

Rheinisch-Westfälische Technische Hochschule Aachen
Lehrstuhl für Informatik II
Prof. Dr. Ir. P.-J. Katoen

Proseminar Verteilte Algorithmen im WS 2006/2007

Selbststabilisierende Algorithmen

Michael Nett

Matrikelnummer 257505

6. Februar 2007

Betreuer: Priv.-Doz. Dr. T. Noll

Inhaltsverzeichnis

1	Einführung	3
2	Dijkstras Definition	4
2.1	Kontext	4
2.2	Topologie	4
2.3	Der zentrale Dämon	5
2.4	Legitimität	6
2.5	Definition	6
2.6	Beispiel	6
2.6.1	Ansatz	7
2.6.2	Realisierung	7
2.6.3	Analyse	7
3	Abweichende Definition	8
3.1	Selbststabilisierung	8
3.2	Stabilisierung	9
3.3	Erreichbare Zustände	9
4	Pros und Kontras	10
5	Fehlermodell	10
5.1	Flüchtige Fehler	10
5.2	Der bösartige Gegenspieler	11
6	Modellierung fehlertoleranten Verhaltens	11
6.1	Inkonsistente Initialisierung	11
6.2	Übertragungsfehler	11
6.3	Flüchtiger Speicherfehler	11
6.4	Wechsel der Betriebsmodi	12
6.5	Betriebspausen von Prozessen	12
7	Selbststabilisierende Algorithmen	12
7.1	Ring-Orientierung	12
7.2	Maximale Paarung	12
7.3	Leiterwahl und Spannbaum Konstruktion	13
8	Entwurf selbststabilisierender Algorithmen	13
8.1	Sequenzielle Komposition	13
8.2	Der Kompositionsoperator	13
8.3	Minimaler-Pfad Probleme	14
9	Zusammenfassung	14

1 Einführung

Geprägt wurde der Begriff „Selbststabilisierung“ 1974 von dem niederländischen Informatiker und Turingpreisträger Edsger W. Dijkstra (1930-2002). Der Begriff kann innerhalb der Informatik dem Bereich verteilter Systeme zugeordnet werden, da seine Anwendung in sequenziellen Systemen eher zweitrangig ist.

In seinem 1974 publizierten Artikel „Self-stabilizing Systems in Spite of Distributed Control“ vermittelt Dijkstra erstmals den Begriff der Selbststabilisierung. Er beschreibt ein verteiltes System als selbststabilisierend genau dann, wenn [Dijkstra74]

[...] unabhängig von seinem anfänglichen Zustand [...] garantiert ist, dass dieses System einen legitimen Zustand in einer endlichen Anzahl von Schritten erreicht.

Auch ohne eine formale Definition von Selbststabilisierung lässt sich bereits erahnen, dass die Implementierung eines solchen Systems gerade durch die Nebenläufigkeit seiner Prozesse Probleme aufwirft: Ein systemweites Ziel muss durch lokale Entscheidungen, basierend auf lokalen Zuständen erreicht werden. Andererseits gehen die später im Detail behandelten Vorteile eines selbststabilisierenden Systems gerade aus der Nebenläufigkeit hervor.

Obschon der Begriff der Selbststabilisierung bereits seit 1974 durch Dijkstra definiert war, gab es lange Zeit nur wenige Publikationen zu diesem Thema. Dijkstra selbst war sich, während seinen ursprünglichen Arbeiten zu diesem Thema, nicht sicher ob überhaupt nicht triviale selbststabilisierende Systeme existieren könnten.

Unlängst jedoch hat die Informatik das Konzept der Selbststabilisierung für die Realisierung fehlertoleranten Verhaltens in verteilten Systemen für sich entdeckt. Hier bietet das Konzept der Selbststabilisierung ein einheitliches und flexibles Modell zur Erholung von Fehlern in einem verteilten System.

2 Dijkstras Definition

2.1 Kontext

Eine Situation, in welcher eine Vielzahl von Prozessen versucht ein gemeinschaftliches Ziel durch eigenständiges Handeln zu verwirklichen, ist in der Informatik nichts Aussergewöhnliches. Beispielsweise arbeiten in einem Betriebssystem viele Prozesse selbstständig und erfüllen Aufgaben, welche ihnen alleine oder auch einer Gruppe von Prozessen zugeteilt wurden. Das gemeinschaftliche Ziel der einzelnen Prozesse könnte man als „Aufrechterhaltung des Betriebes“ sehen. Ein globales Ziel auf welches von einzelnen, lokal beschränkten Prozessen hingearbeitet wird.

In diesem Beispiel stellt die Situation kein Problem dar, denn beteiligte Prozesse können auf den Zustand des Systemes in einem zentralen Speicherplatz zugreifen. Dadurch sind ihre Aktionen immer noch lokal beschränkt, aber ihre Informationsquelle umfasst das gesamte System. Wie entwickelt sich nun solch ein Sachverhalt, falls man diesen auf ein verteiltes System überträgt?

Zuerst existiert in einem verteilten System kein zentraler Speicher in dem der Systemzustand abgelegt wird. Vielmehr setzt sich der Systemzustand als Ganzes aus den Zuständen der beteiligten Prozesse zusammen. Der Zustand ist also über das System verteilt.

Zudem verfügt in einem verteilten System in der Regel ein Prozess nur über die Möglichkeit, mit seinen direkten Nachbarn zu kommunizieren, welches im Allgemeinen nur eine kleine Teilmenge der vorhandenen Prozesse ist. Diese Eigenschaft eines verteilten Systems hat immense Auswirkungen auf die Anforderungen die an einzelne Prozesse gestellt werden [Dijkstra74]:

Lokale Entscheidungen, basierend auf lokalen Informationen, müssen ein globales Ziel erfüllen.

Auch wenn die im Folgenden vorgestellten Inhalte Anwendung in sequenziellen Systemen finden, beschränkt sich der Kontext der Betrachtungen in dieser Ausarbeitung auf verteilte Systeme.

