fi
  do_rollback(me); -- Let's hope this succeeds
  own_initiator = null;

On receipt of rb_req(initiator, last_smsg) from q:
  if own_initiator != null and initiator != own_initiator then
    -- We are already participating in another execution
    -- of the checkpointing/rollback algorithm.
    send(false) to q;
    return;
  fi
  own_initiator = initiator;
  if not (last_rmsg[q] > last_smsg) then
    -- Rollback request not inherited,
    -- wait until protocol finishes
    send(true) to q;
    await(result(success)) from q;
    own_initiator = null;
    return;
  fi
  -- Inherited rollback request
  do_rollback(q);
  own_initiator = null;

do_rollback(q):
  send(rb_req(own_initiator, last_smsg[r])) to every r;
  success = true;
  -- Note: If failure of some other process
  -- is detected, handle as negative reply
  await all replies:
    if rollback request failed:
      success = false;
    fi
  end
  send(success) to q;
```

−− *Wait until first phase is finished and*
−− *DO NOT continue computation until then*
await(result(success)) from q;
send(result(success)) to every other process
**if** success **then**
  −− *The following also resets* last_smsg *and* first_smsg
  rollback_to_latest_permanent_checkpoint;
**fi**

## REFERENCES


[1] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," *IEEE Transactions on Software Engineering*, vol. SE-13, no. 1, pp. 23–31, 1987.

[2] P. Jalote, *Fault tolerance in distributed systems*. Prentice Hall, 1994, vol. 1.

[3] T. Saridakis, "Design patterns for checkpoint-based rollback recovery," 2003.

[4] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Transactions on Computer Systems*, vol. 3, no. 1, pp. 63–75, 1985.

[5] H. Higaki, K. Shima, T. Tachikawa, and M. Takizawa, "Checkpoint and rollback in asynchronous distributed systems," in *In Proceedings IEEE INFOCOM 97. Sixteenth Annual Joint Conference of the IEEE Computer and Communications Societies*, 1997, pp. 998–1005.

# Fehlertolerante Netzwerktopologien

## I. Einleitung

Fehlertolerante Netzwerktopologien befassen sich mit der Art und mit dem Aufbau eines Netzwerkes, welches möglichst fehlertolerant und stabil arbeitet, dabei zugleich möglichst wenig redundante Mittel einsetzt oder durch eine einfache Struktur ausgezeichnet ist. Zuerst werden verwendete Kennzahlen zur Messung der Stabilität bzw. Fehlertoleranz erklärt. Darauffolgend werden verbreitete Netzwerktypen aufgezeigt und erläutert.

## II. Kennzahlen für Stabilität

Bei der Stabilität und der Fehlertoleranz der Netzwerktopologien gibt es verschiedene Ansätze um diese zu überprüfen. Auf der einen Seite existieren Ansätze aus der Graphentheorie, welche vor allem den Zusammenhang innerhalb des Graphen analysieren. Auf der anderen Seite existieren Ansätze, welche direkt auf Computernetzwerke ausgelegt sind. Beide Teilbereiche werden im Folgenden betrachtet. Dabei können aber nicht unbedingt alle Kennzahlen auf alle Netzwerke angewendet werden. Bei allen Methoden wird nur das jeweilige Netzwerk betrachtet, eine Möglichkeit fehlerhafte Knoten zu reparieren wird dabei nicht beachtet, da hiermit eine Aussage über die Stabilität des Netzwerkes getroffen werden soll, ohne das in das Netzwerk eingegriffen werden muss.

### A. Kennzahlen aus der Graphentheorie

*Knoten- und Kanten-Konnektivität:* Bei der graphentheoretischen Betrachtung der Stabilität von Computernetzwerken wird unter anderem oft eine Analyse der Knoten- und Kanten-Konnektivität genutzt. Hierbei wird der Graph überprüft, inwiefern bei einem Fehler, oder einer Fehlfunktion eines bestimmten Knoten, weitere Knoten nicht mehr erreichbar, oder sogar vom Graphen abgeschnitten sind [1, S. 110]. Die Konnektivität wird durch die minimale Anzahl an Knoten ausgedrückt, welche aus dem Graphen entfernt werden müssen, damit Knoten nicht mehr erreichbar sind. Dabei gilt, je höher der Grad der Knoten ist, desto stabiler und fehlertoleranter ist der Graph. Ebenso kann aus einer hohen Knoten- und Kanten-Konnektivität gefolgert werden, dass der Graph eine hohe Stabilität besitzt, da der Graph bei Fehlern nicht auseinander bricht und manche Knoten somit nicht mehr erreichbar sind.

*Durchmesser-Stabilität:* Hierbei wird auf der einen Seite die kürzeste Entfernung zwischen einem Start- und einem Endknoten betrachtet. Auf der anderen Seite wird der Durchmesser des Graphen betrachtet, wobei der Durchmesser der längste Weg zwischen zwei beliebigen Knoten ist. Die Durchmesser-Stabilität zeigt auf, wie schnell die Distanz innerhalb eines Netzwerkes wächst, sofern Knoten fehlerhaft sind und ausfallen [1, S. 110f].

Eine Ausprägung der Durchmesser-Stabilität ist die Beständigkeit, welche die kleinste Anzahl an fehlerhaften Knoten darstellt, die benötigt werden um den Durchmesser zu erhöhen. Als Beispiel ist die Beständigkeit in einem ungerichteten Ring Netzwerk $eins$, denn sobald ein einziger Knoten ausfällt, wächst der Durchmesser.

### B. Kennzahlen für Computer-Netzwerke

*Zuverlässigkeit:* Bei der Zuverlässigkeitsanalyse wird das Zeitintervall betrachtet, in dem das Netzwerk einwandfrei funktioniert. Bei einem Netzwerk ohne Redundanz ist die Zuverlässigkeit gleichbedeutend mit der Zeit in der kein einziger Fehler auftritt. Bei vorhandenen Redundanzen wird dadurch das Zeitintevall ausgedrückt, wobei trotz Fehlern zu jeder Zeit zumindest ein fehlerfreier Pfad zwischen zwei beliebigen Knoten durch das Netzwerk existiert.
Die Zuverlässigkeit eines Netzwerkes wird im Folgenden auch durch $R(t)$ ausgedrückt, wobei mit $t$ das Zeitintervall $[0, t]$ bezeichnet wird, mit $t$ dem ersten Auftreten eines Fehlers [1, S. 111].

*Bandbreite:* Eine Analyse des Netzwerkes hinsichtlich der Bandbreite beachtet hierbei die Geschwindigkeit mit der auf eine bestimmte Datei zugegriffen werden kann. Sind in einem Netzwerk die Knoten auf z.B. 10Mbits/Sekunde ausgelegt, so wird betrachtet wie die Bandbreite verändert wird, sofern Knoten innerhalb des Netzwerkes ausfallen. Die restlichen Knoten arbeiten in dem Fall immer noch mit 10Mbits/Sekunde, aber die Last des ausgefallenen Knotens wird von den verbleibenden, fehlerfreien Knoten übernommen. Sollte keine Verbindung mehr existieren, da ein essentieller Knoten ausgefallen ist und keine Redundanz existiert, so wird die Bandbreite auf 0 reduziert. Die Bandbreite wird im Folgenden auch durch $B(t)$ dargestellt, $t$ steht dabei wieder für das Zeitintervall $[0, t]$ in dem die maximale Bandbreite im gesamten Netzwerk vorhanden ist [1, S. 112].

*Vernetzungsgrad:* Der Vernetzungsgrad betrachtet, anders als die erste Methode, eher den Zerfall des Netzwerkes bevor dies komplett zum Erliegen kommt. Hierbei wird betrachtet, welche Kombinationen von Start-End-Knoten zu einem bestimmten Zeitpunkt $t$, unter der Existenz von Fehlern, noch verbunden sind [1, S. 112].

## III. Verbreitete Netzwerktopologien und Fehlertoleranz

Im Folgenden werden die betrachteten Netzwerktopologien erläutert und auch grafisch dargestellt. Dabei wurde die Auswahl auf häufig genutzte Netzwerktopologien bzw. spezielle Topologien reduziert, welche durch eine gute Fehlertoleranz hervorstechen. In diesem Abschnitt werden die Topologien zuerst erläutert und hinsichtlich der Fehlertoleranz analysiert.

Die Betrachtung liegt hier auf zwei verschiedenen Einsatzbereichen von Netzwerken. Der erste Einsatzbereich liegt im Verbinden von berechnenden Komponenten und verteiltem Speicher. Der zweite Einsatzort sind Computernetzwerke im allgemeinen, bei denen die einzelnen Knoten bereits aus berechnenden Komponenten bestehen. Bei ersteren bestehen die Netzwerke aus Switchboxen, welche eine Menge an $Input$-Knoten mit einer Menge an $Output$-Knoten verbinden. Diese Netzwerktypen werden häufig bei Anwendung von Multi-Prozessor-Systemen mit geteiltem Speicher genutzt. Hierbei wären die Prozessoren bspw. die Input-Knoten und der Speicher wird durch die Output-Knoten dargestellt.

Bei der Analyse wird zu den betrachteten Netzwerktopologien ermittelt, wie diese bei den jeweiligen Kennzahlen für Stabilität abschneiden und welche Verbesserungsmaßnahmen und Restrukturierungen der Netze positiven Einfluss auf die Stabilität besitzen. Dabei werden nicht alle Kennzahlen auf alle

Netzwerktopologien angewendet. Dies hat zwei Gründe, zuerst existieren für einige Kennzahl-Topologie-Kombination keine Berechnungen oder diese können nur begrenzt angewendet werden. Andererseits sind einige Betrachtungen zu komplex um sie in diesem Rahmen darzustellen.

Bei der Darstellung von den Verbesserungen der Fehlertoleranz der Netzwerke, werden die neuen Komponenten, wodurch das jeweilige Netzwerk fehlertoleranter wird, rot hervorgehoben, damit ein sofortiger Vergleich zu der nicht-fehlertoleranten Variante des Netzwerkes möglich ist. Zum Anfang werden dabei Topologien zum Verbinden von berechnenden Komponenten mit verteiltem Speicher betrachtet, später die eigentlichen Computernetzwerke.

### A. Bus Netzwerk

Hierbei handelt es sich um eine linear angeordnete Topologie, welche auf der Abbildung 1 zu erkennen ist. Bei dieser Struktur werden alle Knoten mit einem Bus verbunden, über den sämtliche Nachrichten transportiert werden.

Diese Topologie wurde bei den Anfängen der Computernetzwerke häufig eingesetzt. Aktuelle Netzwerkstrukturen sind dagegen viel zu komplex um sie durch diese Topologie zu realisieren. Diese Struktur wird weiterhin noch häufig intern in Systemen eingesetzt.

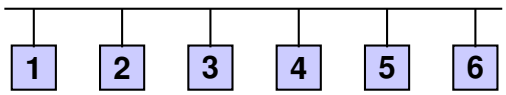



Abbildung 1. Bus Netzwerktopologie

*Analyse der Topologie:* Trotz der einfachen Verbindung der Knoten ist die Knoten-Konnektivität gleichzusetzen mit der Anzahl der vorhandenen Knoten. Bei einem Ausfall des Busses werden sofort alle angebundenen Knoten abgeschnitten. Die Bandbreite wird durch das Minimum von der verfügbaren Bus-Bandbreite und der Bandbreite der einzelnen Knoten berechnet.

Diese Topologie lässt sich durch Hinzufügen eines zusätzlichen Busses fehlertoleranter auslegen. Sofern die Bandbreite des vorhandenen Busses die beschränkende Größe war, wird durch das Hinzufügen eines zusätzlichen Busses die verfügbare Bandbreite ebenfalls heraufgesetzt. Dies ist in der Abbildung 2 zu erkennen. Dabei erhöht sich jeweils der Grad der Knoten um eins.

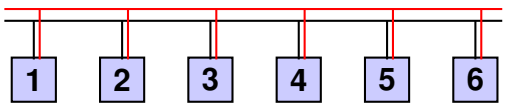



Abbildung 2. Bus Netzwerktopologie

### B. Stern Netzwerk

Eine weitere, recht alte, Netzwerktopologie ist das Stern Netzwerk, hierbei handelt es sich um ein zentralistische Struktur, wobei eine Komponente mittig angeordnet ist und alle anderen Komponenten über diese mit einander verbunden werden. Das Stern Netzwerk wird in Abbildung 3 veranschaulicht.

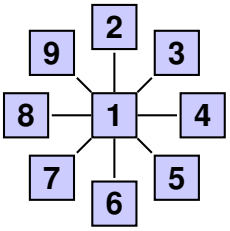



Abbildung 3. Stern Netzwerktopologie

*Analyse der Topologie:* Durch die Struktur des Netzwerkes ist der zentrale Knoten relevant für die Fehlertoleranz des Netzwerkes. So wird die Bandbreite durch die maximale Geschwindigkeit des zentralen Knotens begrenzt. Fällt dieser aus, so werden alle anderen Knoten voneinander abgeschnitten, die Konnektivität beträgt somit eins.

Diese Topologie lässt sich durch einen zusätzlichen zentralen Knoten stabiler auslegen, da hierdurch die Konnektivität auf zwei steigt und die mögliche Bandbreite durch einen zweiten zentralen Knoten ebenfalls steigt. Diese Anpassung ist in der Abbildung 4 zu erkennen.

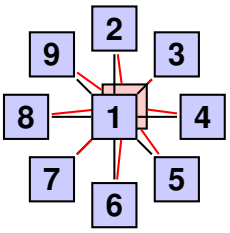



Abbildung 4. Stern Netzwerktopologie

### C. Ring Netzwerk

Das Ring Netzwerk, oder auch als Loop Netzwerk bekannt, ist eine einfache hintereinander Reihung von beliebig vielen Knoten, welche jeweils mit dem Nachfolger verbunden sind. Zusätzlich existiert eine Verbindung von dem letzten zum ersten Knoten, sodass ein Ring entsteht. In der Abbildung 5 ist ein Beispiel zu einem einfachen Ring Netzwerk mit acht Knoten zu erkennen. Das Ring Netzwerk ist das erste, bei dem Verbesserungen zur Fehlertoleranz existieren und auch angewendet werden. Weiterhin wird die Struktur des Ring Netzwerkes in weiteren Netzwerktopologien wiederverwendet.

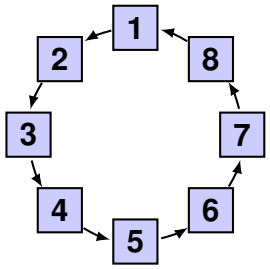



Abbildung 5. Ring. bzw. Loop-Netzwerktopologie

*Analyse des Netzwerkes:* Bei der einfachen Ring-Struktur dieses Netzwerkes ist leicht zu erkennen, dass gerade bei einem gerichteten Netzwerk ein einziger Knoten ausreicht, um das gesamte Netzwerk zu unterbrechen. Bei dem ungerichteten Netzwerk genügen dabei zwei fehlerhafte Knoten damit weitere Knoten abgeschnitten werden und somit nicht mehr erreichbar sind. Der Durchmesser des ungerichteten Netzwerkes beträgt $n-1$, wobei $n$ die Anzahl der vorhanden Knoten ist. Der Grad jedes Knotens beträgt zwei. Eine Ring- bzw. Loop-Struktur ist bei Netzwerken generell recht intolerant, da diese Struktur schnell unterbrochen werden kann. In Verbindung mit anderen Strukturen, wie z.B. dem Cube-Connected Cycles Netzwerk, siehe Abschnitt III-H, kann dieser Typ aber gut eingesetzt werden, da nur wenige Kanten zum Verbinden benötigt werden.

*Verbesserung der Struktur:* Das Ring-Netzwerk kann durch zusätzliche Kanten, welche entgegen der bereits vorhanden verlaufen, verbessert werden. Dadurch entstehen zusätzliche Zyklen innerhalb des Netzwerkes. Die Anzahl der Zyklen ist entsprechend der Anzahl an zusätzlichen Kanten. Diese ergänzenden Kanten werden auch als $Chord$ bezeichnet. Zusätzlich können durch diese Rückwärtskanten auch Knoten übersprungen werden. Die Anzahl an übersprungener Knoten wird durch die sogenannte $Skip-Distance$ dargestellt.

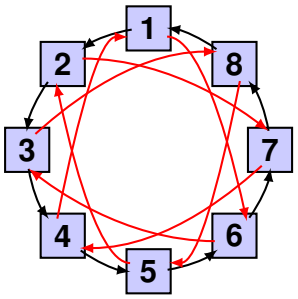



Abbildung 6. Ring. bzw. Loop-Netzwerktopologie

Die Anzahl der Chords wird im Allgemeinen dahin gehend optimiert, dass der Durchmesser des Netzwerkes möglichst minimal wird. Der Durchmesser lässt sich bei einem Ring-Netzwerk mit Chords durch folgende Formel berechnen:
$\lfloor \frac{n}{Skip-Distance+1} \rfloor + (Skip-Distance-1)$.

Die optimale $Skip-Distance$ beträgt dabei $\lfloor \sqrt{n} \rfloor$. Wobei $n$ die Anzahl der Knoten darstellt. In dem obigen Beispiel wäre die optimale Distanz somit $\lfloor \sqrt{8} \rfloor = 2$. Daraus ergibt sich in diesem Beispiel ein optimierter Durchmesser von $\lfloor \frac{8}{3} \rfloor + 2 - 1 = 3$ [1, S. 130ff].

### D. Mehrstufiges Netzwerk

Mehrstufige Netzwerke, auch als Multistage Netzwerke bekannt, werden zumeist für eine Verbindung von verteiltem Speicher und Prozessoren verwendet. Diese bestehen aus Eingangsknoten, welche die Nachrichten mit Hilfe von einfachen Switchboxen an die Ausgangsknoten weiterleiten. Diese Switchboxen haben nur eine begrenzte Möglichkeit die Daten weiterzuleiten, denn sie verfügen nur über folgend abgebildete Funktionen:

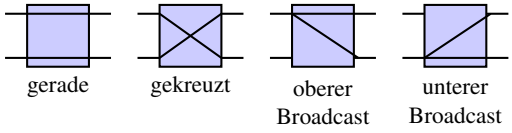



Abbildung 7. Funktionen der Switchboxen

Dabei können die Daten entweder direkt durch geleitet werden, wie es im linken Teil der Abbildung zu erkennen ist. Die Daten können über Kreuz weitergegeben werden und weiterhin können die Daten entweder von dem oberen Eingang an beide Ausgänge oder von dem unteren Eingang an die beiden Ausgänge propagiert werden, dies ist in dem rechten Bereich der Grafik 7 zu erkennen.

Das Netzwerk wird hier einfachheitshalber in einer rechteckigen Struktur dargestellt. Dabei werden die Spalten als $Stage$ oder Stufe bezeichnet. Eine mögliche Ausprägung des mehrstufigen Netzwerkes ist in der Abbildung 8 zu erkennen[1, S. 112ff][2, S 443ff].

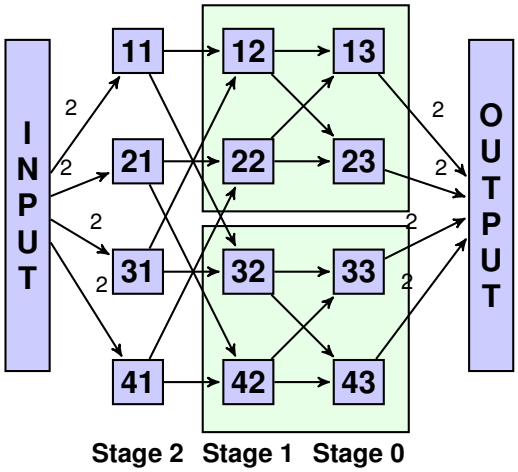



Abbildung 8. Mehrstufige Netzwerktopologie

Das mehrstufige Netzwerk besteht aus Switchboxen, welche jeweils zwei Ein- und zwei Ausgänge besitzen. In dem betrachteten Netzwerk werden insgesamt acht Eingänge mit acht Ausgängen verbunden. Dies geschieht durch zwei sogenannte $Butterfly-Netzwerke$. Diese sind in der Abbildung auf der rechten Seite, jeweils leicht hervorgehoben, zu erkennen. Diese werden durch eine extra Stage miteinander verbunden. So entsteht aus zwei 4x4-Butterfly-Netzwerken ein 8x8-Netzwerk. Diese Konstruktion lässt sich beliebig fortführen. Somit verbindet ein $2^k x 2^k$ Butterfly-Netzwerk $2^k$ Eingänge mit $2^k$ Ausgängen und besteht dabei aus $2^{k-1}$ Stages mit insgesamt $\frac{k}{2} * 2^{k-1}$ Switchboxen [1, S. 112ff].

*Analyse des Netzwerkes:* Bei dem Butterfly-Netzwerk ist zu erkennen (vgl. Abb. 8), dass jede Eingangskante immer nur genau zu einer Ausgangskante führt und kein alternativer Weg existiert. Aus diesem Grund bietet dieser Netzwerktyp keine Fehlertoleranz. Fällt ein Knoten der Stage $i$ aus, so werden $2^{k-i}$ Inputs von $2^{i+1}$ Outputs abgeschnitten. Würde in diesem Beispiel der Knoten 22, der Stage 2 ausfallen, so sind $2^{2-1} = 2$ Verbindungen der Input-Knoten (21 und 41) von $2^{1+1} = 4$ Verbindungen der Output-Knoten (13, 23) abgeschnitten. Die Konnektivität beträgt somit eins.

*Verbesserung der Struktur:* Die obige Struktur könnte um eine zusätzliche Stage erweitert werden. Dies ist in der Abbildung 9 zu erkennen. Die zusätzlichen Knoten und Verbindungen wurden hierbei rot eingefärbt. Die neuen Knoten leiten die eingehenden Nachrichten dabei zusätzlich auch direkt weiter, sodass ein Fehler in der Stage 3 unerheblich ist, die Nachrichten würden direkt an die Stage 2 weitergeleitet. Es existiert also innerhalb des Knotens eine zusätzliche, permanente Kante von dem Eingang zum Ausgang. Wenn in dem Netzwerk aber der Knoten 11 ausfallen würde, so können die Nachrichten von 10 nach 21 und von 30 nach 21 umgeleitet werden. somit würde der defekte Knoten umgangen werden. $B(t)$ wäre nach dem Auftreten eines Fehlers somit eingeschränkt, denn in dem folgenden Beispiel würde die Nachricht auch durch 22 oder 42 geleitet, diese werden aber auch bereits von den Eingängen der Switchbox 41 benutzt. Die Konnektivität beträgt hierbei zwei, denn durch das Einfügen der zusätzlichen Stufe können zwei beliebige Switchboxen ausfallen, ohne das Knoten abgeschnitten werden.

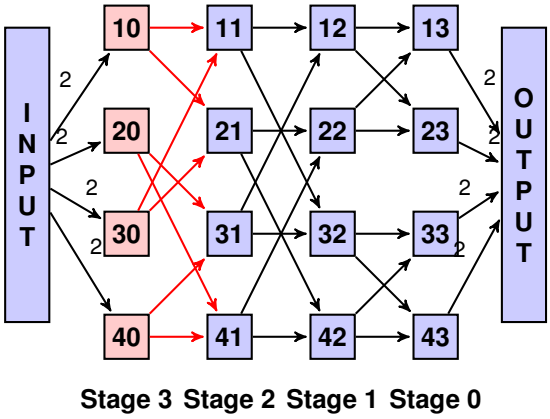



Abbildung 9. Mehrstufige Netzwerktopologie

### E. Verbindungsnetzwerk

Die Struktur des Verbindungsnetzwerkes, auch als Crossbar Netzwerk bekannt, basiert auf der Struktur des mehrstufigen Netzwerkes. Die Verbindungen von den einzelnen Knoten erfolgt aber auf eine andere Art und Weise. Hierbei besitzen die Switchboxen ebenfalls je zwei Ein- und Ausgangskanten, die Switchboxen können die Nachrichten diesmal aber nur in den folgenden Richtungen weitergeben:

- Nachrichten von links werden nach rechts weitergegeben
- Nachrichten von unten werden nach oben weitergegeben
- Nachrichten von links werden nach oben weitergegeben

Dieses Netzwerk wird, ebenfalls wie das mehrstufige Netzwerk, hauptsächlich für eine Kommunikation zwischen Prozessoren und verteiltem Speicher benutzt. Das Crossbar Netzwerk wird in der Abbildung 10 dargestellt.

