Ausfalldetektoren und das Consensus-Problem im Crash-Recovery-Modell

vom Fachbereich der
Technisch-Naturwissenschaftlichen Fakultät der
Universität zu Lübeck genehmigte

Dissertation

zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften - Dr. rer. nat. -

vorgelegt von Dipl.-Ing. Dipl.-Math.
Martin Zinner

Betreuer: Prof. Dr. Rüdiger Reischuk

Eingereicht: Juni 2005

Inhaltsverzeichnis

1. Erklärung

Hiermit erkläre ich, die vorgelegte Arbeit zur Erlangung des akademischen Grades "Dr. rer. nat." mit dem Titel "Ausfalldetektoren und das Consensus-Problem im Crash-Recovery-Modell" selbständig und ausschließlich unter Verwendung der angegebenen Literatur erstellt zu haben. Ich habe bisher noch keinen Promotionsversuch unternommen und auch nicht andernorts die obige Arbeit vorgelegt bzw. einen Promotionsantrag gestellt.

Dresden, den 21. Juni 2005 Martin Zinner

2. Danksagung

An erster Stelle gilt mein Dank meinem Doktorvater Prof. Dr. Rüdiger Reischuk, der in mühevoller Kleinarbeit mich mit den Feinheiten der wissenschaftlichen Forschung vertraut gemacht hat. Durch die vielen Gespräche, aber besonders durch die vielen Fragen, die Herr Prof. Reischuk gestellt hat, habe ich wichtige Anregungen zur Weiterentwicklung und Vervollkommnung der Arbeit erhalten. Erst durch das Verfassen dieser Arbeit wurde mir bewußt, wieviel Aufwand nötig ist um Begriffe und Sachverhalte klar, genau und unmißverständlich zu definieren.

Herrn Prof. Dr. Schill von der TU Dresden bin ich zu großem Dank verpflichtet, daß er meine ersten Schritte geleitet und betreut hat. Daß es nicht eine Arbeit über Erhöhung der Verfügbarkeit geworden ist, wie ich es vorhatte, ist der bahnbrechenden Arbeiten von Chandra und Toueg zu verdanken.

Herrn Prof. Dr. Reichel von der TU Dresden bin ich dankbar für die Gespräche, die wir geführt haben und insbesondere für die Straffung der Metasprache der Darstellung der Formeln.

Frau Mamat bin ich besonders dankbar für die liebevolle Vermittlung von Terminen bei Herrn Prof. Reischuk, Vermittlung die nicht immer leicht und einfach war.

Meiner Familie, meinen Eltern, meiner Ehefrau und meinen Kindern bin ich besonders dankbar, daß sie in all jenen Jahren viel Verständnis für meine Freizeitbeschäftigung gezeigt haben und für das nötige Arbeitsklima gesorgt haben.

Last but not least möchte ich meinen besonderen Dank an den ehemaligen Leiter des Rechenzentrums des Halbleiterwerkes von Siemens in Dresden und ehemaligen Vorgesetzten, Herrn Jürgen Sander aussprechen. Von Herrn Sander habe ich die Aufgabe erhalten, ein hochverfügbares Rechenzentrum zu entwerfen bzw. die ausgearbeiteten Konzepte umzusetzen. Dadurch entstanden die ersten Gedanken der Dissertation, die ich dann später in meiner Freizeit weiter verfolgt, vervollständigt und ausgearbeitet habe.

3. Schlüsselwörter

agreement protocols, algorithm, availability, commit problem, consensus problem, crash failures, distributed applications, distributed databases, distributed systems, failure detectors, fault tolerance, high availability, non blocking atomic commitment, reliability, uniform consensus

4. Abstract (English Version)

The Consensus problem is a fundamental paradigm for fault tolerant asynchronous systems. It abstracts a family of problems known as Agreement (or Coordination) problems. Any solution to Consensus can serve as a basic building block for solving such problems (e.g. Atomic Commitment or Atomic Broadcast). Solving Consensus in an asynchronous system is not a trivial task, it has been proven (1985) by Fischer, Lynch and Paterson that there is no deterministic solution in asynchronous systems which are subject to even a single crash failure. To circumvent this impossibility result, Chandra and Toueg have introduced the concept of unreliable failure detectors (1991), and have studied how these failure detectors can be used to solve Consensus in asynchronous systems with crash failures.

This work discusses the Consensus problem, and the surveillance of applications in asynchronous distributed systems augmented by unreliable failure detectors. Processes may crash, but cannot recover by themselves. Byzantine failures are not considered.

We first define the failure detectors due to their abstract properties, without taking into account their special implementation. We characterise a class of failure detectors by giving the completeness and accuracy properties. We introduce new properties such as *majority completeness*, *majority accuracy* and *p-majority accuracy*. Majority completeness assures that every process that crashes is eventually suspected by a majority of processes. Majority accuracy assures that no process is ever suspected by a majority of processes before it crashes. Analogously p-majority accuracy assures that there is at least one active (correct) process p that is never suspected by a majority of processes.

The main result of this work is that Uniform Consensus is solvable with failure detectors which satisfy weak completeness. Accuracy properties are not assumed. Suspected processes may be shut-down. Since Consensus and Atomic Broadcast are equivalent problems, the result also applies to Atomic Broadcast. Under the above conditions the Non-Blocking Atomic Commitment problem is reducible to Consensus, and hence it can be solved with failure detectors mentioned above.

If the failure detector satisfies majority completeness and p-majority accuracy then Uniform Consensus can be solved in an environment with a majority of correct processes. Erroneously suspected processes may not be shut-down. We also show that the above failure detector is equivalent to the weakest failure detector $\Diamond\mathcal{W}_0$ identified by Chandra and Toueg to solve Consensus in an environment with a majority of correct processes.

We show that crashed applications can be recovered with failure detectors which satisfy weak completeness and present two recovery algorithms. In order to choose a coordinator and distribute the status vector between replicated processes which monitor the surveillance, some sort of agreement between the participants has to be solved. One of the recovery algorithm uses Consensus explicitely.

In this work we first assume that processes which communicate with each other are completely connected through a set of reliable channels, i.e. channels that do not create wrong (spurious) messages, nor duplicate or lose messages. Later on we weaken our assumption considering fair lossy channels, i.e. channels that can lose messages, but if an infinite number of messages are sent, an infinite subset of these messages are received through these channels. We show that any problem that is solvable with reliable links is also solvable with fair lossy links. The idea underlying this result is that reliable channels can be simulated with fair lossy channels, and hence any algorithm based on reliable channels can be transformed to work with fair lossy channels. Especially, our results regarding Consensus and the Atomic Broadcast problem hold if we use fair lossy links instead of reliable links.

5. Zusammenfassung

5.1. Allgemeines. *Consensus* ist ein fundamentales Paradigma für fehlertolerante asynchrone Systeme. *Consensus* zwingt Prozesse eine gemeinsame Entscheidung zu treffen. Jede Lösung des *Consensus* kann als Ausgangspunkt dienen, um weitere mit *Consensus* verwandte Probleme wie *Atomic Broadcast* oder *(Non-Blocking) Atomic Commitment* zu untersuchen. *Atomic Broadcast* stellt sicher, daß Prozesse Nachrichten zuverlässig verschicken, so daß sie über die Nachrichten, die sie verschicken sowie über deren Reihenfolge, einig sind. Das *Non-Blocking Atomic Commitment*-Problem verlangt von den Teilnehmern, daß sie sich auf den Ausgang einer Transaktion einigen, entweder Commit oder Abort, auch wenn Teilnehmer ausfallen können.

5.2. Deterministische Lösung des *Consensus*. Fischer, Lynch und Paterson haben gezeigt, daß in asynchronen Systemen, in welchen Abstürze von Prozessen zugelassen sind, eine deterministische Lösung des *Consensus* nicht möglich ist, siehe [**31**] sowie [**22**]. Um die Unlösbarkeit des *Consensus* zu umgehen sowie um weitere Ergebnisse in asynchronen Systemen zu ermöglichen, haben Chandra und Toueg unzuverlässige externe Ausfalldetektoren in ihrem Modell eingeführt.

5.3. Modellerweiterung. Wir erweitern das übliche Modell, welches auch von Chandra und Toueg verwendet wurde, indem wir weitere Prozeßzustände berücksichtigen, Netzwerkausfälle zulassen sowie *Recovery* in bestimmten Fällen vorsehen. Absturzverdächtige Prozesse werden gegebenenfalls heruntergefahren.

5.4. Ausfalldetektoren. Vereinfacht dargestellt, führt ein Ausfalldetektor die Liste der abgestürzten Prozesse, die in regelmäßigen Abständen aktualisiert wird. Ein *unzuverlässiger* Ausfalldetektor kann aufgrund von fehlerhaften Eingangsdaten auch falsche Einträge, d.h. nicht abgestürzte Prozesse, in seine Liste aufnehmen. Chandra und Toueg haben mit Hilfe von Ausfalldetektoren das *Consensus*-Problem in asynchronen Systemen gelöst.

5.5. Eigenschaften der Ausfalldetektoren. Wir definieren zuerst die Ausfalldetektoren anhand von abstrakten Eigenschaften, ohne auf die spezielle Implementierung einzugehen. Wir charakte-

risieren eine Klasse von Ausfalldetektoren, indem wir die Vollständigkeits- bzw. Genauigkeitseigenschaften angeben. *Vollständigkeit* (completeness) verlangt, daß ein Ausfalldetektor abgestürzte Prozesse erkennt. *Genauigkeit* (accuracy) schränkt die falschen Verdächtigungen ein, die ein Ausfalldetektor machen darf.

Wir führen neue Vollständigkeits- bzw. Genauigkeitseigenschaften ein wie *mehrheitliche Vollständigkeit*, *mehrheitliche Genauigkeit* sowie *p-mehrheitliche Genauigkeit* und berücksichtigen weitere Eigenschaften wie *starke Vollständigkeit* bzw. *schwache Vollständigkeit* sowie *starke Genauigkeit* und *p-starke Genauigkeit*.

Mehrheitliche Vollständigkeit verlangt, daß jeder abgestürzte Prozeß p schließlich von einer Mehrheit von Prozessen, die p überwachen, als abgestürzt eingestuft wird. *Schwache Vollständigkeit* bzw. *starke Vollständigkeit* stellen analog sicher, daß jeder abgestürzte Prozeß p von wenigstens einem Prozeß bzw. von jedem Prozeß, der p überwacht, verdächtigt wird. Entsprechend garantiert *p-starke Genauigkeit* (*weak accuracy* im Sprachgebrauch von Chandra und Toueg) bzw. *p-mehrheitliche Genauigkeit*, daß ein Prozeß p nie von allen bzw. von einer Mehrheit von Prozessen verdächtigt wird, bevor es abstürzt.

Wir unterscheiden *absolute*, *schließliche* bzw. *kontinuierliche* Genauigkeitseigenschaften. Eine *absolute* Eigenschaft garantiert, daß die betreffende Eigenschaft von Anfang an gilt, eine *schließliche* Eigenschaft stellt sicher, daß die entsprechende Eigenschaft nach endlicher Zeit gilt und eine *kontinuierliche* Eigenschaft läßt zu, daß die entsprechende Eigenschaft von Zeit zu Zeit, aber nicht dauernd verletzt werden kann. Entsprechend stellt z.B. *kontinuierlich starke Genauigkeit* sicher, daß aktive Prozesse nicht dauernd als abgestürzt eingestuft werden.

5.6. Äquivalenz von Ausfalldetektoren. Um das Hauptergebnis dieser Arbeit vorzubereiten, zeigen wir, daß die Ausfalldetektoren, die *mehrheitliche Vollständigkeit* und *absolut* bzw. *schließlich p-mehrheitliche Genauigkeit* erfüllen (Ausfalldetektorklasse $(M, \Box pM)$ bzw. $(M, \Diamond pM)$), äquivalent sind zu den Ausfalldetektoren, die *starke Vollständigkeit* sowie *absolut* bzw. *schließlich p-starke Genauigkeit* erfüllen (Klasse $(S, \Box pS)$ bzw. $(S, \Diamond pS)$). Dies bedeutet insbesondere, daß jedes Problem, welches mit Hilfe von Ausfalldetektoren der Klasse $(M, \Diamond pM)$ lösbar ist, auch mit Hilfe von Ausfalldetektoren der Klasse

$(S, \Diamond pS)$ lösbar ist und umgekehrt.

5.7. *Consensus*. Das Hauptergebnis dieser Arbeit zeigt, daß *Consensus* mit Hilfe von Ausfalldetektoren lösbar ist, die *schwache Vollständigkeit* erfüllen. Genauigkeitseigenschaften werden nicht vorausgesetzt, fälschlicherweise verdächtigte Prozesse werden gegebenenfalls heruntergefahren. Um das Hauptergebnis dieser Arbeit zu beweisen, zeigen wir, daß in unserem Ansatz *schwache* und *starke Vollständigkeit* äquivalent sind. Anschließend stellen wir einen Algorithmus vor, der das *Consensus*-Problem mit Hilfe von Ausfalldetektoren löst, die *starke Vollständigkeit* erfüllen und prüfen die Eigenschaften des *Consensus* nach.

Erfüllt der Ausfalldetektor *mehrheitliche Vollständigkeit* und *schließlich p-mehrheitliche Genauigkeit* (Klasse $(M, \Diamond pM)$), dann ist *Consensus* in einer Umgebung mit einer Mehrheit von aktiven (korrekten) Prozessen lösbar. Fälschlicherweise verdächtigte Prozesse müssen nicht zwangsläufig heruntergefahren werden.

Sei $\Diamond\mathcal{W}_0 \in (W, \Diamond pS)$ der Ausfalldetektor, welcher ausschließlich *schwache Vollständigkeit* und *schließlich p-starke Genauigkeit* erfüllt und keine anderen zusätzlichen Eigenschaften. Chandra und Toueg haben gezeigt, daß $\Diamond\mathcal{W}_0$ der schwächste Ausfalldetektor ist, welcher *Consensus* in einer Umgebung mit einer Mehrheit von aktiven (korrekten) Prozessen löst.
Unser Ergebnis ist im Einklang mit der obigen Feststellung von Chandra und Toueg, denn die Ausfalldetektorklassen $(W, \Diamond pS)$ und $(M, \Diamond pM)$ sind äquivalent.

5.8. *Non-Blocking Atomic Commitment*. Es wurde von Guerraoui in [**32**] gezeigt, daß das *Non-Blocking Atomic Commitment*-Problem (*NB-AC*-Problem) im Modell von Chandra und Toueg nicht lösbar ist. Der Grund dafür sind die unzuverlässigen Informationen über Ausfälle, die von den Ausfalldetektoren im Modell von Chandra und Toueg geliefert werden. Allerdings ist infolge der Annahme, daß verdächtigte Prozesse gegebenenfalls heruntergefahren werden können, das *NB-AC*-Problem in unserem Ansatz lösbar und auf *Consensus* reduzierbar. Erfüllt der Ausfalldetektor *mehrheitliche Vollständigkeit* und *mehrheitliche Genauigkeit*, dann ist das *NB-AC*-Problem lösbar. Prozesse werden in diesem Fall nicht fälschlicherweise ver-

dächtigt und dementsprechend nicht heruntergefahren.

5.9. *Fair Lossy* Kanäle. Eine Verbindung ist zuverlässig, falls keine unechten Nachrichten zugelassen werden und falls der Empfänger jede Nachricht genau einmal erhält. Wir nehmen zuerst an, daß die Prozesse, die miteinander kommunizieren, vollständig durch eine Menge von zuverlässigen bidirektionalen Kommunikationskanälen verbunden sind. Im weiteren Verlauf der Arbeit schwächen wir diese Annahme ab, indem wir *Fair Lossy* Kanäle, d.h. Kanäle die zwar Nachrichten verlieren können, aber unendlich viele Nachrichten weiterleiten, falls sie unendlich viele Nachrichten erhalten haben, zulassen. Wir zeigen, daß jedes Problem, welches mit zuverlässigen Verbindungen lösbar ist, auch mit unzuverlässigen Verbindungen, die *Fair Lossy* Kanäle verwenden, lösbar ist. Dazu verwenden wir einen Algorithmus, welcher eine zuverlässige Verbindung mit Hilfe von *Fair Lossy* Kommunikationskanälen simuliert. Insbesondere bleiben unsere Ergebnisse über *Consensus* und *Atomic Broadcast* bzw. über das *NB-AC*-Problem erhalten, falls wir statt zuverlässigen Verbindungen *Fair Lossy* Kanäle verwenden.

5.10. *Recovery*. Abschließend zeigen wir, daß abgestürzte Applikationen mit Hilfe von Ausfalldetektoren, die *schwache Vollständigkeit* erfüllen, wiederhergestellt werden können und stellen zwei *Recovery*-Algorithmen vor. Fälschlicherweise verdächtigte Applikationen werden heruntergefahren. Um Zustandsinformationen zwischen den Prozessen auszutauschen, die die Applikationen überwachen bzw. um einen Koordinator zu wählen, sollten bestimmte Übereinstimmungsprotokolle zwischen den Teilnehmern erfüllt werden. Einer der *Recovery*-Algorithmen verwendet explizit ein *Consensus*-Protokoll.

Damit wir eine vernüftige Überwachung erzielen können, verlangen wir, daß in unserem System die *starke Vollständigkeit* sowie die *kontinuierlich starke Genauigkeit* erfüllt werden.

KAPITEL 1

Einführung

Die ersten Gedanken, eine Arbeit über Ausfalldetektoren zu schreiben, entstanden in der Zeit als der Autor praktische Erfahrung mit hochverfügbaren Systemen gesammelt hatte. Einen entscheidenden Einfluß auf die Arbeit hatten offene Fragen des *Consensus*-Problems.

Die Genauigkeitseigenschaften der Ausfalldetektoren, die vorausgesetzt werden, um die Eigenschaften des *Consensus* nachzuweisen, können bei den praktischen Implementierungen in den früheren Modellen im allgemeinen nicht garantiert werden.

Im weiteren Verlauf der Einführung wird unser Ansatz dargestellt. Es werden Grundbegriffe wie Prozeß, Applikation, Ausfalldetektor anschaulich beschrieben, sowie die Fehleinschätzungen, die ein Ausfalldetektor vornehmen darf, erörtert. Wir unterscheiden nach ihrer dominaten Eigenschaft Anwendungsprozesse, Überwachungsprozesse sowie Entscheidungsprozesse. Eine Applikation ist entweder ein einzelner Prozeß oder eine Menge von Prozessen, die in natürlicher Weise eine übergeordnete Aufgabe erfüllen. Anschaulich ist ein Ausfalldetektor die Menge aller Überwachungsprozesse. Ein Ausfalldetektor kann aktive Prozesse als abgestürzt sowie umgekehrt, abgestürzte Prozesse als aktiv einstufen. Ausfalldetektoren sind aber nur nützlich, falls wir die Fehleinschätzungen, die ein Ausfalldetektor machen darf, einschränken. Wir werden diese Einschränkungen in den nächsten Kapiteln formal definieren und besprechen.

1. Motivation

1.1. Praxisbezogene Erfahrungen. Der Gedanke, eine Arbeit über Ausfalldetektoren zu schreiben, entstand in der Zeit als der Autor die Aufgabe erhalten hatte ein *hochverfügbares* Rechenzentrum zu entwerfen bzw. die ausgearbeiteten Konzepte umzusetzen.

Ein hochverfügbares System enthält redundante Komponenten und erholt sich von Unterbrechungen schneller als gewöhnliche Systeme (siehe [**56**], bzw die Definition der Meta Group). Bei hochverfügbaren Systemen sind die Benutzer aber von den Unterbrechungen betroffen, die Anwendungen müssen neu gestartet werden, Transaktionen gehen möglicherweise verloren.

Die Hardwareverfügbarkeit von *kommerziellen* Rechnersystemen war schon Mitte der Neunziger Jahre zufriedenstellend, Replikationskonzepte wie z.B. *Clustering* ermöglichen es, Applikationen nach einem Hardwareausfall auf einem anderen Rechner automatisch neu zu starten. Die Firma Stratus bietet vollständig replizierte (fully duplex) Rechner an, neuerdings auch mit HP (Hewlett-Packard) Prozessoren. Dank Einsatz von HP Prozessoren ist es möglich, kommerzielle Betriebssyteme wie HP-UX, auf Stratus Rechnern zu verwenden.

Die Anwendungen im Rechenzentrum fielen meistens infolge von *transienten*[1] *Softwarefehlern* aus. Somit richteten sich unsere Anstrengungen in erster Linie auf die Erhöhung der Softwareverfügbarkeit.

Nach dem Ausfall einer Anwendung haben die Betreuer versucht die abgestürzten bzw. die fehlerhaften Module dieser Anwendung zu ermitteln. Danach wurden diese Module bzw. falls nötig die gesamte Applikation heruntergefahren und anschließend neu gestartet.
Dieser Vorgang war fehlerbehaftet. Zum einen konnten Prozesse, die nicht zur ausgefallenen Applikation gehörten, aus Versehen oder aufgrund falscher Diagnose, beendet werden, zum anderen konnten fehlerhafte Module unentdeckt bleiben, was zur Folge hatte, daß nach dem Neustart die Fehlersuche fortgesetzt wurde. Besonders schwierig war es im Falle von *verteilten* Anwendungen. Hier war manchmal die Zusammenarbeit von mehreren Spezialisten nötig. Auch der Versuch, kommerzielle Ausfalldetektoren einzusetzen, brachte nicht den erwünschten Erfolg.

Die evaluierten Ausfalldetektoren wie *Isis Availability Manager* von Stratus und *Hawk* von Tibco überwachen lediglich den Ausfall von Prozessen, nicht aber den Ausfall von Applikationen. Zwar kann man eindeutig einem Prozeß ein Skript zuordnen, das gestartet werden sollte,

[1] d.h. nicht, oder nur sehr schwer reproduzierbar

falls der entsprechende Prozeß ausfällt, eine *Recovery* von Applikationen ist durch diese einfache Strategie nicht immer gewährleistet.
Meistens ist bei einer Applikation wichtig, in welcher Reihenfolge die Prozesse gestartet werden. Bei den oben erwähnten Ausfalldetektoren kann leicht die Reihenfolge vertauscht werden, falls durch unkontrolliertes Starten von Skripten Teilapplikationen herunter- und wieder hochgefahren werden.

Praxisbezogenes Ziel dieser Arbeit ist es zu untersuchen, welche Eigenschaften Ausfalldetektoren aufweisen müssen, damit sie zur Erhöhung der Verfügbarkeit von verteilten Applikationen eingesetzt werden können.