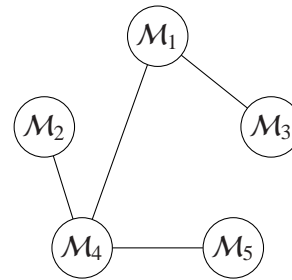


Abbildung 1: Beispielhafte Topologie eines verteilten Systems

2.2 Topologie

Man betrachtet ein verteiltes System als einen ungerichteten, schlaufenfreien Graphen

$$\mathcal{G} = (V, E), V = \{\mathcal{M}_1, \dots, \mathcal{M}_n\}$$

wie er in *Abbildung 1* dargestellt wird.

Die Adjazenzmatrix des Graphs muss allgemein nicht dünn besetzt sein, allerdings würde eine dicht besetzte Matrix einem stark verknüpften Graphen entsprechen. Die Tatsache, dass in einem echt verteilten System nicht alle Prozesse unmittelbar miteinander kommunizieren können, reflektiert also eine dünn besetzte Adjazenzmatrix.

Die Knotenmenge V setzt sich aus den Prozessen des Systems zusammen, während die Kantenrelation E die Fähigkeit zweier Prozesse darstellt Informationen auszutauschen. Die Wege, auf denen Prozesse kommunizieren, hängen natürlich von der Implementation des Systems ab, also ob sich beispielsweise die Prozesse gemeinsamen Speicher teilen, die Kommunikation über ein Netzwerk abgewickelt wird, etc. Nimmt man an, dass die Kommunikation bidirektional funktioniert ist die Kantenrelation E symmetrisch.

Zwei Prozesse $\mathcal{M}_i, \mathcal{M}_j$ sind benachbart, falls sie durch eine Kante verbunden werden, also falls gilt:

$$(\mathcal{M}_i, \mathcal{M}_j) \in E$$

Für die folgenden Definitionen ist es nützlich, die Menge der benachbarten Prozesse zu definieren. Die Menge N_i ist die Menge der zu M_i benachbarten Prozesse und ist definiert durch

$$N_i = \{j \in \mathbb{N} \mid (M_i, M_j) \in E\}$$

Wie bereits gesagt ist der Systemzustand abhängig von den Zuständen aller beteiligten Prozesse. Dementsprechend wechselt das System seinen Zustand, falls einer der Prozesse einen Zustandswechsel vollzieht. Wie Dijkstra feststellte, müssen Prozesse Entscheidungen basierend auf lokalen Informationen treffen. Unter dem Begriff „Entscheidung“ versteht man hier die Wahl des neuen Zustands in den der Prozess wechselt. Die Zustandsübergänge oder auch Transitionen von Prozess M_i kodiert man in einer Zustandsübergangsfunktion δ_i .

In dieser Situation steht der Funktion δ_i nur der lokal verfügbare Teil des Systemzustandes zur Verfügung. Dieser Teil beinhaltet die Zustände der benachbarten Prozesse und des entsprechenden Prozesses selbst. Die Zustandsfunktion δ_i ist also von der Form,

$$\delta_i : \left(Q_i \times \prod_{j \in N_i} Q_j \right) \rightarrow Q_i$$

wobei Q_i die endliche Zustandsmenge des Prozesses M_i bezeichnet.

2.3 Der zentrale Dämon

Bereits zu Anfang seines Artikels weist Dijkstra auf ein erstes Problem hin, das die Modellierung eines verteilten Systems erschwert [Dijkstra74]:

Um die nicht definierten Geschwindigkeitsverhältnisse der verschiedenen Prozesse zu modellieren, führen wir einen zentralen Dämon ein [...]

Die von Dijkstra erwähnten Geschwindigkeitsunterschiede der einzelnen Prozesse sind allgemein nicht auszuschließen, da je nach Implementation des Systems verschiedene Prozesse durch verschieden schnelle Hardware realisiert werden, oder einfach dadurch, dass je nach Komplexität der einzelnen Transitionsfunktionen die Transitionen einzelner Prozesse unterschiedlich viel Zeit in Anspruch nehmen können.

Problematisch wird dieser Unterschied durch die Tatsache, dass generell die Kommunikation von benachbarten Prozessen (also das Abfragen von Zuständen benachbarter Prozesse und das Ändern des eigenen Zustands) keine atomare Operation ist. Um sich zukünftig im Modell von diesen Restriktionen zu befreien stellte Dijkstra einen zentralen Dämon vor.

Dijkstra führt für einen Prozess M_i ein Privileg p_i als boolesche Funktion etwa der Form

$$p_i : \left(Q_i \times \prod_{j \in N_i} Q_j \right) \rightarrow \{0, 1\}$$

in Abhängigkeit vom eigenen Zustand und dem seiner Nachbarn ein. Wird für einen Prozess M_i das Privileg p_i auf 1 abgebildet, dann ist M_i privilegiert.

Für einen Systemzustand kann also eindeutig bestimmt werden, welche Prozesse privilegiert sind und welche nicht.

Die Funktion des zentralen Dämons besteht nun darin, zufällig und fair einen privilegierten Prozess auszuwählen. Dieser Prozess darf nun seine Transition entsprechend seiner Transitionsfunktion vollziehen.

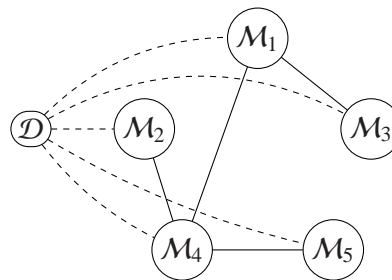


Abbildung 2: Verteiltes System mit zentralem Dämon

In dieser Modellierung koordiniert der Dämon die Zustandswechsel der einzelnen Prozesse, wobei die Wahl des nächsten privilegierten Prozesses als nicht deterministisch angenommen werden kann.

Da nun aber der zentrale Dämon als zentraler Prozess widersprüchlich zu der Idee eines echt verteilten Systems ist, schlägt Dijkstra die Implementation eines verteilten Dämons vor, worauf aber nicht näher eingegangen wird.