*Analyse des Netzwerkes:* Da das Crossbar Netzwerk nur aus gerichteten Kanten besteht und die einzelnen Switchboxen nur in den drei bekannten Richtungen Nachrichten propagieren können ist schnell ersichtlich, dass bei einem Ausfall einer einzigen Switchbox nicht mehr alle Nachrichten weitergegeben werden können. Bei einem NxM Netzwerk, wobei $N$ die Anzahl der Spalten und $M$ die Anzahl der Zeilen darstellt, können bei einem Defekt einer Switchbox bis zu N Ausgänge nicht mehr erreicht werden. Der

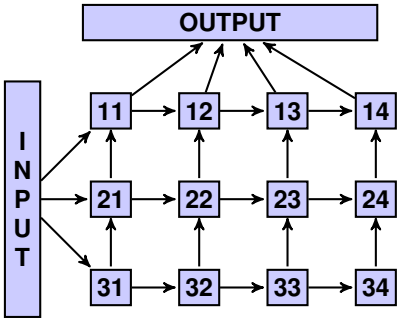



Abbildung 10. Crossbar Netzwerktopologie

Vorteil des Crossbar Netzwerkes gegenüber dem mehrstufigen Netzwerkes liegt in der Anzahl der benötigten Switchboxen. Bei dem mehrstufigen Netzwerk werden, bei $2^k$ Ein- und Ausgängen, $k * 2^{k-1}$ Switchboxen benötigt. Bei einem Crossbar Netzwerk mit der gleichen Anzahl an Ein- und Ausgängen werden dagegen nur $k^2$ Switchboxen benötigt. Hierbei können gleichzeitig pro Verbindung jeweils eine Nachricht übertragen werden, jede Switchbox kann dabei bis zu zwei Nachrichten gleichzeitig verarbeiten. Dadurch wird beim Crossbar Netzwerk die Bandbreite nicht eingeschränkt, da die Nachrichten parallel bearbeitet werden können, solange nicht Verbindungen zwischen zwei Eingangsknoten und dem gleichen Ausgangsknoten aufgebaut werden.

*Verbesserung der Struktur:* Das Crossbar Netzwerk kann durch das Hinzufügen von jeweils einer zusätzlichen Spalte und Zeile und darüber hinaus durch hinzufügen von Switchboxen ergänzt werden. Diese Switchboxen haben die Möglichkeit, eine eingehende Nachricht an die gleiche Zeile oder an den unteren bzw. den linken Nachbarn weiterzuleiten. Durch diese Erweiterung wird eine Fehlertoleranz ermöglicht, so dass ein Ausfall von bis zu einem beliebigen Knoten problemlos aufgefangen werden kann. Die verbesserte Struktur ist auf der folgenden Grafik, Abbildung 11, zu erkennen. Auch hierbei wird bei einem Fehler die Bandbreite eingeschränkt, da die Nachricht auf eine Nachbarspalte bzw. -Zeile übergeben wird und diese auch die eigentlichen Daten verarbeiten muss. Der Umfang ist aber weit geringer wie bei dem mehrstufigen Netzwerk.

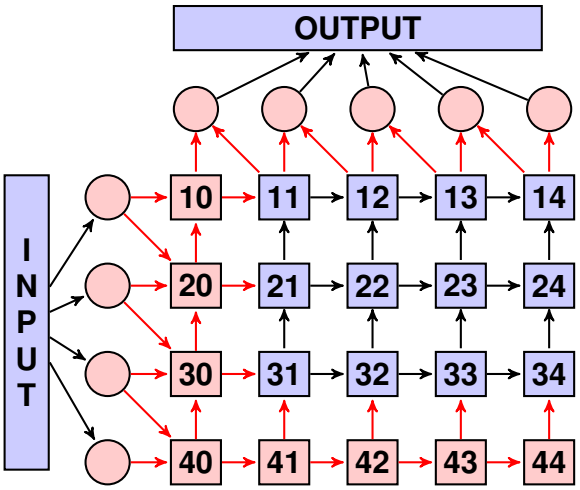



Abbildung 11. Crossbar-Netzwerktopologie

### F. Rectangular Mesh Netzwerk

Netzwerke mit einer Rectangular Mesh-Struktur zeichnen sich durch einen Aufbau aus, wie sie in Abbildung 12 abgebildet wird. Dabei besteht das Netzwerk aus einem rechteckigen Verbund der Komponenten. Jeder innere Knoten hat jeweils vier Nachbarknoten. Rand- und Eckknoten haben jeweils drei bzw. zwei Nachbarn. Hierbei handelt es sich nicht, wie bei dem Crossbar Netzwerk um einen gerichteten Netzwerktyp, sondern um ein bidirektionales Netzwerk. Diese Topologie wird, entgegen den vorangegangen Netzwerken, für einen Verbund von berechnenden Knoten verwendet, wie etwa in einem Computernetzwerk.

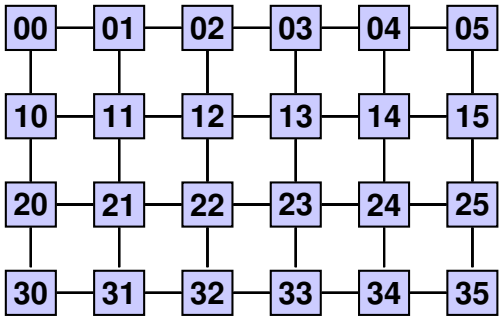



Abbildung 12. Rectangle-Mesh-Netzwerktopologie

*Analyse des Netzwerkes:* Bei dem Rectangular Mesh Netzwerkes handelt es sich erstmalig um eine Topologie, welche für Rechnernetze verwendet werden kann. Hierbei spiegelt jeder Knoten keine Switchbox wider, sondern einen berechnenden Knoten, wie etwa einen Computer. Bei dieser Topologie gibt es trotz der bidirektionalen Verbindungen keine Fehlertoleranz, ohne dass die Eigenschaft des voll vermaschten Netzes verloren geht. Diese wird bei bestimmten Algorithmen vorausgesetzt, da der jeweilige Knoten auf bereits berechnete Werte des Nachbarknotens zugreifen muss.

Anzahl der Knoten bei einem NxM-Netzwerk beträgt $N * M$ Knoten mit jeweils bis zu vier Verbindungen.

*Verbesserung der Struktur:* Diese Netzwerktopologie kann durch einfügen sogenannter Spare-Knoten fehlertoleranter ausgelegt werden. Dabei existieren zwei verschiedene Möglichkeiten. Entweder werden pro vier existenten Knoten ein Spare-Knoten eingefügt, oder pro Knoten existieren vier Spare-Knoten. Die erste Variante ist in der folgenden Grafik blau gestrichelt dargestellt und die ergänzenden Knoten zur zweiten Variante werden rot dargestelt.

Bei der 1,4-Variante, mit einem Spare-Knoten pro vier vorhandenen Knoten, beträgt die Anzahl an zusätzlichen Knoten $\frac{N*M}{4}$ und bei der 4,4-Variante beträgt diese Anzahl zusätzlich $(N-1)*(M-1)$ Knoten. Speziell bei der 4,4-Variante werden sogenannte Cluster gebildet. Dabei umgeben vier Knoten einen Spare-Knoten. Pro Cluster dürfte somit ein beliebiger Knoten ausfallen und die Funktion würde von dem Spare-Knoten sofort übernommen. Ein großer Vorteil dieser Struktur liegt bei der räumlichen Nähe der Spare-Knoten.

### G. Hypercube Netzwerk

Bei der Hypercube Netzwerktopologie wird das gesamte Netzwerk in einem n-Dimensionalen Würfel strukturiert. In der Abbildung 14

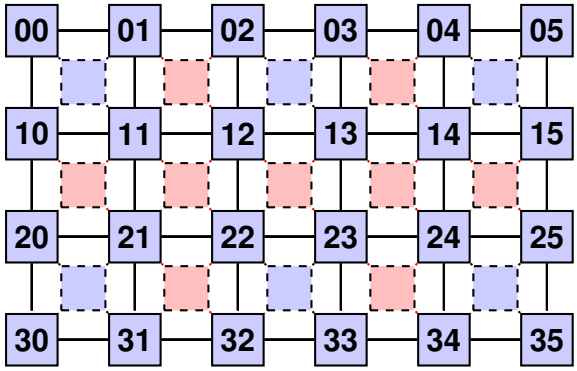



Abbildung 13. Rectangle-Mesh-Netzwerktopologie

ist ein Hypercube der 4. Dimension zu erkennen, im weiteren auch als $H^n$ bezeichnet, wobei $n$ die Dimension ist. Der Hypercube lässt sich rekursiv aufbauen. Ein $H^0$ besteht dabei aus einem einzigen Knoten. Die Erweiterung zum $H^1$ erfolgt durch verbinden von zwei $H^0$. Dieser Vorgang kann beliebig weit fortgeführt werden. Bei dieser Topologie besitzt jeder Knoten jeweils $n$ Kanten, welche die Knoten miteinander verbinden und das vollständige Netzwerk besteht aus $n^2$ Knoten[3, S. 2f][4, S. 3ff].

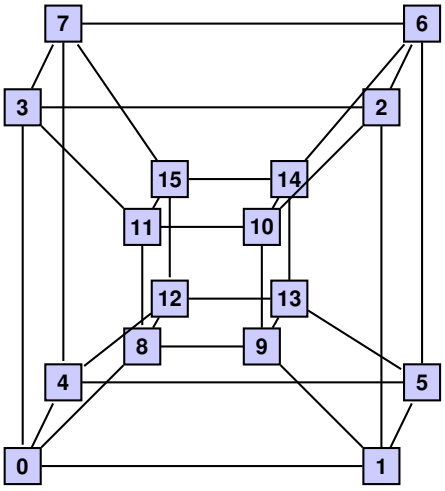



Abbildung 14. Hypercube-Netzwerktopologie

Der Hypercube kann auch noch in einer weiteren Darstellung veranschaulicht werden[1, S. 124ff]. Dabei wird der drei dimensionale Würfel nur auf zwei Dimensionen abgebildet. Dies ist in der Abbildung 15 zu erkennen. Dieser wird anhand des folgenden Schemas aufgebaut. Zuerst wird jeweils einzelne Verbindungen zwischen zwei Knoten geschaffen. Diese wird in der Grafik durch eine graue Kante dargestellt, dies ist auch zwischen den Knoten 0 und 1 zu erkennen. Weiter werden alle Knoten der Dimension 1 zur Dimension 2 verbunden. Dies geschieht in der Grafik durch eine gestrichelte Kante, hier die Knoten 0 und 2. Die Dimension 3 wird wiederum durch verbinden der vorhergehenden erstellt. Diese Kante wird in der Grafik durch eine rote, gestrichelte Linie dargestellt, hier zwischen 0 und 4. Die vierte Dimension ergibt sich, indem die beiden Gruppen der Dimension 3 verbunden werden. Dies geschieht durch die durchgehenden schwarzen Kanten, hier zwischen 2 und 10. Somit ist diese Darstellung äquivalent zu der

Abbildung 14. Dieses Aufbauschema lässt sich beliebig erweitern, sodass auch Graphen mit einer höheren Dimension möglich sind. Der Grad der Knoten ist dabei jeweils abhängig von der Dimension. Bei einem $H^4$ besitzt dabei jeder Knoten den Grad vier.

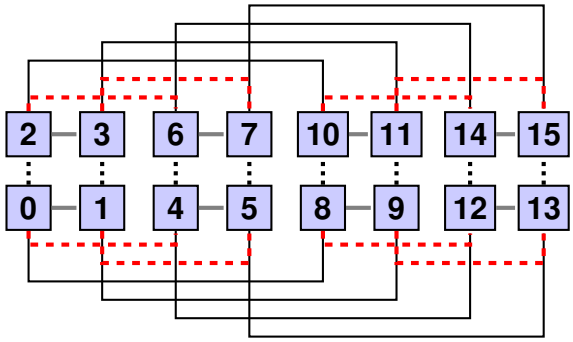



Abbildung 15. Hypercube-Netzwerktopologie - 2D-Darstellung

*Analyse des Netzwerkes:* Dieser Netzwerktyp bietet von vornherein eine geringe Fehlertoleranz, da mindestens zwei Knoten ausfallen müssten um andere Knoten vom restlichen Netzwerk abzuschneiden. Die Konnektivität beträgt hierbei somit 2. Der Durchmesser dieses Netzwerkes ist gleichzusetzen mit der Dimension des Hypercubes. Ein Hypercube besitzt dabei immer $2^n$ Knoten, mit $n$ jeweils der Dimension des Hypercubes.

*Verbesserung der Struktur:* Um die Fehlertoleranz noch weiter zu verbessern, können zusätzliche Verbindungen eingefügt werden, welche die Knoten mit Spare-Knoten verbinden, die bei einem Ausfall einspringen würden. In diesem Fall erhöht sich der Grad der Knoten um eins und es werden pro $2^{n-1}$ Knoten zwei Spare-Knoten benötigt. Diese Erweiterung ist in der folgenden Abbildung gut zu erkennen. Hierbei wird auf Grund der Übersichtlichkeit auf die 2D-Darstellung zurückgegriffen. Um die Anzahl der Ein- und Ausgänge bei den Spare-Knoten zu reduzieren werden hier Switchboxen vorgeschaltet, welche den Grad soweit reduzieren, dass alle Knoten den selben Grad aufweisen.

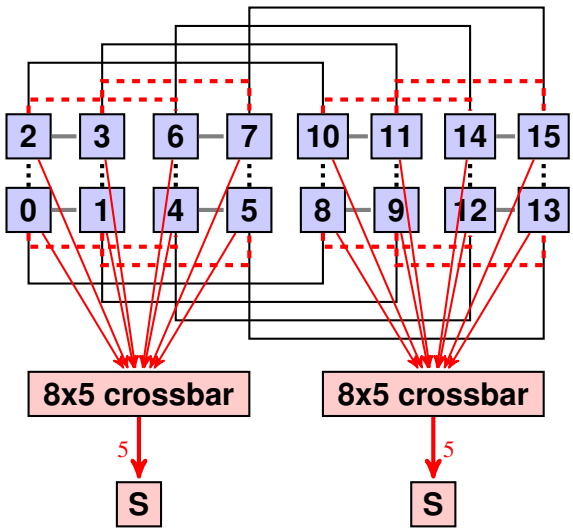



Abbildung 16. Hypercube Netzwerktopologie - 2D-Darstellung

Da hier jeweils noch ein weiterer Knoten und eine weitere Verbindung eingefügt wurde, dürfen im schlechtesten Fall bis zu drei Knoten ausfallen ohne dass Teile des Netzwerkes abgeschnitten werden. Die Konnektivität beträgt somit 3. Dazu existieren jeweils meist multiple Pfade zwischen den Knoten, so dass generell ein niedriger Durchmesser vorliegt und die Durchmesser-Stabilität ebenfalls gering bleibt.

Eine weitere Alternative ist der sogenannte Crossed Cube. Dabei wird die Struktur des $H^3$ dahingehend verändert, dass die Knoten auf einer anderen Art und Weise miteinander verbunden werden. Diese Veränderung ist in dem linken Teil der Grafik 17 zu erkennen.

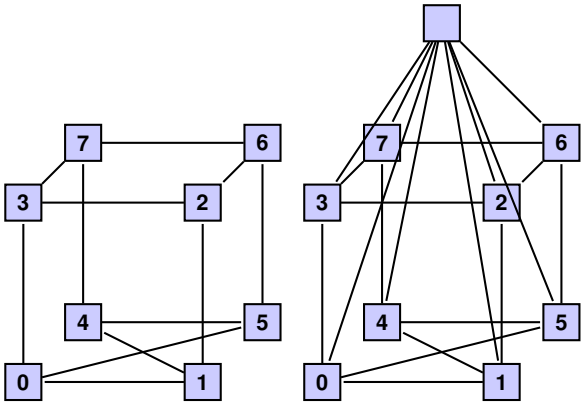



Abbildung 17. Crossed Cube Netzwerktopologie

Diese Struktur kann um einen weiteren Verbindungsknoten erweitert werden, siehe Abbildung 17, rechte Seite. Mit Hilfe dieses Knotens kann der Crossed Cube rekrusiv aufgebaut und erweitert werden. Muss das Netzwerk vergrößert werden, so kann jeder einzelne Knoten aus dem linken Teil der Grafik jeweils so einen kompletten Graphen beinhalten [5, S. 86ff].

### *H. Cube-Connected Cycles Netzwerk*

Bei diesem Netzwerktyp wird ebenfalls eine würfelartige Struktur aufgebaut, wobei die Eckpunkte des Würfels jeweils aus einem kleinen Ring Netzwerk bestehen. Der Aufbau dieser Topologie ist in der Abbildung 18 zu erkennen. Hierbei bestehen die Ecken des Würfels aus mehreren Knoten, welche mit Hilfe eines Ring Netzwerkes miteinander verbunden werden. Diese Teilnetzwerke werden mit Hilfe einer weiteren Kanten zu einem Würfel zusammengefügt. Der Vorteil gegenüber dem Hypercube Netzwerk besteht darin, dass die Knoten dieser Topologie immer exakt den Grad drei besitzen. Wird das Netzwerk erweitert, müssen die Knoten nicht eine zusätzliche Verbindung aufnehmen können, wie dies von der Erweiterung von einem $H^n$ zu einem $H^{n+1}$ der Fall wäre, da der Knotengrad dabei von der Dimension abhängig ist [4, S. 14ff].

Um diese Topologie zu erweitern, um zusätzliche Knoten zu ermöglichen, wird das gleiche Netzwerk nochmals aufgebaut und die beiden Teilnetzwerke besitzen an deren Ecken Loop Netzwerke welche dabei je um einen Knoten erweitert werden. Der zusätzliche Knoten, der bislang nur vom Grad zwei ist, hat jetzt eine Anbindung frei. Mit Hilfe dieser noch freien Verbindung werden die beiden Netzwerke wieder verbunden, so dass ein Cube-Connected Cycles

Netzwerk der vierten Dimension entsteht [3, S. 4f] [1, S. 128ff].

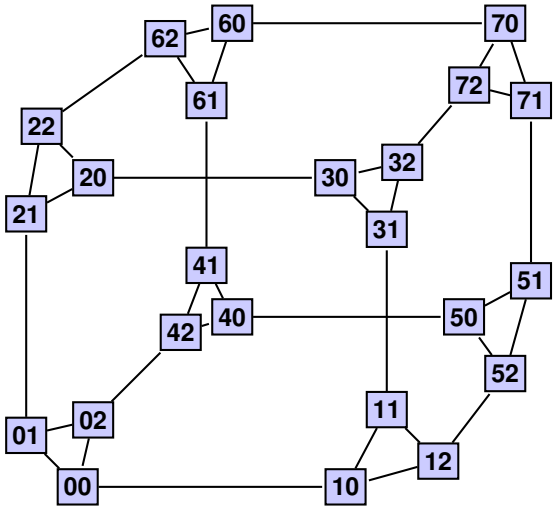



Abbildung 18. Cube-Connected Cycles-Netzwerktopologie

*Analyse des Netzwerkes:* Bei dieser Topologie wird ein Loop Netwerk mit einem Hypercube Netzwerk verbunden. Die Vorteile des Netzwerks liegen genau in dieser Kombination. Wird der Hypercube erweitert, so muss jeder Knoten eine zusätzliche Verbindung aufnehmen können. Das Cube-Connected Cycles Netzwerk behält dabei immer einen festen Knotengrad von drei. Diese Topologie ist einfacher zu erweitern, besitzt aber einen größeren Durchmesser, wie ein Hypercube der gleichen Dimension. Weiterhin besitzt der Hypercube der Dimension n $2^n$ Knoten und ein Cube-Connected Cycles Netzwerk der gleichen Dimension beinhaltet bis zu $n * 2^n$ Knoten.

Der Nachteil dieser Topologie liegt in einem erhöhten Durchmesser gegenüber des Hypercubes mit der gleichen Dimension. Der Durchmesser dieser Struktur lässt sich wie folgt berechnen: $2 * n + \lfloor \frac{n}{2} \rfloor - 2 \approx 2,5n$, wobei $n$ die Dimension des Netzwerkes darstellt.

*Verbesserung der Struktur:* Bei dem Cube-Connectd Cycles Netzwerk wirkt sich der Fehler/Ausfall von einem Knoten genauso aus, wie der Ausfall einer Verbindung bei einem Hypercube der gleichen Dimension. Auf eine weitere Verbesserung dieser Struktur wird nicht weiter eingegangen.

### *I. Ad Hoc Netzwerk*

Die letzte Topologie, welche vorgestellt wird, ist ein Ad Hoc Netzwerk, welches in der Realität sehr häufig vorkommt. Gerade in Computernetzwerken in denen die beteiligten Knoten sehr oft wechseln und somite keine feste Struktur besteht. Eine schematische Abbildung ist unter 19 zu erkennen. Diese Struktur ist nicht an feste Bedingungen und Strukturen gebunden.

*Analyse der Topologie:* Durch die unbeständige Struktur, da das Netzwerk beliebig erweitert und verändert werden kann, und dadurch, dass diese Netzwerktopologie keinen festen geometrischen Strukturen unterliegt, ist eine Betrachtung der Kennzahlen für diesen Netzwerktyp recht komplex. Es gibt verschiedene Möglichkeiten Kennzahlen hierzu zu berechnen. Dazu müssen alle vorhandenen Wege von einem Start- zu einem Zielknoten, zusammen mit der

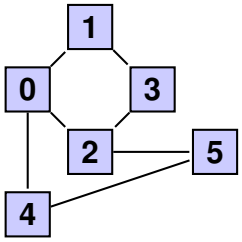



Abbildung 19. Ad Hoc-Netzwerktopologie

Wahrscheinlichkeit, dass diese Wege fehlerhaft sind, betrachtet werden. Alleine in der Abbildung 19 existieren bereits drei Wege von dem Knoten 0 zum Knoten 2. Die Verbesserung dieses Netzwerktyps werden hierbei nicht betrachtet.

## IV. Fazit

Zusammenfassend müssen die vorgestellten Netzwerktopologien in zwei Kategorien unterteilt werden. Auf der einen Seite die Netzwerke, welche aus einfachen Switchboxen bestehen und Nachrichten nur in bestimmten Richtungen weiterleiten können. Diese Verbinden meistens verteilte Speichereinheiten mit den Recheneinheiten. Auf der anderen Seite existieren die Computernetzwerke, welche aus berechnenden Knoten bestehen. Bei beiden Arten existieren verschiedenste Netzwerktopologien, welche diverse Vor- und Nachteile aufweisen. Allen gemein lässt sich aber zusammenfassen, dass eine fehlertolerante Auslegung der Netzwerke zu Lasten von zusätzlichen Verbindungen und Knoten geht. Weiterhin besitzen komplexere und größere Netzwerkstrukturen meistens eine bessere Fehlertoleranz, wie einfache Strukturen, wie Beispielsweise das Loop-Netzwerk gegenüber dem Hypercube.

## Literatur


[1] I. Koren und C. M. Krishna, *Fault-Tolerant Systems*. Morgan Kaufmann, 2007, ISBN:978-0120885251.

[2] G. B. A. III und H. J. Siegel, Hrsg., *The Extra Stage Cube: A Fault-Tolerant Interconnection Network for Supersystems*, Bd. C-31, Ser. IEEE Transactions on Computers 5, IEEE, IEEE, 1982.

[3] R. Elsäßer, *Graphenalgorithmen*, http : / / www2 . cs . uni - paderborn . de/cs/ag - monien/PERSONAL/ELSA/GraphAlg10/ folien1010.pdf, zuletzt besucht: 16.06.2013, 2011.

[4] A. Schwill, *Netzwerke der Hypercube-Familie*, http://ddi.cs.uni-potsdam . de / Lehre / ParalleleAlgorithmen / Vortraege9 - 16 . pdf, zuletzt besucht: 16.06.2013, 2003.

[5] M. N. Adhikari und D. C. R. Tripathy, Hrsg., *Extended Crossed cube: An Improved Fault Tolerant Interconnection Network*, Ser. Fifth International Joint Conference on INC, IMS and IDC, ISBN: 978-0-7695-3769-6, IEEE, IEEE Computer Society, 2009.