1.2. Fragestellungen der Theoretischen Informatik. Das Konzept des *unzuverlässigen Ausfalldetektors* (unreliable failure detector) wurde zum ersten Mal von Chandra and Toueg in [**17**] eingeführt. Ein Ausfalldetektor ist unzuverlässig, falls er abgestürzte Prozesse irrtümlich als aktiv einstuft bzw. den Ausfall von Prozessen nicht erkennt. *Consensus* zwingt Prozesse eine gemeinsame Entscheidung zu treffen trotz fehlerhaften Meldungen der Ausfalldetektoren. Eine verteilte Umgebung ist *asynchron*, falls keine zeitlichen Annahmen über die Dauer der Nachrichtenübertragung bzw. über die Dauer der internen Schritte vorausgesetzt werden.

Chandra und Toueg haben mit Hilfe des Ausfalldetektors das *Consensus*-Problem in asynchronen Systemen gelöst.

Den Anstoß, eine Arbeit über das *Consensus*-Problem zu schreiben, gab die Feststellung, daß die Eigenschaften der Ausfalldetektoren, welche Chandra und Toueg verwendet haben um das *Consensus*-Problem zu lösen, in praktischen (industriellen) Systemen nicht sinnvoll implementierbar sind. Wir erweitern deswegen das übliche theoretische Modell, indem wir *Recovery* zulassen. Absturzverdächtige Prozesse werden in unserem Ansatz gegebenenfalls heruntergefahren.

Durch die oben erwähnten Änderungen ergibt sich in natürlicher Weise die Frage, inwieweit in unserem Ansatz *unzuverlässige Ausfalldetektoren* zum Lösen des *Consensus*-Problems verwendet werden können.

2. Grundlagen

2.1. Prozesse. Um die Übersicht zu bewahren sowie um die Beweise verständlicher zu gestalten, haben wir die Prozesse durch bestimmte dominante Eigenschaften gekennzeichnet. Entsprechend unterscheiden wir drei Arten von Prozessen, *Anwendungsprozesse* (application processes), *Überwachungsprozesse* (monitoring processes) und *Entscheidungsprozesse* (decision taking processes). Ein einzelner Prozeß kann mehrere dieser Aufgaben wahrnehmen. Die Anwendungsprozesse sind die Prozesse, die vom Ausfalldetektor überwacht werden. Die Überwachungsprozesse sind die Prozesse des Ausfalldetektors. Sie sammeln die Zustandsinformationen (abgestürzt, aktiv) der Anwendungsprozesse und leiten diese Werte an die Entscheidungsprozesse weiter. Die Entscheidungsprozesse analysieren die Zustandsinformationen und leiten gegebenenfalls eine *Recovery* der Anwendungsprozesse ein.

Um eine bestmögliche Verfügbarkeit des Systems zu erreichen, werden alle ausgefallenen Prozesse gegebenenfalls hochgefahren. Insbesondere werden auch die Überwachungsprozesse bzw. die Entscheidungsprozesse überwacht und gegebenenfalls hochgefahren, falls sie ausfallen.

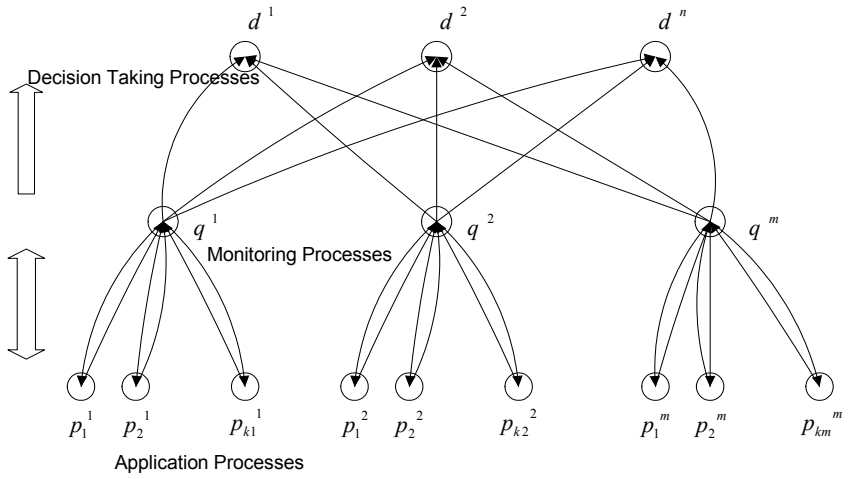


ABBILDUNG 1. Datenfluß

Ein Prozeß p hat sich *anormal* oder *vorzeitig beendet*, falls er sich nicht spezifikationsgemäß beendet hat. Wir sagen: "ein Prozeß *hängt*", falls er nicht mehr spezifikationsgemäß auf äußere Einflüsse reagiert. Somit antwortet ein hängender Prozeß nicht mehr auf ankommende Nachrichten. Im folgenden werden wir nicht mehr zwischen anormal

beendeten und hängenden Prozessen unterscheiden, wir werden von *abgestürzten* Prozessen sprechen.

2.2. Applikationen. Eine Applikation kann anschaulich als ein Baum betrachtet werden, wobei die Knoten weitere Applikationen und die Blätter Prozesse darstellen. Um Hochverfügbarkeit zu erreichen, überwachen wir im allgemeinen Applikationen und nicht ausschließlich nur Prozesse.

Ein Prozeß ist *aktiv*, falls er nicht abgestürzt (spezifikationsgemäß beendet) ist oder sich nicht in der *Recovery*-Phase befindet. Eine Applikation ist *aktiv*, d.h. arbeitet spezifikationsgemäß, falls alle Unterapplikationen aktiv sind.

Somit kann man eindeutig die Knoten im Applikationsbaum identifizieren, die nicht mehr spezifikationsgemäß funktionieren, falls bestimmte Unterapplikationen abstürzen. In unserem Ansatz werden folglich in natürlicher Weise jedem Prozeß p eine oder mehrere Applikationen (Knoten) $A_1, A_2, ..., A_k$ zugeordnet, welche nicht mehr spezifikationsgemäß funktionieren, falls der Prozeß p abstürzt.

Wir lassen nicht zu, daß Prozesse nach einem Absturz selbstständig hochkommen. Würde der Prozeß p nach einem Absturz von selbst hochkommen, dann würden in der Regel die zugewiesenen Applikationen $A_1, A_2, ..., A_k$ nicht mehr spezifikationsgemäß funktionieren. Die *Recovery*-Strategie sieht ein Herunterfahren und einen anschließenden Neustart der Applikationen $A_1, A_2, ..., A_k$ vor. Die eventuell vorhandene Reihenfolge der Prozesse beim Herunter- und Hochfahren sollte berücksichtigt werden.

2.3. Lokales Ausfalldetektormodul. Die Überwachungsprozesse auf einem Rechner, die dieselben Applikationsprozesse überwachen, bilden einen lokalen Ausfalldetektor. Wir verwenden verteilte Ausfalldetektoren, jeder *Applikationsprozeß* wird von einem lokalen Ausfalldetektormodul überwacht.

Jeder Überwachungsprozeß führt eine Liste der überwachten Applikationsprozesse sowie der momentan ermittelten Zustände der jeweiligen Applikationsprozesse. Die Überwachungsprozesse führen keine *History*, d.h. Prozeßzustände von zurückliegenden Zeitpunkten werden von den Überwachungsprozessen nicht aufgehoben.

2.4. Fehleinschätzungen des Ausfalldetektors. Wir lassen zu, daß jeder Überwachungsprozeß q Fehleinschätzungen vornehmen kann,

indem er fälschlicherweise den Zustand von aktiven Prozessen in seine Liste als abgestürzt einträgt, q kann fälschlicherweise den Applikationsprozeß p als abgestürzt betrachten, obwohl p korrekt läuft. Meint q zu einem späteren Zeitpunkt, der Prozeß p würde korrekt laufen, dann stuft er p als aktiv ein.

Außerdem lassen wir zu, daß abgestürzte Prozesse fälschlicherweise als aktiv von Überwachungsprozesse eingestuft werden.

Wir nehmen an, die Überwachungsprozesse und Entscheidungsprozesse sind so sorgfältig getestet worden, daß sie spezifikationsgemäß arbeiten, es sei denn, diese Prozesse stürzen infolge von transienten Fehlern ab. Anwendungsprozesse können sich normal beenden.

Es ist wichtig, daß die Fehler, die ein Ausfalldetektor macht, einen aktiven Prozeß nicht daran hindern, gemäß Spezifikation zu arbeiten, auch wenn dieser Prozeß von allen Überwachungsprozessen fälschlicherweise als abgestürzt eingestuft worden ist, siehe auch [**17**, Seite 227].

2.5. Herunterfahren von verdächtigten Prozessen. Wir lassen zu, daß das System alle aktiven Applikationsprozesse herunterfährt, die fälschlicherweise als abgestürzt eingestuft wurden. Diese zusätzliche Möglichkeit hindert den Prozeß nicht daran, spezifikationsgemäß zu arbeiten. Eine ähnliche Vorgehensweise verwenden auch Ricciardi und Birman, siehe [**55**], der Unterschied liegt in der Überwachungsstrategie. Wir überwachen Applikationen und nicht ausschließlich nur Prozesse, deshalb kann es vorkommen, daß wir auch Applikationsprozesse herunterfahren, die als aktiv erkannt wurden.

2.6. *History*-Auswertung. Um mögliche Fehlentscheidungen von Ausfalldetektoren aufzufangen, führen bzw. werten wir die *History* der Zustandsvektoren aus. Die Entscheidungsprozesse merken sich die zeitlich zurückliegenden Zustände der Applikationsprozesse in der *Zustandsmatrix.* Eine *Recovery* wird aufgrund der *History* gefällt und nicht nur anhand der gerade eingetroffenen Liste der aktuellen Auswertungen. Somit ist sichergestellt, daß während der transienten Phasen keine *Recovery* ausgelöst werden kann. Eine *Recovery* wird erst eingeleitet, falls keine neuen Abstürze in einem bestimmten Zeitraum erfolgen.

KAPITEL 2

Formale Beschreibung des Modells

Um Beweise führen zu können, erstellen wir gegebenenfalls aussagenlogische bzw. prädikatenlogische Formeln für die Begriffe aus dem vorigen Kapitel. Die Terminologie sowie die grundlegenden Definitionen sind der Arbeit [**17**] entnommen. Wir berücksichtigen verteilte asynchrone Systeme, d.h. Systeme in denen keine Zeitbegrenzung vorgegeben wird. Jedes Paar von Prozessen, die miteinander kommunizieren, ist durch einen Kommunikationskanal verbunden.

Wir legen die Zustände fest, die die Prozesse annehmen können und definieren den Systemablauf $\mathcal{S}$ als eine Funktion, die zu jedem Zeitpunkt t die tatsächlichen Zustandswerte der Prozesse angibt. Anschließend definieren wir formal die Zustände über ein Zeitintervall und untersuchen den Zusammenhang zwischen ihnen. Danach definieren wir die Ausfalldetektor-*History* als eine Funktion, die zu jedem Überwachungsprozeß q, zu jedem Applikationsprozess p, die von q überwacht wird und zu jedem Zeitpunkt t, den von q ermittelten Zustand von p angibt. Jetzt sind wir in der Lage, den Ausfalldetektor als eine Funktion zu bestimmen, die jedem Systemablauf $\mathcal{S}$ eine Menge von Ausfalldetektor-*Histories* zuordnet.

Abschließend definieren wir Vollständigkeits- bzw. Genauigkeitseigenschaften eines Ausfalldetektors und vergleichen unseren Ansatz mit dem Modell von Chandra und Toueg.

1. Grundlagen

1.1. Asynchrone Systeme. Wie schon in der Einführung erwähnt, setzen wir keine zeitliche Synchronisation des Systems voraus. Wir berücksichtigen dementsprechend verteilte asynchrone Systeme, bei welchen es keine Zeitbegrenzung gibt hinsichtlich später ankommenden Nachrichten. In diesen Systemen ist Zeitverschiebung durch die Systemuhr zulässig und die Zeit, um einen Schritt auszuführen, ist

nicht begrenzt. Unser Ansatz kann letztendlich auf Fischer zurückgeführt werden, siehe [**31**]. Dieses Modell wurde auch in den Arbeiten von Chandra und Toueg verwendet.

1.2. Aufbau des Systems. Das System Π besteht aus einer endlichen Menge von Prozessen. Die Menge der Applikationsprozesse bezeichnen wir durch App. Analog bezeichnen wir die Menge der Überwachungsprozesse durch Mon bzw. die Menge der Entscheidungsprozesse durch Dec. Sei $p \in App$ beliebig. Die Menge der Überwachungsprozesse, die p überwachen, bezeichnen wir durch $Mon(p)$. Analog bezeichnen wir durch $App\,(q)$ die Menge der Applikationsprozesse, die durch q überwacht werden. Wir bezeichnen durch Mon_q das lokale Ausfalldetektormodul von q. Wie bekannt, bilden die Überwachungsprozesse, die dieselben Applikationsprozesse überwachen ein (lokales) Ausfalldetektormodul. Seien $q_1, q_2 \in Mon(p)$ beliebig. Dann gilt $Mon_{q_1} = Mon_{q_2}$.

Sei $\mathbb{M}$ eine Menge. Wir bezeichnen durch $|\mathbb{M}|$ die Anzahl der Elemente dieser Menge und durch $maj\,(\mathbb{M})$ bezeichnen wir die Menge der Mehrheiten der Elemente dieser Menge. So gilt $M \in maj\,(\mathbb{M}) \iff |M| > \frac{1}{2}\,(|\mathbb{M}| + 1)$.

Um die Führung der Beweise zu erleichtern, nehmen wir an, es gäbe eine diskrete globale Uhr. Die Prozesse haben aber keinen Zugang zu dieser Uhr. Der Wertebereich T der Uhrschläge sei mit $\mathbb{N}$ identisch.

Die Prozesse, die miteinander kommunizieren, sind vollständig durch eine Menge von unidirektionalen Kommunikationskanälen verbunden. Der unidirektionale Kanal, welcher den Prozeß p mit dem Prozeß q verbindet, wird durch $C_{p,q}$ bezeichnet, wobei p der Absender und q der Empfänger ist. Analog bezeichnen wir eine Nachricht m, welche vom Prozeß p an den Prozeß q verschickt wird, durch $m_{p,q}$. Die Menge der Absender bezeichnen wir durch Sen und die Menge der Empfänger bezeichnet wir durch $\mathrm{Re}\,c$. Um eine Nachricht $m_{p,q}$ an den Prozeß q zu veschicken, überreicht der Prozeß p die Nachricht $m_{p,q}$ an den Kommunikationskanal $C_{p,q}$. Die Menge der Nachrichten der Form $m_{p,q}$ bezeichnen wir durch $Mes_{p,q}$.

DEFINITION 1 (**Fair Lossy Channel**). *Ein* Fair Lossy *Kanal $C_{p,q}$ zwischen den Prozessen p und q zeichnet sich durch folgende zwei Eigenschaften aus:*

No Creation: *Erhält der Prozeß q die Nachricht m vom Kanal $C_{p,q}$, dann wurde m von p an den Kanal $C_{p,q}$ überreicht.*

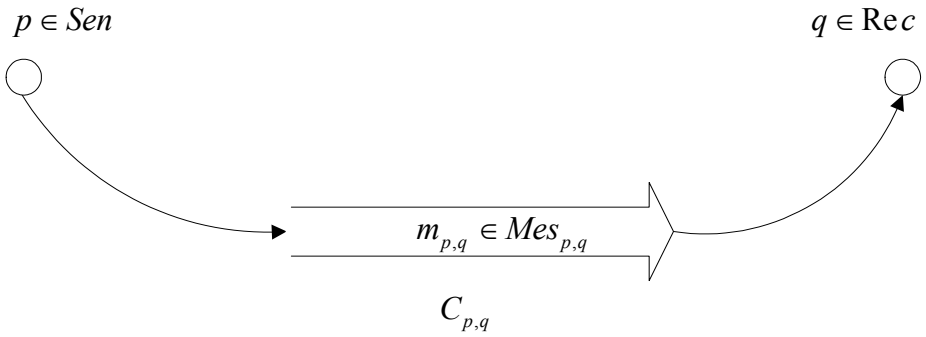


ABBILDUNG 1. Datenfluß

***Fair Loss**: Überreicht p eine unendliche Menge von Nachrichten an den Kanal $C_{p,q}$, dann überreicht der Kanal $C_{p,q}$ eine unendliche Menge von Nachrichten an q.*■

Die Eigenschaft *No Creation* stellt sicher, daß keine korrupten Nachrichten erzeugt werden, Eigenschaft *Fair Loss* sichert die Nützlichkeit des Kanals. Ohne die Eigenschaft *Fair Loss* oder ohne eine ähnliche Eigenschaft wäre jedes wichtige Problem in einer verteilten Umgebung, in der Nachrichten verlorengehen können, trivialerweise unlösbar.

Wir heben im folgenden eine Bemerkungen hervor, die die Kommunikation über Kanäle näher verdeutlicht.

REMARK 1. *Nachrichten können in einem* Fair Lossy *Kanal umgeordnet werden.*■

Fair Lossy Kanäle gewährleisten keine zuverlässige Verbindung, da Nachrichten im Kanal verlorengehen können. Eine *zuverlässige Verbindung* erzeugt keine falschen Nachrichten und jede (echte) Nachricht wird genau einmal zugestellt.

DEFINITION 2 (**Reliable Connection**). *Eine Verbindung ist zuverlässig, falls zusätzlich zur Eigenschaft **No Creation** die folgenden zwei Eigenschaften erfüllt werden:*

***No Duplication**: Überreicht der Absender p eine Nachricht m an den Kanal $C_{p,q}$, dann überreicht der Kanal $C_{p,q}$ die Nachricht m an den Empfänger q höchstens einmal.*

***No Loss**: Überreicht der Absender p eine Nachricht m an den Kanal $C_{p,q}$, dann überreicht der Kanal $C_{p,q}$ die Nachricht m an den Empfänger q wenigstens einmal.*■

Wir nehmen zunächst an, daß in unserem Ansatz die Kommunikation zuverlässig ist. Im weiteren Verlauf der Arbeit schwächen wir diese Annahme insofern ab, daß wir auch unzuverlässige Verbindungen zulassen. Wir geben einen Algorithmus an, welcher eine zuverlässige Verbindung simuliert und *Fair Lossy* Kanäle verwendet.

Prozesse können durch Absturz ausfallen, d.h. Prozesse können vorzeitig beendet werden. Prozesse können aber auch als *nicht vertrauenswürdig* eingestuft werden. Wir sagen: eine Applikation ist nicht vertrauenswürdig, falls es wenigstens einen *nicht vertrauenswürdigen* Prozeß enthält. Unter *Recovery* einer Applikationen verstehen wir das Herunterfahren und anschließendes Hochfahren der betreffenden Applikation. Prozesse, die einmal als *nicht vertrauenswürdig* eingestuft wurden, werden heruntergefahren und werden, solange der Grund für diese Einstufung anhält, während einer *Recovery* nicht mehr hochgefahren. So wird z.B. ein Prozeß, mit welchem im Rahmen des *Consensus*-Algorithmus nicht erwartungsgemäß kommuniziert werden kann, heruntergefahren und während der Ausführung des Algorithmus nicht mehr hochgefahren.

1.3. Zustandswerte. Wir unterscheiden insgesamt folgende Zustände:

$\uparrow$	aktiv
$\dagger$	abgestürzt
$\mp$	als *nicht vertrauenswürdig* eingestuft
$\downarrow$	wurde heruntergefahren
$\nearrow$	im Hochfahren
$\searrow$	im Herunterfahren
$\emptyset$	kein vernünftiges Zusammenfügen möglich
∞	es wurde kein Zustandswert ermittelt

Der Wert $\emptyset$ wird nur von den Entscheidungsprozessen vergeben. Die Entscheidungsprozesse analysieren die verschiedenen Zustandsvektoren, die von den Überwachungsprozessen geliefert werden und ermitteln jeweils für jede Applikation einen gemeinsamen Wert. Liefern die Überwachungsprozesse widersprüchliche Werte, so daß ein plausibler Zustand der betreffenden Applikation sich nicht ermitteln läßt, dann wird $\emptyset$ gesetzt.

Der Wert ∞ wird von den Entscheidungsprozessen vergeben, falls ein Überwachungsprozeß bzw. die ganze Überwachung für einen Prozeß p keinen Zustandswert geliefert hat. Außerdem können Überwachungsprozesse unter bestimmten Umständen den Wert ∞ verschicken, falls man die Abschätzung nicht weiter auswerten möchte. Wird ein Prozeß p von einem Überwachungsprozeß q nicht überwacht, d.h. $q \notin Mon(p)$, so wird unter Umständen der Wert ∞ als Abschätzung von q über den Zustand von p geführt.

Seien:

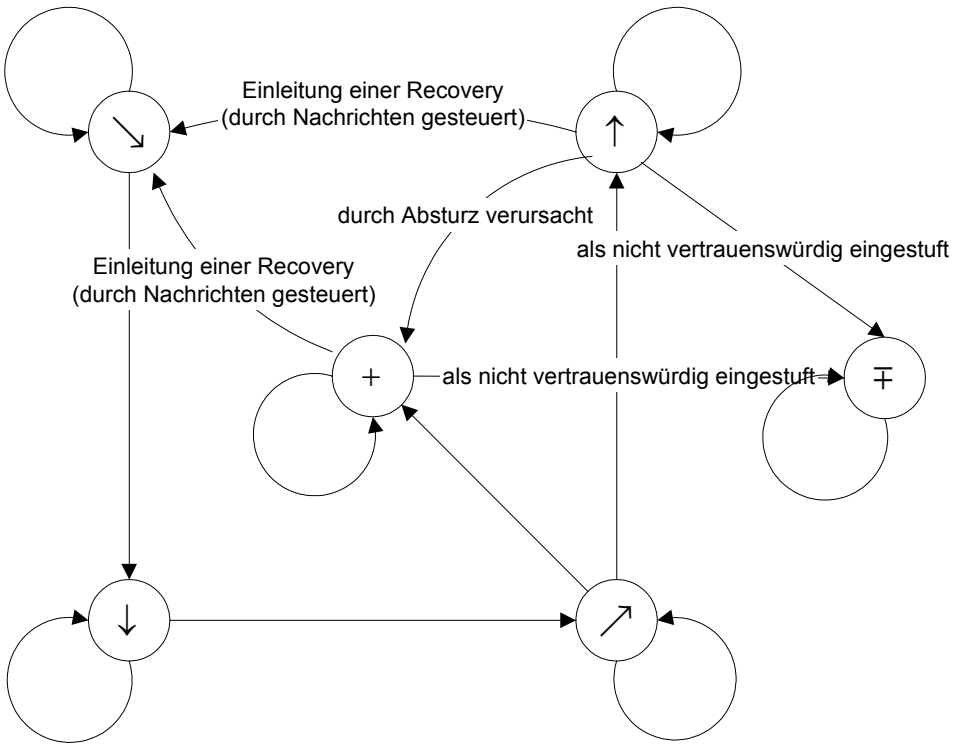
a) $SV_P := \{\uparrow, \dot{+}, \mp\}$ der Wertebereich der Zustandswerte, die von den Überwachungspozessen unterschieden werden können. Entsprechend ermitteln Überwachungsprozesse die Zustände von Applikationsprozessen nur aus diesem Bereich.

b) $SV_P^{\infty} := SV_P \cup \{\infty\}$ der Wertebereich der Zustandswerte, die von den Überwachungspozessen geführt werden können.

c) $SV_A := SV_P \cup \{\downarrow, \nearrow, \searrow\}$ der Wertebereich der Zustandswerte, die von einer Applikation tatsächlich angenommnen werden können.

d) $SV := SV_A \cup \{\emptyset, \infty\}$ der Wertebereich der Zustandswerte, die von den Entscheidungsprozessen bzw. der anschließenden Auswertung für eine Applikation ermittelt werden können.