Mit dieser neuen Modellierung kann man nun bei zukünftigen Betrachtungen Transitionen und Kommunikation von Prozessen als atomare Operationen ansehen.

2.4 Legitimität

Es existiert ein systemweites Kriterium, welches den Zustand des Systems entweder als legitim qualifiziert oder nicht. Je nach Ziel des umgesetzten Algorithmus kann dieses Kriterium sehr unterschiedlich sein.

Dijkstra stellt weiter vier Forderungen an das verteilte System:

1. In jedem legitimen Zustand existiert ein privilegierter Prozess.
2. In jedem legitimen Zustand bringt jeder mögliche Zustandsübergang das System erneut in einen legitimen Zustand.
3. Jeder Prozess muss in mindestens einem legitimen Zustand privilegiert sein.
4. Für jedes Paar von legitimen Zuständen gibt es eine endliche Übergangsfolge, die das System vom einen in den anderen Zustand bringt.

Der erste Punkt fordert, dass ein verteiltes System nicht terminieren darf, also alle Berechnungen unendlich lange fortgeführt werden. Würde nämlich in einem legitimen Zustand kein Prozess privilegiert sein, würde der zentrale Dämon keine Möglichkeit haben das System in einen neuen Zustand zu bringen.

Dijkstras zweite Forderung bezieht sich auf die Abgeschlossenheit der legitimen Zustände unter der Transitionsfunktion. Er fordert, dass eine Transition ausgehend von einem legitimen Zustand wieder in einem legitimen Zustand resultiert. Das System kann also unter korrekter Ausführung nicht von einem legitimen in einen illegitimen Zustand abrutschen.

Im dritten Punkt wird gefordert, dass jeder Prozess zu einem Wechsel zwischen zwei legitimen Zuständen beitragen kann. Das impliziert, dass ein einzelner Prozess niemals so beschaffen sein darf, dass er einen legitimen Systemzustand kompromittieren könnte (ein derartiger Prozess dürfte in keinem legitimen Zustand privilegiert sein, denn sonst könnte der Dämon diesen Prozess auswählen. Die resultierende Transition verletzt dann die zweite Forderung).

In seiner letzten Forderung, stellt Dijkstra den Anspruch, dass es für zwei legitime Zustände eine endliche Transitionsfolge gibt, welche den einen Zustand in den anderen überführt. Also dürfen nicht mehrere in sich abgeschlossene disjunkte Mengen von legitimen Zuständen existieren.

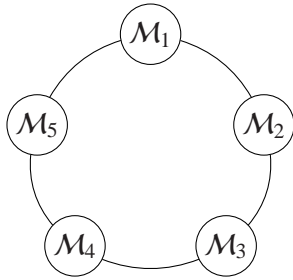
2.5 Definition

Ausgehend von dem beschriebenen verteilten System definiert Dijkstra die Eigenschaft der „Selbststabilisierung“ wie folgt [Dijkstra74]:

Wir nennen das System „selbst-stabilisierend“ genau dann, wenn unabhängig von seinem anfänglichen Zustand und unabhängig von dem jeweils für den nächsten Zustandsübergang ausgewählten Privileg, immer wenigstens ein Privileg vorhanden ist und garantiert ist, dass sich das System nach einer endlichen Anzahl von Zustandsübergängen in einem legitimen Zustand befindet.

2.6 Beispiel

Den ersten selbststabilisierenden Algorithmus präsentierte Dijkstra [Dijkstra74] selbst in seiner ursprünglichen Veröffentlichung zu dem Thema.

Abbildung 3: Dijkstras Token Ring, $N = 5$

Der vorgestellte Algorithmus operiert auf einem Ring von N Prozessen und versucht gegenseitigen Ausschluss (*mutual exclusion*) umzusetzen. Jeder Prozess hat einen als kritische Phase gekennzeichneten Teil. In diesen Phasen kann beispielsweise Zugriff auf eine zwischen den N Prozessen geteilte Ressource stattfinden. Der Algorithmus koordiniert nun gegenseitigen Ausschluss. Also stellt der Algorithmus sicher, dass sich keine zwei Prozesse simultan in ihren kritischen Phasen befinden (keine zwei Prozesse greifen gleichzeitig auf eine geteilte Ressource zu).

2.6.1 Ansatz

Dijkstra verwendet die Privilegienfunktion um zu beschreiben, dass ein Prozess berechtigt ist sich in seiner kritischen Phase zu befinden genau dann, wenn er privilegiert ist. Jetzt lässt sich das Problem des gegenseitigen Ausschlusses nach Tel [Tel94] wie folgt spezifizieren:

1. In jedem Systemzustand existiert höchstens ein privilegierter Prozess.
2. Jeder Prozess ist unendlich oft privilegiert.

Dijkstras Lösung für das Problem involviert ein sogenanntes „Token“, was soviel heißt wie eine „Markierung“, welche um den Ring wandert. Privilegiert ist immer genau der Prozess, der gerade das Token besitzt.

2.6.2 Realisierung

Für einen Ring von N Prozessen

$$\mathcal{M}_0, \dots, \mathcal{M}_{N-1}$$

mit $K > N$ Zuständen $Q = \{0, \dots, K - 1\}$ bezeichnet q_i den Zustand von Prozess \mathcal{M}_i . Die Nachbarn von \mathcal{M}_i sind gegeben durch die Ringgestalt des Systems:

$$N_i = \{\mathcal{M}_{(i+1) \bmod N}, \mathcal{M}_{(i-1) \bmod N}\}$$

Die Privilegienfunktionen sind wie folgt definiert:

$$p_i = 1 \leftrightarrow q_i \neq q_{i-1}, \forall i > 0$$

$$p_0 = 1 \leftrightarrow q_0 = q_{N-1}$$

Der Prozess \mathcal{M}_0 ist also unterschiedlich von den anderen Prozessen. Die Prozesse \mathcal{M}_1 bis \mathcal{M}_{N-1} hingegen sind identisch. Während \mathcal{M}_0 privilegiert ist, wenn sein Zustand dem vom \mathcal{M}_N gleicht, ist jeder andere Prozess genau dann privilegiert, wenn sein Zustand sich von dem seines Vorgängers im Ring unterscheidet (ein System in dem nicht alle Prozesse identisch sind, bezeichnet man als *nicht uniform*).