# Fehlertolerante Routing-Algorithmen

Valentin Spreckels
Department für Informatik
Carl von Ossietzky Universität Oldenburg
26111 Oldenburg, Deutschland
valentin.spreckels@Informatik.Uni-Oldenburg.DE

***Zusammenfassung*—Redundante Verbindungen dienen in Rechnernetzen zur Verbesserung der Eigenschaften, wie zum Beispiel Verfügbarkeit und Bandbreite. Damit diese Redundanz sinnvoll ausgenutzt werden kann, müssen die beteiligten Knoten sie in ihrem Verhalten berücksichtigen. Daher werden in dieser Ausarbeitung drei grundsätzlich verschiedene Algorithmen, die es den einzelnen Knoten ermöglichen, zu entscheiden, wohin Nachrichten weiterzuleiten sind, vorgestellt.**

## I. Einführung

Rechnernetze bestehen aus Knoten und Verbindungen zwischen diesen. Knoten können Nachrichten an andere Knoten senden und Nachrichten empfangen. Damit beliebige Knoten miteinander kommunizieren können, leiten Knoten nicht für sie bestimmte Nachrichten, die sie erhalten weiter, sodass diese ihr Ziel erreichen. Um zu bestimmen, über welche Verbindungen Nachrichten abgesandt bzw. weitergeleitet werden sollen, wird ein Routing-Algorithmus benötigt. Ein solcher Routing-Algorithmus kann entweder vorab einen Pfad zwischen je zwei Knoten festlegen oder bei jedem Knoten für jede Nachricht einzeln die nächste Verbindung, über die die Nachricht weitergeleitet wird, bestimmen. Verfahren der ersten Art heißen *statisch*, die der zweiten Art heißen *adaptiv*. Zusätzlich unterscheiden sich Routing-Algorithmen darin, ob es einen zentralen Knoten gibt, der mehr Aufgaben als die anderen Knoten hat, oder ob der Algorithmus dezentral funktioniert, da jeder Knoten sich gleich verhält.

Für die Betrachtung der Fehlertoleranz der Algorithmen müssen grundsätzlich zwei verschiedene Fehlerarten betrachtet werden: der Ausfall eines Knoten und der Ausfall einer Verbindung. Innerhalb der Algorithmen können diese Ausfälle dann gar nicht unterschieden werden, wenn die Algorithmen dezentral funktionieren, da die einzelnen Knoten dann nur feststellen können, ob sie ihre Nachbarknoten über die Verbindung erreichen. Wenn ein Nachbarknoten nicht zu erreichen ist, können sie nicht erkennen, ob nur die Verbindung oder der darüber erreichbare Knoten ausgefallen ist.

Für die Bewertung von fehlertoleranten Algorithmen werden die Kenngrößen Überlebenswahrscheinlichkeit und Verfügbarkeit benutzt. Beide Kenngrößen hängen jedoch vor allem von der zugrunde liegenden Netzwerktopologie und nicht so sehr von dem darauf operierendem Routing-Algorithmus ab. Da das Routing die Weiterleitung von Nachrichten beeinflusst, können die Pfade, die diese auf dem Weg zum Empfänger zurücklegen, betrachtet werden. Da deren Analyse relativ aufwendig ist, wird im folgendem nur betrachtet, wie lang die Pfade bei der Anwendung der Algorithmen minimal und maximal werden. Es muss berücksichtigt werden, dass diese Werte zwischen den Algorithmen nicht direkt vergleichbar sind, weil diese auf verschiedenartigen Netzen operieren.

## II. Spanning-Tree-Protocol

In Rechnernetzen, die mit Ethernet-Technik aufgebaut sind, werden verschiedene Protokolle genutzt, um mit redundanten Verbindungen umzugehen. Viele Hersteller haben eigene Protokolle entwickelt, mit dem *Spanning-Tree-Protocol* (STP) nach [1] und dem *Rapid-Spanning-Tree-Protocol* (RSTP) nach [2] gibt es jedoch auch zwei herstellerunabhängig spezifizierte Protokolle.

Wie bereits an den Namen dieser Protokolle erkennbar ist, basieren beide darauf einen Spannbaum aufzubauen. Hierfür senden die beteiligten Knoten regelmäßig auf allen Verbindungen Nachrichten, um zu Anfang die Topologie festzustellen und einen Wurzel-Knoten (denjenigen mit der geringsten MAC-Adresse) zu bestimmen bzw. später Änderungen der Netzwerktopologie zu erkennen. Aus der festgestellten Topologie wird dann ein Spannbaum hergestellt, indem über manche Verbindungen keine Nutznachrichten gesandt werden. Auch über alle anderen Verbindungen werden erst nach dieser Festlegung des Spannbaumes Nutz-Nachrichten gesandt. Ansonsten könnten Nachrichten über redundante Verbindungen wieder zurückkehren und somit immer weiter gesandt werden, sodass Teile des Netzwerkes durch das wiederholte Weiterleiten einzelner Nachrichten ausgelastet würden. Wenn festgestellt wird, dass die Netzwerktopologie sich geändert hat, wird dieser Vorgang wiederholt.

Der Unterschied zwischen den beiden Protokollen besteht in Details dieser Vorgänge. Beim STP dauert es aufgrund von Timeouts bis zu dreißig Sekunden bis nach Änderungen der Netzwerktopologie wieder Nutz-Nachrichten versandt werden können. Dadurch, dass RSTP nach dem Erkennen einer Topologieänderung noch weiter den alten Spannbaum verwendet, bis ein neuer Spannbaum konstruiert wurde, beträgt diese Ausfallzeit meistens weniger als eine Sekunde.

Beide Protokolle können in Verbindung mit Rechnernetzen jeder Topologie benutzt werden und sind zentralisiert, da bei der Realisierung des Spannbaumes der Wurzelknoten eine wichtige Funktion übernimmt. Außerdem wird einmalig festgelegt, wie Nachrichten weiterzuleiten sind. Durch die Deaktivierung redundanter Verbindungen kann es dazu kommen, dass Nachrichten über sehr viel mehr Knoten weitergeleitet werden, als es notwendig wäre, außerdem ist dadurch die verfügbare Datenrate geringer. Der Extremfall ist eine Ringtopologie, bei dieser wird eine Verbindung nicht zum Spannbaum gehören, sodass Nachrichten zwischen den beiden Knoten, die sonst

direkt benachbart waren, nun über alle anderen Knoten des Netzwerkes weitergeleitet werden müssen.

Die folgenden Kapitel stellen zwei Algorithmen vor, die dezentral und adaptiv funktionieren, also weder einen Knoten brauchen, der weitergehende Aufgaben übernimmt, noch eine Lernphase brauchen, in der sie festlegen, wie Nachrichten weiterzuleiten sind. Allerdings sind diese Algorithmen nicht auf allgemeine Rechnernetze, sondern nur auf Netze mit bestimmten Topologien anwendbar. Da diese Topologien häufig in Rechenclustern genutzt werden, können die Algorithmen trotz dieser Einschränkung praktisch genutzt werden.

## III. Routing in Hyperkuben

Der erste Algorithmus wird in [3] und [4] beschrieben. Dieser Algorithmus setzt voraus, dass die Topologie des Rechnernetzes ein Hyperkubus ist also ein Quadrat, Würfel bzw. ein noch höherdimensionales Äquivalent hiervon. Hyperkuben haben die Eigenschaft, dass sie aus zwei Hyperkuben der nächst niedrigeren Dimension zusammen gesetzt werden können, indem jeder Knoten des einen Hyperkubus zusätzlich zu den Verbindungen innerhalb des Hyperkubus noch mit dem entsprechendem Knoten des anderen Hyperkubus verbunden wird. Wenn ein Hyperkubus in jeweils zwei Hyperkuben der nächstniedrigeren Dimension zerlegt wird und diese die Zerlegung dann rekursiv solange angewandt wird, bis der Hyperkubus in einzelne Knoten zerlegt wurde, lässt sich jeder Knoten eindeutig darüber beschreiben, in welchem der beiden Hyperkuben er in jedem der Zerlegungsschritte war. Außerdem unterscheiden sich benachbarte Knoten in dieser Benennung nur in genau einer dieser Koordinaten. Für jede Möglichkeit, eine Koordinate zu ändern existiert auch genau ein Nachbarknoten. Abbildung 1 stellt dies anhand der zwei- und dreidimensionalen Hyperkuben dar.

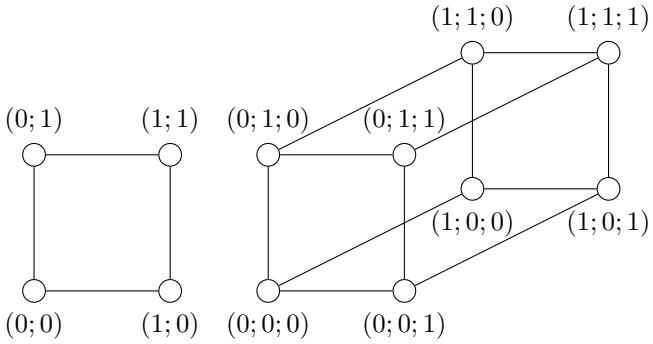



Abbildung 1: Zwei- und dreidimensionale Hyperkuben: Quadrat und Würfel

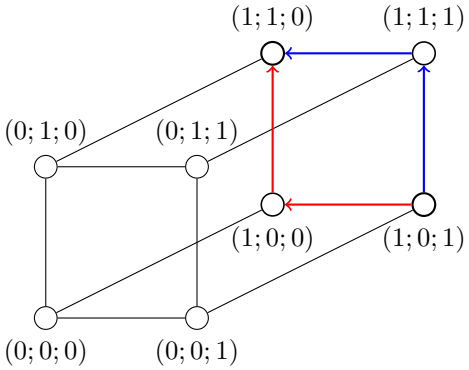



Abbildung 2: Kürzeste Pfade von $(1;0;1)$ zu $(1;1;0)$ im Würfel

Innerhalb von Hyperkuben können diese Koordinaten für das Routing benutzt werden. Bei benachbarten Knoten, zum Beispiel $(1;0;1)$ und $(1;0;0)$ ist leicht ersichtlich, welche Verbindung genutzt werden muss, nämlich die, die entlang der dritten Dimension die beiden Knoten verbindet. Für das Routing zwischen beliebigen Knoten enthalten kürzeste Pfade jeweils genau eine Verbindung entlang der Dimensionen bei denen die Koordinaten der Start- und Ziel-Knoten sich unterscheiden. Da jeder Knoten Verbindungen in den Richtungen aller Dimensionen hat, gibt es in einem vollständigem Hyperkubus für jede Reihenfolge der Dimensionen einen kürzesten Pfad. Zum Beispiel kann eine Nachricht von $(1;0;1)$ zu $(1;1;0)$, wie in Abbildung 2 dargestellt, sowohl zunächst über eine Verbindung in Richtung der dritten Dimension über $(1;0;0)$, als auch über eine Verbindung in Richtung der zweiten Dimension über $(1;1;1)$ zu $(1;1;0)$ gelangen, indem von $(1;0;0)$ bzw. $(1;1;1)$ die Verbindung in Richtung der noch jeweils anderen Dimension genutzt wird. Insgesamt gibt es also zwischen zwei Knoten, deren Koordinaten sich in $m$ Dimensionen unterscheiden $m!$ kürzeste Pfade der Länge $m$. In fehlerfreien Rechnernetzen könnte jeder dieser Pfade genutzt werden und die Nachricht würde ankommen. Da jedoch Verbindungen oder Knoten ausfallen können, muss ein fehlertoleranter Routing-Algorithmus dies berücksichtigen.

Der Algorithmus basiert auf Backtracking, er versucht also zunächst, Nachrichten an andere Knoten weiterzuleiten. Wenn diese Knoten alle von ihnen ausgehenden Pfade vergeblich versucht haben, wird die Nachricht wieder zu dem Knoten, von dem sie empfangen wurde, zurückgesandt, sodass dieser weitere Pfade ausprobieren kann. Insgesamt wird daher der Hyperkubus mittels Tiefensuche nach einem Pfad abgesucht. Damit die Pfade nicht unnötig lang werden, werden zunächst Verbindungen entlang der Dimensionen, in denen sich die Koordinaten des aktuellen und des Ziel-Knoten sich noch unterscheiden, ausprobiert. Wenn immer mindestens eine solche Verbindung benutzbar ist, kommt es niemals zum Backtracking, sodass dann ein kürzester Pfad benutzt wird.

In Abbildung 3 ist der Algorithmus dargestellt, der dieses Verhalten für einen Knoten, der eine Nachricht erhalten hat und sie weiterleiten muss, umsetzt. Nach der Prüfung, ob das Ziel bereits erreicht wurde, wird durch die erste Schleife zunächst versucht, die Nachricht über Verbindungen in den Richtungen, die den Stellen, bei denen sich die Koordinaten des aktuellen und des Ziel-Knotens unterscheiden, entsprechen, zuzustellen. Wenn die Nachricht über keine dieser Verbindungen abgeschickt werden konnte oder wenn die Nachricht von den erreichten Knoten wieder zurück geschickt wurde, werden anschließend durch die zweite Schleife alle anderen vom aktuellem Knoten ausgehende Verbindungen versucht. Falls die Nachricht auch über diese nicht zugestellt werden kann, wird die Nachricht entsprechend dem Backtracking-Prinzip zurück an den Knoten, von dem sie empfangen wurde zurück gesandt, damit dieser weitere Möglichkeiten, sie zuzustellen,

Eingabeparameter:

| | |
|---|---|
| $d$ | Koordinaten des Ziel-Knotens |
| $m$ | Inhalt der Nachricht |
| $visited$ | Menge bereits erreichter Knoten |
| $stack$ | Stack mit dem Pfad zurück zum Absender |

Algorithmus:

**if** $t = me$ **then**
  **exit** ▷ Nachricht ist angekommen
**else**
  $visited$.add($me$)
  $d := me \oplus t$
  **for** $i$ mit $d_i = 1$ **do**
    **if** $me \oplus e_i \notin visited \wedge me \oplus e_i$ erreichbar **then**
      $stack$.push($me$)
      send($me \oplus e_i, m, visited, stack$)
      **exit**
    **end if**
  **end for**
    ▷ Nutzung eines kürzesten Pfades fehlgeschlagen.
  **for** $i$ mit $me \oplus e_i \notin visited$ **do**
    **if** $me \oplus e_i$ erreichbar **then**
      $stack$.push($me$)
      send($d \oplus e_i, m, visited, stack$)
      **exit**
    **end if**
  **end for**
  $t := stack$.pop
  send($t, m, visited, stack$) ▷ Backtracking
**end if**

Abbildung 3: Algorithmus für das Routing, der in den einzelnen Knoten des Hyperkubus benutzt wird. $e_i$ repräsentiert den Vektor, bei dem nur das $i$-te Bit gesetzt ist. Mit $d_i$ wird auf das $i$-te Bit von $d$ zugegriffen.

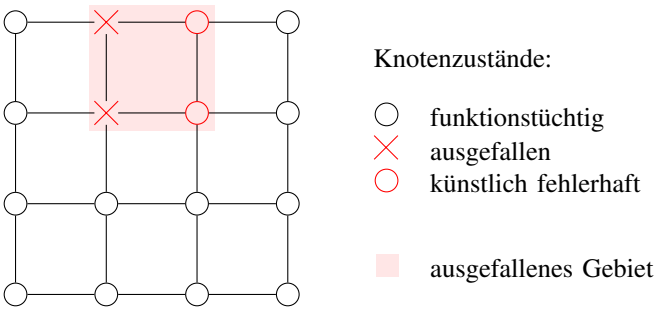



Abbildung 4: Ein Netz mit künstlich fehlerhaften Knoten

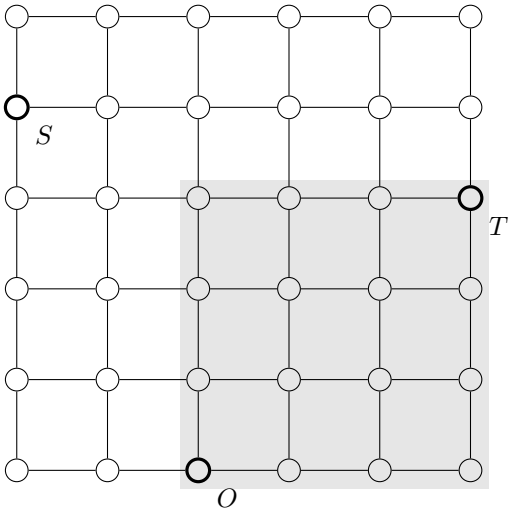



Abbildung 5: Ein Beispiel-Netz für die Betrachtung des Algorithmus mit Startknoten $S$, Zielknoten $T$, Ursprung $O$ und grau markierter Outbox

ausprobieren kann.

Da der Algorithmus das gesamte Netzwerk mittels Tiefensuche nach einem Pfad zum Ziel-Knoten durchsucht, wird die Nachricht immer dann, wenn ein geeigneter Pfad existiert, zugestellt. Im schlimmsten Fall wird die Nachricht von jedem Knoten des Netzwerkes behandelt, bevor sie ihr Ziel erreicht.

## IV. Ursprungsbasiertes Routing in Gittern

In [3] und [5] wird der im folgenden beschriebene Algorithmus vorgestellt, der auf gitterartigen Netzwerken arbeitet. Die Beschreibung wird hier anhand von zweidimensionalen, quadratischen Gittern erfolgen; der Algorithmus lässt sich jedoch auch auf Gitter höherer Dimensionen anwenden. Der Algorithmus setzt außerdem voraus, dass Regionen von Fehlern quadratische Form haben. Hierfür werden ggf., wie in Abbildung 4 gezeigt, Knoten als fehlerhaft gekennzeichnet, die korrekt funktionieren.

Der Algorithmus wählt außerdem einen Knoten als sogenannten Ursprung. Dieser wird für das Routing benötigt und muss die Eigenschaft haben, dass er in einer Zeile und Spalte liegt, in der keine Ausfälle aufgetreten sind. Solange es weniger als $n$ Ausfälle gibt, kann immer ein Ursprung gewählt werden; wenn es mehr Ausfälle gibt, kann oftmals trotzdem ein Ursprung gewählt werden, mit dem das Routing funktioniert, es gibt dann aber auch Fälle, in denen das Routing nicht mehr funktioniert. Das kleinste rechteckige Teilnetz, das den Ursprung und den Ziel-Knoten enthält, hat für den Algorithmus eine wichtige Funktion und wird im folgendem als Outbox bezeichnet. Für manche Knoten der Outbox wird im folgenden gezeigt werden, dass sie *sicher* sind. Aus der Analyse dieser sogenannten sicher-Eigenschaft folgt, dass von diesen Knoten der Weg zum Ziel-Knoten bekannt ist.

Das Routing lässt sich nun grob in drei grundlegende Teile zerteilen. Als erstes werden Verbindungen genutzt, die den Abstand zum Ursprung verringern. Sobald die Outbox erreicht wird, wird versucht, sichere Knoten zu erreichen; solange dies nicht möglich ist, wird weiter der Abstand zum Ursprung verringert. Sobald ein sicherer Knoten erreicht ist, wird von dort der Pfad zum Zielknoten genommen.

Es ist nun zu betrachten, wieso die einzelnen Teile des Algorithmus funktionieren. Dafür wird die Situation aus Abbildung 5 als Beispiel genommen. Es soll eine Nachricht von Knoten $S$ zu Knoten $T$ gesandt werden, als Ursprung wurde der Knoten $O$ gewählt. Im folgendem wird davon ausgegangen, dass die Lage von $S$ zu $O$ bzw. von $O$ zu $T$ so wie in der

Abbildung sind. Dies ist keine Einschränkung, da man die entsprechende Lage einfach durch Drehung bzw. Spiegelung herbeiführen kann.

Für die Analyse des Algorithmus müssen zunächst einige Dinge definiert werden:

**Definition.** Den Knoten seien, wie in Abbildung 4 Koordinaten zugeordnet, hierbei sei $x_k$ die Spalte und $y_k$ die Zeile, in der sich ein Knoten $k$ befindet. Die Länge eines kürzesten Pfades zwischen zwei Knoten $i$ und $o$ wird als Distanz $d(i, o)$ bezeichnet. Da hier die Taxi-Metrik anwendbar ist, ergibt sich:

$$d(i, o) := |x_i - x_o| + |y_i - y_o|$$

Das der erste Teil immer einen Pfad zum Ursprung findet, wird durch den folgenden Satz gezeigt.

**Satz 1.** *Es gibt von jedem nicht ausgefallenem Knoten $k$ einen Pfad der Länge $d(O, k)$ zum Ursprung.*

*Beweis:* Der Beweis erfolgt mittels Induktion über $d(O, k)$. Induktionsanker ist $d(O, k) = 0$: In diesem Fall ist $k$ bereits der Ursprung. In allen anderen Fällen ist die Induktionsvoraussetzung, dass es für alle nicht ausgefallenen Knoten $k'$ mit $d(O, k') < d(O, k)$ entsprechende Pfade gibt. Falls $x_k = x_O$ oder $y_k = y_O$ ist, wurde die Spalte bzw. Zeile, in der der Ursprung liegt, erreicht, diese sind frei von Ausfällen, sodass der Pfad aus den Verbindungen entlang dieser Spalte bzw. Zeile bis zum Ursprung besteht. Ansonsten sind die Knoten $(x_k; y_k - 1)$ und $(x_k + 1; y_k)$ Nachbarn von $k$ mit einer geringeren Distanz zum Ursprung. Mindestens einer dieser Nachbarn ist nicht ausgefallen, da ansonsten auch $k$ ausgefallen sein müsste, damit die ausgefallene Region quadratische Form hat. Dieser Nachbar hat laut Induktionsvoraussetzung einen Pfad zum Ursprung, sodass der Pfad von $k$ zum Ursprung aus der Verbindung zu diesem Nachbarn und dem Pfad des Nachbarn besteht. ■

Für den zweiten Teil des Algorithmus wird noch eine Definition benötigt:

**Definition.** Ein Knoten $k$ ist sicher, wenn er in der Outbox liegt und wenn es für jede Permutation fehlerhafter Knoten, bei der $k$ und $t$ nicht fehlerhaft sind, einen fehlerfreien Pfad zu $T$ gibt.

Für den zweiten und dritten Teil ist der folgende Satz notwendig:

**Satz 2.** *Alle Knoten $k$, die auf einer Diagonale durch $T$ liegen oder zu einem solchem Knoten benachbart sind, sind sicher.*

*Beweis:* Ein Knoten $k$ liegt auf einer Diagonale durch $T$, wenn $x_k - y_k = d$ mit $d := x_T - y_T$ (Eine zweite Diagonale wird durch $x_k + y_k$) beschrieben, diese wird jedoch im beispielhaftem Fall nicht benötigt, der Beweis lässt sich leicht daran anpassen.). Nachbarknoten von Diagonalelementen haben Differenzen mit den Werten $d \pm 1$.

Der Beweis erfolgt mittels Induktion über $d(T, k)$. Falls die Distanz 0 ist, ist $k = T$; offensichtlich ist $T$ sicher. Induktionsvorraussetzung ist, dass alle ausgefallenen Knoten $k'$, die die Bedingungen des Satzes erfüllen und $d(T, k') < d(T, k)$ haben, sicher sind.

Falls $x_k - y_k = d$ ist, erfüllen $(x_k; y_k + 1)$ und $(x_k + 1; y_k)$ die Induktionsvoraussetzung, sodass ein Pfad zu $T$ aus der Verbindung zu einem der Nachbarn und dessen Pfad gebildet werden kann, in Abbildung 6a sind diese Verbindungen rot eingezeichnet. Falls diese beide Knoten ausgefallen sind, muss auch $k$ ausgefallen sein, was für die sicher-Eigenschaft vernachlässigt werden kann. Falls einer der Knoten ausgefallen ist, ergibt sich die Situation in Abbildung 6b, in der offensichtlich über den nicht ausgefallenen Knoten ein Pfad konstruiert werden kann.