Im folgenden geben wir, der besseren Übersicht wegen, die möglichen Zustandsübergänge wieder.



Mögliche Zustandsübergänge

1.4. Systemablauf. Wir führen nun einige zentrale Begriffe unserer Arbeit ein.

DEFINITION 3 (**Systemablauf**). *Ein Systemablauf $\mathcal{S}$ gibt zu jedem Zeitpunkt $t \in T$ den tatsächlichen Zustand aller Prozesse $p \in \Pi$ an. Formal ist $\mathcal{S}$ eine Funktion*

$$\mathcal{S} : T \times \Pi \rightarrow SV_A$$

■

Gilt $\mathcal{S}(t, p) = s$, dann besitzt der Prozeß p zum Zeitpunkt t den tatsächlichen Zustandswert s. Die Menge aller Systemabläufe bezeichnen wir durch $\mathbb{S}$.

Im Gegensatz zum Modell von Chandra und Toueg kann in unserem Modell der Zustand eines Prozesses p von außen durch Herunter- bzw.

Hochfahren verändert werden. Dies kann z.B. im Rahmen einer *Recovery* geschehen, aber auch *Agreement*-Algorithmen können Prozeßzustände verändern.

Sei $\mathcal{S}_i$ ein Systemablauf, im folgenden *interner Systemablauf* genannt, welcher von außen nicht beeinflußt wird. Die Menge aller internen Systemabläufe bezeichnen wir durch $\mathbb{S}_i$.

Sei $\mathcal{S}_e$, vereinfacht dargestellt, eine Teilmenge eines Systemablaufs (im folgenden *externe Systemablauf-Korrektur* genannt), die nur die Zustandswechsel enthält, die von außen (durch Nachrichten) verursacht wurden. Formal:

$$\mathcal{S}_e : T \times \Pi \rightarrow SV_A \cup \{\infty\}$$

$$(t,p) \rightarrow \begin{cases} \infty \text{ falls } t = 1 \\ \underline{\text{tatsächlicher}} \text{ Zustand von } p, \text{ falls Zustandswechsel von} \\ t-1 \text{ zu } t \text{ extern verursacht wurde} \\ \mathcal{S}_e(t-1,p) \text{ sonst} \end{cases}$$

Die Menge aller externen Systemablauf-Korrekturen bezeichnen wir durch $\mathbb{S}_e$. Die triviale Korrektur, welche keinen Zustandswechsel verursacht, bezeichnen wir durch $\mathcal{S}_e^{\infty}$.

Es gilt: $\mathcal{S}_e^{\infty}(t,p) = \infty$ für alle $t \in T$ und $p \in \Pi$.

Sei die Verknüpfung $*$ von $\mathcal{S}_i$ und $\mathcal{S}_e$ wie folgt definiert:

$$\mathcal{S}_e * \mathcal{S}_i : T \times \Pi \rightarrow SV_A$$

$$(t,p) \rightarrow \begin{cases} \mathcal{S}_i\,(t,p) \text{ falls } \mathcal{S}_e\,(t,p) = \infty \\ \mathcal{S}_e\,(t,p) \text{ sonst} \end{cases}$$

Dann ist $\mathcal{S}_e * \mathcal{S}_i$ ein Systemablauf, d.h. $\mathcal{S}_e * \mathcal{S}_i \in \mathbb{S}$.

1.5. Prozesse. Sei $t_0, t_c \in T$ mit $t_0 \leq t_c$. Sei

$$activ\,(\mathcal{S})_{t_0,t_c} := \cap_{t_0 \leq t \leq t_c} \{p \in \Pi \mid \mathcal{S}\,(t,p) = \uparrow\}$$

die Menge der aktiven Prozesse im Zeitintervall $[t_0, t_c]$, sei

$$rec\,(\mathcal{S})_{t_0,t_c} := \cup_{t_0 \le t \le t_c}\{p \in \Pi \mid (\mathcal{S}\,(t,p) = \nearrow) \vee (\mathcal{S}\,(t,p) = \searrow) \vee (\mathcal{S}\,(t,p) = \downarrow)\}$$

die Menge der Prozesse, die sich im Zeitintervall $[t_0, t_c]$ irgendwann im *Recovery*-Zustand befanden, sei

$$crashed\,(\mathcal{S})_{t_0,t_c} := \cup_{t_0 \le t \le t_c}\{p \in \Pi \mid \mathcal{S}\,(t,p) = \dot{+}\}$$

die Menge aller Prozesse, die irgendwann im Zeitintervall $[t_0, t_c]$ abgestürzt sind und sei

$$no\_trust\,(\mathcal{S})_{t_0,t_c} := \cup_{t_0 \le t \le t_c}\{p \in \Pi \mid \mathcal{S}\,(t,p) = \mp\}$$

die Menge aller Prozesse, die irgendwann im Zeitintervall $[t_0, t_c]$ als *nicht vertrauenswürdig* eingestuft wurden.

Da *nicht vertrauenswürdige* Prozesse nicht mehr hochgefahren werden (wie wir dies im weiteren Verlauf der Arbeit präzisieren werden), gilt

$$no\_trust\,(\mathcal{S})_{t_0,t_c} := \{p \in \Pi \mid \mathcal{S}\,(t_c,p) = \mp\}$$

Es gilt

$$activ\,(\mathcal{S})_{t_0,t_c} = \Pi - rec\,(\mathcal{S})_{t_0,t_c} - crashed\,(\mathcal{S})_{t_0,t_c} - no\_trust\,(\mathcal{S})_{t_0,t_c}$$

Um die Darstellungsweise zu vereinfachen, setzten wir:

$$activ\,(\mathcal{S})_{t_0} := \cap_{t_0 \le t < \infty}\{p \in \Pi \mid \mathcal{S}\,(t,p) = \uparrow\}$$

bzw.

$$perm\_activ\,(\mathcal{S}) := activ\,(\mathcal{S})_0$$

Analoge Bezeichnungen gelten für $rec(\mathcal{S})$ sowie für $crashed(\mathcal{S})$.

Falls $p \in crashed(\mathcal{S})$, sagen wir: "$p$ stürzt ab in $\mathcal{S}$" und falls $p \in perm\_activ(\mathcal{S})$, sagen wir: "$p$ ist permanent aktiv in $\mathcal{S}$".

Um unkontrolliertes Hochfahren von Prozessen zu vermeiden, setzen wir fest:

AXIOM 1 (**No Self Recovery**). *Ein abgestürzter Prozeß kommt von selbst nicht hoch, er kann nur vom Überwachungssystem während einer* Recovery *hochgefahren werden. Formal:*

$$\forall t \in T \;.\; \forall p \in \Pi \;.\; \mathcal{S}(t,p) = \dot{+} \implies \mathcal{S}(t+1,p) \in \{\dot{+}, \searrow\}$$

■

Damit die Überwachung nicht zusammenbricht, verlangen wir:

AXIOM 2 (**No Total Crash**). *Bei jedem Systemablauf $\mathcal{S}$ ist immer wenigstens ein Entscheidungsprozeß aktiv, d.h.*

$$\forall t \in T \;.\; \exists d \in Dec \;.\; \mathcal{S}(t,d) = \uparrow$$

■

1.6. Ermittlung von Zuständen. Eine Ausfalldetektor-*History* gibt zu jedem Applikationsprozeß $p \in App$, zu jedem Überwachungsprozeß $q \in Mon(p)$ und zu jedem Zeitpunkt $t \in T$ den von q ermittelten Zustand von p zum Zeitpunkt t an.
Formal:

DEFINITION 4 (**Ausfalldetektor-History**). *Eine Ausfalldektor-*History *H ist eine Funktion:*

$$H : Mon \times App \times T \to SV_P^\infty$$

■

Gilt $H(q,p,t) = \dot{+}$, dann sagen wir: "*p wird von q zum Zeitpunkt t in H verdächtigt*".

REMARK 2.
a) Liefert $q \in Mon$ keine Abschätzung über den Zustand von $p \in App$ zum Zeitpunkt $t \in T$, z.B. weil p von q nicht überwacht wird oder weil q abgestürzt ist, so setzen wir $H(q,p,t) = \infty$.

b) Ist q zum Zeitpunkt t aktiv, so gilt $H(q,p,t) \in SV_P$

c) $H(q,p,t)$ ist die Einschätzung von q über $S(t,p)$.■

Möchten wir uns nicht auf eine Komponente festlegen, so setzen wir " . " statt dieser Komponente.
So bedeutet z.B. $H(q,p,.) = \dot{+}$, daß die Einschätzung $\dot{+}$ von q über den Zustand von p zu einer nicht näher bestimmten Zeit erfolgt.

Im folgenden Beispiel geben wir tabellarisch die Zustände von zwei ausgewählten Prozessen $p \in App$ und $q \in Mon(p)$ sowie die Einschätzung von q über den Zustand von p zu aufsteigenden Zeiten t_0, $t_1, ..., t_{13}$ wieder. Man bemerkt, daß p zu unrecht vom Prozeß q zum Zeitpunkt t_6 verdächtigt wird, d.h. es gilt $H(p,q,t_6) = \dot{+}$, obwohl p zum Zeitpunkt t_6 aktiv ist.

T	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}
q	$\uparrow$	$\uparrow$	$\dot{+}$	$\searrow$	$\downarrow$	$\nearrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\dot{+}$	$\searrow$	$\downarrow$
p	$\uparrow$	$\dot{+}$	$\dot{+}$	$\searrow$	$\downarrow$	$\nearrow$	$\uparrow$	$\searrow$	$\downarrow$	$\nearrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$
H	$\uparrow$	$\dot{+}$	∞	∞	∞	∞	$\dot{+}$	$\dot{+}$	$\dot{+}$	$\dot{+}$	$\uparrow$	∞	∞	∞

Die Menge aller Ausfalldetektor-*Histories* bezeichnen wir durch $\mathbb{H}$.

Informell liefert ein Ausfalldetektor $\mathbb{D}$ (auch inkorrekte) Informationen über den Systemablauf $\mathcal{S}$.

Formal:

DEFINITION 5 (**Ausfalldetektor**). *Ein Ausfalldetektor $\mathbb{D}$ ist eine Funktion, welche jeden Systemablauf $\mathcal{S}$ auf eine Menge von Ausfalldetektor-*Histories $\mathbb{D}(\mathcal{S})$ *abbildet.*

$$\mathbb{D} : \mathbb{S} \rightarrow 2^{\mathbb{H}}$$

■

$\mathbb{D}(\mathcal{S})$ ist die Menge aller Ausfalldetektor-*Histories*, die in der Ausführung mit dem Systemablauf $\mathcal{S}$ und dem Ausfalldetektor $\mathbb{D}$ vorkommen. Wir werden im Laufe der Arbeit die Menge der Ausfalldetektor-*Histories* $\mathbb{D}(\mathcal{S})$ eines Ausfalldetektors $\mathbb{D}$ durch Anforderungen an das System einschränken.

Üblicherweise bezeichnen wir durch $H_{\mathbb{D}}$ eine Ausfalldetektor-*History*[1] des Ausfalldetektors $\mathbb{D}$. Um den Sprachgebrauch zu vereinfachen, werden wir im folgenden von einer *History* des Ausfalldetektors sprechen.

Sei q ein beliebiger Überwachungsprozeß und sei p ein beliebiger Prozeß, welcher von q überwacht wird. Wir bezeichnen durch $\mathbb{D}_q$ **das lokale Ausfalldetektormodul** von q. Wird p von q zum Zeitpunkt t verdächtigt, d.h. $H(q,p,t) = \dot{+}$, so setzen wir $p \in \mathbb{D}_q(t)$. Ist die jeweils zuletzt abgegebene Abschätzung von q über p gleich $\dot{+}$, d.h. p wird von q verdächtigt, so setzen wir $p \in \mathbb{D}_q$.

Gilt $p \in \mathbb{D}_q$ zum Zeitpunkt t', so gibt es einen Zeitpunkt t_0, so daß $p \in \mathbb{D}_q(t_0)$ und zu jedem Zeitpunkt $t \in T\ .\ (t_0 \leq t \leq t')$ gilt $H(q,p,t) \in \{\dot{+}, \infty\}$.

Wie früher festgelegt, bezeichnen wir durch $maj(Mon(p))$ die Menge der Mehrheiten $(> \frac{1}{2}(|Mon(p)|+1))$ der Elemente dieser Menge.

Definition 6 (**Aktive Mehrheit**). *Sei $\mathcal{S}$ ein Systemablauf und sei $\mathbb{M} \subset Mon$ eine Menge von Überwachungsprozessen. Wir bezeichnen durch $act\_maj(\mathbb{M})$ eine* aktive Mehrheit *von Überwachungsprozessen, d.h. $M \in act\_maj(Mon(p))$ genau dann, wenn:*

$$M \in maj(\mathbb{M}) \iff \forall q \in M\ .\ q \in perm\_activ(\mathcal{S})$$

Entsprechend bezeichnen wir durch $even\_act\_maj(\mathbb{M})$ eine schließlich aktive Mehrheit*, d.h.*

[1] In der Regel gibt es mehrere Ausführungen mit demselben Systemablauf $\mathcal{S}$. Die Ausführungen können sich im Zeitverhalten der Nachrichten unterscheiden. Dementsprechend kann $\mathbb{D}(\mathcal{S})$ für jede Ausführung eine andere Ausfalldetektor-*History* haben.

$$M \in even\_act\_maj\,(\mathbb{M}) \iff \exists t' \in T\ .\ \forall q \in M\ .\ q \in activ\,(\mathcal{S})_{t'}$$

■

DEFINITION 7 (**Funktioneller Überwachungsprozeß**). *Ein Überwachungsprozeß q heißt funktionell bezüglich $p \in App$ im Systemablauf $\mathcal{S}$, falls q unendlich viele Abschätzungen über p abgibt, d.h. $H\,(q,p,t) \neq \infty$ für unendlich viele $t \in T$.*■

Die Menge aller funktionellen Überwachungsprozesse bezüglich p bezeichnen wir durch $Func\,(\mathcal{S},p)$.

AXIOM 3 (**No Complete Monitoring Crash**). *Die Überwachung der Prozesse bricht nicht zusammen, d.h. $\forall \mathcal{S} \in \mathbb{S}\ .\ \forall p \in App\ .\ Func\,(\mathcal{S},p) \neq \emptyset$.*■

Das obige Axiom ergänzt Axiom (2). Während einer *Recovery* werden gegebenenfalls abgestürzte Überwachungsprozesse hochgefahren, es sei denn, diese Prozesse sind *nicht vertrauenswürdig.*

AXIOM 4 (**Trustworthness**). Nicht vertrauenswürdige *Prozesse werden nicht hochgefahren.*■

Um die Schreibweise kompakt und verständlich zu halten, werden wir im folgenden, falls dies nicht zu Mißverständnis führt, statt der formalen Schreibweise $\forall t \in T\ .\ (t_1 \leq t)$, die kürzere Form $\forall t \geq t_1$ verwenden. Eine ähnliche Schreibweise wird auch für den Existenzquantor $\exists$ eingeführt.

Sei E eine Eigenschaft und sei $t \in T$. Wir setzen $E\,(t)$ und meinen, die Eigenschaft E gilt für den Zeitpunkt t.
Wir sagen: "eine Eigenschaft E gilt *schließlich*", falls:

$$\exists t_E \in T\ .\ \forall t \geq t_E\ .\ E\,(t)$$

NOTATION 1 (**Correct Failure Detection**). *Wir sagen: "der Überwachungsprozeß $q \in Mon\,(p)$ erkennt den Ausfall des Applikationsprozesses p im Systemablauf $\mathcal{S}$ und der* History $H \in \mathbb{D}(\mathcal{S})$ *zum Zeitpunkt t" (Prädikat $correct\_failure\_\det ec(\mathcal{S},H,q,p,t)$), falls p*

von q zum Zeitpunkt t verdächtigt wird. Formal:

$$\begin{aligned} correct\_failure\_\det ec(\mathcal{S},H,q,p,t) \quad &\equiv \\ [\ \mathcal{S}(t,p) \ &= \ \dot{+} \implies H(q,p,t) = \dot{+}\ ] \end{aligned}$$

■

NOTATION 2 (**False Suspection**). *Wir sagen: "der Applikationsprozeß p wird vom Überwachungsprozeß $q \in Mon(p)$ im Systemablauf $\mathcal{S}$ und der* History *$H \in \mathbb{D}(\mathcal{S})$ zum Zeitpunkt t fälschlicherweise verdächtigt" (Prädikat $false\_susp(\mathcal{S},H,q,p,t)$), falls p zum Zeitpunkt t aktiv ist und falls p zum Zeitpunkt t von q verdächtigt wird. Formal:*

$$false\_susp(\mathcal{S},H,q,p,t) \ \equiv \ [\ \mathcal{S}(t,p) = \uparrow \ \wedge \ H(q,p,t) = \dot{+}\ ]$$

■

NOTATION 3 (**Permanent Crashed**). *Wir sagen: "der Applikationsprozeß p ist im Systemablauf $\mathcal{S}$ ab dem Zeitpunkt t_0* permanent *abgestürzt" (Prädikat $perm\_crash(\mathcal{S},p,t_0)$), falls p im Systemablauf $\mathcal{S}$ zum Zeitpunkt t_0 abgestürzt ist und sein Zustand sich nicht mehr ändert. Formal:*

$$perm\_crash(\mathcal{S},p,t_0) \ \equiv \ [\ \forall t \geq t_0\ .\ \mathcal{S}(t,p) = \dot{+}\ ]$$

■

NOTATION 4 (**Eventually Permanent Suspected**). *Wir sagen: "der Prozeß p wird vom Überwachungsprozeß $q \in Func(\mathcal{S},p)$* permanent *im Systemablauf $\mathcal{S}$ und der* History *$H \in \mathbb{D}(\mathcal{S})$ ab dem Zeitpunkt t_0 schließlich verdächtigt" (Prädikat $event\_perm\_susp(\mathcal{S},H,q,p,t_0)$), falls der Prozeß p im Systemablauf $\mathcal{S}$ und der* History *H ab einem bestimmten Zeitpunkt $t' \geq t_0$ vom Prozeß q zu jedem beliebigen Zeitpunkt $t \geq t'$, wenn überhaupt dann als abgestürzt eingestuft wird. Formal:*

$$\begin{aligned} event\_perm\_susp(\mathcal{S},H,q,p,t_0) \quad &\equiv \quad [\ q \in Func(\mathcal{S},p)\ .\ \exists t' \geq t_0\ . \\ &\qquad \forall t \geq \ t'\ .\ H(q,p,t) \in \{\dot{+},\infty\}\ ] \end{aligned}$$

■

REMARK 3. *Gemäß Umformungsregeln gilt:*

$$\neg\ correct\_failure\_\det ec(\mathcal{S}, H, q, p, t) \quad \equiv$$
$$[\ \mathcal{S}(t,p) \ = \ \dot{+} \ \wedge \ H(q,p,t) \in \{\uparrow, \infty\}\ ]$$

und

$$\neg\ false\_susp(\mathcal{S}, H, q, p, t) \quad \equiv$$
$$[\ \mathcal{S}(t,p) \ = \ \uparrow \implies H(q,p,t) \in \{\uparrow, \infty\}\ ]$$

■

Modellannahme 1 (**Monitoring Inconsistency**) *Verschiedene Überwachungsprozesse eines Ausfalldetektors* $\mathbb{D}$ *können verschiedene Einschätzungen über den Zustand eines Applikationsprozesses* p *zu einem beliebigen Zeitpunkt* t *liefern, d.h.:*
es kann $\mathcal{S} \in \mathbb{S}$, $H \in \mathbb{D}(\mathcal{S})$, $p \in App$, $q_1, q_2 \in Mon(p)$ *und einen Zeitpunkt* $t \in T$ *geben, so daß* $H(q_1, p, t) \neq H(q_2, p, t)$ gilt.■

Modellannahme 2 (**Faulty Behaviour**) *Überwachungsprozesse eines Ausfalldetektors* $\mathbb{D}$ *können inkorrekte Einschätzungen liefern, indem abgestürzte Prozesse nicht verdächtigt werden bzw. aktive Prozesse als abgestürzt eingestuft werden, d.h.:*
es kann $\mathcal{S} \in \mathbb{S}$, $H \in \mathbb{D}(\mathcal{S})$, $p \in App$, $q \in Mon(p)$ *und Zeitpunkte* $t_1, t_2 \in T$ geben*, so daß* $H(q, p, t_1) \neq \infty$ und

$$\neg\ correct\_failure\_\det ec(\mathcal{S}, H, q, p, t_1)$$

bzw.

$$false\_susp(\mathcal{S}, H, q, p, t_2)$$

gilt.■

Modellannahme 3 (**Correct Behaviour**) *Überwachungsprozesse eines Ausfalldetektors* $\mathbb{D}$ ermitteln korrekt den Zustand eines *nicht vertrauenswürdigen* Prozesses*, d.h.:*

$$\forall t \ \in \ T \ .\ H(q,q,t) \ .\ \forall p \in App \ .\ \forall q \in Mon(p) \ .$$
$$\mathcal{S}(t,p) \ = \ \mp \implies H(q,p,t) = \mp$$

■

AXIOM 5 (**No Self-Suspection**). *Überwachungsprozesse eines Ausfalldetektors $\mathbb{D}$ können sich selbst nicht verdächtigen, d.h.: für alle $\mathcal{S} \in \mathbb{S}$, $H \in \mathbb{D}(\mathcal{S})$, $q \in App \cap Mon\,(q)$ und $t \in T$ gilt:*

$$H\,(q,q,t) \;\neq \infty \implies \neg\; false\_susp(\mathcal{S},H,q,q,t)$$

■

REMARK 4. *Da $H\,(q,p,t) = \infty$ genau dann, wenn q abgestürzt ist oder q den Prozeß p nicht überwacht (siehe auch Bemerkung (2) sowie Modellannahme 3), gilt:*

$$\neg\; false\_susp(\mathcal{S},H,q,p,t) \implies$$
$$[\;(q \in Mon\,(p) \;\wedge\; \mathcal{S}\,(t,p) = \uparrow \;\wedge\; \mathcal{S}\,(t,q) = \uparrow) \implies H\,(q,p,t) = \uparrow\;]$$

■

1.7. Hochfahren von Prozessen. Abgstürzte Prozesse bzw. Applikationen werden im Rahmen eines *Recovery*-Verfahrens hochgefahren, allerdings gilt dies nicht für *nicht vertrauenswürdige* Prozesse bzw. Applikationen.