Die Transitionsfunktionen δ_i der Prozesse sind bestimmt durch

$$\delta_i(q_i) = q_{i-1}, \forall i > 0$$

$$\delta_0(q_0) = (q_{N-1} + 1) \bmod K$$

Nach einer Transition übernehmen alle Prozesse $\mathcal{M}_i, i > 0$ also den Zustand ihres Vorgängers. Der Zustand \mathcal{M}_0 hingegen wechselt in einen anderen, jedoch eindeutig bestimmten, Zustand als sein Vorgänger.

2.6.3 Analyse

Angenommen das System befindet sich in einem legitimen Zustand und es existiert ein privilegierter Prozess, dann gilt entweder

1. der privilegierte Prozess ist \mathcal{M}_0 . Dann hat \mathcal{M}_{N-1} also den selben Zustand

$$q_{N-1} = q_0$$

Da ansonsten auch kein anderer Prozess privilegiert ist, gilt nach Definition von p_i , dass alle Prozesse den gleichen Zustand haben. Die einzig mögliche Transition überführt den Zustand q_0 von \mathcal{M}_0 in den neuen Zustand

$$(q_{N-1} + 1) \bmod K$$

Im nächsten Schritt ist also \mathcal{M}_0 nicht mehr privilegiert, dafür aber \mathcal{M}_{N-1} . Die einzig mögliche Transition ändert den Zustand q_0 in den Zustand

$$(q_{N-1} + 1) \bmod K \neq q_0$$

Nach der Transition ist der einzige privilegierte Prozess \mathcal{M}_1 (das Token wurde an den Nachfolger gegeben).

2. der privilegierte Prozess ist $\mathcal{M}_i, i \neq 0$, also ist $q_i \neq q_{(i-1) \bmod K}$. Da sonst kein anderer Prozess privilegiert ist, haben alle Prozesse $\mathcal{M}_j, j > i$ den Zustand $q_j = q_i$. Die Prozesse $\mathcal{M}_j, j < i$ haben den Zustand $q_j = q_0$. Die einzig Mögliche Transition bringt \mathcal{M}_i in den Zustand q_{i-1} . Nach der Transition ist der einzige privilegierte Prozess

$$\mathcal{M}_{(i+1) \bmod N}$$

(das Token wurde an den Nachfolger gegeben).

Man sieht dass ein legitimer Zustand nach einer Transition wieder in einem legitimen Zustand resultiert. Bisher wurden bereits die von Dijkstra gestellten Forderungen an die legitimen Zustände befriedigt. Zu zeigen bleibt noch, dass ein solches System von einem illegitimen auf jeden Fall mit einer endlichen Anzahl von Schritten in einen legitimen Zustand überführt wird.

Angenommen das System wird mit beliebigen Zuständen initialisiert, so dass der resultierende Systemzustand illegitim ist. Tel [Tel94] sagt, dass - da der Algorithmus niemals terminiert - der Prozess \mathcal{M}_0 unendlich oft privilegiert ist, denn es können maximal $\frac{N}{2}(N-1)$ Transitionen vollzogen werden, ohne dass \mathcal{M}_0 privilegiert ist. Das

liegt daran, dass ein Prozess, sofern privilegiert, bei einer Transition auf jeden Fall (siehe Definition $\delta_i, i \neq 0$) sein Privileg verliert. Tel stellt hierzu folgende Funktion vor [Tel94]:

Sei $S = \{i \in \mathbb{N} | i > 1 \wedge p_i = 1\}$ die Menge der privilegierten Prozesse mit einer Nummer größer 1, dann können höchstens

$$F = \sum_{i \in S} (N - i)$$

Schritte ohne \mathcal{M}_0 erfolgen.

In der initialen Zustandskonfiguration der N Prozesse können maximal N verschiedene Zustände auftauchen. Da Prozess \mathcal{M}_0 mit jedem Schritt den Zustand seines Vorgängers \mathcal{M}_{N-1} um 1 inkrementiert (modulo K), vergehen maximal N Schritte, bevor q_0 ein Zustand ist, welcher nicht in der anfänglichen Zustandskonfiguration enthalten war. Alle anderen Zustände kopieren nun reihum den neuen, einzigartigen Zustand q_0 . Der Zustand \mathcal{M}_0 ist nun solange nicht privilegiert, bis der legitime Systemzustand $q_0 = q_1 = \dots = q_{N-1}$ erreicht wurde.

Das System erreicht also einen legitimen Zustand spätestens, nachdem \mathcal{M}_0 $N + 1$ Transitionen vollzogen hat. Zwischen diesen Transitionen liegen im schlechtesten Fall $\frac{N}{2}(N-1)$ Transitionen. Folglich erreicht das System, unabhängig von seinem anfänglichen Zustand, nach $O(N^3)$ Transitionen einen legitimen Zustand.

Entsprechend Dijkstras Definition ist dieses System selbststabilisierend.

3 Abweichende Definition

3.1 Selbststabilisierung

Wie bereits erwähnt, ist Dijkstras ursprüngliche Definition von Selbststabilisierung sehr restriktiv. Überwindet man erst einmal das Problem der Nicht-Atomarität von Interprozesskommunikation und Transitionen durch Hinzunahme eines Dämons in das Modell, so ist es möglich eine einfachere, aber gleich mächtige Definition von Selbst-

stabilisierung zu verwenden, welche Beweisprozesse vereinfachen kann.

Betrachtet man den Systemzustand eines Systems S als Tupel der Prozesszustände $S \in Q_0 \times \dots \times Q_N$, so lässt sich das Kriterium, an welchem die Legitimität des Systemzustands fest gemacht wird, als Prädikat $P \subseteq S^N$ formulieren.