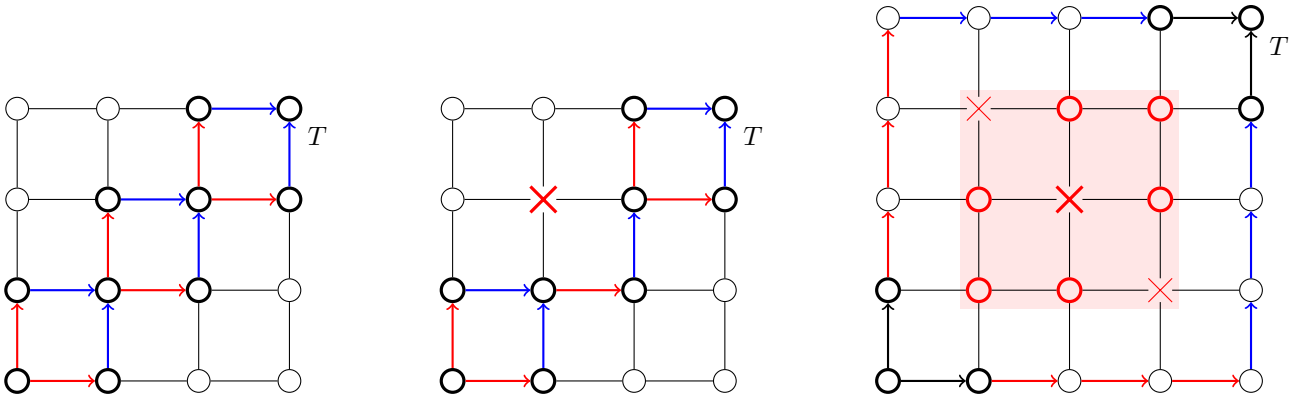
Es bleibt noch, dass $x_y - y_k = d \pm 1$; im folgendem wird $x_y - y_k = d + 1$ angenommen, der andere Fall ist analog. Falls $x_k(y_k + 1)$ nicht ausgefallen ist, kann der Pfad über diesen Knoten genommen werden, da dieser laut Induktionsvoraussetzung sicher ist und damit einen Pfad zu $T$ besitzt. Diese Verbindungen sind in Abbildung 6a blau dargestellt. Ansonsten geht der Pfad, wie in Abbildung 6c rot dargestellt, so lange nach rechts, bis ein Knoten $(x_k + \ell; y_k)$ erreicht wird, dessen Nachbar $(x_k + \ell; y_k + 1)$ nicht ausgefallen ist. Da alle ausgefallenen Regionen quadratisch sind, sind auch die darüber liegenden Knoten bis zu $(x_k + \ell; y_k + \ell)$ nicht ausgefallen. Da $(x_k + \ell) - (y_k + \ell) = x_k - y_k = d + 1$ ist, erfüllt dieser Knoten wieder die Induktionsvoraussetzung, sodass der Pfad, der in der Abbildung blau dargestellt ist, von diesem zu $T$ fortgeführt werden kann. Es wäre denkbar, dass der Pfad an $T$ vorbei geführt hat, in diesem Fall liegt $T$ jedoch im ausgefallenen Gebiet und wäre damit selbst ausgefallen; was ein Ausfallszenario darstellt, dass für die sicher-Eigenschaft nicht relevant ist. ■

Der zweite Teil endet immer damit, dass ein sicherer Knoten erreicht wird, da falls der Ursprung nicht sicher ist, zumindest in der Zeile oder der Spalte, in der der Ursprung liegt, ein sicherer Knoten liegt. Dieser muss erreichbar sein, da der Ursprung so gewählt wurde, dass die Zeile bzw. Spalte frei von Ausfällen ist.

Da im Beweis von Satz 2 die sicher-Eigenschaft durch Konstruktion eines Pfades zum Ziel-Knoten gezeigt wird, kann der dritte Teil des Algorithmus diese Konstruktion übernehmen, um den Pfad zum Zielknoten zu erhalten. Insgesamt erreicht die Nachricht den Ziel-Knoten genau dann, falls er nicht ausgefallen ist.

Der Algorithmus wurde am Beispiel angewandt. Abbildung 7 zeigt zwei mögliche Pfade. Der erste Teil des Algorithmus wählt für den blauen Pfad von $S$ ausgehend Verbindungen nach unten, da diese die Distanz zum Ursprung verringern. Erst mit dem Erreichen der Zeile, in der sich der Ursprung befindet, wird die Richtung geändert. Nun wird dieser Pfad nach rechts zum Ursprung fortgeführt. Da der Ursprung auch der erste Knoten der Outbox ist, der erreicht wird, endet damit der erste Teil des Algorithmus. Der zweite Teil des Algorithmus wird gar nicht angewandt, da der Ursprung auf der Diagonale durch $T$ liegt und daher sicher ist. Im dritten Teil kann der Pfad nach rechts oder nach oben fortgesetzt werden. Der Algorithmus hat die weiterhin die Richtung nach rechts gewählt. Der Pfad muss dann auch in den nachfolgenden Knoten nach rechts fortgesetzt werden, da die darüber liegenden Knoten ausgefallen sind. Erst in der Spalte, in der $T$ liegt kann die Richtung nach oben gewählt und damit $T$ erreicht werden.

Da der Algorithmus mehrere Verbindungen auswählen




(a) ohne ausgefallene Knoten (b) mit einem ausgefallenen Knoten (c) mit einem ausgefallenen Gebiet

Abbildung 6: Verschiedene Fehler-Situationen aus dem Beweis von Satz 2, die innerhalb der Outbox auftreten können. Die Knoten, die auf der Diagonale liegen, sind fett dargestellt.

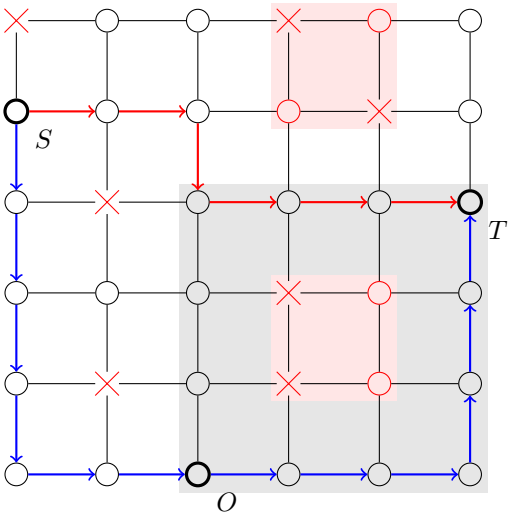



Abbildung 7: Das Beispiel-Netz mit ausgefallenen Bereichen und zwei durch den Algorithmus gewählte Pfade

kann, kann als erstes auch eine Verbindung nach rechts gewählt werden und so der rote Pfad begonnen werden. In diesem Fall wird nach einer weiteren Verbindung nach rechts die Spalte, in der der Ursprung liegt, erreicht. Der Pfad wird entlang dieser Spalte in Richtung des Ursprunges fortgesetzt. Da jedoch bereits nach der ersten Verbindung in dieser Richtung die Outbox erreicht wird, tritt dann bereits der zweite Teil des Algorithmus in Kraft. Da von dem erreichten Knoten ausgehend nach rechts mit dem Knoten $T$ ein sicherer Knoten zu erreichen ist, wird der Pfad in dieser Richtung fortgesetzt. Da $T$ auch das Ziel der Nachricht ist, endet der Algorithmus ohne das der dritte Teil ausgeführt werden musste.

Anhand dieser Pfade ist zu erkennen, dass die Pfade je nach Entscheidungen, die der Algorithmus trifft, unterschiedlich lang werden können. Da der blaue Pfad nicht durch den zweiten Teil des Algorithmus verkürzt wird, erreicht er wirklich den Ursprung und hat damit die maximal mögliche Länge $d(S,O)+d(O,T)=12$. Der rote Pfad verkürzt hingegen mit jeder gewählten Verbindung die Distanz zu $T$ und hat daher die minimale Distanz $d(S,T)=6$.

Bevor eine Nachricht verschickt werden kann, benötigt der Algorithmus Informationen. Einerseits muss dem Absender ein geeigneter Ursprung bekannt sein; andererseits benötigt der zweite Teil des Algorithmus auch das Wissen, wie weit er Nachrichten in die einzelnen Richtungen senden kann, ohne dass sie auf einen ausgefallenen Knoten treffen, um zu wissen ob ein sicherer Knoten erreicht werden kann. Indem entlang der Spalten und Zeilen Informationen ausgetauscht werden, ob in diesen Richtungen die jeweiligen Nachbarknoten erreichbar sind, können die Knoten diese Information einfach auf eine nicht zentral gesteuerte Art und Weise erhalten und aktualisieren. Falls ein Knoten feststellt, dass entlang aller vier Richtungen keine ausgefallenen Knoten liegen, ist er ein Kandidat für den Ursprung. Jeder Knoten muss eine Menge von Kandidaten für den Ursprung verwalten. Wenn ein Knoten feststellt, dass er ein Ursprungskandidat ist oder ein Nachbarknoten ihn über eine Änderung informiert, muss er seine Nachbarn über diese Änderung informieren. Dadurch können alle erreichbaren Knoten ihre Liste von Ursprungskandidaten aktualisieren und bei Versenden einer Nachricht einen Ursprung auswählen. Auch die quadratische Form ausgefallener Gebiete ergibt sich nicht einfach so. Zwar können Knoten teilweise erkennen, dass sie selbst in einem solchem Gebiet liegen, wenn Nachbarknoten ausgefallen sind; jedoch ist es für einzelne Knoten nicht überprüfbar, ob das gesamte Gebiet eine quadratische Form hat. Daher müssen Nachbarn ausgefallener Knoten ihre Nachbarn über diese Ausfälle informieren und auf geeignete Weise erkennen, ob das ausgefallene Gebiet quadratisch ist.

Es stellt sich die Frage, ob der Algorithmus verbessert werden kann, sodass er möglichst kurze Pfade wählt. Ein erste Erweiterung des Algorithmus ist daher, dass im erstem Teil bevorzugt Verbindungen gewählt werden sollen, bei denen nicht nur die Distanz zu $O$ sondern auch die Distanz zu $T$ verringert wird. Allerdings ergibt sich direkt beim Absenden der Nachricht von $S$ das Problem, dass sowohl der rote als auch der blaue Pfad aus Abbildung 7 die Distanz zu $S$ zunächst verringern. Für den blauen Pfad kann allerdings keine geeig-

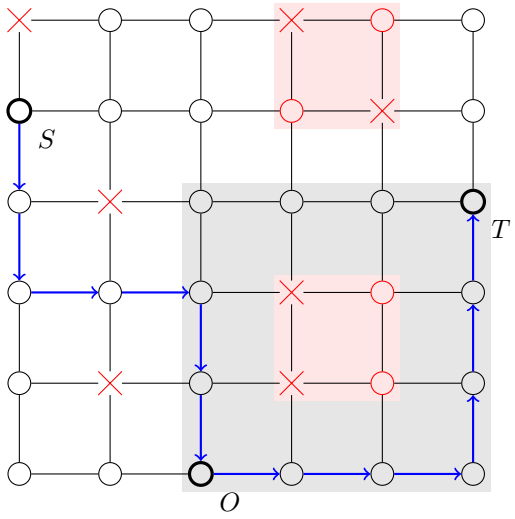



Abbildung 8: Das Beispiel-Netz mit ausgefallenen Bereichen und einem weiterem Pfad, bei dem versucht wurde, einen möglichst kurzen Pfad zu wählen

nete zweite Verbindung ausgewählt werden, da die Verbindung nach unten die Distanz zu $T$ erhöht und die Verbindung nach rechts nicht möglich ist. Wenn der Pfad trotzdem nach unten fortgesetzt wird, können anschließend Verbindungen nach rechts genommen werden, die die Distanz zu $T$ wieder verringern, bis die Outbox erreicht wird. Der zweite Teil des Algorithmus kann aufgrund des ausgefallenen Gebietes innerhalb der Outbox keinen sicheren Knoten erreichen und muss den Pfad bis zum Ursprung fortsetzen. Dieser in Abbildung 8 dargestellte Pfad verläuft damit wieder durch den Ursprung, sodass er wieder die Länge $d(S,O) + d(O,T) = 12$ hat. Diese Erweiterung des Algorithmus reicht also nicht aus, damit der Algorithmus immer kürzere Pfade wählt. Der Algorithmus würde weitere Informationen benötigen, um früh genug zu erkennen, ob ein Pfad, der zu Anfang die Distanz zu $T$ verkürzt auch wirklich kürzer ist. Dadurch würde jedoch jeder Knoten den gesamten Zustand des Netzes kennen, wodurch sich ein Algorithmus ergeben würde, der wesentlich von diesem Algorithmus abweicht, da kein Ursprung mehr benötigt wird, sondern direkt ein minimaler Pfad bestimmt und die Nachricht auf diesem verschickt werden kann.

## V. Fazit

Die drei vorgestellten Algorithmen sind grundsätzlich verschieden. Das STP stellt eine universell anwendbare Lösung bereit, die jedoch bzgl. des Zeitverhaltens problematisch ist und die Redundanzen nur zur Überbrückung von Ausfällen nutzt. Die anderen beiden Algorithmen deaktivieren hingegen keine Verbindungen, sodass alle fehlerfreien Verbindungen genutzt werden und dadurch in einigen Situationen ein höherer Nachrichtendurchsatz möglich ist. Diese beiden Algorithmen haben in jedem Knoten nur relativ wenig Informationen (Welche meiner Nachbarn sind nicht erreichbar? bzw. wie viele Knoten erreiche ich in jeder der Richtung und welche Knoten können als Ursprung genutzt werden?), sind dezentral und adaptiv. Daher sind sie besser für spezialisierte Netzwerke geeignet, bei denen dann auch eine geeignete Netzwerktopologie verwendet werden muss. Allerdings wird die Implementierung sehr wahrscheinlich nicht in der hier vorgestellten Form erfolgen. Beim Routing im Gitter mussten Knoten deaktiviert werden, um die quadratische Form ausgefallener Gebiete zu erzielen. Eine Implementierung wird vermutlich untersuchen, ob ein Knoten wirklich ausgefallen ist, bevor sie die Zustellung aufgibt, weil er aus Sicht des Algorithmus ausgefallen ist.

## Literatur

[1] “802.1d-1998 - IEEE Standard for Local Area Network MAC (Media Access Control) Bridges,” http://standards.ieee.org/findstds/standard/802.1D-1998.html.

[2] “802.1w-2001 - IEEE Standard for Information Technology - Telecommunications and Information Exchange Between Systems - Local and Metropolitan Area Networks - Common Specifications - Part 3: Media Access Control (MAC) Bridges: Amendment 2 - Rapid Reconfiguration,” http://standards.ieee.org/findstds/standard/802.1w-2001.html.

[3] I. Koren and C. M. Krishna, *Fault-Tolerant Systems*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[4] M.-S. Chen and K. Shin, “Depth-first search approach for fault-tolerant routing in hypercube multicomputers,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 2, pp. 152–159, 1990.

[5] R. Libeskind-Hadas and E. Brandt, “Origin-based fault-tolerant routing in the mesh,” in *First IEEE Symposium on High-Performance Computer Architecture. Proceedings.*, 1995, pp. 102–111.

# N-Version Programming als fehlertolerantes System

Carsten Krüger
Carl von Ossietzky Universität Oldenburg

## I. Einleitung/Motivation

Ein Konzept zum Erreichen von fehlertoleranter Systeme ist der Einsatz von Redundanz in Form von N-Version Programming(NVP). Dazu wird die Funktionalität des Systems $n$ mal implementiert. Diese einzelnen Teilimplementierungen werden Versionen genannt. Die Implementierung der einzelnen Versionen wird von verschiedenen Gruppen bearbeitet, die unabhängig voneinander arbeiten. Jedoch wird eine einheitliche Spezifikation verwendet und alle Versionen erhalten die gleichen Eingabeparameter. Die erzeugten Ergebnisse der einzelnen Versionen werden dann durch eine Vergleichssoftware ausgewertet und erzeugt eine Ausgabe für das System[Vgl. Abbildung 1].
Durch NVP ist es möglich, dass das System bei Implementierungsfehlern einzelner Versionen, trotzdem eine richtige Ausgabe produziert.

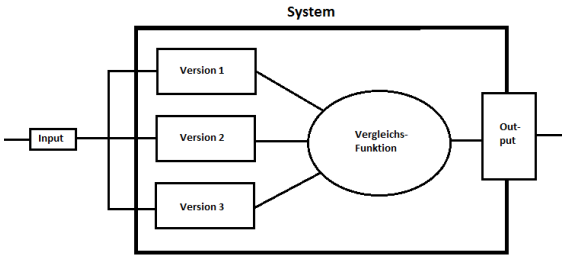



Abbildung 1. Aufbaustruktur eines Systems mit NVP [1]

Um das allgemeine Interesse an diesem Thema zu wecken, soll folgendes Beispiel für ein Softwareunglück zeigen, was durch NVP hätte verhindert werden können.

*Ariane 5*

Die Rakete Ariane 5 wurde am 4.Juni 1996 gestartet. Dabei sprengte sie sich und die mitgeführten vier Cluster-Satelliten bereits nach 36,7 Sekunden. Ursache dieses kurzen Fluges lag in der Software, die teilweise aus dem Vorgängermodell der Ariane 4 übernommen wurden. Diese übernommene Software entsprach jedoch nicht den nötigen Anforderungen. Die Ariane 5 erzeugte eine 64-Bit Gleitkommazahl, die dann in ein 16-Bit integer Variable umgewandelt wurde. Dies führte zu einem Overflow, der dafür sorgte, dass das Lenksystem zusammenbrach und die Ariane 5 ihre Selbstzerstörung aktiviert. Ausgehend von den 370 Millionen US-Doller Schaden, gilt dies als teuerster Softwarefehler der Geschichte [2][3].
Wäre das System durch NVP erstellt worden, dann hätte es die Möglichkeit gegeben, dass andere Versionen keinen Overflow erzeugt hätten, wodurch es den Absturz nicht gegeben hätte.

## II. Funktionsaufbau

Bei dem Funktionsaufbau des zu realisierenden Systems, sollen die einzelnen Versionen so unabhängig wie möglich voneinander implementiert werden. So entstehen weniger gemeinsame Programmierfehler.Die Spezifikation jedoch wird nur einmal für das ganze System erstellt.

*Spezifikation*

Bei der Spezifikation müssen alle Anforderungen eindeutig definiert werden, denn sämtliche Beschreibungsfehler in der Spezifikation können ggf. dazu führen, dass Implementierungsfehler in den Versionen entstehen.
Eine Spezifikation muss die implementierte Funktion. Die Art der Eingabe und Ausgabe muss festgelegt werden. Zusätzlich müssen verwendete Entwicklungsumgebungen und Programmiersprachen angegeben werden.

Um die Thematik besser verfolgen zu können, wird ein Beispiel präsentiert, welches über das ganze Thema hinweg behandelt wird. Das System, soll die Nullstellenberechnung einer Polynomfunktion 2. Grades durchführen.

*Hier Beispiel: Spezifikation Polynomfunktion*

Das System muss in der Lage sein eine Nullstelle einer Polynomfunktion vom Grad 2 zu berechnen. Dabei soll eine möglichst hohe Genauigkeit erreicht werden. Es werden zwei Funktionen bereitgestellt, die Erste entspricht der Polynomfunktion und die Zweite deren Ableitung.

1) Funktion: double funk(double x)
   Eingabe: x entspricht dem Wert auf der x-Achse.
   Ausgabe: Wert an der Stelle x der Polynomfunktion.

2) Funktion: double fd(double x)
   Eingabe: x entspricht dem Wert auf der x-Achse.
   Ausgabe: Wert an der Stelle x der abgeleiteten Polynomfunktion.

Als Eingabeparameter der Versionen kann ein Intervall[links,rechts] übergeben werden. Dabei muss das Intervall für die Polynomfunktion geeignet gewählt werden. Unter geeignet gewählt ist zu verstehen, dass der kleinere

Wert der unteren Intervallgrenze entspricht und der größere Wert der oberen Intervallgrenze entspricht. Außerdem muss das Intervall nahe einer Nullstelle liegen, sodass eindeutig ist, welche Nullstelle berechnet wird.
Die Programmierung muss in der Entwicklungsumgebung Eclipse geschrieben werden. Als Ausgabe der einzelnen Versionen wird ein double Wert zurückgegeben.

Mit dieser Spezifikation wird nun begonnen die zu implementierende Funktion zu entwickeln. Dies geschieht jedoch nicht nur einmal, sondern es werden $n$ verschiedene Gruppen damit beauftragt, jeweils unabhängig von einander die Versionen zu implementieren. Dies führt dazu, dass mögliche Implementierungsfehler nicht bei allen Versionen auftreten.
Ich werde das Beispiel jetzt um drei Versionen der Nullstellenberechnung erweitern und anschließend eine Vergleichsfunktion dazu vorstellen.

*Hier Beispiel: Version 1*

Diese Version benutzt das binäre Teilungsverfahren in einem vorgegebenen Intervall.

```
public static double Binärteilung(double links, double rechts) {
    double wertLinks = funk(links);
    double wertRechts = funk(rechts);
    double mitte = 0;
    for (int i = 0; i <= 30; i++) {
        mitte = (rechts + links) / 2;
        double wertMitte = funk(mitte);
        if (wertMitte == 0)
            break;
        if (wertMitte * wertLinks > 0) {
            links = mitte;
            wertLinks = wertMitte;
        } else {
            rechts = mitte;
            wertRechts = wertMitte;
        }
    }
    return mitte;
}
```

Abbildung 2. Binärteilung

*Hier Beispiel: Version 2*

Diese Version benutzt, ähnlich wie beim binären Teilungsverfahren, ein Intervall und verringert dieses immer wieder, jedoch wird es nicht genau in der Mitte halbiert.

```
public static double Intervallteilung(double x1, double x2) {
    double wertLinks = funk(x1);
    double wertRechts = funk(x2);
    double mitte = 0;
    for (int i = 0; i <= 30; i++) {
        mitte = x1
                + (x2 - x1)
                * (Math.abs(funk(x1)) / (Math.abs(funk(x1)) + Math
                        .abs(funk(x2))));
        double wertMitte = funk(mitte);
        if (wertMitte == 0)
            break;
        if (wertMitte * wertLinks > 0) {
            x1 = mitte;
            wertLinks = wertMitte;
        } else {
            x2 = mitte;
            wertRechts = wertMitte;
        }
    }
    return mitte;
}
```

Abbildung 3. Intervallteilung

*Hier Beispiel: Version 3*

Bei diesem Verfahren wird das Newton Raphson Verfahren zur Nullstellenberechnung verwendet.

```
84    public static double NewtonRaphson() {
85        double x = 0;
86        for (int i = 0; i <= 30; i++) {
87            double funk = funk(x);
88            if (funk == 0)
89                break;
90            double ablt = fd(x);
91            double diff = funk(x) / ablt;
92            x -= diff;
93        }
94        return x;
95    }
```

Abbildung 4. Newton Raphson, Wolfgang P.Kowalk [4]

*Hier Beispiel: Vergleichsfunktion*

Nach der Implementierung muss eine Vergleichsfunktion erstellt werden. Diese entscheidet welche Ausgaben der Versionen als richtig gewertet werden und als Ausgabe erscheinen. Es gibt verschiedene Verfahren, nach denen die Vergleichsfunktion die Ausgaben bewertet. Bei dem Beispiel aus [Abb.5], wird ein Relativtest mit indeterministischen Prozessen verwendet. Dies bedeutet, dass Ergebnisse eine gewisse Ungenauigkeit haben, sodass Vergleiche nicht durch eine Gleichheitsabfrage, sondern durch eine Ähnlichkeitsfrage erfolgen. [5].
Bei diesem Relativtest wird versucht, Ausreißer zu erkennen und vom Ergebnis auszunehmen. Dabei werden die Ergebnisse der einzelnen Versionen miteinander verglichen. Anschließend wird das arithmetische Mittel [7] aus den zwei Ergebnissen mit der geringsten Differenz zueinander gebildet und als Ausgabe des Systems wiedergegeben. Dieses Verfahren ähnelt dem winsorisiertem Mittel [8].

```
static double LINKS = 0;
static double RECHTS = 2;

public static void main(String[] args) {
    double Nullstelle;
    double erg1 = Binärteilung(LINKS, RECHTS);
    double erg2 = Intervallteilung(LINKS, RECHTS);
    double erg3 = NewtonRaphson();
    System.out.println("Erg1: "+erg1);
    System.out.println("Erg2: "+erg2);
    System.out.println("Erg3: "+erg3);
    //Hier wird überprüft welche beiden Ergebnisse am nächsten
    //aneinander liegen. Die summer/2 wird dann ausgegeben.
    if (Math.abs(erg1 - erg2) < Math.abs(erg1 - erg3)
            && Math.abs(erg1 - erg2) < Math.abs(erg2 - erg3)) {
        Nullstelle = (erg1 + erg2) / 2;

    } else {
        if (Math.abs(erg1 - erg3) < Math.abs(erg1 - erg2)
                && Math.abs(erg1 - erg3) < Math.abs(erg2 - erg3)) {
            Nullstelle = (erg1 + erg3) / 2;
        } else {
            Nullstelle = (erg2 + erg3) / 2;
        }
    }
    System.out.println("Ergebnis: "+Nullstelle);

}

public static double funk(double x) {
    return x * x + 3 * x - 4.5;
}
public static double fd(double x) {
    return 2 * x + 3;
```

Abbildung 5. Vergleichsfunktion

## III. Entwicklungsansätze

Jetzt wird darauf eingegangen, welche verschiedenen Entwicklungsansätze es gibt, um das System zuverlässiger zu machen.

*Verschiedene Entwicklungsgruppen*

Nun werden verschiedene Entwicklungsgruppen für die einzelnen Versionen eingeteilt. Dabei sollte besonders auf die Zusammenstellung der Entwicklungsgruppen geachtet werden. Durch verschiedene Herangehensweisen können Implementierungsfehler verhindert werden.

*Programmiersprache*

Hierbei unterscheidet man vor allem, ob höhere Programmiersprachen wie Java/C++ oder maschienennahe Sprachen, wie Assembler genutzt werden. Dabei haben beide Sprachklassen ihre Vor- und Nachteile, die abgewogen werden müssen. Bei höheren Programmiersprachen sind die individuellen Programmierfehler geringer. Dafür entstehen jedoch häufiger gemeinsame Programmierfehler aufgrund ähnlicher Konzepte der Programmiersprachen. Bei den maschienennahen Sprachen ist es genau anders herum.