Wir können in unserem Modell nicht sicherstellen, daß hochgefahrene Prozesse in der Lage sind weitere Schritte auszuführen.

Wie nehmen an, daß die Prozesse über einen stabilen Speicher verfügen, wohin sie ihre Zustandsinformationen während eines Prozeßabsturzes retten können.

2. Eigenschaften des Ausfalldetektors

Wir spezifizieren den Ausfalldetektor $\mathbb{D}$ anhand von zwei abstrakten Eigenschaften, Vollständigkeit (completeness) und Genauigkeit (accuracy). Dies erlaubt uns die Beweise ausgehend von diesen Eigenschaften zu führen. Somit werden spezifische Implementierungsdetails in den Beweisen nicht berücksichtigt. Die Genauigkeitseigenschaften teilen wir in drei große Klassen ein: *absolute Eigenschaften*, *Sicherheitseigenschaften* (safety properties) und *Lebendigkeitseigenschaften* (liveness properties). Die *absoluten Eigenschaften* sind zeitunabhängig, sie gelten für alle Zeitpunkte $t \in T$, eine *Sicherheitseigenschaft* besagt, daß ein bestimmtes Prädikat in jedem erreichbaren globalen Zustand,

d.h. quasi immer gilt. Auf der anderen Seite stellt eine *Lebendigkeitseigenschaft* sicher, daß ab einem nicht näher bestimmbaren Zeitpunkt $t \in T$ ein Prädikat für immer gilt.

DEFINITION 8 (**starke Vollständigkeit**). *Jeder permanent abgestürzte Applikationsprozeß p wird schließlich von allen bezüglich p funktionellen Überwachungsprozessen q permanent verdächtigt. Formal erfüllt* $\mathbb{D}$ starke Vollständigkeit*, falls:*

$$\begin{aligned} \forall \mathcal{S} \in \; & \mathbb{S} \;.\; \forall H \in \mathbb{D}(\mathcal{S}) \;.\; \forall p \in App \;.\; \forall t \in T \;. \\ & perm\_crash(\mathcal{S}, p, t) \Longrightarrow \\ \forall q \in \; & Func\,(\mathcal{S}, p) \;.\; event\_perm\_susp(\mathcal{S}, H, q, p, t) \end{aligned}$$

■

Im Gegensatz zur *starken Vollständigkeit*, verlangt *schwache Vollständigkeit*, daß mindestens ein in Frage kommender Überwachungsprozeß einen abgestürzten Prozeß verdächtigt.

DEFINITION 9 (**schwache Vollständigkeit**). *Jeder permanent abgestürzte Applikationsprozeß p wird schließlich von wenigstens einem bezüglich p funktionellen Überwachungsprozeß q permanent verdächtigt. Formal erfüllt* $\mathbb{D}$ schwache Vollständigkeit*, falls:*

$$\begin{aligned} \forall \mathcal{S} \in \; & \mathbb{S} \;.\; \forall H \in \mathbb{D}(\mathcal{S}) \;.\; \forall p \in App \;.\; \forall t \in T \;. \\ & perm\_crash(\mathcal{S}, p, t) \Longrightarrow \\ \exists q \in \; & Func\,(\mathcal{S}, p) \;.\; event\_perm\_susp(\mathcal{S}, H, q, p, t) \end{aligned}$$

■

Formal erhalten wir *schwache Vollständigkeit*, indem wir $\forall q \in Func\,(\mathcal{S}, p)$ in der Formel für *starke Vollständigkeit* durch $\exists q \in Func\,(\mathcal{S}, p)$ ersetzen. Dabei ist gemäß Axiom (3) die Menge $Func\,(\mathcal{S}, p)$ nicht leer. Somit impliziert *starke Vollständigkeit* trivialerweise *schwache Vollständigkeit*.

Wie wir zeigen werden, sind in unserem Ansatz *starke* und *schwache Vollständigkeit* äquivalent.

Wir führen noch eine weniger abgeschwächte Form von *starker Vollständigkeit* ein, und zwar die *mehrheitliche Vollständigkeit*. Sei p ein

permanent abgestürzter Applikationsprozeß. *Mehrheitliche Vollständigkeit* verlangt, daß die Mehrheit der bezüglich p funktionellen Überwachungsprozesse den Prozeß p permanent verdächtigen.

DEFINITION 10 (**mehrheitliche Vollständigkeit**). *Jeder permanent abgestürzte Applikationsprozeß p wird schließlich von einer Mehrheit der bezüglich p funktionellen Überwachungsprozesse permanent verdächtigt. Formal erfüllt* $\mathbb{D}$ mehrheitliche Vollständigkeit*, falls:*

$$\begin{aligned} \forall \mathcal{S} \in\ & \mathbb{S} \,.\, \forall H \in \mathbb{D}(\mathcal{S}) \,.\, \forall p \in App \,.\, \forall t \in T \,.\, \exists M \in act\_maj\,(Mon\,(p)) \;. \\ & perm\_crash(\mathcal{S}, p, t) \Longrightarrow \\ \forall q \in\ & M \,.\, event\_perm\_susp(\mathcal{S}, H, q, p, t) \end{aligned}$$

■

Vollständigkeit allein ist nicht ausreichend. Um dies zu sehen, nehme man einen Ausfalldetektor, wo jeder Prozeß jeden anderen Prozeß im System verdächtigt, siehe auch [**17**]. Dieser Ausfalldetektor erfüllt trivialerweise *starke Vollständigkeit*, ist aber nicht nützlich, denn er liefert keine Informationen über den Zustand der Prozesse. Um nützlich zu sein, sollte ein Ausfalldetektor bestimmte Genauigkeitseigenschaften erfüllen, welche die falschen Einschätzungen, die ein Ausfalldetektor machen darf, einschränken.

Wir berücksichtigen acht absolute Genauigkeitseigenschaften: *absolut starke Genauigkeit*, *absolut mehrheitliche Genauigkeit*, *absolut p-starke Genauigkeit*, *absolut p-mehrheitliche Genauigkeit absolut q-schwache Genauigkeit*, *absolut pq-starke Genauigkeit*, *absolut pq-schwache Genauigkeit* sowie *absolute Pfad-Genauigkeit*. Die erste und die dritte Genauigkeitseigenschaft wurden zuerst von Chandra und Toueg eingeführt. Im Sprachgebrauch von Chandra und Toueg wird die Eigenschaft *p-starke Genauigkeit* unter dem Namen *weak accuracy* verwendet.

DEFINITION 11 (**absolut starke Genauigkeit**). *Kein Applikationsprozeß wird verdächtigt, bevor er abstürzt. Formal erfüllt* $\mathbb{D}$ absolut starke Genauigkeit*, falls:*

$$\begin{aligned} \forall \mathcal{S} \ \in\ & \mathbb{S} \,.\, \forall H \in \mathbb{D}(\mathcal{S}) \,.\, \forall p \in App \,. \\ \forall q \ \in\ & Mon\,(p) \;.\, \forall t \in T \,.\, \neg\ false\_susp(\mathcal{S}, H, q, p, t) \end{aligned}$$

■

Wie im Falle von *starker Vollständigkeit* schwächen wir die *absolut starke Genauigkeit* ab, indem wir Mehrheitsentscheidungen zulassen.

DEFINITION 12 (**absolut mehrheitliche Genauigkeit**). *Kein Applikationsprozeß wird von einer Mehrheit der Überwachungsprozesse verdächtigt, bevor er abstürzt. Formal erfüllt* $\mathbb{D}$ absolut mehrheitliche Genauigkeit*, falls:*

$$\begin{aligned} \forall \mathcal{S} &\in \mathbb{S} \,.\, \forall H \in \mathbb{D}(\mathcal{S}) \,.\, \forall p \in App \,.\, \exists M \in act\_maj\,(Mon\,(p)) \;. \\ \forall q &\in M \,.\, \forall t \in T \,.\, \neg\; false\_susp(\mathcal{S}, H, q, p, t) \end{aligned}$$

■

Wir schwächen die *absolut starke Genauigkeit*seigenschaft ab, indem wir falsche Verdächtigungen für mindestens einen aktiven Applikationsprozeß ausschließen.

DEFINITION 13 (**absolut p-starke Genauigkeit**). *Wenigstens ein permanent aktiver Applikationsprozeß wird nie verdächtigt. Formal erfüllt* $\mathbb{D}$ absolut p-starke Genauigkeit*, falls:*

$$\begin{aligned} \forall \mathcal{S} &\in \mathbb{S} \,.\, \forall H \in \mathbb{D}(\mathcal{S}) \,.\, \exists p \in App \cap perm\_activ\,(\mathcal{S}) \;. \\ \forall q &\in Mon\,(p) \;.\, \forall t \in T. \;\neg\; false\_susp(\mathcal{S}, H, q, p, t) \end{aligned}$$

■

Wir schwächen die *absolut p-starke Genauigkeit* ab, indem wir falsche Verdächtigungen für mindestens einen aktiven Applikationsprozeß seitens einer Mehrheit der Überwachungsprozesse ausschließen.

DEFINITION 14 (**absolut p-mehrheitliche Genauigkeit**). *Wenigstens ein permanent aktiver Applikationsprozeß wird nie von einer Mehrheit der Überwachungsprozesse verdächtigt. Formal erfüllt* $\mathbb{D}$ absolut p-mehrheitliche Genauigkeit*, falls:*

$$\begin{aligned} \forall \mathcal{S} &\in \mathbb{S} \,.\, \forall H \in \mathbb{D}(\mathcal{S}) \,.\, \exists p \in App \cap perm\_activ\,(\mathcal{S}) \;. \\ \exists M &\in act\_maj\,(Mon\,(p)) \;.\, \forall q \in M \;. \\ \forall t &\in T. \;\neg false\_susp(\mathcal{S}, H, q, p, t) \end{aligned}$$

■

Im Modell von Chandra und Toueg ist wenigstens ein Applikationsprozeß permanent aktiv, d.h. $perm\_activ\,(\mathcal{S}) \neq \emptyset$. Schwächen wir *absolut starke Genauigkeit* ab, indem wir falsche Verdächtigungen nur jeweils von mindestens einem Überwachungsprozeß ausschließen, dann erhalten wir *absolut q-schwache Genauigkeit*.

DEFINITION 15 (**absolut q-schwache Genauigkeit**). *Kein Überwachungsprozeß verdächtigt alle permanent aktiven Prozesse. Formal erfüllt* $\mathbb{D}$ absolut q-schwache Genauigkeit*, falls:*

$$\begin{aligned} \forall \mathcal{S} &\in \mathbb{S} \;.\; \forall H \in \mathbb{D}(\mathcal{S}) \;.\; \forall q \in Mon \;.\; \exists p \in App(q) \cap perm\_activ\,(\mathcal{S}) \;. \\ \forall t &\in T \;.\; \neg\; false\_susp(\mathcal{S}, H, q, p, t) \end{aligned}$$

■

Schließen wir falsche Verdächtigungen für aktive Applikationsprozesse von mindestens einem aktiven Überwachungsprozeß aus, dann erhalten wir *absolut pq-starke Genauigkeit*.

DEFINITION 16 (**absolut pq-starke Genauigkeit**). *Kein Applikationsprozeß wird von allen Überwachungsprozessen verdächtigt, bevor er abstürzt. Formal erfüllt* $\mathbb{D}$ absolut pq-starke Genauigkeit*, falls:*

$$\begin{aligned} \forall \mathcal{S} &\in \mathbb{S} \;.\; \forall H \in \mathbb{D}(\mathcal{S}) \;.\; \forall p \in App \;. \\ \exists q &\in Func\,(\mathcal{S}, p) \;.\; \forall t \in T \;.\; \neg\; false\_susp(\mathcal{S}, H, q, p, t) \end{aligned}$$

■

Schließen wir falsche Verdächtigungen für einen aktiven Applikationsprozeß von mindestens einem aktiven Überwachungsprozeß aus, dann erhalten wir *absolut pq-schwache Genauigkeit*. Anschließend führen wir die *absolute Pfad-Genauigkeit* ein. Anschaulich erfüllt ein Ausfalldetektor $\mathbb{D}$ *absolute Pfad-Genauigkeit*, falls zu jedem permanent aktiven Applikationsprozeß p und zu jedem permanent aktiven Überwachungsprozeß q eine Menge (Pfad) von Prozessen existiert, so daß entlang des Pfades Prozesse sich nicht gegenseitig verdächtigen. Wie wir später sehen werden, ist somit eine Kommunikation von p nach q indirekt möglich.

DEFINITION 17 (**absolut pq-schwache Genauigkeit**). *Wenigstens ein permanent aktiver Applikationsprozeß p wird von min-*

destens einem bezüglich p funktionellen Überwachungsprozeß *nie verdächtigt. Formal erfüllt* $\mathbb{D}$ absolut pq-schwache Genauigkeit*, falls:*

$$\begin{aligned} \forall \mathcal{S} &\in \mathbb{S} \;.\; \forall H \in \mathbb{D}(\mathcal{S}) \;.\; \exists p \in App \cap perm\_activ\,(\mathcal{S}) \;. \\ \exists q &\in Func\,(\mathcal{S}, p) \;.\; \forall t \in T.\; \neg\; false\_susp(\mathcal{S}, H, q, p, t) \end{aligned}$$

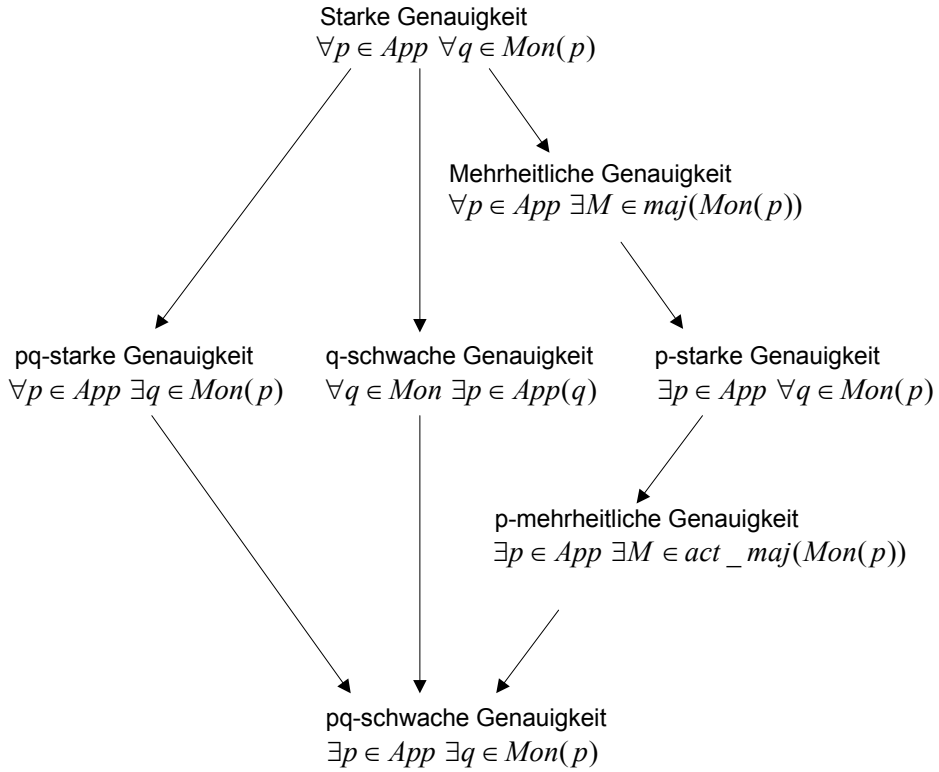
■

DEFINITION 18 (**absolute Pfad-Genauigkeit**). *Zu jedem permanent aktiven Applikationsprozeß p und zu jedem permanent aktiven Überwachungsprozeß q gibt es permanent aktive Prozesse $q_1, q_2, ..., q_k$ mit $p = q_1$ und $q = q_k$, so daß der Prozeß q_{i+1} den Prozeß q_i nie verdächtigt. Formal erfüllt* $\mathbb{D}$ absolute Pfad-Genauigkeit*, falls:*

$$\begin{aligned} &\forall \mathcal{S} \in \mathbb{S} \;.\; \forall H \in \mathbb{D}(\mathcal{S}) \;.\; \forall p \in App \cap perm\_activ\,(\mathcal{S}) \;.\; \forall q \in Mon\,(p) \;. \\ &\exists\; q_1, q_2, ..., q_k \in App \cap perm\_activ\,(\mathcal{S}) \;\wedge\; q_1 = p \;\wedge\; q_k = q \;. \\ &\forall t \in T \;.\; t \geq t_0 \;.\; \forall i \in \{1, 2, ..., k\} \;.\; \neg\; false\_susp(\mathcal{S}, H, q_{i+1}, q_i, t) \end{aligned}$$

■

Analog berücksichtigen wir acht **Lebendigkeitseingenschaften**: *schließlich starke Genauigkeit* (eventual strong accuracy), *schließlich mehrheitliche Genauigkeit, schließlich p-starke Genauigkeit, schließlich p-mehrheitliche Genauigkeit, schließlich q-schwache Genauigkeit, schließlich pq-starke Genauigkeit, schließlich pq-schwache Genauigkeit* sowie *schließliche Pfad-Genauigkeit.* Die erste und die dritte Genauigkeitseigenschaft wurde zuerst von Chandra und Toueg eingeführt. Chandra und Toueg verwenden den Begriff eventual weak accuracy für die Eigenschaft *schließlich p-starke Genauigkeit.*



Abschwächung der Genauigkeitseigenschaften

Wir definieren im folgenden exemplarisch und stellvertretend die Eigenschaft *schließlich p-starke Genauigkeit* und verzichten auf die entsprechende Definition der anderen Eigenschaften.

DEFINITION 19 (**schließlich p-starke Genauigkeit**). *Wenigstens ein aktiver Applikationsprozeß wird schließlich nie verdächtigt. Formal erfüllt* $\mathbb{D}$ schließlich p-starke Genauigkeit*, falls:*

$$\forall \mathcal{S} \in \mathbb{S} \,.\, \forall H \in \mathbb{D}(\mathcal{S}) \,.\, \exists p \in App \cap perm\_activ(\mathcal{S}) \,.\, \forall q \in Mon(p) \,.$$
$$\exists t_0 \in T \,.\, \forall t \geq t_0 \,.\, \neg\, false\_susp(\mathcal{S}, H, q, p, t)$$

■

Die anderen Lebendigkeitseigenschaften werden formal in analoger Weise definiert.

Wir führen in unserem Ansatz acht neue **Sicherheitseigenschaften** ein: *kontinuierlich starke Genauigkeit*, *kontinueirlich mehrheitliche Genauigkeit, kontinuierlich p-starke Genauigkeit*, *kontinuierlich p-mehrheitliche Genauigkeit, kontinuierlich q-schwache Genauigkeit*, *kontinuierlich pq-starke Genauigkeit, kontinuierlich pq-schwache Genauigkeit* sowie *kontinuierliche Pfad-Genauigkeit.*

DEFINITION 20 (**kontinuierlich starke Genauigkeit**). *Kein aktiver Applikationsprozeß wird permanent verdächtigt. Formal erfüllt* $\mathbb{D}$ kontinuierlich starke Genauigkeit*, falls:*

$$\begin{aligned} \forall \mathcal{S} &\in \mathbb{S} \,.\, \forall H \in \mathbb{D}(\mathcal{S}) \,.\, \forall p \in App \,.\, \forall q \in Func\,(\mathcal{S}, p) \;. \\ \forall t &\in T \,.\, \exists t' \geq t \,.\, H\,(q, p, t') \neq \infty \;\wedge\; \neg\; false\_susp(\mathcal{S}, H, q, p, t') \end{aligned}$$

■

Lassen wir aber Mehrheitsentscheidungen zu, indem wir die *kontinuierlich starke Genauigkeit* abschwächen, dann erhalten wir *kontinuierlich mehrheitliche Genauigkeit.*

DEFINITION 21 (**kont. mehrheitliche Genauigkeit**). *Kein aktiver Applikationsprozeß wird permanent von einer Mehrheit der Überwachungsprozesse verdächtigt. Formal erfüllt* $\mathbb{D}$ kontinuierlich starke Genauigkeit*, falls:*

$$\begin{aligned} \forall \mathcal{S} &\in \mathbb{S} \,.\, \forall H \in \mathbb{D}(\mathcal{S}) \,.\, \forall p \in App \,.\, \forall q \in act\_maj\,(Mon\,(p)) \;. \\ \forall t &\in T \,.\, \exists t' \geq t \,.\, H\,(q, p, t') \neq \infty \;\wedge\; \neg\; false\_susp(\mathcal{S}, H, q, p, t') \end{aligned}$$

■

Schwächen wir die Eigenschaft *kontinuierlich starke Genauigkeit* ab, indem wir dauernd falsche Verdächtigungen mindestens für einen Überwachungsprozeß ausschließen, so erhalten wir *kontinuierlich pq-starke Genauigkeit.*

DEFINITION 22 (**kontinuierlich pq-starke Genauigkeit**). *Kein aktiver Applikationsprozeß wird permanent von allen Überwachungsprozessen verdächtigt. Formal erfüllt* $\mathbb{D}$ kontinuierlich *pq*-starke Genauigkeit*, falls:*

$$\begin{aligned} \forall \mathcal{S} \quad &\in \quad \mathbb{S} \;.\; \forall H \in \mathbb{D}(\mathcal{S}) \;.\; \forall p \in App \;.\; \exists q \in Func\,(\mathcal{S}, p) \;\; . \\ \forall t \quad &\in \quad T \;.\; \exists t' \geq t \;.\; H\,(q, p, t') \neq \infty \;\wedge\; \neg\; false\_susp(\mathcal{S}, H, q, p, t') \end{aligned}$$

■

Formal erhalten wir *kontinuierlich pq-starke Genauigkeit*, indem wir in der formalen Definition der *kontinuierlich starken Genauigkeit* den Term $\forall q \in Func\,(\mathcal{S}, p)$ durch $\exists q \in Func\,(\mathcal{S}, p)$ ersetzen.

Wir können die obigen Formeln für *kontinuierlich starke Genauigkeit* bzw. *kontinuierlich q-starke Genauigkeit* etwas vereinfachen, indem wir den Term $H\,(q, p, t') \neq \infty \wedge \neg\; false\_susp(\mathcal{S}, H, q, p, t')$ durch $H(q, p, t') =\uparrow$ ersetzen. Wir haben uns für die etwas längere Darstellungsweise mit dem Prädikat $false\_susp$ entschieden, um die Unterschiede zu den anderen Genauigkeitseigenschaften besser zu verdeutlichen.