Schneider definiert die Eigenschaft der Selbststabilisierung für ein verteiltes System nun wie folgt [Schneider93]:

Wir definieren ein System S als selbststabilisierend bezüglich dem Prädikat P , falls S die folgenden zwei Eigenschaften befriedigt.

1. Abgeschlossenheit - P ist abgeschlossen unter Ausführung von S . Das heißt, sobald PS gilt kann P nicht falsifiziert werden.
2. Konvergenz - Ausgehend von einem beliebigen globalen Zustand ist garantiert, dass S einen legitimen Zustand in einer endlichen Anzahl von Transitionen erreicht.

Schneiders Abgeschlossenheits-Kriterium deckt die zweite und dritte Forderung von Dijkstra ab. Dijkstras erstes Kriterium worin er fordert, dass ein solcher Algorithmus nicht terminieren darf, ist unnötig restriktiv. Schneiders Kriterien fordern implizit allerdings, dass ein selbststabilisierendes System nur dann terminieren darf, falls es sich in einem legitimen Zustand befindet. Dijkstras vierte Forderung, die Menge der legitimen Zustände dürfe nicht in abgeschlossene Teilmengen bezüglich der Ausführung von S zerfallen, ist nach Schneider unnötig.

Schneiders Konvergenz-Kriterium ist äquivalent zu Dijkstras Forderung, ein System müsse nach endlich vielen Schritten legitim sein.

3.2 Stabilisierung

Schneider [Schneider93] führt zusätzlich den verallgemeinerten Begriff von Stabilisierung zurück-

gehend auf Arora [Arora92] ein:

Wir definieren Stabilisierung für ein System S bezüglich den Prädikaten P und Q über die Menge der globalen Zustände. S befriedigt $Q \rightsquigarrow P$ (gelesen als Q stabilisiert zu P) falls folgende Eigenschaften befriedigt werden.

1. Abgeschlossenheit - P ist abgeschlossen unter Ausführung von S . Das heißt, sobald PS gilt kann P nicht falsifiziert werden.
2. Konvergenz - Ausgehend von einem beliebigen globalen Zustand der Q erfüllt ist garantiert, dass S einen legitimen Zustand in einer endlichen Anzahl von Transitionen erreicht.

Schneider verwendet den Begriff der Stabilisierung also, falls ein System unter bestimmten Umständen sein Verhalten stabilisiert. Die Menge der Zustände von welchen aus dies möglich ist, identifiziert das Prädikat Q . Bei einem System, welches stabilisierend aber nicht selbststabilisierend ist, kann es also vorkommen, dass das System konsequent zwischen Zuständen wechselt, die nicht in Q oder P liegen.

Falls es beispielsweise in einer bestimmten Situation ausreichend ist, das ein System stabilisierend ist, kann ein Beweis dieser Eigenschaft unter Umständen durch diese Definition erleichtert werden.

3.3 Erreichbare Zustände

Im Folgenden ist es nützlich Schneiders Idee [Schneider93] von erreichbaren Zuständen aufzugreifen:

Es ist oftmals der Fall, dass beim Schreiben eines Programmes der Autor keine bestimmte Vorstellung davon hat,

welche Zustände legitim sind und welche nicht, das Programm aber designed wird um von einer bestimmten Menge von Startzuständen zu funktionieren.

Schneider verwendet diese Menge von legitimen Startzuständen und erweitert sie unter der fehlerfreien Ausführung des Systems zu der Menge der erreichbaren Zustände. In dieser Menge ist also jeder Systemzustand enthalten, welcher bei fehlerfreier Ausführung des Systems aus einem gültigen Startzustand hervorgehen kann.

Ist die Menge der erreichbaren Zustände bekannt, so muss beispielsweise kein Prädikat mehr angegeben werden, da implizit alle erreichbaren Zustände als legitim angesehen werden. Als Teilmenge aller möglichen Systemzustände entspricht diese Menge sogar formal einem Prädikat P .

Ebenso ist die Menge der erreichbaren Zustände per Definition unter der Ausführung des Systems abgeschlossen.

4 Pros und Kontras

In der Einführung wurde bereits erwähnt, dass sich Selbststabilisierung als Mittel zur Bekämpfung von Fehlern in einem verteilten System eignet. Hierbei stellen selbststabilisierende Algorithmen eine wesentlich elegantere Lösung als die robusten Algorithmen dar. Schneider schreibt hierzu [Schneider93]:

Traditionell wurde jeder dieser Fehler separat behandelt und doch besitzen diese scheinbar unvereinbaren Fehlerphänomene ein gemeinsames Gegenmittel, dass der selbststabilisierenden Algorithmen. [...] Jede Hinzunahme eines Ausnahmefalls kann zwar die die Wahrscheinlichkeit eines Fehlers reduzieren, aber ohne eine formale Basis kann dessen Elimination nicht gewährleistet werden.

Eine solche formale Basis stellt Schneider [Schneider93] zur Verfügung, indem er Dijkstras

Definition von Selbststabilisierung [Dijkstra74] zusammen mit einem klaren Fehlermodell zu einem gemeinsamen Modell vereint.

Natürlich hat das Modell der Selbststabilisierung auch Nachteile. Tel [Tel94] hebt hervor, dass ein selbststabilisierendes System keine passablen Leistungen erbringt, falls das System einen Dienst ununterbrochen zur Verfügung stellen muss.

Ebenfalls kann sich der Prozess der Stabilisierung nach einem Fehler, welcher nur endlich viele Schritte benötigt, über eine immens große Anzahl von Schritten ziehen. Dies bedeutet, dass für eine relativ lange Phase kein konsistenter Systemzustand garantiert werden kann.

Allerdings beschreibt Tel [Tel94], dass sich jeder verteilte Algorithmus der sich unter entsprechender Interpretation in ein „Minimaler-Pfad Problem“ überführen lässt, eine selbststabilisierende Lösung hat. So ist zumindest für eine gewisse Klasse von Algorithmen die Existenz eines äquivalenten selbststabilisierenden Algorithmus gewährleistet.