Eine Variation an Programmiersprachen verringert die Anzahl der gemeinsamen Implementierungsfehler.

*Entwicklungsumgebung*

Es ist sinnvoll verschiedene Entwicklungsumgebungen zu verwenden, wie Eclipse oder CodeLite. Diese einzelnen Entwicklungsumgebungen werden auch jeweils unterschiedlich realisiert, sodass gemeinsame Fehler verringert werden können.

*Compiler und Übersetzungsprogramme*

Ähnlich wie bei der Entwicklungsumgebung, ist es zu der Verringerung von gemeinsamen Programmierfehlern wichtig, unterschiedliche Compiler und Übersetzungsprogramme zu verwenden. Denn die Übersetzung durch verschiedene Compiler kann zu unterschiedlichen Instruktionen führen.

*Fazit*

Bezogen auf das Beispiel wurden die drei vorgestellten Versionen, durch zwei verschiedene Personen implementiert. Version1 und Version2 wurden von mir selbst implementiert. Die Version 3 stammt aus der Literatur von Wolfgang P.Kowalk [4].
Es kann als Fazit festgestellt werden, dass das Beispielprogramm nicht gegen Compiler-Fehler oder Interpreter-Fehler geschützt ist. Auch wurde nicht jede Version unabhängig voneinander implementiert. Es wurde eine gemeinsame Entwicklungsumgebung und Programmiersprache verwendet, sodass auch hier noch gemeinsame Programmierfehler entstehen können.

## IV. Fehlerreduzierung

N-Version Programming zielt darauf ab, die Implementierungsfehler in einem System zu verringern. Es ist nicht möglich, Imlementierungsfehler vollständig zu vermeiden, da Fehler auch in allen Versionen gleichzeitig auftreten können. Dennoch wird das System durch die Fehlerreduzierung zuverlässiger, sodass es sinnvoll ist NVP zu verwenden.
Wenn man die Fehlerreduzierung genauer betrachtet, lassen sich die behobenen Fehler in verschiedene Kategorien einteilen.

*Programmierfehler*

Bei Programmierfehlern in einzelnen Versionen, die zu falschen Berechnungen oder Rundungsfehlern führen, muss es nicht unbedingt zu einem Systemversagen kommen. Mit Hilfe des N-Version Programming können Berechnungsfehler von einzelnen Versionen, durch das Vergleichsprogramm erkannt werden.

*Programmfehler*

Auch Fehler, die nicht durch die Programmierung, sondern durch das Programm entstehen können abgefangen werden. Dies sind Fehler in Betriebssystemen oder fehlerhafte Treiber. Aber auch Programmfehler von Programmiersprachen können vermieden werden. Die hier benannten Fehler können nur dann verhindert werden, wenn sie nicht in allen Versionen auftreten.

*Hardwarefehler*

Eine weitere Möglichkeit, um ein System zuverlässiger zu machen, ist der Einsatz von verschiedener Hardware. Wobei Hardware als technische Komponente verstanden wird, auf dem eine Version die Berechnungen durchführt.
Dadurch kann bei Ausfällen von einzelnen Hardware-Komponenten, ein Systemversagen vermieden werden.

*Fehlerquellen in NVP*

Es gibt auch einige Fehler, die nicht von N-Version verhindert werden können. Dies sind beispielsweise Beschreibungsfehler in der Spezifikation. Des Weiteren können auch falsche Eingabeparameter dazu führen, dass fehlerhafte Berechnungen in allen Versionen auftreten. Aber auch die Vergleichsfunktion kann eine mögliche Fehlerquelle, denn bei falscher Auswertung der Berechnungen, würden alle Ausgaben falsch in Relation gesetzt werden und die Systemausgabe wäre möglicherweise nicht korrekt.

*Fazit*

In dem Beispiel werden alle Versionen auf der gleichen Hardware-Komponente angewendet. Daher kann ein Systemversagen bei einem Hardwareausfall nicht verhindert werden.

Dadurch das die Berechnungen von drei Versionen durchgeführt werden, können Implementierungsfehler verhindert werden. Jedoch darf höchstens eine Version gleichzeitig einen Implementierungsfehler aufweisen.

## V. Arten von NVP

N-Version Programming ist ein Fehlertoleranz Konzept, von dem verschiedene Ansätze existieren. Dabei unterscheidet man vor allem:

1) **Zeit:** Wie häufig eine Version die Berechnung wiederholt.
2) **Hardware:** Versionen laufen auf verschiedenen technischen Komponenten.
3) **Software:** Die Anzahl der Versionen.

Diese verschiedenen Ansätze können nun in Varianten kombiniert werden, so lassen sich Systeme zuverlässiger implementieren.

### *Variante 1*

Es existiert eine Version, die auf einer Hardware-Plattform läuft. Diese Version wird $n$ mal durchgeführt und die Ergebnisse werden durch eine Vergleichsfunktion ausgewertet. Diese Variante ist eine Abwandlung von NVP, da keine unabhängigen Versionen erstellt werden. Stattdessen werden die Berechnungen häufiger durchgeführt, um sie anschließend zu vergleichen. Damit schützt man sich gegen einmalige Programmfehler.
Typ: (Zeit, Hardware, Software)=(N,1,1)

### *Variante 2*

Es existieren $n$ Versionen, die zusammen auf einer Hardware-Plattform laufen. Diese $n$ Versionen werden jeweils einmalig durchgeführt. Anschließend wird die Vergleichsfunktion genutzt, um die verschiedenen Ergebnisse auszuwerten. So schützt man sich gegen Implementierungsfehler einzelner Versionen.
Typ: (Zeit, Hardware, Software)=(1,1,N)

### *Variante 3*

Bei dieser Variante werden $n$ Versionen auf $n$ Hardware-Plattformen realisiert. Dabei wird jede Version einmalig durchgeführt und anschließend durch die Vergleichsfunktion ausgewertet. Diese Variante hat den Vorteil, dass beim Ausfall einzelner Hardware-Komponenten, nicht alle Versionen davon betroffen sind und daher ein Systemversagen verhindert werden kann.
Typ: (Zeit, Hardware, Software)=(1,N,N)

Aus diesen drei Variationen können auch weitere Kombinationen erstellt werden. Jedoch muss beachtet werden, dass sich die Kosten und die Entwicklungszeit erhöhen.

### *Hier Beispiel: Variante*

Im Beispiel wurde Variante 2 verwendet, da jede der drei Version einmal gestartet wird und alle auf den gleichen Hardware-Komponenten laufen.
Typ: (Zeit, Hardware, Software)=(1,3,1)

## VI. Probleme

Durch NVP können auch einige Probleme entstehen. Diese Probleme werden jetzt genauer betrachtet.

### *Synchronisation*

Wenn verschiedene Versionen und Hardware-Plattformen verwendet werden, müssen die verschiedenen Ausgaben alle zur Vergleichsfunktion gelangen. Dieser Vorgang ist nicht einfach zu realisieren. Oft entstehen die Probleme bei der Übertragung von Ergebnissen einer Hardware-Plattform auf eine andere.

### *Anzahl Versionen gegen steigende Kosten*

Für jede weitere Version erhöhen sich die Kosten, ausgenommen von der Spezifikation und der Vergleichsfunktion. Es wird darüber hinaus ein hohes Maß an Fachkräften benötigt, denn jede Version soll unabhängig voneinander entstehen.
Dennoch steigen die Gesamtkosten nicht linear zu der Anzahl an Versionen, da die Kosten für Spezifikation, Planung, Hardware etc. einen Großteil ausmachen.

### *Testen*

Ein weiteres Problem ist das Testen aller Versionen. Das Testen an sich, ist eine sehr komplexe Aufgabe und bedarf einer langen Bearbeitungszeit. Da es sich bei N-Version Programming allerdings um n-Versionen handelt, die jeweils unterschiedliche Strukturen beinhalten, wird das Testen schwierig und zeitintensiv. Somit steigert sich der Testaufwand fast linear zu der Anzahl entwickelter Versionen.

### *Hier Beispiel: Tests*

Polynomfunktion: $x^2 + 3x - 4,5$

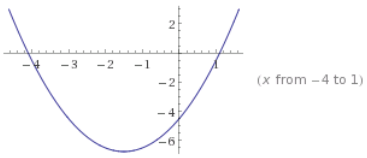



Abbildung 6. Polynomfunktion $x^2 + 3x - 4,5$

Um einen Test darzustellen, werden verschiedene Eingabeparameter verwendet und die dabei entstehenden Ausgaben betrachtet.

Eingabe: Intervall[0,2]
Ausgabe:
Ergebnis Version1: 1.098076212219894
Ergebnis Version2: 1.098076211353316
Ergebnis Version3: 1.098076211353316
Systemergebnis: 1.098076211353316

Alle Versionen führen zu den gleichen Ergebnissen, nur die Genauigkeit unterscheidet sich.

Eingabe: Intervall[0,1]
Ausgabe:
Ergebnis Version1: 0.9999999995343387
Ergebnis Version2: 0.9999999999531494
Ergebnis Version3: 1.098076211353316
Systemergebnis: 0.9999999997437441

Da zwischen dem Intervall keine Nullstelle existiert, berechnen die ersten zwei Versionen einen falschen Wert, so ist das Systemergebnis falsch.

Eingabe: Intervall[-6,0]
Ausgabe:
Ergebnis Version1: -4.098076212219894
Ergebnis Version2: -4.098076211353316
Ergebnis Version3: 1.098076211353316
Systemergebnis: -4.098076211786605

Die Polynomfunktion besitzt zwei Nullstellen, sodass bei diesem Intervall die ersten zwei Versionen die negative Nullstelle berechnen und die 3. Version die positive Nullstelle berechnet.
Anhand dieser Tests wird schnell erkenntlich, dass das gewählte Intervall eine wichtige Rolle spielt. Daraus folgt, dass die gegebene Spezifikation nicht eindeutig genug war, sodass teilweise falsche Systemergebnisse ausgegeben werden.

## VII. Stochastisches Beispiel

Als Nächstes soll ein stochastisches Beispiel verdeutlichen, wie sich die Fehlerwahrscheinlichkeit durch Verwenden von NVP verringert.
In der nachfolgenden Tabelle sind zwei Versionen, mit ihren einzelnen Fehlerwahrscheinlichkeiten angegeben. [6].

| Version | S1 | S2 |
|---|---|---|
| V1 | 0,010 | 0,001 |
| V2 | 0,020 | 0,003 |

Die stochastische Fehlerwahrscheinlichkeit der einzelnen Versionen ergibt sich aus:

**Probe(V1 fails)** = 0,01*0,5 + 0,001*0,5 = 0,0055
**Probe(V2 fails)** = 0,02*0,5 + 0,003*0,5 = 0,0115

Die Stochastische Vereinigung der beiden Versionen ergibt dann:

Probe(V1 fails)*Probe(V2 fails) $= 0,0055 * 0,0115 = 6,33 * 10^{-}5$

Probe(V1,V2 beide fails)$= 0,01 * 0,02 * 0,5 + 0,001 * 0,003 * 0,5 = 1,02 * 10^{-}4$

Es ist deutlich zu sehen, dass aus stochastischer Sicht die Fehlerwahrscheinlichkeit enorm verringert wird.

## VIII. Fazit

Das Fehlertoleranz-Konzept N-Version Programming ist ein sehr umfangreiches Thema. Es bietet eine Vielzahl an Möglichkeiten, um Systeme zuverlässiger zu realisieren.
Die Implementierung der Vergleichsfunktion wird häufig als triviale Aufgabe in den Hintergrund gestellt, dabei ist die Vergleichsfunktion ein wichtiger Bestandteil von NVP. Deshalb muss das Entscheidungsverfahren der Vergleichsfunktion sehr sorgfältig ausgewählt werden.
Ein weiterer Punkt, den man bei diesem Verfahren nicht aus den Augen verlieren darf, ist die Anzahl der Versionen.Die Qualität der einzelnen Versionen darf nicht bei der Erhöhung der Anzahl an Versionen vernachlässigt werden.
Sinnvoll ist meiner Meinung nach NVP lediglich in groß angelegten Projekten. Hingegen der Einsatz in alltäglichen Computeranwendungen darauf verzichten kann. Gute Einsatzgebiete sind beispielsweise Programme, die für Menschenleben verantwortlich sind.

## IX. Quellen/Links

### Literatur


[1] Israel Koren and L.Mani Krishna*Fault-Tolerant Systems*, 2007
[2] Software Desaster *http://www4.in.tum.de/lehre/seminare/ps/WS0203/desaster/Riedl-Arianne5-Ausarbeitung-27-11-02.pdf* (am 01. Juni 2013)
[3] Ariane 5 *http://de.wikipedia.org/wiki/Ariane_5* (am 19. Juni.2013)
[4] Wolfgang P.Kowalk *System Modell Programm, Vom GOTO zur objekt-orientierten Programmierung* 1996, in Oldenburg
[5] Universität Karlsruhe, Fehlerdiagnose*http://goethe.ira.uka.de/seminare/ftv/diagnose/#ToC17* (am 16. Juni 2013)
[6] Laura L. Pullum *Software Fault Tolerance/Techniques and implentation* , 2001
[7] Fahrmeir, Ludwig Pigeot, Iris ; Tutz, Gerhard: *Statistik : der Weg zur Datenanalyse, mit 162 Abbildungen und 25 Tabellen. 4. verb. Aufl.. Heidelberg: Springer-Verlag GmbH, 2003.*
[8] Winsorisiertes Mittel*http://www.uni-protokolle.de/Lexikon/Mittelwert.html* ( 18. Juni 2013)

# Fehlertoleranz in VLSI Schaltkreisen

Clemens Büse, Fachbereich II, Informatik
Universität Oldenburg

***Zusammenfassung*—In dieser Ausarbeitung geht es um die Fehlertoleranz in „monolithischen Halbleiterschaltung mit sehr hohem Integrationsgrad“ (Very-large-scale integration (VLSI)). Es wird ein Überblick über Fehler, Fehlerarten sowie über Fehlerumgehungs/- Fehlertolerierungsstrategien gegeben.**

## I. Einleitung

In der heutigen Zeit, in der alles kleiner wird, macht die Industrie keinen Halt vor der Chiptechnologie. Auch Chips könnten immer kleiner hergestellt werden. Die ersten Computer arbeiteten noch mit Röhren. Diese waren mehrere Zentimeter groß. Um 1954 wurde der Transistor mit Silizium als Schaltelement erfunden und ersetzte die Röhren in der Computerindustrie, die im Vergleich zu den Röhren nur wenige Zentimeter bis Millimeter groß waren.
Der entscheidende Durchbruch gelang Jack Kilby, der es 1958 schaffte, ein Flip-Flop auf einem Wafer herzustellen. Hierfür verwendete er das Verfahren der Lithographie. In mehreren Durchgängen wurden auf dem Siliziumwafer Masken aufgebracht und entweder wurde ein Material abgeschieden oder durch ätzen entfernt. 1970 wurden mittels dieser Technik 2000 Transistoren auf einem Chip gefertigt. Ein Jahr später, am 15. November 1971, wurde von Intel der erste Mikroprozessor vorgestellt, der in Serie produziert wurde. Hierbei handelte es sich um den Intel 4004. Dieser besaß 2250 Transistoren, welche ca. 10 µm an Größe aufwiesen (vgl. [3]).

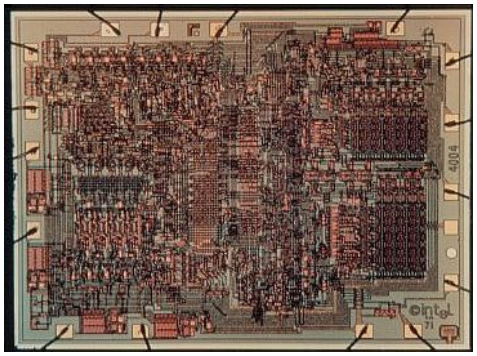



Abbildung 1. Intel 4004, bestehend aus 2250 Transistoren ([2])

Seitdem gab es alle paar Jahre einen neuen Technologieknoten. Ein solcher Technologieknoten wird immer dann festgelegt, wenn sich die Abmessung des Chips bei gleichbleibendem Schaltplan halbiert hat. Dieses konnte bisher durch neue Fertigungstechnologien oder ihre Optimierung erreicht werden. Derzeit forscht die Industrie daran, Prozessoren mit einer Strukturbreite von 10-5 nm ($10^{-8}$-5*$10^{-9}$ m) herzustellen (vgl. [5] [4]). Zum Vergleich: Das menschliche Haar ist ca. 100 µm ($10^{-4}$ m) und ein Atom 0.1 nm ($10^{-11}$ m) dick. Bei diesen minimalen Stärken der Bauteile wird deren Fehlertoleranz immer wichtiger.

## II. Fehlerarten

Es gibt verschiedene Arten von Fehlern, die bei der Chipproduktion auftreten können. Zum einen gibt es „großflächige“, zum anderen „punktuelle Fehler“.

- großflächige Fehler
  Diese Fehler können im Herstellungsprozess des Wafers auftreten. Eventuelle Verunreinigungen im Siliciumdioxid können dazu beitragen, dass Lithographietechniken nicht optimal durchgeführt werden können. Defekte oder unreine Masken, die für den Lithographie Prozess verwendet werden oder auch fehlerhaftes ätzen oder abscheiden, sind Fehlerquellen, die zu großflächigen Fehlern auf dem Wafer führen können, wobei ein größerer Teil des Wafers davon betroffen sein muss. Eventuelle Beschädigungen des Wafer beim Transport fallen ebenfalls in diese Kategorie (vgl. [1]).
- punktuelle Fehler
  Sie treten eher sporadisch auf und verteilen sich unregelmäßig auf der Chipoberfläche. Ihren Ursprung können sie - wie die großflächigen Fehler auch - von mangelhaften Masken, fehlerhaften ätzen oder unbrauchbaren Abscheidungen haben. Punktuelle Fehler entstehen jedoch eher durch Materialfehler oder schadhafte Partikel, die sich während des Lithographieverfahrens auf der Oberfläche des Wafers absetzen und somit eine erfolgreiche Lithographie verhindern(vgl. [1]).

Beide Fehler verringern die Ausbeute an Chips, die man aus einem Wafer gewinnen kann. Großflächige Fehler können durch eine Optimierung oder eine verbesserte Handhabung des Wafers während des Herstellungsprozesses weitestgehend minimiert werden. Schwieriger ist es, die unregelmäßig auftretenden punktuellen Fehler zu vermeiden. Diese sind zudem schwerwiegender, da sie nicht so schnell erkennbar sind, wie großflächige Fehler. Außerdem steigt die Anzahl punktueller Fehler gegenläufig zur Chipgröße. Dies erscheint logisch. Denn je kleiner die Strukturen, desto wahrscheinlicher ist es, dass ein Fehler in der Herstellung geschieht, obwohl sich ebenfalls die Herstellungstechnik und Herstellungsmethoden ständig im Wandel befinden (vgl. [4][1]).

Punktuelle Fehler lassen sich in verschiedene Fehlerklassen einteilen. Hauptsächlich wird unterschieden zwischen:

- Intralayer (Oberfläche) Fehler sind Mängel, die sich nur auf eine Ebene beziehen, wie zum Beispiel fehlerhafte

Strukturen (eindimensional). Ein Beispiel ist in Abbildung 2 (a) zu sehen. Hier wurde mehr Material als gewollt entfernt. Gegenteiliges gilt bei (b): Hier wurde zu viel Material abgeschieden, was ebenfalls zu Fehlern führen kann. Aber auch Strukturfehler, Fehler im Silicium oder in dem abzuscheidenden Material sowie auch fehlerhaftes Anbringen (falsche Ausrichtung) einer Maske sind mögliche Ursachen für eine solche Störung.

- Beim Interlayer (Zwischenschicht) Fehler gibt es einen Defekt zwischen unterschiedlichen Schichten, welche durch die Lithurgrapieprozedur aufgebaut werden. Eventuell ist die dielektrische Zwischenschicht nicht sauber abgeschieden oder auch ein VIA (Kontaktloch) ist nicht korrekt aufgebaut worden. Dies geschieht häufig durch lokale Konterminationen der Schichten, z.B. durch Staubpartikel.

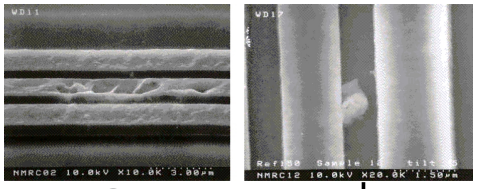



Abbildung 2. (a) Fehlerhafte Leiterbahn, (b) Anlagerung an einer Leiterbahn [6]

Nicht alle Fehler machen einen Chip unbrauchbar. Die Strukturänderungen in Abbildung 2 (b) müssen nicht zwangsläufig zu einem Leitungsschluss oder zu einem Leitungsbruch wie in Abbildung 2 (a) führen. Hierbei kommt es auch darauf an, welchem Zweck die Struktur dient. Die in Abbildung 2 (b) erkennbare Ablagerung könnte den Chip empfindlicher gegenüber elektromagnetischen Wellen machen, wobei bei (a) Komplikationen mit benötigtem Strom auftreten könnten, da die Leiterbahn nicht vollständig ist. Ähnlich wie bei einer Schmelzsicherung kann die Leiterbahn schmelzen und somit den Stromfluss verhindern (vgl. [1]).

## III. Fehlerwahrscheinlichkeit

Die Fehlerwahrscheinlichkeit (POF, probability of failure) hängt von verschiedenen Faktoren ab. Zum einen vom Fehlertypen, zum anderen von der Größe des betroffenen Gebietes. Nur weil ein Chip einen Fehler aufweist, muss dies nicht zwangsläufig zu einer Fehlfunktion des Chips führen.
Einige Methoden zur Fehlerwahrscheinlichkeitsberechnung arbeiten nach dem geometrischen Verfahren. Hier wird zuerst der kritische Bereich ($A_i^{(c)}$) berechnet.

$$A_i^{(c)} = \int_{x_0}^{x_m} A_i^{(c)} f_d(x) dx \tag{1}$$

$x_m$ steht für die maximale Defektgröße und $x_0$ für die Auflösung, die beim Lithographieprozess verwendet wird. Zudem wird für die Berechnung noch das $f_d(x)$ benötigt, welches durch Formel (2) abgebildet wird.

$$f_d(x) = \begin{cases} kx^{-p} & x_0 \leq x \leq x_m \\ 0 & sonst \end{cases} \tag{2}$$

Wobei $k$ für eine Normierungskonstante

$$k = \frac{(p-1)x_0^{p-1}x_m^{p-1}}{x_m^{p-1} - x_0^{p-1}} \tag{3}$$

steht.

Abbildung 3 zeigt eine geometrische Herangehensweise. Sollte die Fehlergröße (x) kleiner sein, als die Breite einer Leiterbahn (w), fällt der Fehler nicht ins Gewicht, da die Leiterbahn nicht durchtrennt ist (Fehler a und b) (vgl. Formel (1). (s. [1])

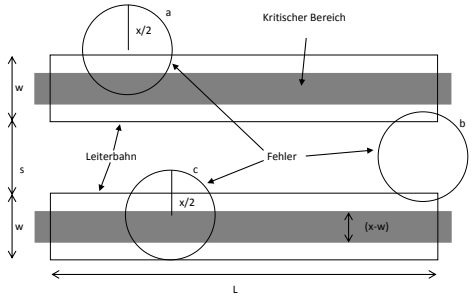



Abbildung 3. Schematische Darstellung für die geometrische Herangehensweise an die Fehlerwahrscheinlichkeit (nach [1])

## IV. Grundlegende Ausbeutungsmodelle

Auch wenn angestrebt wird Fehler möglichst gering zu halten, kann ihr Vorkommen nicht vollständig ausgeschlossen werden. Daher wird stetig versucht, das Maximum an fehlerfreien Chips aus einem Wafer zu bekommen. Bei den Ausbeutungsmodellen wird unterschieden zwischen Chips, die keine Redundanz bzw. welchen die Redundanz in ihrem Schaltplan integriert haben. Allgemein lässt sich der Ausbeutungserfolg durch folgende Formel berechnen(s. [1]):

$$Y = \frac{\text{Anzahl der funktionieren Chips auf einem Wafer}}{\text{Anzahl aller Chips auf einem Wafer}} \tag{4}$$

### A. Ausbeutungsmodelle ohne Redundanz

Die in der Literatur am häufigsten anzutreffenden Ausbeutungsmodelle sind das Poisson und das gemischte Poisson Modell [1]. Sei zum Beispiel $\lambda$ die durchschnittliche Fehleranzahl auf einem Chip und der Chip wird in $n$ unabhängige Abschnitte aufgeteilt. Jeder Abschnitt hat dann eine Fehlerwahrscheinlichkeit von $\frac{\lambda}{n}$. Aus diesen Schlussfolgerungen kann man binominal schließen, dass sich die Anzahl der Fehler auf einem Chip ($Y$) folgendermaßen darstellt(vgl. [1]):

$$Y = \binom{n}{k} \left(\frac{\lambda}{k}\right)^k \left(1 - \frac{\lambda}{n}\right)^{n-k} \tag{5}$$

Lässt man nun $n$ gegen $\infty$ laufen (Aufteilung in immer kleinere Abschnitte), so erhält man die Poisson-Verteilung, welche sich durch die folgende Formel darstellen lässt.