Wir erhalten *kontinuierlich p-starke Genauigkeit* bzw. *kontinuierlich pq-schwache Genauigkeit* analog zu den absoluten Eigenschaften bzw. Lebendigkeitseigenschaften, indem wir die entsprechenden Allquantoren durch Existenzquantoren ersetzen.

REMARK 5. *Eine alternative Darstellungsweise für* kontinuierlich *pq*-starke Genauigkeit *wäre:*

$$\begin{aligned} \forall \mathcal{S} \in \quad & \mathbb{S} \;.\; \forall H \in \mathbb{D}(\mathcal{S}) \;.\; \forall p \in App \;.\; \forall t \in T \;. \\ & p \in activ(\mathcal{S})_t \Longrightarrow \\ \exists q \in \quad & Func\,(\mathcal{S}, p) \;\; . \neg\; event\_perm\_susp(\mathcal{S}, H, q, p, t) \end{aligned}$$

■

Analoges gilt für die *kontinuierlich starke Genauigkeit.*

Sei $(S, .)$ die Klasse der Ausfalldetektoren, die *starke Vollständigkeit* erfüllen, sei $(W, .)$ die Klasse der Ausfalldetektoren, die *schwache Vollständigkeit* erfüllen bzw. sei $(M, .)$ die Klasse der Ausfalldetektoren, die *mehrheitliche Vollständigkeit* erfüllen.

Wir fassen die verschiedenen Klassen von Ausfalldetektoren, die eine bestimmte Genauigkeitseigenschaft erfüllen, tabellarisch zusammen.

Genauigkeit	absolut	schließlich	kontinuierlich
stark	$(., \Box S)$	$(., \Diamond S)$	$(., \sim S)$
mehrheitlich	$(., \Box M)$	$(., \Diamond M)$	$(., \sim M)$
Pfad	$(., \Box P)$	$(., \Diamond P)$	$(., \sim P)$
$p-$stark	$(., \Box pS)$	$(., \Diamond pS)$	$(., \sim pS)$
$p-$mehrheitlich	$(., \Box pM)$	$(., \Diamond pM)$	$(., \sim pM)$
$q-$schwach	$(., \Box qW)$	$(., \Diamond qW)$	$(., \sim qW)$
$pq-$stark	$(., \Box pqS)$	$(., \Diamond pqS)$	$(., \sim pqS)$
$pq-$schwach	$(., \Box pqW)$	$(., \Diamond pqW)$	$(., \sim pqW)$

Ausfalldetektor–Klassen (**Genauigkeitseigenschaften**)

So erfüllen z.B. die Ausfalldetektoren der Klasse $(S, \Diamond pS)$ *starke Vollständigkeit* und *schließlich p-starke Genauigkeit.*

Wir führen abschließend den Begriff *kommunikativ* nicht *disjunkte Mengen* ein. Sei $M \subset Mon$ und sei $M' \subset Mon$. Dann sind M und M' *kommunikativ* nicht *disjunkt*, falls M und M' disjunkte Mengen sind und falls Prozesse q aus M und q' aus M' existieren, so daß q und q' sich nicht gegenseitig verdächtigen. In diesem Fall können die Prozesse aus der Menge M mit den Prozessen der Menge M' kommunizieren, wie wir dies später sehen werden. Wir halten fest:

DEFINITION 23 (**kommunikativ nicht disjunkte Mengen**).
Sei $M \subset App$ und sei $M' \subset Mon$. Wir sagen: "die Mengen M und M' sind absolut kommunikativ *nicht* disjunkt" *(als Zeichen $M \,\Box\!\!\!*\, M' = \emptyset$), falls M und M' disjunkte Mengen sind, d.h. $M \cap M' = \emptyset$ und falls:*

$$(\exists q \in M \;.\; \exists p' \in M' \;.\; \forall t \in T \;.\; H(q, p', t) = \uparrow) \wedge$$
$$(\exists p \in M \;.\; \exists q' \in M' \;.\; \forall t \in T \;.\; H(q', p, t) = \uparrow)$$

Analog sagen wir: "die Mengen M und M' sind schließlich kommunikativ *nicht* disjunkt" *(als Zeichen $M \Diamond\!\!\!*\, M' = \emptyset$), falls M und M' disjunkte Mengen sind, d.h. $M \cap M' = \emptyset$ und falls:*

$$(\forall p \in M \;.\; \forall p' \in M' \;.\; \forall t' \in T \;.\; \exists t \geq t' \;.\; H(p, p', t) \neq \uparrow) \vee$$
$$\left(\forall p'' \in M \;.\; \forall p^{(3)} \in M' \;.\; \forall t' \in T \;.\; \exists t \geq t' \;.\; H(p'', p^{(3)}, t) \neq \uparrow\right)$$

■

Entsprechend sagen wir: "Mon ist <u>*nicht*</u> *absolut separiert*" (als Zeichen $\Box Mon \neq \|$) bzw. "<u>*nicht*</u> *schließlich separiert*" (als Zeichen $\Diamond Mon \neq \|$), falls für jede Menge $\emptyset \subsetneq M \subsetneq Mon$ die Mengen M und $Mon \backslash M$ absolut kommunikativ <u>nicht</u> disjunkt bzw. schließlich kommunikativ <u>nicht</u> disjunkt sind.

3. Vergleich mit dem Modell von Chandra und Toueg

In diesem Abschnitt vergleichen wir unseren Ansatz mit dem Modell von Chandra und Toueg bzw. mit seinen Nachfolgemodellen und vermerken die wesentlichen Unterschiede. Anschließend vergleichen wir anhand von Beispielen die Vollständigkeits- bzw. die Genauigkeitseigenschaften in beiden Modellen. Wir zeigen, daß in unserem Ansatz die Anforderungen an die Vollständigkeits- bzw. Genauigkeitseigenschaften der Ausfalldetekoren, um hohe Verfügbarkeit des Systems bzw. *Consensus* zu erreichen, gegenüber dem Modell von Chandra und Toueg abgeschwächt werden können. Außerdem zeigen wir, daß es alternative Darstellungsformen zu den Vollständigkeits- bzw. Genauigkeitseigenschaften gibt, die von Chandra und Toueg verwendet wurden, um *Consensus* zu lösen.

3.1. Unterschiede zum Modell von Chandra und Toueg bzw. zu den Nachfolgemodellen. Die wesentlichen Unterschiede zum Modell von Chandra und Toueg bzw. zu den Nachfolgemodellen können letztendlich auf die Möglichkeit, abgestürzte (verdächtigte) Prozesse in unserem Ansatz hochzufahren, zurückgeführt werden.

3.1.1. Recovery. Abgestürzte Prozesse können in unserem Ansatz hochgefahren werden. Durch Herunterfahren von verdächtigten Prozessen werden falsche Verdächtigungen aufgehoben. Im Modell von Chandra und Toueg sowie in allen uns bekannten Nachfolgemodellen werden verdächtige Prozesse nicht gezielt heruntergefahren. Die *Recovery*-Strategie in den Nachfolgemodellen von Chandra und Toueg sieht ein ad hoc Hochfahren der abgestürzten Prozesse vor, während in unserem Ansatz gezielt *Recovery* eingeleitet wird. Somit eignet sich unser Ansatz auch zur Erhöhung der Verfügbarkeit von Applikationen, wie wir dies in den nächsten Kapiteln sehen werden.

3.1.2. *Überwachung der Überwachung.* Um hohe Verfügbarkeit zu erreichen, werden die Überwachungsprozesse in unserem Ansatz genau so behandelt wie die Applikationsprozesse, d.h. die Überwachungsprozesse werden selber überwacht und gegebenenfalls werden abgestürzte Überwachungsprozesse hochgefahren.

3.1.3. *Unendlich lange Verdächtigung.* In unserem Ansatz entfallen die Anforderungen nach unendlich langer Verdächtigung, wie sie im Modell von Chandra und Toueg für *schwache* bzw. *starke Vollständigkeit* üblich sind. Diese Anforderungen werden entsprechend abgeschwächt, indem sich die Anforderungen nur auf die Zeit beziehen,

während dessen die Überwachungsprozesse aktiv sind.

3.1.4. *Unendlich viele falsche Einschätzungen.* Um die *kontinuierlich starke* bzw. *kontinuierlich q-schwache Genauigkeit*seigenschaften in unserem Ansatz zu erfüllen, können Überwachungsprozesse unendlich viele falsche Einschätzungen liefern. Im Gegensatz dazu sind im Modell von Chandra und Toeg nur endlich viele falsche Einschätzungen (falsche Verdächtigungen) möglich, sonst kann die *schließlich p-starke Genauigkeit* sowie die *schließlich starke Genauigkeit* in diesem Modell nicht erfüllt werden.

3.2. Beispiel. Im folgenden Beispiel vergleichen wir anhand von ausgewählten Verhaltensmustern von Überwachungsprozessen die Vollständigkeit- und die Genauigkeitseigenschaften in unserem Ansatz bzw. im Modell von Chandra und Toueg. Das Verhalten wird tabellarisch erfaßt, als Spaltenattribute werden diskrete Zeitwerte genommen.

Sei p ein ausgewählter Applikationsprozeß, sei q ein beliebiger Überwachungsprozeß und sei t ein beliebiger Zeitpunkt. Der tatsächliche Zustand von q zum Zeitpunkt t sowie der von q zum Zeitpukt t ermittelte Zustand von p wird als Eintrag zum Zeitpunkt t in der Tabelle dargestellt. Formal gilt für q und p zum Zeitpunkt t der Eintrag $(\mathcal{S}(t,q), H(q,p,t))$.

Da durch die Tabelle nur endlich viele Werte darstellbar sind, aber die Überwachung nicht zeitlich begrenzt ist, setzen wir:

$$(\mathcal{S}(t_{k+6},q), H(q,p,t_{k+6})) := (\mathcal{S}(t_k,q), H(q,p,t_k))$$

für alle $k > 0$ sowie $t_{k+1} := t_k + 1$ für alle $k \geq 0$. Dabei bedeutet die letzte Einschränkung, daß alle Zeitpunkte ab einem Anfangszeitpunkt t_0 erfaßt werden. Es gibt keine Zeitlücken, in denen das Verhalten der Überwachungsprozesse nicht definiert ist.

3.2.1. *Vollständigkeit.* Sei $p \in App$ ein Applikationsprozeß, welcher ab dem Zeitpunkt t_0 permanent abgestürzt ist, d.h. $\forall t \geq t_0$. $\mathcal{S}(t,p) = \dot{+}$, seien $q_1, q_2, ..., q_4 \in Mon(p)$. Die folgende Tabelle gibt die Einschätzungen von $q_1, q_2, ..., q_4$, wie eingangs besprochen, in unserem Beispiel wieder.

T	t_0	t_1	t_2	t_3	t_4	t_5	t_6
q_1	$(\uparrow, \uparrow)$	$(\uparrow, \dot{+})$	$(\uparrow, \dot{+})$	$(\uparrow, \dot{+})$	$(\uparrow, \dot{+})$	$(\uparrow, \dot{+})$	$(\uparrow, \dot{+})$
q_2	$(\uparrow, \uparrow)$	$(\uparrow, \dot{+})$	$(\uparrow, \dot{+})$	$(\uparrow, \dot{+})$	$(\uparrow, \dot{+})$	$(\uparrow, \dot{+})$	$(\uparrow, \uparrow)$
q_3	$(\uparrow, \uparrow)$	$(\uparrow, \dot{+})$	$(\dot{+}, \infty)$	$(\searrow, \infty)$	$(\downarrow, \infty)$	$(\nearrow, \infty)$	$(\uparrow, \dot{+})$
q_4	$(\uparrow, \dot{+})$	$(\dot{+}, \infty)$	$(\dot{+}, \infty)$	$(\dot{+}, \infty)$	$(\dot{+}, \infty)$	$(\dot{+}, \infty)$	$(\dot{+}, \infty)$

TABELLE 1. Einschätzungen von $q_1, ..., q_4$

Der Überwachungsprozeß q_1 erfüllt die Bedingungen für *schwache Vollständigkeit* sowohl in unserem Ansatz als auch im Modell von Chandra und Toueg. Ab dem Zeitpunkt t_1 verdächtigt q_1 den Prozeß p dauernd, d.h. $\forall t > t_0 \; . \; H\,(q_1, p, t) = \dot{+}$.

Im Gegensatz dazu erfüllt q_2 die Bedingungen für *schwache Vollständigkeit* weder in unserem Ansatz noch im Modell von Chandra und Toueg. Man bemerkt, daß $\forall k > 0 \; . \; \forall t > t_0 \; . \; H\,(q_2, p, t_{k+6}) = \uparrow$.

Der Überwachungsprozeß q_3 erfüllt in unserem Ansatz ab dem Zeitpunkt t_1 die Bedingungen für *schwache Vollständigkeit*. Man bemerkt, daß q_3, falls er aktiv ist, d.h. falls er überhaupt in der Lage ist Auswertungen vorzunehmen (abgestürzte Prozesse verschicken keine Nachrichten), ab dem Zeitpunkt t_1 den Prozeß p dauernd verdächtigt.
Formal: $\forall t > t_0 \; . \; (\mathcal{S}(t, q) = \uparrow \implies H\,(q_1, p, t) = \dot{+})$.
Trivialerweise erfüllt q_3 im Modell von Chandra und Toueg die Bedingungen für *schwache Vollständigkeit*, denn q_3 stürzt in bestimmten Zeitabständen ab.

Der Überwachungsprozeß q_4 erfüllt in unserem Ansatz die Bedingungen für *schwache Vollständigkeit* für den Zeitpunkt t_0, aber für keinen anderen Zeitpunkt $t > t_0$, da q_4 dauernd abgestürzt ist.

3.2.2. *Genauigkeit.* Sei $p \in App$ ein Applikationsprozeß, welcher ab dem Zeitpunkt t_0 permanent aktiv ist, d.h. $\forall t \geq t_0 \; . \; \mathcal{S}(t, p) = \uparrow$, seien $q_1, q_2, ..., q_5 \in Mon(p)$. Die folgende Tabelle gibt die Einschätzungen von $q_1, q_2, ..., q_5$, wie eingangs besprochen, in unserem Beispiel wieder.

T	t_0	t_1	t_2	t_3	t_4	t_5	t_6
q_1	$(\uparrow, \uparrow)$	$(\uparrow, \uparrow)$	$(\uparrow, \uparrow)$	$(\uparrow, \uparrow)$	$(\uparrow, \uparrow)$	$(\uparrow, \uparrow)$	$(\uparrow, \uparrow)$
q_2	$(\uparrow, \uparrow)$	$(\uparrow, \dot{+})$	$(\uparrow, \dot{+})$	$(\uparrow, \uparrow)$	$(\uparrow, \uparrow)$	$(\uparrow, \uparrow)$	$(\uparrow, \uparrow)$
q_3	$(\uparrow, \dot{+})$	$(\uparrow, \dot{+})$	$(\dot{+}, \infty)$	$(\searrow, \infty)$	$(\downarrow, \infty)$	$(\nearrow, \infty)$	$(\uparrow, \uparrow)$
q_4	$(\uparrow, \uparrow)$	$(\uparrow, \dot{+})$	$(\uparrow, \dot{+})$	$(\uparrow, \dot{+})$	$(\uparrow, \dot{+})$	$(\uparrow, \dot{+})$	$(\uparrow, \dot{+})$
q_5	$(\uparrow, \uparrow)$	$(\dot{+}, \infty)$	$(\dot{+}, \infty)$	$(\dot{+}, \infty)$	$(\dot{+}, \infty)$	$(\dot{+}, \infty)$	$(\dot{+}, \infty)$

TABELLE 2. Einschätzungen von $q_1, ..., q_5$

Wie ersichtlich, wird im Modell von Chandra und Toueg *schwache Genauigkeit* erfüllt, falls alle Überwachungsprozesse sich so verhalten wie q_1. Verhalten sich alle Überwachungsprozesse gegenüber allen aktiven Anwendungsprozessen wie q_1, dann wird *starke Genauigkeit* im Modell von Chandra und Toueg erfüllt. Analoge Aussagen für *kontinuierlich starke Genauigkeit* bzw. *kontinuierlich q-schwache Genauigkeit* gelten auch in unserem Ansatz.

Der Überwachungsprozeß q_2 erfüllt in unserem Ansatz die Bedingungen für *kontinuierlich starke Genauigkeit* bzw. *kontinuierlich q-schwache Genauigkeit*, er erfüllt aber nicht die Bedingungen für *absolut (schließlich) starke* bzw. *absolut (schließlich) p-starke Genauigkeit* im Modell von Chandra und Toueg. Man bemerkt, daß q_2 den Prozeß p nicht dauernd verdächtigt, d.h. $\forall t > t_0 \; . \; \exists t' \geq t \; . \; H\,(q_2, p, t') = \uparrow$. Das gleiche gilt für q_3.

Der Überwachungsprozeß q_4 erfüllt weder in unserem Ansatz noch im Modell von Chandra und Toueg die untersuchten Genauigkeitseigenschaften, denn der Überwachungsprozeß q_5 ist dauernd abgestürzt.

KAPITEL 3

Stand der Forschung

1. Wichtigste Arbeiten

Wir stellen die wichtigsten Arbeiten, die sich mit Ausfalldetektoren befassen, kurz vor.

1.1. Non-Blocking (Weak) Atomic Commitment. Rachid Guerraoui untersucht in [**32**] den Zusammenhang zwischen *Non-Blocking Atomic Commitment* (*NB-AC*) und *Consensus* in asynchronen Systemen, die mit einem unzuverlässigen Ausfalldetektor versehen sind. Sei f die Anzahl der abgestürzten Prozesse. Guerraoui verwendet das Modell von Chandra und Toueg, siehe auch [**17**] und stellt fest, daß *Consensus* und *Uniform Consensus* äquivalent sind, siehe [**32**, Section 4.2] bzw. daß das *NB-AC* stärker ist als *Consensus*. Gilt $f > 0$, so ist in diesem Modell das *NB-AC*-Problem für die Ausfalldetektor-Klassen $(S, \Box pS)$, $(S, \Diamond S)$, siehe [**32**, Theorem 1] sowie $(S, \Diamond pS)$, $(W, \Diamond S)$ oder $(W, \Diamond pW)$, siehe [**32**, Corollary 1] nicht lösbar.

Guerraoui hat die *Non-Triviality*-Bedingung (Commit muß entschieden werden, falls alle Teilnehmer *Yes* beschließen) des *NB-AC*-Problems abgeschwächt, und zwar Commit muß entschieden werden, falls alle Teilnehmer *Yes* beschließen und kein Teilnehmer ist je verdächtigt). Unter diesen Umständen kann das neue Problem, genannt *Non-Blocking Weak Atomic Commitment* (*NB-WAC*), gelöst werden. *NB-WAC* ist in asynchronen Systemen, die mit unzuverlässigen Ausfalldetektoren versehen sind, auf *Consensus* reduzierbar, d.h. wann immer *Consensus* lösbar ist, ist das *NB-WAC*-Problem auch lösbar. Somit ist das *NB-WAC*-Problem mit Ausfalldetektoren der Klasse $(S, \Box pS)$ lösbar, falls wenigstens ein Teilnehmer aktiv ist und mit Ausfalldetektoren der Klasse $(S, \Diamond pS)$ lösbar, falls die Mehrheit der Teilnehmer aktiv ist.

1.2. Schließlich mehrheitsstabile Systeme. Christian Fetzer und Flaviu Cristian untersuchen in [**27**], ob *Consensus* in asynchronen Systemen lösbar ist. Es wird gezeigt, daß eine deterministische Lösung für *Election*, *Rotating Leadership* und *Consensus* in schließlich mehrheitsstabilen (*eventually majority stable*) Systemen immer möglich ist. Dieses Modell setzt voraus, daß jede instabile Phase des Systems schließlich von einer Phase gefolgt wird, in welcher die Kommunikation der Mehrheit der Prozesse innerhalb einer festen und bekannten Zeitschranke erfolgt. In diesem Modell sind auch *Restart*s von Prozessen nach einem Ausfall erlaubt.

Ein Vergleich mit Ausfalldetektoren im Sinne von Chandra und Toueg wird nicht unternommen, da das Modell von Fetzer und Cristian zeitlos (*time-free*) ist, es setzt voraus, daß die entsprechenden Eigenschaften des Ausfalldetektors schließlich erfüllt werden und im Gegensatz zum Modell von Chandra und Toueg ist das Hochfahren von Prozessen nach einem Ausfall erlaubt.

Christian Fetzer und Flaviu Cristian zeigen, daß das *Leader-Election-* bzw. das *Consensus*-Problem in verteilten asynchronen Systemen lösbar ist, falls die Mehrheit der Prozesse schließlich immer imstande ist für genügend lange Zeit zu kommunizieren. Ausfälle und *Recoveries* der anderen Prozesse sowie die Kommunikation zwischen ihnen verhindern den *Consensus* nicht.

1.3. Ausfallbewußte-Ausfalldetektoren. Christian Fetzer und Flaviu Cristian untersuchen in [**28**] Ausfallbewußte-Ausfalldetektoren (*fail aware failure detectors*). Es wird festgestellt, daß in zeitlosen (*time-free*) asynchronen verteilten Systemen unmöglich ist, perfekte Ausfalldetektoren zu implementieren. Ein perfekter Ausfalldetektor verdächtigt nur abgestürzte Prozesse und sie verdächtigt schließlich alle abgestürzten Prozesse.

Das *Highly Available Leader Election*-Problem verlangt, daß zu jedem Zeitpunkt s ein Zeitpunkt t mit einem aktiven *Leader* existiert. Die Autoren erwähnen auch, daß im allgemeinen in zeitlosen *(time-free)* asynchronen Systemen Ausfalldetektoren nicht implementierbar sind.

Um *Consensus* zu lösen, werden die Eigenschaften des Ausfalldetektors um Ausfallbewußtsein (*fail awareness*) ergänzt.

Der Ausfallbewußte-Ausfalldetektor ist streng schwächer als der perfekte Ausfalldetektor, er ist nach Meinung der Autoren in asynchronen Systemen, in denen die Kommunikation innerhalb einer festen und bekannten Zeitschranke erfolgt, implementierbar und erlaubt eine deterministische Lösung des *Consensus*- bzw. des *Election*-Problems.

Die Grundidee des Ausfallbewußten-Ausfalletektors ist, daß der Ausfalldetektor eines Prozesses p weiß, wenn "zu viele" Prozesse fälschlicherweise p verdächtigen und setzt p davon in Kenntnis. Diese Eigenschaft kann verwendet werden, um zu sichern, daß es zu jedem Zeitpunkt nur einen *Leader* gibt. Ist p der gegenwärtige *Leader* und ist p von einer Mehrheit von Prozessen verdächtigt, die einen neuen *Leader* wählen könnten, dann wird p auch von seinem eigenen Ausfalldetektormodul verdächtigt und diese "Selbstverdächtigung" sagt p, daß er Platz für einen neuen *Leader* schaffen sollte.