5 Fehlermodell

Wie eingehend erwähnt kann Selbststabilisierung verwendet werden um fehlertolerantes Verhalten zu implementieren, und zwar tolerant gegenüber sogenannten flüchtigen Fehlern (*transient failures*).

5.1 Flüchtige Fehler

Schneider [Schneider93] grenzt zu Beginn der Vorstellung seines Fehlermodells die Beschaffenheit eines flüchtigen Fehlers klar ein:

Ein flüchtiger Fehler ist ein Ereignis, welches den Zustand des Systems nicht aber sein Verhalten modifiziert.

Ein flüchtiger Fehler kann also Systemzustände korrumpieren, aber nicht das Verhalten des Systems. Die Transitions- und Privilegienfunktionen δ_i und p_i bleiben also unbeeinflusst von

flüchtigen Fehlern. Hervorgerufen werden solche Fehler z.B. durch Umwelteinflüsse (beschädigte Leitungen, fehlerhafte Soft- oder Hardware).

Konkretisiert man das Modell eines verteilten Systems in einem Programm wird klar, dass Umwelteinflüsse nicht nur den Systemzustand im Prozessspeicher korrumpieren können, sondern auch das Programm, welches ebenfalls im Speicher des Prozesses abgelegt ist. Dies verstößt aber gegen Schneiders Einschränkung. Möchte man also mit Schneiders Modell vom flüchtigen Fehler arbeiten, müsste man den Programmcode in einem Read-Only Speicher ablegen, welcher von Umwelteinflüssen unberührt bleibt. Da dies ohne großen Aufwand möglich ist, kann man also das Programmverhalten als unfehlbar annehmen.

Diese Erkenntnis kann man auf Dijkstras anfängliche Erkenntnis beziehen [Dijkstra74]:

Lokale Entscheidungen, basierend auf lokalen Informationen, müssen ein globales Ziel erfüllen.

Flüchtige Fehler entstehen also dadurch, dass eine Entscheidung auf verfälschten lokalen Informationen getroffen wird. Diese Entscheidung führt zu einem verfälschten Systemzustand.

5.2 Der bösertige Gegenspieler

Bringt man eine virtuelle Person in die Lage nach Belieben flüchtige Fehler in einem verteilten System erzeugen zu können, dann nennt man diese Person einen „bösertigen Gegenspieler“. Diese Person versucht nun mit aller Macht das System daran zu hindern sein globales Ziel zu erreichen, indem sie an gewählten Stellen flüchtige Fehler erzeugt. Das Verhalten des Gegenspielers kann sogar bis zu dem Punkt reichen, dass dieser ganze Prozesse dauerhaft lahmlegt (dieser Fall wird dann in der Zustandsmenge der Prozesse mit modelliert).

Dieser Gegenspieler stellt im Prinzip einen Worst-Case Fehlerfall dar, also ein Fehlverhalten des Systems das maximalen Einfluss auf dessen korrekte Ausführung hat.

6 Modellierung fehlertoleranten Verhaltens

Möchte man ein Problem mit einem verteilten Algorithmus berechnen können sich mehrere Vor- und Nachteile ergeben, falls man einen selbststabilisierenden Algorithmus wählt.

Schneider [Schneider93] stellt fünf mögliche Fehlerquellen vor, welche im Bezug auf das vorgestellte Fehlermodell, durch Selbststabilisierung entkräftet werden können.

6.1 Inkonsistente Initialisierung

Ein selbststabilisierendes System darf inkonsistent initialisiert werden (Die lokalen Systemzustände müssen nicht im Bild des globalen Zustandes zusammenpassen, sie können sogar widersprüchlich sein). Nach Dijkstra [Dijkstra74] findet sich das System nach endlich vielen Schritten in einem legitimen Zustand wieder, also benötigt ein System ohne spezielle Initialisierung unter Umständen eine gewisse Zeit um ein korrektes Verhalten zu entwickeln. Hierbei benötigt man dann aber keine Algorithmen für eine verteilte Initialisierung (siehe *distributed reset*).

6.2 Übertragungsfehler

Es kann je nach Implementierung der Prozesskommunikation zu Verlust, Korruption oder Umsortierung von einzelnen Nachrichten zwischen Prozessen kommen. Auch wenn die Prozesse selbst keinen Mechanismus besitzen (siehe *robust algorithms*) um solche Fehler zu beheben, bleibt die Ausführung des System fast unbeeinflusst. Unter der Annahme dass ein solcher Fehler ein legitimes System in einen illegitimen Zustand abrutschen lässt, findet sich nach Dijkstra [Dijkstra74] das System nach endlich vielen Schritten wieder in einem legitimen Zustand wieder.

6.3 Flüchtiger Speicherfehler

Ein Prozess könnte durch Umwelteinflüsse seinen Speicherinhalt verlieren. Der resultierende lokale Zustand kann dann widersprüchlich zum

Rest des Systems stehen. Dank Selbststabilisierung findet sich das System nach einer endlichen Anzahl von Schritten in einem legitimen Zustand wieder.

6.4 Wechsel der Betriebsmodi

Algorithmen können in mehreren Modi oder Stufen betrieben werden (siehe beispielsweise *merge-sort*). Falls der Betriebsmodus eines verteilten Systems geändert werden soll, so ist es nicht allen Prozessen möglich den Moduswechsel gleichzeitig zu vollziehen. Das System befindet sich also in einem Zustand, in dem ein Teil der Prozesse im alten Betriebsmodus operiert, während der Rest bereits in den neuen Modus gewechselt hat. Nachrichten zwischen Nachbarn welche in verschiedenen Betriebsmodi sind können sich wie fehlerhafte Nachrichten auswirken. Aus diesem Grund sorgt die Selbststabilisierung hier ebenfalls für Abhilfe.