$$Y = \frac{e^{-\lambda}\lambda^k}{k!} \tag{6}$$

In der Realität zeigt sich, dass die Poisson-Verteilung zu pessimistisch ist. Denn in der Praxis lag die Ausbeute höher,

als die errechnete Voraussagen. Dies erklärt sich durch die erhöhte Wahrscheinlichkeit, dass ein aufgetretener Fehler weitere Fehler nach sich zieht. Dies ist in Abbildung 4 visualisiert.

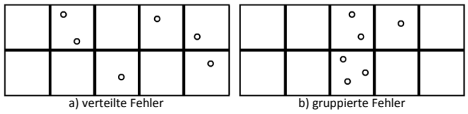



Abbildung 4. Der Effekt von Fehlergruppierung auf die Ausbeute (nach [1])

Beim linken Wafer gibt es eine Ausbeute von 50 %, wohingegen der rechte Wafer eine Ausbeute von 70 % aufweist. Dies nur, weil sich die Fehler gruppiert haben.
In den vorangegangen Überlegungen wurde von gleichmäßig verteilten Fehlern $\lambda$ ausgegangen, da sich die Fehler gruppieren musste anders an das Problem herangegangen werden. Daher gab es Überlegungen $\lambda$ nicht durch eine Konstante, sondern durch einen Zufallswert abzubilden. Mit diesen Überlegungen, erhält man:

$$Y = \int_0^\infty \frac{e^{-\lambda}\lambda^k}{k!} f_\Lambda(\lambda) d\lambda \tag{7}$$

Die Funktion $f_\Lambda(\lambda)d\lambda$ stellt hierbei die Dichte der Fehler da. In der Praxis werden für diese Funktion nur Häufigkeitsverteilungen der Gamma-Verteilungen, Logarithmische-Normalverteilungen und Inverse Gauß-Verteilungen benutzt. Für alle eingesetzten Dichtefunktionen muss gelten (s. [1]):

$$\int_0^\infty f_\Lambda(\lambda)d\lambda = 1, \qquad E(\Lambda) = \int_0^\infty \lambda f_\Lambda(\lambda)d\lambda = \lambda \tag{8}$$

### *B. Ausbeutungsmodelle mit Redundanz*

Hier werden Verfahren beschrieben, bei denen in den Chips Redundanzen schon mit eingearbeitet wurden. Das bedeutet, dass bei der Planung des Chips mit dem Auftreten von Fehlern gerechnet und sich eine Fehlerumgehung überlegt wurde. Dies ist besonders wichtig, wenn es um große Chips geht, auf denen kein Produktionsfehler sein darf. Durch das Hinzufügen von Reserven auf dem Chip, wird zwar der Chip wieder größer und es passen weniger Chips auf einen Wafer, dafür ist die Wahrscheinlichkeit einen fehlerfreien Chip zu erhalten höher. Je höher der Wert $Y$ der Formel (9) desto optimaler ist die Integration von redundanten Bereichen auf dem Chip (vgl. [1]).

$$Y = \frac{\text{Gebiete auf dem Chip ohne Redundanz}}{\text{Gebiete auf dem Chip mit Redundanz}} \tag{9}$$

Eine Berechnung der Ausbeute gestaltet sich hierbei schwieriger als bei Chips ohne Redundanz.

#### *1) Speicher Arrays mit Redundanz:*

Techniken der Fehlertoleranz wurden erfolgreich in die Designs von Speicherbausteinen integriert. Aufgrund ihres gleichmäßigen Aufbaus wurde das Hinzufügen von Redundanzen stark vereinfacht. So wurden zum Beispiel zusätzliche Reihen und Spalten hinzugefügt, welche aktiviert werden können, falls eine Hauptreihe oder Spalte einen Fehler aufweist, oder es wurde mit Fehlerkorrekturcodes (z.B. Gray-Code) gearbeitet. Diese Techniken werden von vielen Halbleiterherstellern eingesetzt. Bei frühen Prototypen konnte die Ausbeute an fehlerfreien Chips, die aus einem Wafer gewonnen wurden, um das bis zu 30-Fache erhöht werden. Bei schon in Serie produzierten Chips konnte durch dieses Verfahren eine Steigerung um das 1,5 bis 3-Fache erreicht werden.
Die am häufigsten eingesetzte Technik beinhaltet das Hinzufügen zusätzlicher Speicherreihen bzw. Speicherspalten. Wie in Abbildung 5 zu sehen ist, besteht das Speichermodul lediglich aus einem sechs mal sechs Feld. Es wurden aber zwecks der Redundanz um jeweils zwei Reihen und Spalten erweitert (s. [1]).

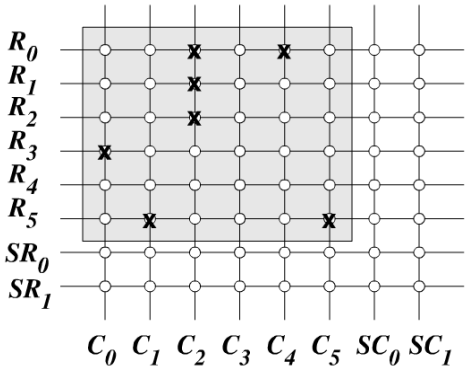



Abbildung 5. Sechs mal sechs Speicherarray mit zwei zusätzlichen Reihen und Spalten, zudem sieben defekte Zellen [1]

In den Anfängen dieser Technik bediente man sich Schmelzsicherungen wobei Bits in Reihen gespeichert wurden. War eine Reihe defekt, so wurde sie durch das Auslösen der Schmelzsicherung deaktiviert. Durch den Wegfall dieser Reihe musste an anderer Stelle eine neue Reihe hinzugefügt werden. Dies wurde mittels einem programmierbaren Decoder und wiederum schmelzbaren Verbindungen gelöst.
Die ersten Entwürfe, die zusätzliche Reihen und Spalten beinhalteten, arbeiteten mit Lasersicherungen. Hierbei wurden Leiterbahnen mittels speziellen Lasergeräten durchtrennt und Verbindungen zu Reservereihen hergestellt. Die Lasersicherungen nahmen nicht nur viel Platz in Anspruch, sondern benötigten zudem ein spezielles Equipment, um die Sicherungen auszulösen. Da dies jedoch alles sehr umständlich war, wurden die Lasersicherungen durch CMOS Sicherungen ersetzt. Diese mussten sehr zuverlässig und fehlerfrei funktionieren. Sollte nun ein Fehler in der CMOS Schaltung auftreten, die zur Fehlerminimierung gedacht war, hätte man mit an Sicherheit grenzender Wahrscheinlichkeit einen defekten Chip. Dies wurde von den Speicher-Designern durch selbst korrigierende Fehlerkodes gelöst (vgl. [1]).
Natürlich mussten erst einmal die defekten Zeilen und Spalten identifiziert werden. Hierfür wurde ein Selbsttest (BIST [Built-In Self-Sesting]) in den Chip integriert. Durch diesen werden Testmuster erzeugt und deren Ergebnisse kontrolliert. Defekte Speicherstellen können so identifiziert

werden (s. [1] [6]).
Durch den Selbsttest ist jetzt zwar bekannt, welche Speicherstellen defekt sind, aber es gibt mehrere Möglichkeiten die Fehler zu eliminieren. Es kann entweder die Spalte oder die Reihe getauscht werden, wobei ein zufälliges Tauschen nicht zwingend zielführend ist. Dies wird mittels der Abbildung 5 veranschaulicht (s. [1]).

Würde man zuerst die Reihen ersetzen, hätte man die Auswahl zwischen den Reihen $R_{0-3}$ und $R_5$. Wählt man $R_0$ und $R_1$ für die zu ersetzenden Reihen, verbleiben noch vier Fehler. Diese können nicht durch die zwei verbleibenden Spalten eliminiert werden und der Speicherbaustein bliebe defekt.
Würden aber andere Reihen ausgewählt werden, wie beispielsweise die Reihen mit den meisten Fehlern ($R_0$ und $R_5$), so verblieben noch Fehler in den Spalten $C_0$ und $C_2$. Diese könnten durch die redundanten Spalten $SC_0$ und $SC_1$ ersetzt werden und der Speicherbaustein hätte keinen Fehler mehr.
Es handelt sich hier um ein NP-Vollständiges Problem, für welches es zur Zeit keine Lösung durch einen Algorithmus mit polynomialer Komplexität gibt. Es gibt zwar die Möglichkeit, das Problem zum Beispiel dadurch zu lösen, dass nur noch Reihen ersetzt werden dürfen. Hier hat man dann aber den Nachteil, dass - wie in Abbildung 5 zu sehen ist - drei Reihen ersetzt werden müssten, wobei es praktikabler wäre nur Spalte $C_2$ zu ersetzt. Hinzu kommt noch, dass es in der Praxis immer häufiger vorkommt, dass ganze Reihen oder Spalten fehlerhaft sind. Je nachdem welcher Algorithmus zur Verwendung implementiert wurde, nur Reihen oder Spalten, gibt es keine Lösung.
Um das Problem möglichst einfach zu halten, geht man in zwei Schritten vor. Zuerst wird analysiert, in welcher Reihe oder Spalte die meisten Fehler vorliegen. Ist die Anzahl der Defekte höher als die Anzahl der zur Verfügung stehenden Ersatzreihen oder Ersatzspalten, so muss diese ersetzt werden. Eine solche Spalte wäre zum Beispiel $C_2$. Diese enthält drei Fehler und es gibt zwei Ersatzspalten. Danach müsste wieder überprüft werden, welche Fehler noch ersetzt werden müssen. Als Nächstes würde der Algorithmus die Reihe $R_5$ auswählen, da diese zwei Fehler hat und es noch zwei Ersatzreihen gibt. Dies würde so lange fortgeführt werden, bis entweder alle Fehler beseitigt oder keine Reservereihen und Spalten mehr zur Verfügung stehen (s. [1]).
Auch, wenn die Ausbeute durch den Algorithmus erhöht werden kann, so gibt es keine Garantie, dass alle Chips eines Wafers am Ende fehlerfrei sind. Gerade in den frühen Phasen eines neuen Technologieknotens sind die Fehler besonders hoch. Diese fehlerhaften Chips werden nicht unbedingt vernichtet. Es gibt Hersteller, die diese bedingt funktionsfähigen Chips vertreiben. Bei diesen Chips funktionieren, trotz Belegung aller Redundanzen, nicht alle Speicherbausteine.
Ein bekanntes Beispiel für Speicherbausteine sind die Caches bei einem Prozessor. Dort gibt es eigentlich immer mehrere schnelle Speicher, die direkt im Prozessorchip integriert sind und zur Zwischenpufferung dienen.

Eine Optimierung des Verfahrens kann noch durch das Hinzufügen von Fehlerkorrekturcodes erreicht werden. Als Beispiel dient der Aufbau eines 16-Mb-DRAM-Chips. Dieser enthält vier voneinander unabhängige Subarrays mit jeweils 16 redundanten Bitleitungen und 24 redundanten Wortleitungen. Den 128 Datenbits wurden neun Prüfbits hinzugefügt. So konnte eine Struktur geschaffen werden, welche Einzelbitfehler ausgleicht. Da ein Fehler aber selten einzeln auftritt, wurden noch je acht benachbarte Bits zu acht separaten Wörtern zusammengefasst.
Hierbei wurde festgestellt, dass die Kombinationen dieser Strategien mit redundanten Reihen und Spalten mehr Ausbeute erbrachte, als jede Optimierung für sich allein. Die Fehlerkorrekturcodes sind eher geeignet für die Anwendung gegen Einzelfehler, wobei redundante Reihen und Spalten optimal gegen mehrere Defekte eingesetzt werden können. Besonders bei großen Speichern wird dies verwendet, um gegen äußere Einflüsse (beispielsweise ein Elektromagnetischer Impuls oder eine Spannungsspitze ein oder mehrere Bit beeinflusst) stabil zu sein. Zudem wurde Speicher, aus Performanzgründen, immer in kleinere Speicherbausteine aufgeteilt und in Untergruppen zusammengefügt. Mit dem Verfahren der redundanten Reihen und Spalten bleiben - aufgrund der Fehlergruppierung - einige Reserven ungenutzt. Daher wurden die lokalen redundanten Reihen und Spalten globalisiert. Hierbei wird wiederum mehr Platz für die programmierbaren Sicherungen benötigt, da diese nun ein größeres Areal bedienen müssen.
Bei der Konstruktion eines 1-Gbit-DRAM Chips wurde auf andere Herangehensweisen genutzt. Der Entwurf verwendet eine geringere Anzahl redundanter Reihen und Spalten, welche nicht global, sondern wie in den ersten Überlegungen, lokal angeordnet waren. Um nun nochmals eine erhöhte Fehlertoleranz zu erhalten, wurde jede Untergruppe - mit einer Größe von 256 MB - so produziert, dass vier unterschiedliche Speicherbausteine entstanden. Einige wurden nicht - wie üblich - mit festen Verbindungen hergestellt, um einen flexiblen Anschluss gewährleisten zu können. Damit dies durchgeführt werden konnte hätte die Chipfläche um zwei Prozent vergrößert werden müssen. Um dies zu vermeiden wurde die Reihenredundanz ausgeschlossen. Die Spaltenredundanz hingegen wurde weiterhin angewendet, um Fehler zu vermeiden (s. [1])

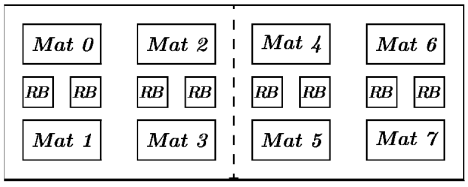



Abbildung 6. Dargestellt ist ein 1-Gb Chip aufgeteilt in acht Bereiche zu je 128 Mbits (Mat 0 bis Mat 7) und acht redundanten Blöcken zu je 1 Mbit (RB) [1].

Ein anderer Ansatz zur Fehlertoleranz kombiniert die Redundanz der Reihen und Spalten mit mehreren redundanten Un-

tergruppen. Diese Untergruppen sollen Bereiche eines Chips ersetzen, welche selbst nach der Anwendung der Reihen- und Spaltenredundanz nicht fehlerfrei sind. Auch diese Technik wurde verwendet, um einen 1-Gbit-DRAM Chip herzustellen. Hierbei wurde der Chip in acht Bereiche zu je 128 Mbit aufgeteilt. Für jeden dieser Bereiche ist ein Redundanzblock mit jeweils einem Mbit vorhanden, wie in Abbildung 6 zu sehen. Jeder ein Mbit Block besteht wiederum aus vier Standard 256 Kbit Speicherbausteinen. Zudem hat der Redundanzblock acht eigene redundante Reihen und vier redundante Spalten (siehe Abbildung 7). Dies erhöht die Wahrscheinlichkeit, dass der redundante Block keinen Fehler aufweist und verwendet werden kann, um einen Chip-Defekt zu verhindern (s. [1]).

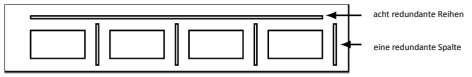



Abbildung 7. Der Aufbau eines Blocks, bestehend aus vier 256-Kbit Speicherbausteinen und acht redundanten Reihen, sowie vier redundanten Spalten (s.[1]).

Die 128 Mbit Bereiche bestehen aus 512 Standard Speicherbausteinen, mit jeweils 256 Kbit. Zudem hat jeder der acht Bereiche jeweils 32 redundante Reihen und Spalten, wobei zu beachten ist, dass diese nicht alle global sind. Vier Reihen werden jeweils 16-Mbit und acht Spalten jeweils 32-Mbit vom Hauptchip zugeordnet (vgl. [1]).

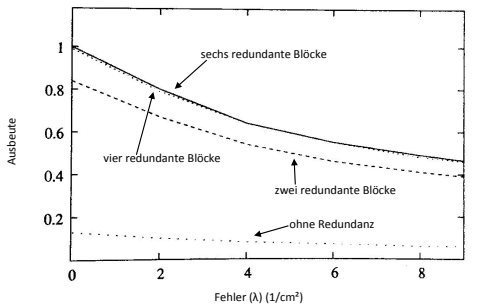



Abbildung 8. Ausbeutungsberechnung für verschiedene Techniken der Redundanz pro halbem Chip. Wobei angenommen wird, dass der Chip-Fehler bei einer Wahrscheinlichkeit von $5 * 10^{-4}$ liegt (vgl. [1]).

Diese neue Art der Aufteilung der Redundanzen lässt sich mit dem Standardverfahren vergleichen. Dadurch aber, dass sich die Redundanzen teilweise überdecken wird eine weitaus höhere Ausbeute erzielt. Dies ist graphisch in Abbildung 8 dargestellt. Die zwei Prozent mehr Platz die für redundante Blöcke benötigt wurde, lohnten sich wenn man bedenkt, welche höhere Ausbeute man erreichen konnte (zwei redundante Blöcke). Beim Beispiel mit vier redundanten Blöcken wird die Ausbeute noch einmal erhöht, wobei eine erneute Erhöhung der Redundanz keine signifikate Änderung aufzeigt.

*2) Logische Schaltungen:*
Im Gegensatz zu Speicherbausteinen werden nur wenige logische Schaltungen redundant konstruiert. Da logische Schaltungen nicht wie Speicherbausteine eine gleichmäßige Struktur aufweisen gibt, es als Redundanztechnik nur das Verfahren der Vielfachheit. Auch das Testen der Logikbausteine gestaltet sich als schwierig, denn es müssen alle möglichen Testmuster geprüft werden. Dies ist von der Anzahl der ein- und Ausgänge des Logikbausteins abhängig. Eine Vorgehensweise wird durch Abbildung 9 dargestellt (vgl. [1], [6]).

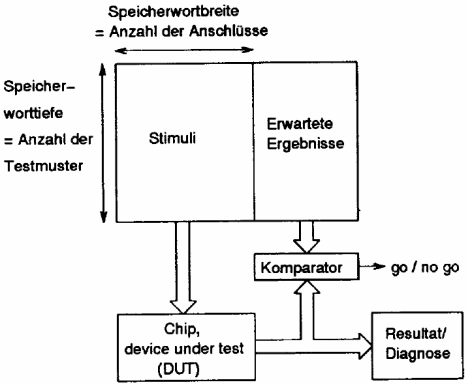



Abbildung 9. Aufbau eines Testautomaten für Logikbausteine (s. [6]).

Dem Logikbaustein wird vom Testautomat ein Muster vorgegeben. Der Ausgang des Chips wird mit dem zu erwartenden Ergebnis verglichen. Sollten die Ergebnisse nicht übereinstimmen, dann liegt beim Logikbaustein ein Defekt vor. Für eine kleine Anzahl an Eingängen am Logikchip können die Testmuster relativ schnell überprüft werden. Mit zunehmender Anzahl der Eingänge steigt die Überprüfungszeit exponentiell. Bei 30 Eingängen und einer Testgeschwindigkeit von einem Testmuster pro µs, dauert der Test eines solchen Logikbausteines ca. 18 Minuten. Bei 40 Eingängen beträgt die Testzeit schon 12,73 Tage [7].
Aus diesen Gründen wird bei Logikbausteinen das Verfahren der Vielflachheit angewandt. Zu diesem Zweck wird der Chip einfach mehrmals eingebaut und gleich beschaltet. Bei einer Verdoppelung der Hardware, kann das Ergebnis zweier Logikbausteine verglichen werden. Sollten die Ergebnisse nicht übereinstimmen, so hat man bei der verdoppelten Methode nur die Möglichkeit eines Abbruchs. Sollten aber eine ungerade Anzahl von Logikbausteinen die Berechnung durchführen, könnte mit dem Mehrheitsergebnis weitergearbeitet werden (s.[6]).

*3) Ändern des Grundrisses:*
Bisher hatte der Grundriss eines Chips keinen Einfluss auf die Ausbeute. Dies gilt aber nur für kleine Chips. Bei Chips die eine Grundfläche von zwei oder mehr $cm^2$ haben, gilt dies nicht. Solche Chips werden in der Regel aus verschiedenen Komponenten zusammengesetzt, wobei jede Komponente eigene Fehlerwahrscheinlichkeiten und eigene Redundanztechniken beinhaltet. Wird so ein Chip durch gruppierte Fehler beeinflusst, kann ihre Lage auf dem Chip sehr wohl einen Einfluss auf die Ausbeute eines Wafers haben (s. [1]).
Der in Abbildung 10 a) dargestellte Chip, besteht aus vier Modulen ($M_{1-4}$). Der Chip selbst beinhaltet keinerlei Fehlertoleranz und es werden alle vier Module benötigt, um einen ordnungsgemäßen Betrieb es Chips gewährleisten zu

können. Für diesen Grundriss gibt es 4!=24 mögliche Anordnungsmöglichkeiten. Wenn Drehungen und Spiegelungen vernachlässigt werden, bleiben noch drei Möglichkeiten, wie die vier Module im Grundriss angeordnet werden können (Abbildung 10 a, b, c) (s. [1]).
Unter der Voraussetzung, dass die Fehlerverteilung mittelmäßig ist und die einzelnen Blöcke unterschiedlich stark von Fehlern beeinflusst werden ( $M_2$ ist unempfindlicher oder gleich empfindlich gegen Fehler als $M_1$, $M_3$ ist unempfindlicher oder gleich empfindlich gegen Fehler als $M_2$u.s.w.), kann errechnet werden, das Grundriss a) und c) in Abbildung 10 zu einer höheren Ausbeute führt als Grundriss b). Dabei errechnet man bei Grundriss a) die höchste Ausbeute (vgl.[1]). Dies hätte auch intuitiv gelöst werden können, denn beim Grundriss a) sind die Module die am empfindlichsten gegenüber Fehlern sind nahe zusammen (vgl. [1]).

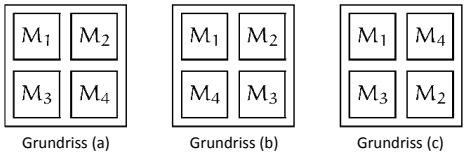



Abbildung 10. Drei Grundrisse eines vier mal vier Bausteins (vgl. [1]).

Anders verhält es sich bei einem Chip mit drei mal drei Modulen wie es in Abbildung 11 veranschaulicht wird. Auch in diesem Beispiel muss gelten, dass die Fehlerunempfindlichkeit mit dem Index steigt $M_{1-9}$. Hier lässt sich kein optimaler Grundriss festlegen. Es kann jedoch angenommen werden, dass das Modul mit der höchsten Fehlerunempfindlichkeit mittig und die Module mit den höchsten Fehlerempfindlichkeit in den Ecken zu Platzieren sind. Diese Vorgehensweise verringert das Risiko, das ein Fehler-Cluster mehrere benachbarte Module auf einem Chip beschädigt. Wird die Platzierung der Module nach dieser Vorgehensweise durchgeführt, könnte das ergebnis so aussehen wie in Abbildung 11 b) (vgl. [1]).

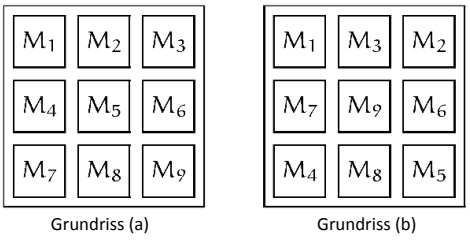



Abbildung 11. Zwei Grundrisse eines drei mal drei Bausteins ([1]).

Juni 23, 2013

## V. Fazit

Das Erkennen und Korrigieren von Fehlern in „monolithischen Halbleiterschaltung mit sehr hohem Integrationsgrad“ wird auch künftig die Wissenschaft und Industrie beschäftigen. Dadurch können hohe kosten eingespart werden. In der Abbildung 12 ist sind die Kosten eines fehlerhaften Chips - von der Fertigung bis zum Endprodukt - dargestellt.

| | |
|---|---|
| Wafer | 0,01 - 0,10 $ |
| Gehäuster Chip | 0,10 - 1 $ |
| Platine | 1 - 10 $ |
| System | 10 - 100 $ |
| Feld (im Einsatz) | 100 - 1000 $ |

Abbildung 12. Ungefähre Kosten eines festgestellten Fehlers in den unterschiedlichen Produktionsstufen (vgl. [6]).