Die Eigenschaft *Ausfallbewußtsein* verlangt, daß das Ausfalldetektormodul eines jeden Prozesses p weiß, ob p von mehr als $k > 0$ Prozessen verdächtigt wird. Somit setzt die Eigenschaft Ausfallbewußtsein voraus, daß weniger als k Prozesse p verdächtigen, falls p nicht von seinem eigenem Ausfalldetektormodul verdächtigt wird.

Folglich wurden zusätzlich zu den Vollständigkeits- bzw. Genauigkeitseigenschaften aus dem Modell von Chandra und Toueg zwei neue Ausfallbewußtsein-Eigenschaften eingeführt: schwaches und starkes Ausfallbewußtsein.

Sei n die Anzahl der Prozesse des Systems. Die Eigenschaft **starkes Ausfallbewußtsein** erfordert, daß ein Prozeß p sich selbst verdächtigt, falls ein anderer Prozeß q den Prozeß p verdächtigt, **schwaches Ausfallbewußtsein** erfordert hingegen, daß ein Prozeß p nur dann sich selbst verdächtigt, falls eine Mehrheit von Prozessen p verdächtigen.

1.4. *Message Omission Failure*-Umgebung. Dolev, Keidar, Friedman und Malkhi untersuchen in [**24**] Ausfalldetektoren in sogenannter *Message Omission Failure*-Umgebung. In dieser Umgebung können Prozesse abstürzen sowie wieder hochkommen, Nachrichten

können verlorengehen sowie willkürlich verzögert werden, das Netzwerk kann partitionieren bzw. die Netzwerkpartitionen können sich zusammenschließen.

Es wird die Definition von Chandra und Toueg bezüglich Vollständigkeit und Genauigkeit übernommen und es wird der schwache Ausfalldetektor $<> W(om)$ definiert, welcher jeder Mehrheit von Prozessen, die miteinander verbunden sind, erlaubt *Consensus* zu erreichen trotz unbegrenzter Anzahl von transienten Kommunikationsfehlern in der Vergangenheit. Im verwendeten Protokoll ist es nicht nötig, daß man die alten Nachrichten aufbewahrt, um sie nochmals zu senden. Prozesse können abstürzen und wieder hochkommen, der feste Speicher bleibt intakt und jeder Zustandswechsel im Prozeß wird im festen Speicher protokolliert. In diesem Modell gibt es keinen Unterschied zwischen abgestürzten Prozessen, welche später hochkommen bzw. Prozessen, die ihre Netzwerkverbindungen verloren haben.

In diesem Modell konvergiert schließlich das System gegen einen stabilen Zustand, in welchem alle korrekten Prozesse zuverlässig miteinander kommunizieren und in welchem die korrekten Prozesse von allen fehlerhaften Prozessen entbunden sind. Der stabile Zustand kann nur von außen beobachtet werden und die Prozesse müssen dessen nicht bewußt sein. Ein Prozeß muß nicht von sich selber wissen, ob er korrekt ist oder nicht. Stabilität muß fomal für immer gelten, für jede Ausführung des Algorithmus muß Stabilität aber nur eine endliche Zeit dauern.

Es wird gezeigt, daß *Consensus* in diesem Modell lösbar ist, falls nicht mehr als $\left[\frac{n}{2}\right]$ Prozesse fehlerhaft sind und die Umgebung mit einem Ausfalldetektor[1] $<> W(om)$ versehen ist.

1.5. Kollektive Konsistenz. Wir stellen kurz der besseren Übersicht wegen die Ideen über kollektive Konsistenz aus der Arbeit von Cynthia Dwork (siehe [**25**]) vor.

Kollektive Konsistenz ist eine schwache Form von *Consensus*, in welcher die Prozesse, die am Protokoll teilnehmen, versuchen eine gemeinsame Sicht (d.h. eine gemeinsame Liste von verdächtigten Prozessen) über die Gruppenmitgliedschaft [2]zu erreichen. Ein Prozeß beginnt

[1]Streng genommen handelt es sich um eine Ausfalldetektor-Klasse

[2]Prozesse die nicht abgestürzt sind, und für die die Berechnungen relevant sind

die Berechnung mit einer initialen Sicht und verläßt das *CCP* (*Collective Consistency Protocol)* mit einer Ausgabesicht.

Kollektive Konsistenz verlangt, daß die Ausgabesicht V_p jedes Prozesses p mit den Ausgabesichten der Prozesse übereinstimmt, die p nicht verdächtigen. Mit anderen Worten: die Ausgabe von p müßte mit der Ausgabe der Prozesse übereinstimmen, die nicht in V_p sind.

Wird somit ein Absturz konsistent an alle überlebenden Teilnehmer gemeldet, dann können die Überlebenden zum letzten *Checkpoint* zurückrollen, sich umorganisieren und können somit die Berechnung wieder aufnehmen (*roll forward*). Solange aber die Teilnehmer verschiedene Sichten auf die Menge der Prozessen haben, die nach der Reorganisation zur Verfügung steht, kann die Berechnung nicht fortgesetzt werden.

Das Ziel der kollektiven Konsistenz ist, daß alle Teilnehmer eine konsistente Sicht über die Menge der abgestürzten Prozesse haben. In einem kollektiven Konsistenz-Protokoll besteht keine Anforderung, daß ein Prozeß eine Sicht zurückgibt. Somit kann kollektive Konsistenz auch in Systemen gelöst werden, die keine Lösung des *Consensus* erlauben.

Speziell das kollektive Konsistenz-Protokoll macht sicher, daß zwei Prozesse, die sich gegenseitig nicht als abgestürzt betrachten und die sich gegenseitig nicht blockieren, nach dem Verlassen des Protokolls eine identische Sicht auf die Gruppenmitgliedschaft haben. Wir unterscheiden kollektive Konsistenz und starke kollektive Konsistenz:
a) **Kollektive Konsistenz**: Ist V_{p_j} die Sicht, die vom Prozeß p_j zurückgegeben wird und ist V_{p_i} die Sicht, die vom Prozeß p_i zurückgegeben wird und ist $p_i \notin V_{p_j}$, dann gilt $V_{p_i} = V_{p_j}$.
b) **Starke kollektive Konsistenz***:* Die zurückgegebenen Sichten aller Prozesse sind identisch.

Starke kollektive Konsistenz entspricht *Consensus*. Der Hauptunterschied zwischen *Consensus* und kollektiver Konsistenz besteht darin, daß die Definition der kollektiven Konsistenz unter bestimmeten Umständen zwei Prozessen p und q erlaubt, verschiedene Sichten zu haben. Dwork ist in erster Linie an monotonen Lösungen interessiert, in welchen Verdächtigungen nie wegfallen. Verdächtigt am Anfang q den Prozeß p, dann kann q nichts weiter davon abhalten, während des gesamten *CCP* p weiter zu suspektieren.

Kollektive Konsistenz wurde zuerst im Bereich *Parallel Engineering* und für mathematische Berechnungen verwendet. In dieser Umgebung ist es zulässig, daß die Prozesse sich in zwei Gruppen spalten und jede Gruppe die Berechnung unabhängig weiterführt.

1.6. Vertrauenswürdigkeit. Aguilera, Chen und Toueg untersuchen in [**3**] das Problem der Fehlererkennung und *Consensus* in asynchronen Systemen, in welchen Prozesse abstürzen und wieder hochkommen können und Netzwerkverbindungen Nachrichten verlieren können. Es wird ein Ausfalldetektor vorgestellt, welcher besonders geeignet für das *Crash-Recovery*-Modell ist.

Anschließend wird untersucht unter welchen Bedingungen das Speichern von Daten auf festem Medium nötig ist, um *Consensus* in diesem Modell zu lösen. Es werden zwei Algorithmen vorgestellt, die Ausfälle von Netzwerkverbindungen tolerieren und die besonders effizient für Abläufe sind, die häufig in der Praxis vorkommen, und zwar: Abläufe ohne Ausfälle und Abläufe ohne Fehleinschätzungen des Ausfalldetektors. In solchen Abläufen ist *Consensus* in 3δ Zeit erreicht, wobei δ die maximale Verzögerung der Nachrichten darstellt.

In diesem Modell werden nicht die vermutlich abgestürzten oder instabilen Prozesse ermittelt. Statt dessen ermittelt der Ausfalldetektor die Liste der Prozesse, von denen er meint, daß sie aktiv sind, sowie für jeden Prozeß aus der Liste die *Epoch*-Zahl. Befindet sich ein Prozeß p in dieser Liste, dann sagen wir: "p ist *vertrauenswürdig*". Die *Epoch*-Zahl ist eine grobe Abschätzung der Anzahl der Abstürze und *Recoveries* in der Vergangenheit. Die Autoren unterscheiden zwei Arten von Prozessen, *böse* und *gute* Prozesse. Die *bösen* Prozesse sind instabil bzw. stürzen permanent ab. Die *guten* Prozesse stürzen nie ab oder bleiben schließlich hoch.

Aguilera, Chen und Toueg schlagen einen Ausfalldetektor vor, genannt $<> S_e$, welcher grob formuliert folgende Eigenschaften aufweist:

Vollständigkeit: Zu jedem *bösen* Prozeß p, zu jedem *guten* Prozeß q, welcher p überwacht, gibt es einen Zeitpunkt, nachdem entweder p für q nicht mehr vertrauenswürdig ist oder die *Epoch*-Zahl von p ständig wächst.

Genauigkeit: Einige *gute* Prozesse sind für alle *guten* Prozesse vertrauenswürdig und deren *Epoch*-Zahl wächst nicht.

Der Ausfalldetektor, der **Vollständigkeit** und **starke Genauigkeit** (einige *gute* Prozesse sind schließlich vertrauenswürdig für alle *guten* und instabilen Prozesse und deren *Epoch*-Zahl hört auf zu wachsen) erfüllt, wird mit $<> S_u$ bezeichnet. Im Schreiben von Aguilera, Chen und Toueg wird gezeigt wie man $<> S_e$ in $<> S_u$ transformiert, gesetzt der Fall, die Mehrheit der Prozesse sind korrekt.

1.7. Heartbeat-Ausfalldetektor. Aguilera, Chen und Toueg untersuchen in [**5**], [**7**], [**6**] sowie in [**4**] die Möglichkeiten um zuverlässige Kommunikation mit ruhenden (*quiescent*) Algorithmen in asynchronen Systemen zu erreichen. In diesem Modell können Prozesse ausfallen sowie Netzwerkverbindungen verlorengehen. Die ruhenden Algorithmen hören schließlich auf, Nachrichten zu verschicken. Es wird gezeigt, daß es unmöglich ist, zuverlässige Kommunikation mit ruhenden (*quiescent*) Algorithmen in asynchronen Systemen ohne Ausfalldetektoren zu lösen.

Um zuverlässige Kommunikation zu gewährleisten, wurde ein neuer Ausfalldetektor, der *Heartbeat*-Ausfalldetektor eingeführt. Die Autoren meinen, daß im Gegensatz zu den vorhergehenden Ausfalldetektoren, die verwendet wurden, um Unmöglichkeitsergebnisse zu umgehen, der *Heartbeat*-Ausfalldetektor implementierbar ist und seine Implementierung verwendet keine *Timeouts*. Die Ergebnisse können verwendet werden, um existierende Algorithmen, die nur Prozeßausfälle tolerieren, in ruhende Algorithmen zu transformieren, die zusätzlich Nachrichtenverlust tolerieren können. Diese Ergebnisse können beim *Consensus*, *Atomic Broadcast*, *k-set Agreement*, *Atomic Commitment* usw. angewendet werden.

Vereinfacht dargestellt, gibt das Ausfalldetektormodul eines Prozesses p einen Vektor von Zählern aus, einen für jeden Nachbarn q von p. Stürzt q nicht ab, dann wächst sein Zähler grenzenlos, stürzt q ab, dann hört schließlich sein Zähler auf zu wachsen. Die Grundidee für eine Implementierung eines *HB*(*Heartbeat*)-Ausfalldetektors ist naheliegend: jeder Prozeß schickt periodisch eine "*I am alive*" Nachricht, d.h. ein *Heartbeat* und jeder Prozeß, welcher das *Heartbeat* erhält, erhöht den entsprechenden Zähler. Der *HB*-Ausfalldetektor zählt die Anzahl der *Heartbeats*, die von jedem Prozeß kommen und gibt diese

Werte aus ohne zusätzliche Bearbeitung oder Interpretation.

Die Eigenschaften des Ausfalldetektors werden entsprechend verändert, und zwar:
HB-Vollständigkeit: Bei jedem korrekten Prozeß ist die *Heartbeat*-Sequenz jedes abgestürzten Nachbarn beschränkt.
HB-Genauigkeit: 1) Bei jedem Prozeß ist die *Hearbeat*-Sequenz jedes Nachbarn nicht absteigend. 2) Bei jedem Prozeß ist die *Heartbeat*-Sequenz jedes korrekten Nachbarn uneingeschränkt.

REMARK 6. *Unserer Meinung nach verwendet die praktische Implementierung von* HB*-Ausfalldetektoren dennoch verdeckt* timeout*-ähnliche Auswertungskriterien. Die Anzahl der* Heartbeats *werden für die Feststellung, ob ein Prozeß noch korrekt ist, herangezogen. Empfängt ein Prozeß q keine* Heartbeats *von einem Prozeß p, dann kann q noch nicht feststellen, ob tatsächlich p keine* Heartbeats *geschickt hat oder nur der Empfang verzögert wurde.*
In praktischen Systemen spielen die Auswertungszeiten genau die Rollen von Timeouts*, irgendwann muß eine Entscheidung getroffen werden, ob eine Auswertung stattfindet oder nicht. Durch die zweite Genauigkeitsanforderung wird implizit angenommen, daß zwischen zwei Auswertungen für einen korrekten Prozeß der Zähler wächst.*

1.8. *Quittable Consensus*. Delporte-Gallet, Hadzilacos, Fauconnier, Kouznetsov, Guerraoui und Toueg ermitteln in [**21**] den schwächsten Ausfalldetektor, um bestimmte Probleme in verteilten Systemen, in denen beliebig viele Prozesse abstürzen können, zu lösen. Die Probleme, die in der oben genannten Arbeit untersucht werden, beinhalten die Implementierung eines atomaren Registers, die Lösbarkeit des *Consensus*, des *Quittable Consensus* (eine Variante des *Consensus* in dem die Prozesse die Option haben, *quit* zu entscheiden, falls Prozesse abstürzen) und des *Non-Blocking Atomic Commitments*. Die Autoren betrachten beliebige Umgebungen, wo es keine Einschränkungen gibt bezüglich der Anzahl der abgestürzten Prozesse, der Reihenfolge der Abstürze, usw. und verwenden Ausfalldetektoren mit ausgewählten Eigenschaften. Im Sprachgebrauch der Autoren stehen diese Eigenschaften stellvertretend für die Ausfalldetektoren. Die Autoren berücksichtigen folgende drei Ausfalldetektoren:

a) den *Quorum*-Ausfalldetektor Σ. Dieser Ausfalldetektor ermittelt zu jedem Prozeß p einen Satz von Prozessen. Der Durschnitt von je zwei Sätzen (an verschiedenen Prozessen) ist nicht leer und der ermittelte Satz an jedem aktiven Prozeß besteht nur aus aktiven Prozessen.

b) den *Leader*-Ausfalldetektor Ω. Dieser Ausfalldetektor ermittelt bei jedem Prozeß p die *ID* eines Prozesses. Schließlich wird bei jedem aktiven Prozeß die *ID* desselben aktiven Prozesses ermittelt.

c) den Ausfallsignal-Ausfalldetektor $\mathcal{FS}$ (*failure signal failure detector*). Dieser Ausfalldetektor gibt bei jedem Prozeß entweder grün oder rot aus. Solange kein Prozeß ausfällt, gibt $\mathcal{FS}$ bei jedem Prozeß grün aus. Fällt wenigstens ein Prozeß aus, so gibt $\mathcal{FS}$ schließlich permanent bei jedem Prozeß rot aus.

Die Autoren untersuchen auch Ausfalldetektoren, die über die Eigenschaften von zwei der obigen Ausfalldetektoren verfügen sowie auch Ausfalldetektoren, die aus den obigen Ausfalldetekoren abgeleitet werden. Dementsprechend verfügt der Ausfalldetektor (Ω, Σ) über die Eigenschaften des Ausfalldetektors Ω sowie des Ausfalldetektors Σ. Die Autoren zeigen, daß der *Quorum*-Ausfalldetektor Σ der schwächste Ausfalldetektor ist, um Register zu implementieren. Mit Hilfe dieses Ergebnisses zeigen die Autoren, daß (Ω, Σ) der schwächste Ausfalldetektor ist, welcher *Consensus* in jeder Umgebung löst.

Sei Ψ ein Ausfalldetektor wie folgt definiert:
Am Anfang ermittelt Ψ bei jedem Prozeß den Wert $\perp$. Nach endlicher Zeit verhält sich Ψ jedoch wie (Ω, Σ) oder falls ein Ausfall geschieht und nur dann wie $\mathcal{FS}$. Der Übergang von $\perp$ nach (Ω, Σ) bzw. $\mathcal{FS}$ muß nicht gleichzeitig geschehen, aber der Übergang ist zwingend für alle Prozesse. Im Rahmen der obigen Arbeit wird gezeigt, daß Ψ der schwächste Ausfalldetektor ist, welcher *Quittable Consensus* löst. Anschließend wird gezeigt, daß $(\Psi, \mathcal{FS})$ der schwächste Ausfalldetektor ist, welcher das *NB-AC*-Problem löst.

1.9. Stubborn Kommunikationskanal. Rachid Guerraoui, Rui Oliveira und André Schiper haben in [**34**] das Konzept des *Stubborn* Kommunikationskanals eingeführt. Es verwendet *Fair Lossy* Kommunikationskanäle. Formal wird ein *Stubborn* Kommunikationskanal anhand von zwei Primitiven $S\text{-}send_{p,q}$ und $S\text{-}receive_{p,q}$ eingeführt. Ein Kanal ist *stubborn*, falls zusätzlich zur Eigenschaft *No Creation* des *Fair Lossy* Kommunikationskanals folgendes erfüllt ist:

Stubborn: Seien p und q zwei aktive (d.h. nicht abgestürzte) Prozesse. Sendet p durch die Primitive $S\text{-}send_{p,q}$ eine Nachricht m an q und

verzögert p für eine unbestimmte Zeit das Verschicken von Nachrichten an q, dann erhält q schließlich m durch die Primitive $S\text{-}receive_{p,q}$.

In [**52**] wurde dieses Konzept verallgemeinert und auf *k-Stubborness* erweitert, indem nicht einzelne Nachrichten, sondern ein Satz $m_\psi, m_{\psi+1}, ..., m_{\psi+k-1}$ verschickt wird und dessen Empfang garantiert wird. In [**52**] wird zusätzlich noch die Eigenschaft *No Duplication* vorausgesetzt.

Das Hauptanliegen in den obigen Arbeiten besteht darin, zu untersuchen, inwieweit das *Consensus*-Problem in Modellen lösbar ist, die keine zuverlässige Netzwerkverbindung voraussetzen.

In [**34**] wird gezeigt, daß *Consensus* mit Hilfe eines Ausfalldetektors der Klasse $(S, \Diamond pS)$ sowie unter Verwendung von *Stubborn* Kommunikationskanälen lösbar ist, falls die Mehrheit der am *Consensus*-Algorithmus teilnehmenden Prozesse korrekt (aktiv) ist. Die Ausfalldetektor-Klasse $(S, \Diamond pS)$ verlangt *starke Vollständigkeit* (schließlich wird jeder abgestürzte Prozeß permanent verdächtigt) und *schließlich p-starke Genauigkeit* (schließlich wird wenigstens ein aktiver Prozeß nie verdächtigt).

In [**52**] werden zuerst grüne und rote Prozesse definiert (g/r Modell), um anschließend zum Begriff des roten Ausfalldetektors zu kommen. Ein Prozeß p führt einen Null Schritt aus, falls der Prozeß keine Nachrichten empfängt, seinen Status nicht ändert sowie keine Nachrichten verschickt. Ein Prozeß, welcher abstürzt und wieder hochkommt, führt eine endliche Mengen von Null Schritten zwischen dem Absturz und dem Hochkommen aus, ein Prozeß, welcher nie hochkommt, führt unendlich viele Null Schritte aus. Grüne Prozesse führen nur eine endliche Menge von Null Schritte, rote Prozesse führen aber unendlich viele Null Schritte aus. Durch $<> S_r$ bezeichnen wir die Klasse der Ausfalldetektoren, die *starke Vollständigkeit* (schließlich wird jeder rote Prozeß permanent von jedem grünen Prozeß verdächtigt) sowie *schließlich schwache Genauigkeit* (schließlich wird wenigstens ein grüner Prozeß nie von grünen Prozessen verdächtigt) erfüllen. Es wird gezeigt, daß der Ausfalldetektor der Klasse $<> S_r$ der schwächste Ausfalldetektor ist, welcher *Consensus* im g/r Modell mit einer Mehrheit von grünen Prozessen und *Fair Lossy* Kommunikationskanälen erfüllt.

Leider untersuchen die Autoren die Eigenschaft *stubborn* nicht näher, speziell wird nicht die Zeit untersucht, während dessen eine Nachricht im Kanal stabil vorhanden sein muß, um zuverlässig verschickt zu werden. In unserem Ansatz ist diese Verzögerungszeit, falls der Empfänger aktiv ist, durch die Versendezeit der Nachricht bzw. der Bestätigung im wesentlichen beschränkt.

1.10. Umwandlung in zuverlässige Kanäle. In [**9**] wird gezeigt, daß jedes Problem, welches mit Hilfe von zuverlässigen Kanälen in asynchronen Systemen lösbar ist, auch mit Hilfe von *Fair Lossy* Kanälen lösbar ist vorausgesetzt, daß wenigstens die Hälfte der Prozesse nicht abstürzt. Es werden in der obigen Arbeit zuverlässige Kommunikationskanäle anhand von *Fair Lossy* Kanälen simuliert, so daß jeder Algorithmus, welcher zuverlässige Kommunikationskanäle voraussetzt, auch über *Fair Lossy* Kanälen verwendet werden kann.

Leider müssen in [**9**] wegen des Simulationsalgorithmus alle Nachrichten gespeichert werden, so daß unendlicher Speicherplatz vorausgesetzt wird. Die Länge der Nachrichten wird auch nicht begrenzt. Unser Simulationsalgorithmus verwendet einen endlichen Speicherplatz, die durch die Länge der längsten Nachricht begrenzt ist.

2. Der *Consensus*-Algorithmus von Chandra und Toueg

Der folgende Algorithmus von Chandra und Toueg (**Alg. 4**) löst das *Consensus*-Problem mit jedem Ausfalldetektor $\mathbb{D} \in (S, \Box pS)$, siehe [**17**, Fig. 5]. Sei n die Anzahl der Prozesse im System, die am *Consensus* teilnehmen. Der *Consensus*-Algorithmus von Chandra und Toueg (**Alg. 4**) toleriert den Absturz von bis zu $n-1$ Prozessen.