6.5 Betriebspausen von Prozessen

Ein Prozess könnte beispielsweise wegen Wartungsarbeiten aus dem Netz genommen werden und später seinen Betrieb fortsetzen. Der Systemzustand, egal ob zwischengespeichert oder willkürlich initialisiert, ist dann inkonsistent mit dem Rest des Systems. Der resultierende widersprüchliche Systemzustand bewegt sich wegen der gewährleisteten Konvergenz jedoch wieder zu einem legitimen Zustand.

7 Selbststabilisierende Algorithmen

Neben Dijkstras Token Ring zur Realisierung gegenseitigem Ausschlusses gibt es bis heute eine Vielzahl von selbststabilisierenden Problemen für verschiedene Probleme.

7.1 Ring-Orientierung

Ein solcher selbststabilisierender Algorithmus wurde 1990 von Israeli und Jalfon [IJ90] eingeführt.

Der Algorithmus operiert auf einem ungerichteten Ring von N Prozessen, bei dem jeder Prozess über zwei Anbindungen verfügt. Das Orientierungsproblem in einem Ring (*ring-orientation problem*) besteht nun darin, jeder Kante von Prozess M_i zu M_j entweder mit „Nachfolger“ oder mit „Vorgänger“ zu beschriften. Der von Israeli und Jalfon [IJ90] vorgestellte verteilte Algorithmus ist selbststabilisierend, auch wenn er unter Umständen nicht terminiert.

Nach Tel [Tel94] befindet sich der Algorithmus, nachdem keine Fehler mehr auftraten, nach maximal N^2 Schritten in einem legitimen Zustand.

7.2 Maximale Paarung

Eine maximale Paarung bezeichnet einen Begriff auf der Graphentheorie. Eine Paarung M eines Graphen G mit

$$G = (V, E), M \subseteq E$$

ist eine Teilmenge der Kanten für die gilt, dass keine zwei Kanten aus M sich einen Knoten teilen. Eine Paarung ist maximal, falls jede Hinzunahme einer Kante dazu führt, dass die Paarungseigenschaft verletzt wird.

Zum Finden einer maximalen Paarung existiert ein sequenzieller Greedy-Algorithmus, welcher allerdings für Anwendung auf einem verteilten System nicht in Frage kommt. 1992 haben Hsu und Huang [HH92] einen verteilten und sogar selbststabilisierenden Algorithmus für dieses Problem vorgestellt.

Der Algorithmus besteht aus drei verschiedenen Teilen, die je nach lokaler Situation auf einem Prozess zur Anwendung gebracht werden. Tel [Tel94] zeigt, dass der von Hsu und Huang vorgeschlagene Algorithmus nach $O(N)$ Schritten in einem legitimen und terminalen Zustand befindet.

7.3 Leiterwahl und Spannbaum Konstruktion

Zusammengesetzt aus dem Problem der Leiterwahl und der Konstruktion eines Spannbaums in einem verteilten System ergibt ein neues Problem. Es kann nötig sein, einen Prozess in einem verteilten System zum Leiter zu ernennen und zugleich eine umfassende Hierarchie in Form eines Spannbaumes auf dem verteilten System einzuführen.

Afek, Kutten und Yung [AKY90] stellten einen selbststabilisierenden Algorithmus vor, welcher beide diese Probleme löst. Der ernannte Leiter im System wird automatisch zur Wurzel des Spannbaums.

Obwohl nicht formal bewiesen beschreiben Afek et. al intuitiv, dass der von ihnen vorgestellte Algorithmus selbststabilisierend und korrekt arbeitet.

8 Entwurf selbststabilisierender Algorithmen

Für den Entwurf eines selbststabilisierenden Algorithmus existiert keine Patentlösung und kein Rezept. Dennoch existieren Methoden, welche den Entwurf vereinfachen. Im folgenden werden einige dieser Methoden nun kurz erläutert.

8.1 Sequenzielle Komposition

Ein Algorithmus versucht in einem System eine Nachbedingung ψ herzustellen. Nun arbeiten viele Algorithmen in Stufen, so dass etwa zuerst eine Zwischenbedingung θ hergestellt wird und danach, unter der Vorbedingung θ die Nachbedingung ψ hergestellt werden kann. Tel [Tel94] gibt einige Beispiele für Algorithmen, welche als mehrstufige Algorithmen Kompositionen von Teilalgorithmen sind:

1. *Routing*. Die meisten Routing Methoden beginnen damit Routing-Tabellen in jedem Prozess zu berechnen, nach denen dann Pakete weitergeleitet werden.

2. *Leiterwahl*. Der Grund für die Wahl eines Leiters ist in der Regel der Wunsch einen zentralisierten Algorithmus in dem Netzwerk auszuführen.
3. *Graphenfärbung*. Eine Sechsfärbung eines Graphen kann berechnet werden indem man eine passende azyklische Orientierung in dem Graphen findet nach der die Knoten, in einer Reihenfolge die mit der Orientierung übereinstimmt, eingefärbt werden können.

Als klassische Lösung den Übergang zwischen zwei Stufen eines verteilten Algorithmus zu koordinieren beschreibt Tel [Tel94] ein Terminations-Aufspürungs Protokoll (*termination detection protocol*). Dieser Ansatz schlägt bei selbststabilisierenden Algorithmen fehl, denn es ist im Allgemeinen für einen Prozess in einem selbststabilisierenden Algorithmus nicht klar, ob die gewünschte Nachbedingung θ der aktuellen Stufe bereits systemweit hergestellt ist.

Nach Tel [Tel94] umgeht man dieses Problem, dadurch dass die zweite Stufe des Algorithmus selbststabilisierend ist. Ist dies gewährleistet, so kann die Ausführung der zweiten Stufe beginnen, auch wenn die Vorbedingung θ noch nicht vollständig hergestellt wurde. Dies wirft die zweite Stufe zwar in einen momentan inkonsistenten Systemzustand, aber nach endlich vielen Schritten stellt sich wieder ein legitimer Zustand ein, da die zweite Stufe selbststabilisierend war.