Die Industrie versucht natürlich zu verhindern, dass fehlerhafte Chips in den Handel gelangen, was nicht immer verhindert werden kann. So geschehen beim Chiphersteller Intel: 2011 wurden von Intel Chipsatze der „Sandy Bridge“-Prozessoren zurückgerufen. Hierbei führte ein Fehler dazu, dass mit einem Leistungsabfall an betroffenen SATA-Controllern gerechnet werden musste. Die Kosten für die Rückrufaktion wurden auf 700 Millionen Dollar geschätzt [8]. Dieses Beispiel zeigt auf, das es wichtig ist Chipfehler möglichst früh im Fertigungsprozess zu beheben. Hierfür sind neue Herangehensweisen und Ideen bzw. Technologien notwendig, um die Chipproduktion effektiver zu gestalten.Zudem wird es - zu mindestens solange es keine Begrenzung gibt - immer wieder neue Technologieknoten und somit neue Herstellungsverfahren und -techniken geben, die möglicherweise andere Fehlertoleranztechniken benötigen. Halbleiterhersteller versuchen derzeit Chips zu produzieren, bei denen die kleinsten Teile nur noch wenige Nanometer groß sind [5].
Interessant wird es dann, wenn die Dielektischen schichten und die Abstände einzelner Bauteile nur noch Atomdicke betragen. Schon jetzt haben Halbleiterhersteller Probleme mit Leckströmen. Wie werden Hersteller mit diesen Problem umgehen? Um ein leistungsfähigeres System zu entwickeln, wird derzeit die Anzahl der beinhaltenden Prozessoren erhöht. Quad-Core (Vierkernprozessoren) oder Octa-Core (Achtkernprozessoren) sind entwickelt worden und im Handel erhältlich. Dies macht den Eindruck, dass die Grenze der Transistoren erreicht ist. Wird es in Zukunft möglich sein Chips zu erstellen, welche aus Teilchen gebaut werden, die kleiner als ein Atom sind? Neue technische Entwicklungen werden mit hoher Wahrscheinlichkeit nicht auf Fehlertoleranztechniken verzichten können, sodass eine stetige Forschung notwendig ist.

## Literatur


[1] Israel Koren and C. Mani Krishna, *Fault-tolerant Systems*, Chapter 8, Morgan Kaufmann, 2007
[2] Bernd Leitenberger, *Computergeschichte(n): Die ersten Jahre des PC*, erste Auflage, Books on Demand GmbH, Norderstedt, 2012.
[3] http://www.cpu-galaxy.at/CPU/Intel%20CPU/3002-8008/Intel%204004%20Section.htm [17.06.2013]
[4] Philipp Laube, *Halbleitertechnologie von A bis Z*, http://www.halbleiter.org/downloads/pdf/1/ [18.06.2013]
[5] Anton Shilov, *Intel Discloses Fab Transition Plans*, xbitlabs.com, http://www.xbitlabs.com/news/other/display/20120513115821_Intel_Begins_Work_on_7nm_5nm_Process_Technologies.html [18.06.2013]
[6] Prof. Dr. Dirk Timmermann, Vorlesung „Hochintegrierte Systeme I“, Universität-Rostock, http://www.imd.uni-rostock.de/lehre/vlsi_i/ [18.06.2013]
[7] http://www.wolframalpha.com/input/?i=2%5E30+*+1%C2%B5s+%3D+hour [23.06.2013]
[8] Matthias Wellendorf, http://www.tomshardware.de/Intel-Ruckruf-Chipsatz-H67-P67,news-245291.html [23.06.2013]

# Selbststabilisierung

Jan Steffen Becker
Universität Oldenburg
E-Mail: jan.steffen.becker@uni-oldenburg.de

***Zusammenfassung*—In diesem Paper werden selbststabilisierende Algorithmen als Fehlertoleranzkonzept sowie eine allgemeine Definition (nicht-maskierender) Fehlertoleranz vorgestellt. Anhand zweier Beispiele wird gezeigt, wie Selbststabilisierung mittels einer Variant- und einer Lyapunov-Funktion nachgewiesen werden kann. Anschließend wird an denselben Beispielen vorgeführt, wie selbststabilisierende Algorithmen kombiniert werden können, um aus bekannten Algorithmen neue selbststabilisierende Algorithmen zu gewinnen, deren selbststabilisierende Eigenschaften nicht noch einmal nachgewiesen werden müssen. Zuletzt wird ein kurzer Ausblick auf aktuelle Entwicklungen gegeben.**

***Index Terms*—Selbststabilisierung, Fehlertoleranz, nicht-maskierende Fehlertoleranz, Variant-Funktion, Lyapunov-Funktion, Bubblesort, Fault Containment**

## I. Einleitung

Es sind mehrere Arten bekannt, wie Systeme gegen Fehler abgesichert werden können. Selbststabilisierende Algorithmen gehören wohl zu der mächtigsten Art der fehlertoleranten Algorithmen. Sie zeichnen sich durch die Eigenschaft aus, von jedem beliebigen Zustand nach endlicher Zeit wieder in einen Zustand zurückzukehren, in dem sie fehlerfrei ihre Aufgabe erfüllen. Durch diese Eigenschaft eignen sich selbststabilisierende Algorithmen besonders in solchen Bereichen, wo eine lange Laufzeit erfordert ist. Zusätzlich kann das System auf der einen Seite starken Einflüssen unterworfen sein, die zu Fehlern führen können. Auf der anderen Seite ist ein manuelles Eingreifen aber nicht unbedingt möglich oder nicht erwünscht. Das System ist so zu sagen „sich selbst überlassen“.

Einsatz finden selbststabilisierende Algorithmen daher vor allem in Netzwerken. Routing-Protokolle sind hier ein populäres Beispiel [1]. Hier handelt es sich oftmals um große Rechnernetze, die lange Zeit autonom arbeiten und dabei dynamisch auf Änderungen ihrer Struktur reagieren müssen. Neue Knoten können hinzu kommen oder bestehende Knoten ausfallen. Das System muss sich diesen Änderungen möglichst schnell automatisch anpassen.

Im Folgenden wird zunächst eine formale Definition für Fehlertoleranz und insbesondere Selbststabilisierung aufgestellt. Darauf aufbauend werden zwei Beispielalgorithmen vorgestellt und formal gezeigt, dass diese selbststabilisierend sind. Die beiden Algorithmen werden dann exemplarisch zu einem weiteren selbststabilisierenden Algorithmus kombiniert. Zum Abschluss wird ein Ausblick auf weitere Entwicklungen auf dem Gebiet der Selbststabilisierung gegeben.

## II. Definition von Fehlertoleranz

Aus der Spezifikation der Aufgabe (*Task*) eines (fehlertoleranten) Systems ergeben sich zwei Anforderungen, die der Algorithmus erfüllen soll [2]. Eine Bedingung live, die *Lebendigkeit* garantiert, und eine Bedingung safe, die *Sicherheit* garantiert.

Bei einer Verkehrsampel könnten diese Anforderungen folgendermaßen aussehen:

- live: Kein Verkehrsteilnehmer soll unendlich lange auf ein grünes Signal warten müssen.
- safe: Zu keiner Zeit dürfen Fahrzeuge aus verschiedenen Richtungen gleichzeitig die Kreuzung überqueren dürfen.

Im Allgemeinen beschreibt die safe-Bedingung einen Zustand, der immer (bzw. nie) gegeben sein soll, wo hingegen die live-Bedingung einen Zustand beschreibt, der nach endlicher Zeit eintreten soll. Ein einfacher Fall einer Lebendigkeitsbedingung ist die Terminierung eines Algorithmus.

live und safe sind im Allgemeinen von einander unabhängig. Ein Algorithmus, der *liveness* garantiert, in dem er terminiert, muss nicht *save* sein, da er z.B. ein falsches Ergebnis liefern kann. Umgekehrt kann ein Algorithmus sich in einem Zustand befinden, der zwar *safe* ist, aus dem er aber beispielsweise nie terminiert.

### A. Vier Klassen der Fehlertoleranz

Je nachdem welche der beiden Bedingungen ein Algorithmus noch garantiert, wenn Fehler auftreten, werden vier Klassen der Fehlertoleranz unterschieden [2]. Diese sind in Tabelle I aufgeführt.

| $\wedge$ | live | $\neg$live |
|---|---|---|
| safe | maskierend fehlertolerant | *fail-stop* |
| $\neg$safe | nicht-maskierend fehlertolerant | nicht fehlertolerant |

Tabelle I
Fehlertoleranzklassen

*Maskierend fehlertolerant* werden die Algorithmen genannt, die zu jeder Zeit sowohl Sicherheit als auch Lebendigkeit garantieren. Das Auftreten von Fehlern ist nach außen transparent.

Als *fail-stop*-Systeme werden die Systeme bezeichnet, die ihre Arbeit einstellen, bevor ein Zustand erreicht wird, der nicht mehr *save* ist. Diese Systeme werden dort eingesetzt, wo Sicherheit wichtiger ist als Lebendigkeit.

Im Gegensatz dazu sind *nicht-maskierend-fehlertolerante* Systeme solche, bei denen *Lebendigkeit* zu jedem Zeitpunkt garantiert wird, die safe-Bedingung aber beim Auftreten von

Fehlern zeitlich begrenzt[1] verletzt wird. Insbesondere die *selbststabilisierenden* Algorithmen gehören dieser Klasse an.

Die vierte Klasse ist die der nicht-fehlertoleranten Algorithmen.

### B. Formale Definition eines Algorithmus

Im vorherigen Abschnitt sind bereits die Bedingungen safe und live eingeführt worden, um einen Algorithmus zu charakterisieren. Hier wird nun eine formale Definition eines fehlertoleranten Systems vorgestellt, um Selbststabilisierung formal zu definieren und für einen gegebenen Algorithmus auch nachweisen zu können.

Ein (verteiltes) System hat eine Menge $\mathcal{C}$ an möglichen Konfigurationen. Eine Konfiguration $c$ fasst die Zustände aller Subsysteme des Systems zu einem Zeitpunkt zusammen. Wenn z.B. ein System aus einer Menge von $n$ Rechnerknoten besteht, die ihren eigenen Zustand in lokalen Variablen $s_1, s_2, \ldots, s_n$ speichern und über öffentliche Register $r_1, r_2, \ldots, r_n$ untereinander Werte austauschen, so ist eine Konfiguration des Systems von der Form $c = (s_1, s_2, \ldots, s_n, r_1, r_2, \ldots, r_n)$. $c$ ist ein Tupel aus den Inhalten aller Variablen des Systems. $\mathcal{C}$ ist somit das kartesische Produkt der Wertebereiche aller Variablen.

Die einzelnen Subsysteme *(Prozesse)* führen jeder einen (lokalen) Algorithmus aus. Zur Notation eines Algorithmus werden in dieser Arbeit *Guarded Commands* [5] benutzt. Ein *Guarded Command* ist ein Paar $\mathbb{B} \rightarrow st$ aus einem bool'schen Ausdruck $\mathbb{B}$, dem *Guard*, und einem Statement $st$, dem *Command*. Ein *Guarded Command* ist (unter einer Konfiguration $c$) *aktiviert*, wenn sein *Guard* unter der Belegung von $c$ wahr ist. In dem Fall kann das System von der Konfiguration $c$ in die Konfiguration $c'$ übergehen.

E. Dijkstra führt in [5] zwei Formen von Statements ein, die *Guarded Commands* zusammenfassen. Hier wird nur die zweite Form, das *repitative* Statement **do** ... **od**, benötigt, das eine Menge *Guarded Commands* umschließt. Es hat die einfache Semantik, dass der Prozess, der das Statement ausführt, wiederholt einen *Guarded Command* auswählt, dessen *Guard* aktiviert ist, und dessen *Command* ausführt. Die Auswahl eines aktivierten *Guarded Command* geschieht nichtdeterministisch.

Die Ausführung eines Algorithmus ist somit eine Folge von Berechnungsschritten, die sich in Übergängen von einer Konfiguration $c$ in eine Folgekonfiguration $c'$ äußern. Die Menge der Konfigurationsübergänge, die durch Berechnungsschritte auftreten können, wird als $\mathcal{A}$ bezeichnet. $\mathcal{A}$ stellt damit quasi den Gesamtalgorithmus des Systems dar, der aus den Algorithmen aller Prozesse besteht.

### C. Fehlerklassen

Um einen gegebenen Algorithmus auf Fehlertoleranz zu untersuchen und insbesondere nachzuweisen, dass ein Algorithmus fehlertolerant ist, reicht es nicht aus, nur den Algorithmus selbst zu betrachten. Man muss vielmehr auch festlegen, welche Fehler der Algorithmus, bzw. das System, tolerieren soll. Im Allgemeinen ist es nicht möglich, ein System gegen alle möglichen Fehler abzusichern.

Bei selbststabilisierenden Algorithmen nimmt man meist an, dass ein Fehler beliebig viele Variablen beliebiger Knoten beliebig (innerhalb ihres Wertebereichs) verändert. Diese Fehler sind allerdings *transient*, d.h. dass Fehler zwar Werte verändern können, diese aber durch den Algorithmus wieder korrigiert werden können. Somit sind beispielsweise solche Fehler ausgeschlossen, nach denen Register nicht mehr beschrieben werden können und dauerhaft den falschen Wert beinhalten. Es ist leicht einzusehen, dass das System sonst unter Umständen nicht mehr in einen stabilen Zustand zurückkehren kann.

Die Gesamtheit der tolerierten Fehler wird als *fault class* $\mathcal{F}$ – im Deutschen *Fehlerklasse* oder *Fehlermodell* – bezeichnet.

Aufgrund der Eigenschaft selbststabilisierender Algorithmen, aus jedem beliebigen Zustand in einen stabilen Zustand zurückzukehren, wird meist darauf verzichtet, die Fehlerklasse explizit anzugeben. Ansonsten kann eine Fehlerklasse auch auf die gleiche Art wie der Algorithmus selbst spezifiziert werden [3], beispielsweise also ebenfalls durch *Guarded Commands*.

### D. Formale Definition von Fehlertoleranz

Auf der Grundlage dieser Definitionen haben A. Arora und M. Gouda 1993 folgende Definition von *Fehlertoleranz* vorgeschlagen [3].

**Definition II.1** (Fehlertoleranz nach [3])**.** *Es sei $\mathcal{A}$ ein Algorithmus, $\mathcal{F}$ eine Fehlerklasse und* S *ein Prädikat auf den Konfigurationen von $\mathcal{A}$. $\mathcal{A}$ ist* $\mathcal{F}$-fehlertolerant bezüglich S *g.d.w. ein weiteres Prädikat* T *existiert, so dass folgende Bedingungen gelten:*

1) S *impliziert* T*: Alle Konfigurationen, die* S *erfüllen, erfüllen auch* T*.*
2) *$\mathcal{F}$ und $\mathcal{A}$ sind abgeschlossen in* T*: Für alle Konfigurationen, in denen* T *gilt, gilt* T *auch in allen Folgekonfigurationen, die durch einen Schritt aus $\mathcal{F}$ oder $\mathcal{A}$ erreichbar sind.*
3) *$\mathcal{A}$ ist abgeschlossen in* S*: Für alle Konfigurationen, in denen* S *gilt, gilt* S *auch in allen Folgekonfigurationen, die durch einen Schritt aus $\mathcal{A}$ erreichbar sind.*
4) T *konvergiert mit $\mathcal{A}$ gegen* S*: Wenn nur Schritte aus $\mathcal{A}$ ausgeführt werden, gelangt man aus einer Konfiguration, in der* T *gilt, nach endlich vielen Schritten immer in eine Konfiguration, in der auch* S *gilt.*

Abbildung 1 veranschaulicht diese Definition. Durch einen Fehler (durch „$\rightsquigarrow$“ angedeutet) verlässt das System die Menge der Konfigurationen, in denen S gilt. Es ist dann nur noch T erfüllt. Nach endlich vielen Schritten kehrt das System wieder in eine Konfiguration zurück, in der auch S gilt.

T wird auch *Faultspan* genannt. S nennt man eine *Invariante* von $\mathcal{A}$. Meist will man zeigen, dass ein System schließlich seine Spezifikation erfüllt, also fehlertolerant für $\mathsf{S} = \mathsf{safe} \wedge \mathsf{live}$

[1] Diese Definition von *non-masking fault tolerant* findet sich in [3]. Eine alternative Definition sind Systeme, die safe zwar dauerhaft verletzen, diese Abweichung aber begrenzt ist, so dass immer noch ein gewisses Maß an Sicherheit garantiert ist [4].

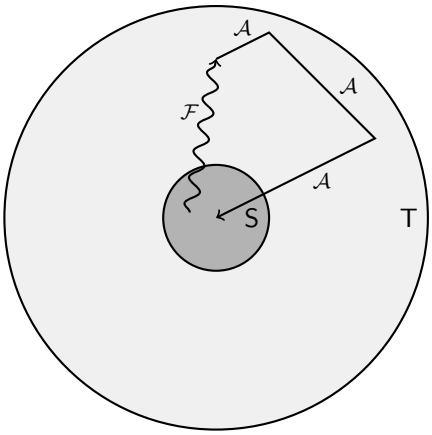



Abbildung 1. Fehlertoleranz nach [3]. Nach einem Fehler ($\rightsquigarrow$) gilt immer noch T. Wenn kein weiterer Fehler auftritt, gilt nach endlich vielen Schritten wieder die Spezifikation S.

ist. Im Allgemeinen kann für einen Algorithmus mehr als ein Prädikat T gefunden werden, unter dem die obigen Bedingungen gelten. Dabei sind das *stärkste*, $\mathsf{T}_s$, und das *schwächste*, $\mathsf{T}_w$, besonders interessant, da anhand von ihnen ebenfalls die Fehlertoleranzklasse unterschieden werden kann: Wenn $\mathsf{T}_s = \mathsf{S}$ gilt, ist S beim Auftreten eines Fehlers immer noch erfüllt. Somit liegt dann *maskierende Fehlertoleranz* vor, andernfalls *nicht-maskierende Fehlertoleranz*. Wenn $\mathsf{T}_w = \mathbf{true}$ gilt, stabilisiert der Algorithmus sich aus jedem Zustand (Arora und Gouda nennen dies *global stabilizing*). Der Algorithmus ist somit *selbststabilisierend*.

## III. Beispiele

Bisher wurde gezeigt, wie ein Algorithmus und die fehlerbehaftete Umgebung, in der der Algorithmus ausgeführt wird, formal beschrieben werden kann. Darauf aufbauend ist eine formale Definition vorgestellt worden, anhand der nachgewiesen werden kann, ob ein gegebener Algorithmus selbststabilisierend ist. Intuitiv muss dafür bewiesen werden, dass der Algorithmus aus *jeder* möglichen Konfiguration ($\mathsf{T}_w = \mathbf{true}$ ist per Definition immer erfüllt) nach einer *endlichen Anzahl an Schritten* eine *sichere* Konfiguration erreicht, ab der die Spezifikation S gilt.

Abgeschlossenheit und Konvergenz werden dabei getrennt bewiesen. Der Beweis der Selbststabilisierung teilt sich somit in zwei Teile.

Dies wird im Folgenden an zwei Beispielen demonstriert und dabei zwei Methoden – das Finden einer *Variant-Funktion* und das Finden einer *Lyapunov-Funktion* – zum Zeigen von Konvergenz vorgeführt.

### A. Variant-Funktion

Das erste Beispiel eines selbststabilisierenden Algorithmus ist besonders von historischer Bedeutung. Abbildung 3 zeigt den ersten von drei selbststabilisierenden Algorithmen, die 1974 von E. Dijkstra in [6] vorgestellt worden sind. Sie sind deshalb interressant, da nach Ausssage Dijkstras zuvor nicht bekannt war, ob es einen nichttrivialen selbststabilisierenden Algorithmus gibt.

Die von Dijkstra vorgestellten Algorithmen stellen gegenseitigen Ausschluss zwischen $n$ ringförmig angeordneten Prozessorknoten $P_0, P_1, \ldots P_{n-1}$ her. In einer stabilen Konfiguration kann immer genau ein Prozessor einen Berechnungsschritt ausführen. Dies ist dann immer der rechte Nachbar des Prozessors, der den letzten Berechnungsschritt ausgeführt hat. Das System ist somit ein Tokenring, in dem ein imaginäres Token reihum gereicht wird. Dies ist in Abbildung 2 dargestellt. Ein Prozess, der einen Berechnungsschritt ausführen kann, wird im Folgenden *aktiviert* genannt. safe und live lassen sich wie folgt formulieren:

- live: Es ist mindestens ein Prozess aktiviert. Diese Bedingung ist immer erfüllt und schließt Deadlocks aus.
- safe: Es ist höchstens ein Prozess aktiviert. Dies sichert den gegenseitigen Ausschluss.

Aus safe und live folgt die Spezifikation

S := „Es ist genau ein Prozess aktiviert“

.

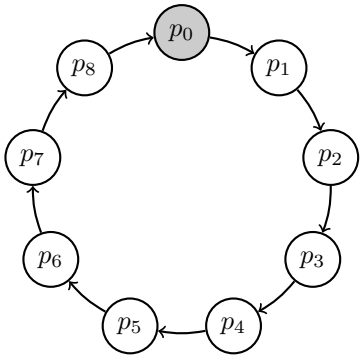



Abbildung 2. Ringtopologie mit $p_0$ als *special processor*. In einer stabilen Konfiguration führen die Prozessoren nacheinander einen Berechnungsschritt aus.

In dieser Arbeit wird der erste der drei Algorithmen betrachtet, wie er in Abbildung 3 dargestellt ist. Jeder Prozessor $P_i$ hat ein öffentliches Register $r_i$, das ganzzahlige Werte von 0 bis $n$ (einschließlich) annehmen kann (Dijkstra spricht daher von einer Lösung mit $(n+1)$-*state machines*) und das er lesen und beschreiben kann. Er kann zudem das Register seines linken Nachbarn, $r_{i-1}$ bzw. $r_{n-1}$ für $P_0$, lesen. $P_0$ verhält sich anders als die anderen Prozessoren und wird daher *special processor* genannt.

Es wird im Allgemeinen angenommen, dass das Lesen und das Schreiben eines Registers atomar ist (dies wird *read/write atomicity* genannt [7]), nicht aber der gesamte *Guarded Command*. Dijkstra nimmt der Einfachheit halber (zusätzlich) die Existens eines *central daemons* an, der nichtdeterministisch einen Prozess mit aktiviertem Guard auswählt, um einen Schritt auszuführen. Dies hat zur Folge, dass die Auswertung eines Guards und die Ausführung seines Commands

- für Knoten 0:
  **do**
  $\langle r_0 = r_{n-1} \to r_0 := r_0 + 1 \pmod{n+1}\rangle$
  **od**
- für Knoten $i$, $i = 1, 2, \ldots, n-1$:
  **do**
  $\langle r_{i-1} \neq r_i \to r_i := r_{i-1}\rangle$
  **od**

Abbildung 3. Dijkstras erster selbststabilisierender Algorithmus. Die spitzen Klammern $\langle \ldots \rangle$ deuten atomare Ausführung an.

zusammen atomar erfolgen. Dieses Prinip wird hier zunächst beibehalten. Atomare Anweisungen werden in den Beispielen in spitze Klammern $\langle \ldots \rangle$ eingeschlossen.

In Tabelle II ist eine beispielhafte Ausführung des Algorithmus für $n = 5$ aufgeführt. In Schritt 7 wird erstmals eine stabile Konfiguration erreicht.

| Schritt | $r_0$ | $r_1$ | $r_2$ | $r_3$ | $r_4$ |
|---|---|---|---|---|---|
| 0 | 3 | 0 | 1 | 3 | 2 |
| 1 | 3 | 0 | 0 | 3 | 2 |
| 2 | 3 | 0 | 0 | 3 | 3 |
| 3 | 4 | 0 | 0 | 3 | 3 |
| 4 | 4 | 4 | 0 | 3 | 3 |
| 5 | 4 | 4 | 0 | 0 | 3 |
| 6 | 4 | 4 | 4 | 0 | 3 |
| 7 | 4 | 4 | 4 | 4 | 3 |
| 8 | 4 | 4 | 4 | 4 | 4 |
| 9 | 5 | 4 | 4 | 4 | 4 |
| 10 | 5 | 5 | 4 | 4 | 4 |
| 11 | 5 | 5 | 5 | 4 | 4 |
| 12 | 5 | 5 | 5 | 5 | 4 |
| 13 | 5 | 5 | 5 | 5 | 5 |
| 14 | 0 | 5 | 5 | 5 | 5 |
| ... | ... | ... | ... | ... | ... |

Tabelle II
BEISPIELHAFTE AUSFÜHRUNG VON DIJKSTRAS ERSTEM SELBSTSTABILISIERENDEN ALGORITHMUS

Mithilfe einer *Variant-Funktion* [8] $f$ wird nun gezeigt, dass der Algorithmus selbststabilisierend ist. Man nimmt dazu eine Funktion her, die „misst“, wie weit das System noch von einer stabilen Konfiguration „entfernt“ ist. Der Funktionswert nimmt mit jedem Berechnungsschritt ab, bis eine stabile Konfiguration erreicht ist. Eine *Variant-Funktion* ist eine Funktion von der Menge aller Konfigurationen $\mathcal{C}$ in eine Menge $M$ auf der eine *wohlfundierte Relation* $<$ existiert und die bezüglich $<$ nach unten begrenzt ist. Es ist dann zu zeigen, dass immer, wenn das System durch einen Berechnungsschritt von einer Konfiguration $c$ in eine Konfiguration $c'$ wechselt, $f(c') < f(c)$ gilt, und dass eine Konfiguration $c$ stabil ist, wenn $f(c)$ minimal ist. Da $<$ wohl fundiert ist, muss das Minimum nach endlich vielen Schritten erreicht werden.