Der *Consensus*-Algorithmus kann in drei Phasen gegliedert werden. In der ersten Phase (Zeile 5 bis 17) durchlaufen die Prozesse $n-1$ asynchrone Runden. Wir bezeichnen durch r_p jeweils die aktuelle Runde, wobei der Index p auf den Prozeß hinweist, welcher den Run des Algorithmus gestartet hat. In der ersten Phase des Runs verschickt p seinen Vorschlag an alle Prozesse, die am *Consensus* teilnehmen und wartet auf die entsprechenden Vorschläge der Teilnehmer. Wird der Prozeß q von p verdächtigt, d.h. $q \in \mathbb{D}_p$, dann wartet p nicht mehr auf die Antwort von q und fährt mit der nächsten Runde fort. Am Ende der Phase 2 enthält $V_p[q]$ entweder den vom Prozeß q vorgeschlagenen Wert oder $\perp$. In der nächsten Phase einigen sich die Prozesse auf die erste nichttriviale Komponente dieses Vektors.

Jeder Prozeß p, welcher am *Consensus*-Algorithmus beteiligt ist, führt folgende Folge von Anweisungen aus.

```
01 procedure propose (v_p) {
02      V_p ← (⊥, ⊥, ..., ⊥) ;
03      V_p ← v_p; /* p's estimate of the proposed value */
04      Δ_p ← V_p;

        /* Phase 1: asynchronous rounds r_p, 1 ≤ r_p ≤ n − 1 */
05      for r_p ← 1 to n − 1{
06          send (r_p, Δ_p, p) to all;
07          wait until [∀q : received (r_p, Δ_q, q) or q ∈ 𝔻_p] ;
08          msgs_p [r_p] ← ((r_p, Δ_q, q) | received (r_p, Δ_q, q)) ;
09          Δ_p ← (⊥, ⊥, ..., ⊥) ;
10          for k ← 1 to n{
11              if V_p [k] =⊥ and ∃ (r_p, Δ_q, q) ∈ msgs_p [r_p] with
12                                                  Δ_q [k] ≠⊥ then{
13                  V_p [k] ← Δ_q [k] ;
14                  Δ_p [k] ← Δ_q [k] ;
15              }
16          }
17      }

        /∗ Phase 2: ∗/
18      send V_p to all;
19      wait until [∀q : received V_q or q ∈ 𝔻_p] ;
20      lastmsgs_p ← (V_q | received V_q) ;
21      for k ← 1 to n{
22          if ∃V_p ∈ lastmsgs_p with V_p [k] =⊥ then{
23              V_p [k] ←⊥;
24          }
25      }

26      /∗ Phase 3: ∗/
27      decide (first non− ⊥ component of V_p ) ;
28 }
```

Alg. 4. Algorithmus zum Lösen des *Consensus* mit einem Ausfalldetektor der Klasse $(S, \Box pS)$

Der nächste Algorithmus (**Alg. 5**), siehe [**17**, Fig. 6], löst das *Consensus*-Problem mit jedem Ausfalldetektor $\mathbb{D} \in (S, \Diamond pS)$, vorausgesetzt, daß wenigstens die Hälfte der Prozesse, die am *Consensus* beteiligt sind, während des *Consensus* nicht abstürzen.

Der Algorithmus verwendet das Paradigma des rotierenden Koordinators[3] und führt mehrere, aber eine endliche Anzahl von asynchronen Runden aus . Bei jedem Run des Algorithmus wird in der Runde r der Prozeß $c = (r \mod n) + 1$ als Koordinator verwendet. Jede Runde r kann in vier Phasen aufgeteilt werden.

In der ersten Phase schicken alle Prozesse p ihre Vorschläge $estimate_p$ durch die Anweisung: "$send\,(p, r_p, estimate_p, ts_p)$ *to* c_p;" zum Koordinator. Die Variable ts_p enthält einen Zeitstempel und zwar ist es die letzte Runde in der p seinen Vorschlag $estimate_p$ geändert hat.

In der zweiten Phase wartet der Koordinator auf das Eintreffen von $\left\lceil \frac{(n+1)}{2} \right\rceil$ Vorschlägen, wählt einen Vorschlag mit dem größten Zeitstempel aus und schickt diesen Vorschlag mittels der Anweisung: "$send\left(c_p, r_p, estimate_{c_p}\right)$ *to all*;" an alle Prozesse, die am *Consensus* beteiligt sind.

In der dritten Phase warten alle Prozesse p auf das Eintreffen der Nachricht vom aktuellen Koordinator c_p, es sei denn, der Ausfalldetektor $\mathbb{D}_p$ von p verdächtigt c_p. Hat der Prozeß p den Vorschlag $estimate_{c_p}$ vom Koordinator erhalten, so überschreibt p seinen eigenen Vorschlag $estimate_p$, d.h. p setzt $estimate_p := estimate_{c_p}$, aktualisiert den Zeitstempel ts_p und schickt eine Bestätigung (ack) zurück zum Koordinator. Meint der Prozeß p, der Koordinator sei abgestürzt, dann schickt p eine negative Bestätigung (nack) zurück.

In der vierten Phase wartet der Koordinator c_p auf $\left\lceil \frac{(n+1)}{2} \right\rceil$ Antworten. Sind wenigstens $\left\lceil \frac{(n+1)}{2} \right\rceil$ Antworten gleich *ack*, dann ist der Wert $estimate_{c_p}$ entschieden und der Koordinator schickt durch die Anweisung: "$R\text{-}broadcast(c_p, r_p, estimate_{c_p}, decide)$;" eine entsprechende Nachricht an alle Prozesse. Erhält der Prozeß p die obige Nachricht, dann entscheidet p den Wert $estimate_{c_p}$.

Jeder Prozeß p, welcher am *Consensus* beteiligt ist, führt folgende Anweisungen aus:

[3] d.h. in jeder Runde wird ein neuer Koordinator gewählt

```
01 procedure propose (v_p) {
02      estimate_p ← v_p;
03      state_p ← undecided;
04      r_p ← 0;
05      ts_p ← 0;

  /* Rotate through coordinator until a decision is reached */

06      while state_p = undecided{
07          r_p ← r_p + 1;
08          c_p ← (r_p mod n ) + 1;

      /* Phase 1: */
  /* All processes p send estimate_p to the current coordinator */

09          send (p, r_p, estimate_p, ts_p) to c_p;

      /* Phase 2:
  /* The current coordinator gathers ⌈(n+1)/2⌉ estimates
          and proposes a new estimate */

10          if p = c_p then{
11              wait until [for ⌈(n+1)/2⌉ processes q : received
                                (q, r_p, estimate_q, ts_q) from q];
12              msgs_p [r_p] ← (q, r_p, estimate_q, ts_q) | p received
                                (q, r_p, estimate_q, ts_q) from q];
13              t ← largest ts_q such that (q, r_p, estimate_q, ts_q)
                                ∈ msgs_p [r_p] ;

14              estimate_p ← select one estimate_q such that
                            (q, r_p, estimate_q, t) ∈ msps_p [r_p] ;
15              send (p, r_p, estimate_p) to all;
16          }

      /* Phase 3: */
  /* All processes wait for the new estimate proposed by the
          current coordinator */
```

```
17          wait until [received (c_p, r_p, estimate_{c_p}) from c
                                or c_p ∈ 𝔻_p];
18          if [received (c_p, r_p, estimate_{c_p}) from c_p] then{
19              estimate_p ← estimate_{c_p};
20              ts_p ← r_p;
21              send (p, r_p, ack) to c_p;
22          }else /* p suspects that c_p crashed */
23              send (p, r_p, nack) to c_p;
```

```
    /* Phase 4 */
 /* The current coordinator waits for ⌈(n+1)/2⌉ replies. If they
        indicate that ⌈(n+1)/2⌉ processes adopted the estimate,
         the coordinator R-broadcast a decide message */

24          if p = c_p then{
25              wait until [for ⌈(n+1)/2⌉ processes q : received
                                (q, r_p, ack) or (q, r_p, nack)];
26              if [for ⌈(n+1)/2⌉ processes q : received (q, r_p, ack)]
                                                          then
27                  R-broadcast(p, r_p, estimate_p, decide) ;
28          }

    /* if p R-delivers a message, p decides accordingly */

29          when R-deliver (q, r_q, estimate_q, decide)
30              if state_p = undecided then{
31                  decide (estimate_p) ;
32                  state_p ← decided;
33              }
34      }/* while */
35  }
```

Alg. 5. Algorithmus zum Lösen des *Consensus* mit einem Ausfalldetektor der Klasse $(S, \Diamond pS)$

3. Vergleich der Ansätze

Im folgenden fassen wir tabellarisch die verschiedenen Ansätze zusammen. Um die Größe der Tabelle in Grenzen zu halten, kürzen wir die Themenkreise wie folgt ab:

WAC	*Non-Blocking (Weak) Atomic Commitment* ([**17**],[**32**])
SmS	Schließlich mehrheitsstabile Systeme ([**27**])
FA	Ausfallbewußte-Ausfalldetektoren ([**28**])
MOF	*Message Omission Failure* Umgebung ([**24**])
KK	Kollektive Konsistenz ([**25**])
VW	Vertrauenswürdigkeit ([**3**])
HB	*Heartbeat*-Ausfalldetektor ([**5**],[**7**],[**6**],[**4**])
SK	*Stubborn* Kommunikationskanal ([**34**],[**52**])
Consensus	Ist *Consensus* lösbar ?
NB-AC	Ist das *NB-AC*-Problem lösbar ?
Ausf.-De.	Werden Ausfalldetektoren verwendet ?
Restart	Sind *Restart* von Prozessen erlaubt ?
Netz.-Part.	Netzwerkpartitionierung erlaubt ?
Sys.-konv.	Systemkonvergenz gegen eine stabile Phase ?
Her. Pr.	Herunterfahren von absturzverdächtigen Prozessen ?
Erw. Eig.	Erweiterung der Eigenschaften der Ausfalldetektoren ?
Unk. Impl.	Unkonventionelle Implementierungsmethoden ?

Wir verwenden folgende Legende:

+	Die Eigenschaft/Problem trifft zu
-	Die Eigenschaft/Problem trifft nicht zu
+/-	Die Eigenschaft/Problem trifft nur teilweise zu
?	Die Eigenschaft/Problem wurde nicht untersucht

Zuerst halten wir fest, daß **Consensus** in allen vorgestellten Modellen untersucht wurde und lösbar ist. Das **NB-AC**-Problem wurde in asynchronen Systemen unter Zuhilfenahme von Ausfalldetektoren, soweit es

	WAC	SmS	FA	MOF	KK	VW	HB	SK
Consensus	+	+	+	+	+	+	+	+
NB-AC	-	?	?	?	?	?	?	?
Ausf.-De.	+	-	+	+	-	+	+	+
Restart	-	+	-	+	-	+	+	+
Netz.-Part.	-	+	-	+	-	+	+	+
Sys.-konv.	-	+	-	+	-	-	-	-
Her. Pr.	-	-	-	-	-	-	-	-
Erw. Eig.	?	?	+	-	?	-	-	+
Unk. Impl.	-	?	?	?	?	?	+	-

TABELLE 1. Vergleich der Ansätze

uns bekannt ist, nur in Modell von Chandra und Toueg untersucht und hat sich als unlösbar erwiesen. Im Gegensatz dazu ist das *NB-AC*-Problem in unserem Ansatz lösbar.

Nicht alle Autoren verwenden **Ausfalldetektoren**, um die Lösbarkeit des *Consensus* in asynchronen Systemen zu untersuchen. So wird *Consensus* in asynchronen, schließlich mehrheitsstabilen (*eventually majority stable*) Systemen gelöst, siehe [**27**]. Dieses System setzt voraus, daß jede instabile Phase des Systems schließlich von einer Phase gefolgt wird, in welcher die Kommunikation der Mehrheit der Prozesse innerhalb einer festen und bekannten Zeitschranke erfolgt.

Restarts von Prozessen, die am *Consensus* teilnehmen, wird von manchen Autoren auch berücksichtigt, allerdings wird das Hochfahren der abgestürzten Prozesse nicht im Rahmen des *Consensus*-Algorithmus angestoßen, Prozesse können abstürzen und wieder von sich selbst hochkommen. In unserem Ansatz kommen abgestürzte Prozesse von sich selbst nicht hoch.

Einige Autoren erweitern das übliche Modell von Chandra und Toueg, um **Netzwerkpartitionierungen** aufzufangen. Die Verfügbarkeit der Netzwerke in der Industrie ist mittlerweile so hoch, daß keine *Single Points of Failure* vorhanden sind, beim Ausfall einer Netzwerkkomponente wird die Nachricht transparent für den Sender über einen anderen Weg zum Ziel geleitet.

Einige Systeme **konvergieren** gegen eine stabile Phase, die mit anderen Mitteln untersucht werden kann. So wird jede instabile Phase in [**27**] von einer Phase gefolgt, in welcher die Kommunikation der Mehrheit der Prozesse innerhalb einer festen und bekannten Zeitschranke erfolgt. In [**24**] konvergiert schließlich das System gegen einen stabilen Zustand, in welchem alle korrekten Prozesse zuverlässig miteinander kommunizieren und in welchem die korrekten Prozesse von allen fehlerhaften Prozessen entbunden sind.

Absturzverdächtige Prozesse werden nur in unserem System heruntergefahren.

Einige Autoren **erweitern die Eigenschaften** des Ausfalldetekors. So werden in [**28**] die üblichen Eigenschaften um *Fail Awareness* ergänzt.

Unkonventionelle Implementierungsmethoden verwenden z.B. Aguilera, Chen und Toueg in [**5**], [**7**], [**6**] sowie in [**4**]. Jeder Prozeß schickt periodisch ein *Heartbeat* und jeder Prozeß, welcher dieses *Heartbeat* erhält, erhöht den entsprechenden Zähler. Der Ausfalldetektor zählt die Anzahl der *Heartbeats*, die von jedem Prozeß kommen. Im allgemeinen legen sich die Autoren aber nicht auf eine bestimmte Implementierungsmethode fest, vielmehr sind die Eigenschaften des Ausfalldetektors unabhängig von der Wahl der Implementierung.

KAPITEL 4

Eigener wissenschaftlicher Beitrag

Im folgenden führen wir die wichtigsten Beiträge dieser Arbeit auf.

1. Änderung des Modells

Wir haben das übliche Modell, welches auch von Chandra und Toueg verwendet wurde und letztendlich auf Fischer zurückgeführt werden kann, erweitert, siehe [**31**], indem wir neben Prozeßabstürzen weitere Zustände eingeführt haben sowie *Recovery* zugelassen haben. *Recovery* wurde auch in [**3**] bzw. [**24**] untersucht, der Unterschied zu den erwähnten Arbeiten besteht darin, daß in unserem Ansatz die Prozesse nach einem Absturz nicht selber hochkommen, sondern im Rahmen einer *Recovery* hochgefahren werden.

Alle uns bekannten Arbeiten, die sich mit Ausfalldetektoren befassen, untersuchen, inwieweit *Consensus* und die mit *Consensus* verwandten Probleme lösbar sind. Eine Überwachung von Anwedungen bzw. von Prozessen, um höhere Verfügbarkeit zu erreichen, ist nicht Ziel dieser Arbeiten.

Alle Prozesse bzw. Applikationen, die schließlich als abgestürzt eingestuft wurden, können in unserem Ansatz heruntergefahren werden. Eine ähnliche Entscheidung treffen auch Ricciardi und Birman, siehe [**55**]. Uns ist keine Arbeit über Ausfalldetektoren bekannt, welche diese Herangehensweise verwendet.

2. Neue Vollständigkeits- bzw. Genauigkeitseigenschaften

Wir haben in unserer Arbeit neue Vollständigkeits- bzw. Genauigkeitseigenschaften eingeführt und untersuchen, inwieweit diese Eigenschaften zur Lösung der von uns berücksichtigten Übereinstimmungsprobleme (*agreement problems*) beitragen können. So haben wir außer der *starken Vollständigkeit* und der *schwachen Vollständigkeit* eine neue Eigenschaft, und zwar *mehrheitliche Vollständigkeit* eingeführt.

Analog haben wir *mehrheitliche Genauigkeit* bzw. *p-mehrheitliche Genauigkeit* als Abschwächung der entsprechenden starken Eigenschaften definiert. Wir zeigen, daß im allgemeinen die mehrheitlichen Eigenschaften ausreichen, um die erwünschten starken Eigenschaften zu erhalten. Auch die Eigenschaft *q-schwache Genauigkeit* sowie das Konzept der *Pfad-Genauigkeit* ist neu.

3. Formale Modellierung

Damit wir die Begriffe präzise einführen und untersuchen können, haben wir in unserem Modell eine entsprechende Metasprache eingeführt und die in der Literatur übliche formale Modellierung weiterentwickelt. Dementsprechend haben wir das Modell axiomatisiert und drei Modellannahmen eingeführt sowie mehrere Prädikate definiert, wodurch sich die Eigenschaften des Ausfalldetektors prägnanter und einfacher sich darstellen lassen.

4. Emulation von Ausfalldetektor-Eigenschaften

Wir untersuchen welche stärkere Eigenschaften durch schwächere Eigenschaften in unserem Modell emuliert werden können. So kann z.B. die *starke Vollständigkeit* mit Hilfe von *mehrheitlicher Vollständigkeit* und die *absolut starke Genauigkeit* mit Hilfe von *absolut mehrheitlicher Genauigkeit* emuliert werden. Somit kann die Ausfalldetektor-Klasse $(M, \Box M)$ in $(S, \Box S)$ überführt werden. Dieses Ergebnis zeigt insbesondere, daß wir einen perfekten Ausfalldetektor erhalten, falls der Ausfalldetektor für eine Mehrheit seiner Prozesse perfekt ist. Außerdem zeigen wir, daß $(S, \Diamond pS)$ mit Hilfe von $(M, \Diamond pM)$ emuliert werden kann.

5. Simulation von zuverlässigen Netzwerkverbindungen

Wir zeigen, daß in unserem Modell jedes Problem, welches mit zuverlässigen Verbindungen lösbar ist, auch mit Hilfe von *Fair Lossy* Kanälen lösbar ist. Im Gegensatz zu [**9**] nehmen wir an, daß die Prozesse, die den Simulationsalgorithmus ausführen, hoch verfügbar sind, siehe auch [**34**] sowie [**52**]. Dies bedeutet insbesondere, daß eine Nachricht erfolgreich verschickt wird, sobald sie dem Kanal übergeben wurde.

6. Das *Consensus*-Problem

In unserem Ansatz reicht es, um *Consensus* zu lösen, daß der Ausfalldetektor *schwache Vollständigkeit* erfüllt (Ausfalldetektor-Klasse $(W, .)$). Genauigkeitseigenschaften werden nicht vorausgesetzt. Verdächtigte Prozesse werden gegebenenfalls heruntergefahren. Abgestürzte Prozesse können hochgefahren werden.

Erfüllt der Ausfalldetektor:

a) *starke Vollständigkeit* sowie

-*schließliche Pfad-Genauigkeit* oder

-*schließlich q-schwache Genauigkeit,* falls Mon nicht schließlich separiert ist

bzw.

b) *mehrheitliche Vollständigkeit* sowie *schließlich p-mehrheitliche Genauigkeit*,

dann kann in unserem Modell *Consensus* gelöst werden, ohne falsch verdächtigte Prozesse herunterfahren zu müssen. In unserer Bezeichnung ist dies unter Zuhilfenahme von Ausfalldetektoren der Klassen $(S, \Diamond P)$ oder $(S, \Diamond qW)$ falls $\Diamond Mon \neq \|$ sowie $(M, \Diamond pM)$ möglich.

In [**15**] wird z.B. gezeigt, daß der Ausfalldetektor der Klasse $(W, \Diamond pS)$ der schwächste Ausfalldetektor ist, mit dessen Hilfe *Consensus* gelöst werden kann, falls eine Mehrheit der Prozesse aktiv ist. Wir zeigen, daß die Genauigkeitseigenschaften von $(W, \Diamond pS)$ noch abgeschwächt werden können, falls die Anforderungen an die Vollständigkeitseigenschaften erhöht werden, und zwar, daß es reicht, daß die obigen Eigenschaften des Ausfalldetektors für eine Mehrheit der Prozesse erfüllt werden (Ausfalldetektor-Klasse $(M, \Diamond pM)$). Unser Ergebnis ist im Einklang mit [**15**], denn wir zeigen, daß $(S, \Diamond pS)$ und $(M, \Diamond pM)$ äquivalent sind. Chandra und Toueg haben in [**17**] die Äquivalenz von $(W, \Diamond pS)$ und

$(S, \Diamond pS)$ bewiesen. Unser Ergebnis zeigt insbesondere, daß bestimmte Übereinstimmungsprobleme lösbar sind, falls eine Mehrheit der Überwachungsprozesse den Erwartungen entsprechend funktionieren.

7. Das *NB-AC*-Problem

In unserem Ansatz kann das *Non-Blocking Atomic Commitment*-Problem (*NB-AC*-Problem) gelöst werden. Zur Unlösbarkeit des *NB-AC*-Problems im Modell von Chandra und Toueg, siehe [**32**, Theorem 1 und Corollary 1].

Wir zeigen, daß in unserem Ansatz, in dem (falsch) verdächtigte Prozesse gegebenenfalls heruntergefahren werden können, das *NB-AC*-Problem auf *Consensus* reduzierber ist und somit mit Hilfe von Ausfalldetektor die *schwache Vollständigkeit* erfüllen lösbar ist. Abgestürzte Prozesse können gegebenenfalls hochgefahren werden.

Erfüllt der Ausfalldetektor *mehrheitliche Vollständigkeit* sowie *mehrheitliche Genauigkeit*, dann gibt es keine falschen Verdächtigungen und das *NB-AC*-Problem kann, ohne Prozesse herunterzufahren, gelöst werden.

8. Vergleich der Ansätze

Abschließend ergänzen wir die zusammenfassende Tabelle vom Ende des Kapitels Stand der Forschung durch den Vergleich mit unserem neuen *Forced-Shutdown*-Ansatz (**FS**).