Ist die Termination der ersten Stufe mit θ feststellbar, dann startet man natürlich die zweite Stufe erst dann, wenn die Termination der ersten Stufe erkannt wurde.

8.2 Der Kompositionsoperator

Bezüglich der Komposition von Ausführungsstufen, formalisierte Herman [Her91] einen Kompositionoperator für Algorithmen. Seien hierzu P_1, P_2 Programm, welche atomare Operationen auf gewissen Variablen darstellen un zwar so, dass die Mengen der von P_1 und P_2 verwendeten Variablen disjunkt sind.

Die sequenzielle Komposition beider Algorithmen notiert man als

$$P_1 \triangleright P_2$$

Der Algorithmus $P_1 \triangleright P_2$ verwendet die Variablen, welche von P_1 oder P_2 verwendet werden und enthält genau die selben atomaren Operationen wie P_1 und P_2 .

P_1 stellt die erste Stufe des Algorithmus dar, welche aus einer Eingabe eine Ausgabe erzeugen. P_2 wird nun auf dieser Ausgabe gestartet mit der Einschränkung, dass die Variablen, welche in P_1 verwendet wurden in P_2 Konstanten sind. P_1 stellt also eine Informationsbasis für P_2 zur Verfügung, während implizit aber gilt, dass P_2 keine Rückmeldung an P_1 macht.

Nun interpretiert man die Zwischenbedingung θ als Prädikat über der Variablenmenge von P_1 und die Nachbedingung ψ als Prädikat über der Variablenmenge von P_2 . Stellt man nun sicher, dass eine gewissen Fairness (beschrieben in Tel [Tel94]) zwischen P_1 und P_2 in der Ausführung von $P_1 \triangleright P_2$ vorliegt, dann lässt sich die Selbststabilisierung von $P_1 \triangleright P_2$ an vier Bedingungen festmachen.

Tel [Tel94] formuliert diese vier Bedingungen folgendermaßen:

1. P_1 stabilisiert zu θ ,
2. P_2 stabilisiert zu ψ , falls θ gilt,
3. P_1 ändert keine Variablen mehr, welche von P_2 gelesen werden sobald θ gilt und
4. alle Programmschritte sind fair für P_1 und P_2 .

8.3 Minimaler-Pfad Probleme

Tel [Tel94] zeigt, dass jedes Problem welches sich in ein „Problem des minimalen Pfades“ umformulieren lässt, eine selbststabilisierende Lösung besitzt. Hierzu präsentiert Tel [Tel94] auch gleich einen selbststabilisierenden Algorithmus für die Berechnung minimaler Pfade in einem verteilten System.

Also kann man den Versuch, ein Problem in ein solchen Graphenproblem umzuformen, als Entwurfsmethode werten. Gelingt die Transformation, so steht nach Tel [Tel94] ein selbststabilisierender Algorithmus zur Verfügung, welcher das Problem löst.

9 Zusammenfassung

Obwohl selbststabilisierenden Algorithmen unmittelbar nach ihrer Einführung durch Dijkstra nur relativ wenig Beachtung geschenkt wurden ist diese Klasse von Algorithmen heute Schwerpunkt einiger aktuellerer Forschungsvorhaben.

Weiter stellen selbststabilisierende Algorithmen ein abstraktes und flexibles Modell für Fehlertoleranz zur Verfügung in dem sich ein System nach einem Fehler, sofern dieser nur von endlicher Dauer ist, selbstständig in einen legitimen Zustand stabilisiert.

Auch wenn die Vielzahl von offensichtlichen Vorteilen welche mit einem selbststabilisierenden Algorithmus verknüpft sind ein überzeugendes Argument ist, ist ein selbststabilisierender Algorithmus nicht die Lösung aller Probleme. Einerseits kann die Verwendung eines selbststabilisierenden Algorithmus mit sehr hohen Entwicklungskosten verknüpft sein, zum anderen gibt es Szenarien in welchen der Dienst den ein System bereitstellt zu keinem Zeitpunkt unterbrochen werden kann. In solchen Situation ist es sicherlich nicht sinnvoll eine Lösung in Form eines selbststabilisierenden Algorithmus anzustreben.

Wenn auch ein selbststabilisierender Algorithmus eine unheimlich elegante Lösung für Fehlertoleranz ist, so hängt es schlussendlich von der gegebenen Situation ab, ob Selbststabilisierung eine Option ist.

Abbildungsverzeichnis

1	Beispielhafte Topologie	4
2	Zentraler Dämon	5
3	Token Ring	7

Literatur

- [Dijkstra74] DIJKSTRA, E. W. 1973. Self-stabilization in spite of distributed control. In *Selected Writing on Computing: A Personal Perspective*. Springer Verlag, Berlin, 1982, 41-46.
- [Schneider93] SCHNEIDER, M. 1993. Self-Stabilization. In *ACM Computing Surveys, Vol. 25, No. 1*.
- [Tel94] TEL, G. 1994. *Introduction to Distributed Algorithms*. Cambridge University Press, Cambridge, 2000.
- [Arora92] ARORA, A 1992. A foundation for fault-tolerant computing. Ph.D. dissertation, Dept. of Computer Sciences, Univ. of Texas at Austin.
- [IJ90] ISRAELI, A. und JALFON, M. Self-stabilizing ring orientation. In *4th Int. Workshop on Distributed Algorithms*, Springer-Verlag, 1990.
- [HH92] HSU, S.-C. und HUANG, S.-T. A self-stabilizing algorithm for maximal matching. *Inf. Proc. Lett.* 43, 1992.
- [AKY90] AFEK, Y., KUTTEN, S. und YUNG, M. Memory-efficient self stabilizing protocols for general networks. In *4th Int. Workshop on Distributed Algorithms*, Springer-Verlag, 1990.
- [Her91] HERMAN, T. Adaptivity through distributed convergence. Ph.D. thesis, Dept. Computer Science, University of Texas at Austin, 1991.