Beim vorliegenden Algorithmus kann folgendes Verhalten beobachtet werden, das hilft, eine *Variant-Funktion zu finden*: Solange das System nicht stabil ist, nähert sich der Wert von $r_0$ einem Wert, der in keinem Register eines aktivierten Prozessors steht. Ist dieser Wert erreicht, wird er von $P_1$ übernommen, dann von $P_2$, und so fort. Da nach einem Schritt $r_{i-1} = r_i$ für den entsprechenden Prozessor $P_i$ gilt, ist dann kein Prozessor mit diesem Wert mehr aktiviert, bis er auch von $P_{n-1}$ übernommen wurde.

Um unterscheiden zu können, ob der Inhalt einer Variablen in der letzten Konfiguration $c$ oder in der Folgekonfiguration $c'$ gemeint ist, wird im Folgenden die Konfiguration einer Variablen vorangestellt. Es bezeichnet $c.r$ den Inhalt von $r$ in der Konfiguration $c$, $c'.r$ den Inhalt in der Konfiguration $c'$.

*Variant-Funktion* sei nun $f : \mathcal{C} \longrightarrow \mathbb{N} \times \mathbb{N}$ mit

$$f(c) = \begin{cases} (A, B) & \text{falls } |\{P_i | P_i \text{ aktiviert in } c\}| > 1 \\ (0, 0) & \text{sonst} \end{cases} \quad (1)$$

wobei

$$A = \min\{k \in \mathbb{N} \mid c.r_i \neq (c.r_0 + k) \mod (n+1) \\ \forall\ P_i \text{ aktiviert in } c\} \quad (2)$$

$$B = \sum_{\substack{0 \leq i \leq n-1 \\ P_i \text{ aktiviert in } c}} (n - i) \quad (3)$$

$A$ gibt den Abstand von $r_0$ bis zum nächsten „unbenutzten“ Wert an.

Tupel $(a, b) \in \mathbb{N} \times \mathbb{N}$ lassen sich lexikographisch ordnen. Es ergibt sich die wohl fundierte Relation

$$(a, b) < (c, d) :\Leftrightarrow a < c \vee (a = c \wedge b < d). \quad (4)$$

$(0, 0)$ ist minimal.

Im Folgenden wird bewiesen, dass $A$ abnimmt, wenn $P_0$ einen Schritt macht, und $B$ abnimmt, wenn ein anderer Prozessor einen Schritt macht. Folglich ist $f$ dann in jedem Schritt fallend.

Dazu wird zunächst gezeigt, dass die Menge der Werte, die in den Registern von *aktivierten* Prozessoren stehen, sich durch einen Berechnungsschritt nicht vergrößert.

*Beweis:* Angenommen, nach einem Berechnungsschritt gibt es einen aktivierten Prozessor $P_i$, in dessen Register ein Wert steht, der vorher in keinem Register gestanden hat. Der Wert in $r_i$ kann sich in dem Schritt nicht geändert haben, da $P_i$ dann nicht mehr aktiviert wäre. (Dies ergibt sich aus dem Algorithmus.) Ebenso kann $P_i$ zuvor nicht aktiviert gewesen sein (dann wäre der Wert von $r_i$ enthalten gewesen) und als Letzter muss der linke Nachbar von $P_i$ seinen Wert geändert haben (das ist das Einzige, was zu einer Aktivierung von $P_i$ führen kann). Dieser war also in der letzten Konfiguration aktiviert.

Man nehme nun an, der neu aktivierte Prozessor sei $P_0$. Da das System sich noch nicht in einem stabilen Zustand befindet, muss es einen zweiten Prozessor $P_k$, $k > 0$, geben, der zuvor bereits aktiviert war und für den gilt $r_{k-1} \neq r_k = r_{k+1} = \cdots = r_{n-1} = r_0$. Damit war der Wert von $r_0$ bereits zuvor mit $r_k$ enthalten. Dies ist ein Widerspruch zur Annahme.

Nehme man stattdessen an, dass der neue Wert in $r_i$, $i > 0$, steht. Nach der obigen Argumentation hat dann $P_{i-1}$ den

letzten Schritt gemacht. Es muss aber auch $r_{i-1} = r_i$ in der vorherigen Konfiguration gegolten haben, weil sonst $P_i$ schon aktiviert gewesen wäre. Somit war der Wert von $r_i$ aber schon enthalten, was ebenfalls ein Widerspruch ist. ■

Nun kann gezeigt werden, dass $f$ fallend ist, also $f(c') < f(c)$ für eine Folgekonfiguration $c'$ von $c$ ist.

*Beweis:* Es sei $P_i$ der Prozessor, der den Berechnungsschritt ausgeführt hat, der von $c$ nach $c'$ führt. Es sei $f(c) = (A, B) \neq (0, 0)$ und $f(c') = (A', B')$. Es gibt hier zwei Fälle:

- $i = 0$: Es gilt nach obigem Beweis

$$\begin{aligned} A' &= \min\{k \in \mathbb{N} \mid c'.r_i \neq (c'.r_0 + k) \mod (n+1) \\ &\qquad \forall\ P_i \text{ aktiviert in } c'\} && (5) \\ &\leq \min\{k \in \mathbb{N} \mid c'.r_i \neq (c.r_0 + k) \mod (n+1) \\ &\qquad \forall\ P_i \text{ aktiviert in } c'\} - 1 && (6) \\ &= A - 1 && (7) \end{aligned}$$

  wegen $c'.r_0 = (c.r_0 + 1) \mod (n+1)$ und $A > 0$, da $P_0$ in $c$ aktiviert war. Es folgt $(A', B') < (A, B)$.
- $0 < i < n - 1$ sonst: Nach obigem Beweis gilt mit $c.r_0 = c'.r_0$

$$A' = A \tag{8}$$

  und

$$\begin{aligned} B' &= \sum_{\substack{0<j\leq n-1 \\ P_j \text{ aktiviert in } c'}} (n-j) && (9) \\ &\leq \sum_{\substack{0<j\leq n-1 \\ P_j \text{ aktiviert in } c}} (n-j) \\ &\qquad - (n-i) + \begin{cases} (n-(i+1)) & \text{falls } i < n-1 \\ & \text{und } P_i \text{ nicht} \\ & \text{aktiviert in } c \\ 0 & \text{sonst} \end{cases} && (10) \\ &\leq B - 1 && (11) \end{aligned}$$

  da $P_i$ in $c'$ nicht mehr aktiviert ist, allerdings $P_{i+1}$ möglicherweise aktiviert wird. Auch hier folgt $(A', B') < (A, B)$.

Damit gilt in allen Fällen, in denen $c$ nicht stabil ist, $f(c') < f(c)$. Es wird somit nach endlicher Zeit $f(c) = (0, 0)$ erreicht. Es sind alle Konfigurationen $c$ mit $f(c) = (0, 0)$ stabil, da bei mehr als einem aktivierten Prozessor nach Gleichung (3) $f(c) = (A, B)$ mit $B > 0$ gilt. Das System konvergiert also zu einer Menge stabiler Konfigurationen. ■

Es ist noch zu zeigen, dass die stabilen Konfigurationen abgeschlossen sind.

*Beweis:* Zunächst einmal ist live immer erfüllt. Dies zeigt sich durch Widerspruch: Angenommen, es ist kein Prozessor aktiviert, ist aufgrund der *Guards* für $P_i$, $i \neq 0$, $r_0 = r_1 = \cdots = r_{n-1}$. Damit ist aber der *Guard* für $P_0$ aktiviert.

Stabile Konfigurationen sind hier genau die Konfigurationen, in denen genau ein Prozessor aktiviert ist. Da ein Prozessor, nachdem er einen Schritt gemacht hat, nicht mehr aktiviert ist, dafür aber sein rechter Nachbar aktiviert wird, bleibt das System in einer stabilen Konfiguration. Der Algorithmus ist also *selbststabilisierend.* ■

### B. Lyapunov-Funktion

Als nächstes wird anhand eines weiteren Beispiels gezeigt, wie mittels einer *Lyapunov-Funktion* Konvergenz eines Systems bewiesen werden kann. Dies ist eine Vorgehensweise, die dem Finden einer *Variant-Funktion* sehr ähnlich ist und von J. Oehlerking, A. Dhama und O. Theel in [9] vorgestellt wird. Der Vorteil dieser Methode ist, dass bei bestimmten selbststabilisierenden Systemen – *discrete-time hybrid systems* – Konvergenz automatisiert gezeigt werden kann. Bei allgemeinen *Variant-Funktionen* ist dies meist, zumindest bisher, nicht möglich.

Sei $n \in \mathbb{N}$ und $e \in \mathbb{R}^n$ eine Konfiguration[2]. $e$ ist ein *equilibrium point* (zu Deutsch *Gleichgewichtspunkt*) g.d.w. für alle Folgekonfigurationen $e'$ von $e$ gilt $e' = e$. *Equilibrium points* sind somit *stabile Konfigurationen*. Die Menge aller *equilibrium points* sei $\mathcal{E}$. Es ist $V : \mathbb{R}^n \longrightarrow \mathbb{R}$ eine *Lyapunov-Funktion* g.d.w. für beliebige Konfigurationen $c$ und beliebige Folgekonfigurationen $c'$ von $c$ gilt

- $V$ ist positiv definit, also

$$V(c) = 0, \text{ falls } c = 0 \text{ und } V(c) > 0 \text{ sonst.} \tag{12}$$

- $$V(c') - V(c) = 0, \text{ falls } c = 0 \text{ und } V(c') - V(c) < 0 \text{ sonst} \tag{13}$$

- $x \to \infty \Rightarrow V(x) \to \infty$

Da man allerdings Konvergenz nicht unbedingt gegen den Nullpunkt sondern die Menge der *equilibrium points* beweisen möchte, wird Gleichung (13) zu

$$V(c') - V(c) = 0, \text{ falls } c \in \mathcal{E} \text{ und } V(c') - V(c) \leq \delta \text{ sonst} \tag{14}$$

abgewandelt [9].

Die Beweistechnik wird jetzt anhand eines zweiten selbststabilisierenden Algorithmus verdeutlicht, der gemeinhin als Sortieralgorithmus *Bubblesort* bekannt ist. Er ist in Abbildung 4 beschrieben.

Das System besteht aus $n$ Prozessorknoten $P_i$, $0 \leq i \leq n-1$, die jeder ein Register $r_i$ schreiben und die Register $r_i$ und $r_{i+1}$ lesen können. Die $r_i$ können beliebige positive reelle Werte annehmen. Dem Register $r_n$ ist kein Knoten zugeordnet. Es wird auch hier ein *central daemon* angenommen.

Es wird gezeigt, dass der Algorithmus gegen Konfigurationen konvergiert, in denen $r_0 \leq r_1 \leq \cdots \leq r_n$ gilt. Um Gleichung (12) auf jeden Fall zu erfüllen, werden für den Beweis die Registerinhalte quadriert. Da nur positive Werte

[2]Mit $\mathbb{R}^n$ ist hier die Menge der $n$-dimensionalen Vektoren aus den reellen Zahlen gemeint.

sortiert werden, folgt aus $r_i < r_j$ auch $r_i^2 < r_j^2$ für beliebige $i, j$. Es ist die Menge der *equilibrium points*

$$\mathcal{E} := \{(r_0, r_1, \ldots, r_n) \in \mathbb{R}^{n+1} \mid r_0^2 \leq r_1^2 \leq \cdots \leq r_n^2\}. \quad (15)$$

Dies sind nämlich genau die Konfigurationen, in denen kein Knoten mehr einen Berechnungsschritt ausführen kann.

Die Idee, um die Konvergenz des *Bubblesort*-Algorithmus zu beweisen, ist, eine Funktion zu finden, die besonders kleine Werte Liefert, wenn große Werte in den Variablen mit höherem Index liegen. Dazu wird einfach eine gewichtete Summe verwendet, bei denen höhere Indices kleinere Koeffizienten bekommen. Als Kandidat für eine *Lyapunov-Funktion* eignet sich so besonders

$$V(r_0, r_1, \ldots, r_n) := \sum_{k=0}^{n} r_k^2 2^{-k} \quad (16)$$

Es ist klar, dass Gleichung (12) gilt. Ebenso gilt Gleichung (14), wenn $c \in \mathcal{E}$ gilt. (Dann gilt $c' = c$ und damit $V(c') = V(c)$ nach Definition eines *equilibrium points*.) Es muss noch Gleichung (14) für $c \notin \mathcal{E}$ gezeigt werden: Man betrachte den Fall, dass das System von einer Konfiguration $c$ nach $c'$ wechselt, indem $P_i$ einen Berechnungsschritt ausführt. Es gilt dann $c.r_{i+1} < c.r_i$ (der *Guard*) und $c'.r_i = c.r_{i+1}, c'.r_{i+1} = c.r_i$ (der *Command*). (Es wird die gleiche Notation wie in Abschnitt III-A verwendet.) Daraus folgt

$$\begin{aligned} V(c') &= \sum_{k=0}^{n} c'.r_k^2 2^{-k} && (17)\\ &= \sum_{k=0}^{n} c.r_k^2 2^{-k} - c.r_i^2 2^{-i} - c.r_{i+1}^2 2^{-(i+1)} \\ &\quad + c.r_i^2 2^{-(i+1)} + c.r_{i+1}^2 2^{-i} && (18)\\ &= V(c) + (-2c.r_i^2 - c.r_{i+1}^2 \\ &\quad + c.r_i^2 + 2c.r_{i+1}^2) 2^{-i-1} && (19)\\ &= V(c) + (-c.r_i^2 + c.r_{i+1}^2) 2^{-i-1} && (20) \end{aligned}$$

wegen $c.r_{i+1} < c.r_i$ folgt

$$V(c') - V(c) = (-c.r_i^2 + c.r_{i+1}^2) 2^{-i-1} \leq \delta \quad (21)$$

mit

$$\delta = -2^{-n} \min\{c.r_j^2 - c.r_i^2 \mid c.r_i < c.r_j\} < 0 \quad (22)$$

Da nur Werte getauscht werden, bleibt $\delta$ konstant. Der *Bubblesort*-Algorithmus erreicht damit nach endlicher Zeit eine Konfiguration $c \in \mathcal{E}$. Dies sind hier genau die Konfigurationen, in denen die Werte in den $r_i$ sortiert sind, weshalb die Spezifikation des Algorithmus erfüllt wird. Da dann auch keine Berechnungsschritte mehr ausgeführt werden, ist Abgeschlossenheit ebenfalls erfüllt.

## IV. Kombination selbststabilisierender Algorithmen

In beiden Beispielen wurde der Einfachheit halber angenommen, dass ein einzelner Schritt eines Prozessors – das Auswählen und Ausführen eines aktivierten *Guarded Commands* – atomar erfolgt. Wie bereits angedeutet, trifft dies in der Praxis meist nicht zu. Die Annahme eines *central daemons* vereinfacht aber die Beweisführung.

*Bubblesort*-Algorithmus für Knoten $i$, $i = 0, 1, \ldots, n-1$:
**do**
$\langle r_{i+1} < r_i \to swap(r_i, r_{i+1}) \rangle$
**od**

Abbildung 4. *Bubblesort*-Algorithmus. Die Funktion *swap* tauscht den Inhalt zweier Register.

Man kann zeigen, dass Dijkstras erster selbststabilisierender Algorithmus auch noch korrekt und selbststabilisierend ist, wenn auf einen *central daemon* verzichtet und stattdessen einfach *read/write atomicity* angenommen wird. Dazu muss aber der Wertebereich der $r_i$ verdoppelt werden [7]. Die Beweisidee ist hier, dass der Wert eines Vorgängers von einem Prozess zunächst in ein privates Register kopiert wird. Auf diese Art wird *read/write atomicity* simuliert. Dieses Register wird quasi als weiterer Knoten zwischen zwei benachbarten Prozessoren eingefügt. Die Beweisführung ist danach analog zur bisherigen Version des Algorithmus, es muss nur kein *central daemon* mehr angenommen werden. Für Details muss hier aber auf [7] verwiesen werden.

Wenn man mehrere selbststabilisierende Algorithmen *hierarchisch* zu einem neuen Algorithmus kombiniert, ist dieser ebenfalls selbststabilisierend. Man spricht von einer *fair composition* [7]. Die Algorithmen werden dabei so kombiniert, dass das Ergebnis des ersten Algorithmus quasi als Eingangswerte des nächsten Algorithmus dienen. Sobald der erste Algorithmus – unabhängig von den anderen Algorithmen – einen stabilen Zustand erreicht hat, kann sich der nächste Algorithmus in der Kette stabilisieren. Abbildung 5 verdeutlicht dies.

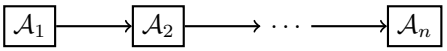



Abbildung 5. *Fair composition* von selbststabilisierenden Algorithmen $\mathcal{A}_1, \ldots, \mathcal{A}_n$.

Da der Entwurf und Beweis neuer selbststabilisierender Algorithmen meist schwierig ist, kann man auf diese Art bekannte selbststabilisierende Algorithmen als „Bausteine“ für neue (komplexere) Algorithmen verwenden.

So kann man den *Bubblesort*-Algorithmus aus dem zweiten Beispiel mit Dijkstras Algorithmus aus dem ersten Beispiel – allerdings in der eben angesprochenen Form ohne *central daemon* – kombinieren, um daraus einen *Bubblesort*-Algorithmus zu generieren, der ebenfalls ohne *central daemon* auskommt. Das Ergebnis ist in Abbildung 6 dargestellt. Der gegenseitige Ausschluss aus Dijkstras Algorithmus „umhüllt“ sozusagen den *Guarded Command* aus dem Sortieralgorithmus.

Es wurde bereits gezeigt, dass in einem stabilen Zustand

- für Knoten 0:

  **do**

  $r_0 = r_{n-1} \wedge v_1 < v_0 \rightarrow swap(v_i, v_{i+1})$;
  $r_0 := r_0 + 1 \pmod{2n+2}$

  $\square$

  $r_0 = r_{n-1} \wedge v_1 \geq v_0 \rightarrow$
  $r_0 := r_0 + 1 \pmod{2n+2}$

  **od**

- für Knoten $i$, $i = 1, 2, \ldots, n-1$:

  **do**

  $r_{i-1} \neq r_i \wedge v_{i+1} < v_i \rightarrow swap(v_i, v_{i+1})$;
  $r_i := r_{i-1}$

  $\square$

  $r_{i-1} \neq r_i \wedge v_{i+1} \geq v_i \rightarrow r_i := r_{i-1}$

  **od**

Abbildung 6. Kombination der Algorithmen. Die Guards müssen jeweils von links nach rechts ausgewertet werden. Es sind keine atomaren Blöcke mehr nötig.

von Dijkstras Algorithmus die Bedingung $r_i \neq r_{i-1}$ bzw. $r_0 = r_{n-1}$ immer für *genau einen* Knoten wahr ist. Der entsprechende Prozessor kann dann also gefahrlos einen Schritt aus dem *Bubblesort*-Algorithmus ausführen. Danach wird dann die Zuweisung für den neuen Wert von $r_i$ aus Dijkstras Algorithmus ausgeführt, so dass der nächste Prozessor an die Reihe kommt.

Um zu beweisen, dass die Selbststabilisierung wirklich erhalten bleibt, kann man argumentieren, dass Dijkstras Algorithmus quasi unverändert übernommen wurde. Da dieser selbststabilisierend ist, ist irgendwann ein Zustand erreicht, in dem gegenseitiger Ausschluss unter den Knoten herrscht. Da dann reihum ein Prozessor einen Schritt macht, ist auch die Funktion des eingebetteten *Bubblesort*-Algorithmus nicht beeinträchtigt, sodass nach endlicher Zeit auch bezüglich dieses Algorithmus' ein stabiler Zustand erreicht ist.

## V. Ausblick

In der vorliegenden Arbeit wurden zwei selbststabilisierende Algorithmen vorgestellt.

Bei der genaueren Betrachtung selbststabilisierender Algorithmen zeigen sich trotz ihrer wünschenswerten Eigenschaft, eine große Zahl an Fehlern zu tolerieren, einige Probleme auf.

Zum einen ist es keine einfache Aufgabe, für ein gegebenes Problem einen selbststabilisierenden Algorithmus zu finden.

Wie man auch schon anhand des Beweises für Dijkstras ersten selbststabilisierenden Algorithmus gesehen hat, kann auch schon bei einem so simplen Algorithmus die Beweisführung verhältnismäßig kompliziert werden. Oehlerking et al. weisen in [9] dadurch, dass sie Lyapunov-Funktionen einsetzen, einen Weg dahin auf, Selbststabilisierung automatisch zu beweisen.

Weiterhin fällt auf, dass im Allgemeinen keine Einschränkungen bestehen, welche Berechnungsschritte auf dem Weg zurück in einen stabilen Zustand ausgeführt werden. So kann die Korrektur eines einzelnen Fehlers viel Zeit in Anspruch nehmen, da z.B. auch primär nicht betroffene Knoten ihren Zustand in der Folge des Fehlers ändern. A. Gupta demonstriert deshalb in [10] das Prinzip des *Fault-containment*. Hier ist das Ziel einen selbststabilisierenden Algorithmus so zu designen (bzw. einen bestehenden Algorithmus so zu modifizieren), dass ein Fehler, der nur Variablen *eines* Prozesses verändert, in $\mathcal{O}(1)$ – also in konstanter Zeit, unabhängig von der Gesamtgröße des Systems – korrigiert wird.

Allein schon an diesen Punkten zeigt sich, dass das Gebiet der selbststabilisierenden Algorithmen und Systeme auch für künftige Forschung noch viel Raum bietet. Vielleicht können selbststabilisierende Systeme zukünftig in noch mehr Bereichen verstärkt eingesetzt werden.

## Literatur


[1] C. Adam and R. Stadler, "Patterns for routing and self-stabilization," in *in Proc. of Network Operations & Management Symposium (NOMS 2004), Seoul, Korea*, 2004, p. 200.

[2] F. C. Gärtner, "Fundamentals of fault-tolerant distributed computing," *ACM Computing Surveys*, vol. 31, no. 1, March 1999.

[3] A. Arora and M. Gouda, "Closure and convergence: A foundation of fault-tolerant computing," *IEEE Transactions on Software Engineering*, vol. 19, pp. 1015–1027, 1993.

[4] T. Warns, "Structural failure models for fault-tolerant distributed computing," Ph.D. dissertation, Universität Oldenburg, 2009.

[5] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Commun. ACM*, vol. 18, no. 8, pp. 453–457, Aug. 1975. [Online]. Available: http://doi.acm.org/10.1145/360933.360975

[6] E. W. Dijkstra, "Self-stabilizing systems in spite of distributed control," *Commun. ACM*, vol. 17, no. 11, pp. 643–644, Nov. 1974. [Online]. Available: http://doi.acm.org/10.1145/361179.361202

[7] S. Dolev, *Self-stabilization*. Cambridge, MA, USA: MIT Press, 2000.

[8] J. L. W. Kessels, "An exercise in proving self-stabilization with a variant function," *Inf. Process. Lett.*, vol. 29, no. 1, pp. 39–42, Sep. 1988. [Online]. Available: http://dx.doi.org/10.1016/0020-0190(88)90131-7

[9] J. Oehlerking, A. Dhama, and O. Theel, "Towards automatic convergence verification of self-stabilizing algorithms," in *Self-Stabilizing Systems*, ser. Lecture Notes in Computer Science, S. Tixeuil and T. Herman, Eds. Springer Berlin Heidelberg, 2005, vol. 3764, pp. 198–213. [Online]. Available: http://dx.doi.org/10.1007/11577327_14

[10] A. Gupta, "Fault-containment in self-stabilizing distributed systems," Ph.D. dissertation, University of Iowa, May 1997.