	FS	WAC	SmS	FA	MOF	KK	VW	HB	SK
Consensus	+	+	+	+	+	+	+	+	+
NB-AC	+	-	?	?	?	?	?	?	?
Ausf.-De.	+	+	-	+	+	-	+	+	+
Restart	+/-	-	+	-	+	-	+	+	+
Netz.-Part.	-	-	+	-	+	-	+	+	+
Sys.-konv.	-	-	+	-	+	-	-	-	-
Her. Pr.	+	-	-	-	-	-	-	-	-
Erw. Eig.	+	?	?	+	-	?	-	-	+
Unk. Impl.	-	-	?	?	?	?	?	+	-

TABELLE 1. Vergleich der Ansätze (Ergänzung)

KAPITEL 5

Simulation von zuverlässigen Netzwerkverbindungen

Bevor wir unsere eigentlichen Themen angehen, untersuchen wir in diesem Kapitel, inwieweit in unserem Ansatz auf zuverlässige Netzwerkverbindungen verzichtet werden kann. Unser Augenmerk gilt den *Fair Lossy* Kanälen, die bei der Modellbeschreibung formal eingeführt und eingehend beschrieben wurden. Informell ist ein *Fair Lossy* Kanal ein unzuverlässiger Kanal, welcher keine Nachrichten erzeugt und welcher eine unendliche Teilmenge einer unendlichen Menge von Nachrichten weiterleitet.

Wir zeigen, daß in unserem Modell jedes Problem, welches mit zuverlässigen Verbindungen lösbar ist, auch mit Hilfe von *Fair Lossy* Kanälen lösbar ist. Dazu geben wir einen Algorithmus an, welcher eine zuverlässige Verbindung simuliert und dabei *Fair Lossy* Kanäle verwendet. Im Gegensatz zu [**9**] nehmen wir an, daß vereinfacht ausgedrückt, die Prozesse, die den Simulationsalgorithmus ausführen, hoch verfügbar sind, siehe auch [**34**] sowie [**52**]. Dies bedeutet insbesondere, daß eine Nachricht erfolgreich verschickt wird, sobald sie dem Kanal übergeben wird. Stürzt der Prozeß p ab, nachdem er erfolgreich eine Nachricht $m_{p,q}$ an den Kanal $C_{p,q}$ übergeben hat, so sorgt der Simulationsalgorithmus dafür, daß $m_{p,q}$ an q weitergereicht wird.

1. Grundlagen

Der folgende Algorithmus simuliert eine zuverlässige Verbindung und verwendet *Fair Lossy* Kommunikationskanäle. Der Algorithmus generiert zu jeder Nachricht $m_{p,q}$ eine fortlaufende Sequenznummer $s_{p,q}$. Die Menge der Sequenznummern der Form $s_{p,q}$ wird durch $Seq_{p,q}$ bezeichnet.

Der Algorithmus enthält fünf globale Variablen $OutBuf_{p,q}$, $OutSeqN_{p,q}$, $AckSeqN_{p,q}$, $InBuf_{p,q}$ und $InSeqN_{p,q}$. Die ersten drei Variable sind auf der Seite des Senders, die letzten zwei Variablen auf

der Seite des Empfängers angelegt.
Durch $OutBuf_{p,q}$ bzw. $InBuf_{p,q}$ wird Puffer für eine Nachricht beim Sender bzw. beim Empfänger bereitgestellt. $OutSeqN_{p,q}$ enthält die Sequenznummer der Nachricht $m_{p,q}$, die dem Kanal erfolgreich übergeben wurde. Die Variable $OutSeqN_{p,q}$ wird beim jeden erfolgreichen Übergeben einer Nachricht $m_{p,q}$ an den Kanal hochgezählt.
Die Variable $AckSeqN_{p,q}$ enthält die Sequenznummer der Nachricht, dessen Empfang zum letzten Mal vom Emfänger q bestätigt wurde. Analog enthält $InSeqN_{p,q}$ die Sequenznummer der letzen Nachricht, die beim Empfänger q eingegangen ist.

Der Kanal, dem die Nachricht $m_{p,q}$ übergeben wird, wird durch $C^m_{p,q}$ bezeichnet. Um die Nachricht $m_{p,q}$ dem Kanal $C^m_{p,q}$ zu übergeben, ruft der Prozeß p die Prozedur $R\text{-}send_{p,q}(m_{p,q})$ auf.

Wir sagen: "*p übergibt erfolgreich die Nachricht* $m_{p,q}$ *an den Kanal* $C^m_{p,q}$", falls die Prozedur $R\text{-}send_{p,q}(m_{p,q})$ erfolgreich ausgeführt wurde.

Wir sagen: "*p hat erfolgreich die Nachricht* $m_{p,q}$ *verschickt*", falls die Nachricht $m_{p,q}$ erfolgreich an den Kanal $C^m_{p,q}$ übergeben wurde.

Wir sagen: "*ein Prozeß q erhält eine Nachricht* $m_{p,q}$", falls $m_{p,q}$ in die globale Variable $InBuf_{p,q}$ geschrieben wird.

Der Algorithmus verwendet zwei unterschiedliche Kanäle $C^m_{p,q}$ sowie $C^s_{q,p}$. Der erste Kanal wird zur Übertragung der Nachricht aus $OutBuf_{p,q}$ verwendet, der zweite Kanal dient zur Übertragung der Bestätigung s von q nach p.

Um die Darstellungsweise zu vereinfachen, verwenden wir die Schreibweise: $(m, s, t) \in OutBuf_{p,q}$, falls zum Zeitpunkt t die Variable $OutBuf_{p,q}$ den Wert (m, s) enthält. Ist der Zeitpunkt t nicht relevant, so schreiben wir $(m, s) \in OutBuf_{p,q}$. Ähnliches gilt für die anderen globalen Variablen.

In den folgenden formalen Darstellungen bedeutet $succ\_send_{p,q}(m)$, daß die Prozedur $R\text{-}send_{p,q}(m)$ erfolgreich ausgeführt wurde.

Wir sagen: "$m_{p,q}$ *ist von* $m'_{p,q}$ *verschieden*" genau dann, wenn $m_{p,q}$ und $m'_{p,q}$ verschiedene Nachrichten sind, auch wenn sie inhaltlich identisch sind. Formal:

$$m_{p,q} \neq m'_{p,q} \Longleftrightarrow succ\_send_{p,q}\left(m_{p,q}\right) \neq succ\_send_{p,q}\left(m'_{p,q}\right)$$

2. Simulationsalgorithmus

2.1. Beschreibung des Algorithmus. In der Initialisierungsphase werden sowohl $OutBuf_{p,q}$ als auch $InBuf_{p,q}$ auf $\perp$ gesetzt. Dementsprechend wird sowohl $AckSeqN_{p,q}$ als auch $OutSeqN_{p,q}$ auf 0 gesetzt. Da die erste Nachricht, die erfolgreich verschickt wird, die Sequenznummer 1 hat (man beachte, daß in der Prozedur $R\text{-}send_{p,q}$ zuerst $OutSeq_{p,q}$ inkrementiert wird, bevor es dem Kanal übergeben wird), wird die Variable $InSeqN_{p,q}$ auf 1 initialisiert.

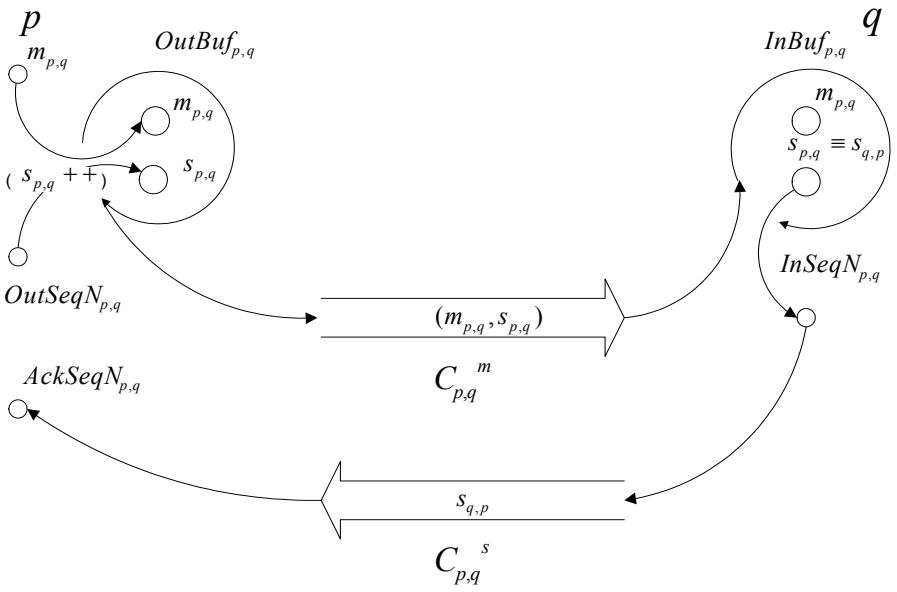


ABBILDUNG 1. Datenfluß

Um die Nachricht $m_{p,q}$ dem Kanal $C^m_{p,q}$ zu übergeben, ruft der Prozeß p die Prozedur $R\text{-}send_{p,q}\left(m_{p,q}\right)$ auf. Die Prozedur $R\text{-}send_{p,q}$ arbeitet transaktionsorientiert. Zuerst wird $OutSeqN_{p,q}$ gesperrt. Damit

wird erreicht, daß nur diese *Instance* von $R\text{-}send_{p,q}$ den Wert der Variable $OutSeqN_{p,q}$ ändern kann. Anschließend wird $OutSeqN_{p,q}$ innerhalb der Transaktion inkrementiert. Stürzt die *Instance* ab, dann wird $OutSeqN_{p,q}$ durch die *Rollback*-Anweisung auf den alten Wert zurückgesetzt.

Ist der Puffer $OutBuf_{p,q}$ gleich $\perp$, d.h. leer, dann wird das Tupel $(m_{p,q}, OutSeqN_{p,q})$ an die Variable $OutBuf_{p,q}$ zugewiesen. Somit ist die Nachricht m dem Kanal $C^m_{p,q}$ übergeben worden. Die Variable $OutSeqN_{p,q}$ wird durch die anschließende Anweisung Commit zum Schreiben freigegeben. Ist der Puffer $OutBuf_{p,q}$ nicht leer (d.h. gilt $OutBuf_{p,q} <> \perp$), so bedeutet dies, daß der Kanal $C^m_{p,q}$ die vorige Nachricht noch nicht erfolgreich [1]verschickt hat. In diesem Fall wird so lange innerhalb der Prozedur $R\text{-}send_{p,q}$ gewartet, bis $OutBuf_{p,q}$ wieder eine neue Nachricht aufnehmen kann. Nachdem der Kanal $C^m_{p,q}$ die Nachricht m erfolgreich an q verschickt hat, setzt er die Variable $OutBuf_{p,q}$ zurück (auf $\perp$).

Der Task *sender* verschickt über den Kanal $C^m_{p,q}$ periodisch den Inhalt von $OutBuf_{p,q}$. Ist $AckSeqN_{p,q}$ gleich der Sequenznummer $SeqN$ aus $OutBuf_{p,q}$, dann bedeutet dies, daß der Empfänger q die Nachricht mit der Sequenznummer $SeqN$ erhalten hat. Somit kann $OutBuf_{p,q}$ zurückgesetzt werden (auf $\perp$) und ist bereit eine neue Nachricht aufzunehmen. Ansonsten hat der Empfänger den Empfang der Nachricht mit der Sequenznummer $SeqN$ noch nicht bestätigt oder die Bestätigung hat die Senderseite des Kanals noch nicht erreicht und die Nachricht in $OutBuf_{p,q}$ wird über den Kanal $C^m_{p,q}$ (eventuell nochmals) verschickt.

Auf der Empfängerseite sorgt der Task $receiver\_m$ dafür, daß die erhaltene Nachricht m in den Puffer $InBuf_{p,q}$ geschrieben wird und somit dem Empfänger q zugänglich ist, sowie daß die Sequenznummer s der erhaltenen Nachricht an den Sender p zurückgeschickt wird. Ältere, außer der Reihe kommende Nachrichten, die schon einmal bestätigt wurden, werden weggeworfen.

Auf der Senderseite sorgt der Task $receiver\_s$, daß die Sequenznummer der letzten Nachricht, die vom Empfänger bestätigt wurde, in $AckSeqN_{p,q}$ geschrieben wird.

[1]d.h. der Kanal ist noch nicht davon überzeugt, daß der Empfänger die Nachricht erhalten hat

```
      /* Global Variables */
01          OutBuf_{p,q} ←⊥ /* on p side */
02          InBuf_{p,q} ←⊥ /* on q side */
03          OutSeqN_{p,q} ← 0 /* on p side */
04          AckSeqN_{p,q} ← 0 /* on p side */
05          InSeqN_{p,q} ← 1 /* on q side */

06      procedure R-send_{p,q}(m) {
07          begin
08          lock OutSeqN_{p,q};
09          OutSeqN_{p,q} + +;
10          repeat
11              if (OutBuf_{p,q} =⊥) then{
12                  OutBuf_{p,q} ← (m, OutSeqN_{p,q}) ;
13                  Commit;
14                  break;
15              }else
16                  sleep(δ) ;
17          forever;
18          end
19          exception
20              rollback;
21      };

22      cobegin

23      task sender : /* task executed by the sender p */
24          periodically execute{
25              (if AckSeqN_{p,q} = SeqN from OutBuf_{p,q}) then
26                  OutBuf_{p,q} ←⊥;
27              else
28                  send^m_{p,q}(OutBuf_{p,q}) ;
29          }
```

```
30    task receiver_m : /* task executed by the receiver q */
31        upon receive (m, s) from p{
32            if (s = InSeqN_{p,q}) then{
33                InSegN_{p,q} + +;
34                InBuf_{p,q} ← m;
35                send^s_{q,p} (s) ;
36            }else
37                send^s_{q,p} (s) ;
38        }
39    task receiver_s : /* task executed by the sender p */
40        upon receive (s) from q{
41            AckSeqN_{p,q} ← s;
42        }

43    coend
```

Simulationsalgorithmus (Code für den Kanal $C^m_{p,q}$ & $C^s_{q,p}$)

2.2. Korrektheitsbeweis.

LEMMA 1 (**Eindeutigkeit der Sequenznummer**). *Die Sequenznummer s jeder Nachricht m in $OutBuf_{p,q}$ ist eindeutig, d.h.:*

$$\begin{aligned} &\forall p \in Sen \,.\, \forall q \in \mathrm{Re}\, c \,.\, \forall\, (m,s)\,, (m',s') \in OutBuf_{p,q} \\ &m \neq m' \implies s \neq s' \end{aligned}$$

und wird fortlaufend vergeben.

BEWEIS. Die einzige Möglichkeit, einer Nachricht m eine Sequenznummer zuzuordnen, ist durch einen Aufruf der Prozedur *R-send*$_{p,q}$. Hier wird zuerst die globale Variable $OutSeqN_{p,q}$ gesperrt, d.h. werden zwei *Instance*s von *R-send*$_{p,q}$ fast gleichzeitig gestartet, so wartet eine *Instance* solange, bis die Sperre der Variablen $OutSeqN_{p,q}$ aufgehoben wird. Somit ist sichergestellt, daß die wartende *Instance* auf die Variable $OutSeqN_{p,q}$ nicht zugreifen kann, solange die Sperre nicht durch das Commit in der *repeat* Schleife aufgehoben wurde.

Insgesamt kann somit eine Sequenznummer höchstens nur einmal vergeben werden.

Stürzt eine *Instance* von $R\text{-}send_{p,q}$ ab, nachdem sie die globale Variable $OutSeq_{p,q}$ gesperrt und erhöht hat, so wird die Änderung des Statements $OutSeqN_{p,q}{+}{+}$ zurückgerollt, d.h. die Variable $OutSeqN_{p,q}$ wird nicht global erhöht. Das gleiche geschieht, falls die *Instance* $R\text{-}send_{p,q}$ auf einen Fehler läuft. Somit wird die Sequenznummer fortlaufend vergeben. ∎

LEMMA 2 (**Bestätigung**). *Jede Nachricht* (m, s)*, welche von* q *empfangen wurde, wird von* q *bestätigt und der Sender erhält schließlich die Bestätigung. Formal:*

$$\begin{aligned} \forall p \;&\in\; Sen \;.\; \forall q \in \mathrm{Re}\, c \;.\; \forall m \in Mes_{p,q} \;.\; \forall s \in Seq_{p,q} \;. \\ (m,s) \;&\in\; InBuf_{p,q} \implies s \in AckSeqN_{p,q} \end{aligned}$$

BEWEIS. Sei (m, s) eine Nachricht, die von p verschickt wurde und von q empfangen wurde. Dann wird im Task $receiver\_m$ die empfangene Sequenznummer s an p zurückgeschickt. Erhält der Prozeß p die Nachricht s und ist $AckSeqN_{p,q}$ gleich 0, dann wird $AckSeqN_{p,q}$ auf s gesetzt.
Hat der Prozeß p die Nachricht s nicht erhalten, dann verschickt der Task *sender* die Nachricht (m, s) noch einmal und q schickt wieder s als Bestätigung. Schickt p die Nachricht (m, s) unendlich oft, dann erhält q gemäß Eigenschaft *Fair Loss* des Kanals die Nachricht (m, s) unendlich oft und verschickt die Bestätigung s unendlich oft. Somit erhält p schließlich die Bestätigung s. ∎

LEMMA 3 (**Übergabe der Nachrichten**). *Jede Nachricht* $m_{p,q}$ *wird schließlich an den Kanal* $C^m_{p,q}$ *übergeben, falls die* Instance $R\text{-}send_{p,q}\,(m_{p,q})$ *nicht abstürzt. Formal:*

$$\begin{aligned} \forall p \quad &\in \quad Sen \;.\; \forall q \in \mathrm{Re}\, c \;.\; \forall m \in Mes_{p,q} \;. \\ succ\_send_{p,q}\,(m) \;&\implies\; (\exists s \in Seq_{p,q} \;.\; (m,s) \in OutBuf_{p,q}) \end{aligned}$$

BEWEIS. Sei $m_{p,q} \in Mes_{p,q}$ beliebig. Wir müsssen lediglich zeigen, daß die aufgerufene Prozedur $R\text{-}send_{p,q}\,(m_{p,q})$ nach endlicher Zeit

beendet wird, d.h. es gibt eine endliche Zeit, nachdem die globale Variable $OutBuf_{p,q}$ auf $\perp$ gesetzt wird. Die Aussage des Lemmas folgt dann sofort gemäß dem Algorithmus von $R\text{-}send_{p,q}$, da $OutBuf_{p,q}$ nur geändert werden kann, falls es vorher zurückgesetzt wurde (auf $\perp$) .

Die einzige Stelle, wo $OutBuf_{p,q}$ zurückgesetzt wird, befindet sich im Task *sender*. Hier kann $OutBuf_{p,q}$ nur zurückgesetzt werden, falls der Empfänger $receiver\_m$ die Bestätigung vom Empfänger erhält und falls die Sequenznummer $AckSeqN_{p,q}$ vom Sender gleich der Sequenznummer $SeqN$ aus $OutBuf_{p,q}$ ist. Da jede Nachricht, welche von p verschickt wird auch bestätigt wird (siehe Lemma (2)), wird $OutBuf_{p,q}$ schließlich immer zurückgesetzt. ∎

LEMMA 4 (**Nachrichten gehen nicht verloren**). *Jede Nachricht $m_{p,q}$, welche zum Kanal $C^m_{p,q}$ geleitet wurde, wird vom Prozeß q empfangen. Formal:*

$$\begin{aligned} \forall p \;&\in\; Sen \;.\; \forall q \in \mathrm{Re}\, c \;.\; \forall m \in Mes_{p,q} \;.\; \forall s \in Seq_{p,q} \;. \\ (m,s) \;&\in\; OutBuf_{p,q} \implies (m,s) \in Inbuf_{p,q} \end{aligned}$$

BEWEIS. Jede Nachricht $m_{p,q}$, welche zum Kanal geleitet wurde, wird solange an den Empfänger q durch $C^m_{p,q}$ verschickt, bis p die Bestätigung von q erhalten hat, eventuell wird $m_{p,q}$ unendlich oft verschickt. Aufgrund der Eigenschaft *Fair Loss* des Kanals $C^m_{p,q}$ erhält q schließlich die Nachricht $m_{p,q}$. ∎

LEMMA 5 (**Totale Ordnung**). *Der Prozeß q erhält alle Nachrichten $m_{p,q}$, und zwar genau in der Reihenfolge in der sie p verschickt hat. Formal:*

$$\begin{aligned} \forall p \;&\in\; Sen \;.\; \forall q \in \mathrm{Re}\, c \;.\; \forall\, (m,s,t)\,, (m',s',t') \in OutBuf_{p,q} \;. \\ (\; t \;&<\; t' \;.\; s \neq s') \\ &\implies \\ \exists t^{(2)}, t^{(3)} \;&\in\; T \;.\; (m,s,t^{(2)}), (m',s',t^{(3)}) \in InBuf^{t^{(2)}}_{p,q} \;.\; t^{(2)} < t^{(3)} \end{aligned}$$

BEWEIS. Gemäß Lemma (3) und Lemma (4) erhält q alle Nachrichten $m_{p,q}$, die p verschickt hat. Aufgrund des Lemmas (1) folgt sofort $s < s'$. Da (m', s') erst durch den Kanal $C^m_{p,q}$ verschickt werden kann, nachdem q den Erhalt von (m, s) bestätigt hat und die Bestätigung in $AckSeqN_{p,q}$ geschrieben wurde, folgt sofort $t^{(2)} < t^{(3)}$ ∎

LEMMA 6 (**Kein Duplizieren**). *Der Empfänger q erhält keine Nachrichten $m_{p,q}$ außer der Reihe.*

$$\begin{aligned} \forall p \;&\in\; Sen \;.\; \forall q \in \mathrm{Re}\, c \;.\; \forall (m,s,t),(m',s',t') \in InBuf_{p,q} \;. \\ s \;&<\; s' \implies t < t' \end{aligned}$$

BEWEIS. Folgt sofort aus der Eindeutigkeit der Sequenznummer (Lemma (1)) sowie aus der totalen Ordnung der Nachrichten (Lemma (5)). ∎

THEOREM 1. *Der obige Simulationsalgorithmus verwandelt* Fair Lossy *Kanäle in zuverlässige Kommunikationskanäle. Die Nachrichten werden total geordnet, falls die Prozedur R-send sequentiell ausgeführt wird.*

BEWEIS. Die Eigenschaft *No Creation* folgt aus der entsprechenden Eigenschaft des *Fair Lossy* Kanals, die Eigenschaft *No Duplication* folgt aus Lemma (6). Die Eigenschaft *No Loss* folgt aus Lemma (4). Die totale Ordnung der Nachrichten ist Inhalt des Lemmas (5). ∎

REMARK 7. *Die Aussage des Satzes (1), bis auf die Totale Ordnung der Nachrichten, bleibt erhalten, falls die Prozedur R-send konkurrent (z.B. in einem* Thread*) aufgerufen wird. Die Reihenfolge der Nachrichten hängt davon ab, wie einzelnen* Threads *Ressourcen zugewiesen werden.*■

KAPITEL 6

Äquivalenz von Ausfalldetekor-Eigenschaften

1. Einleitung

In diesem Kapitel setzen wir die Beschreibung der theoretischen Grundlagen fort, indem wir Begriffe wie Algorithmus, Run eines Algorithmus, Problem sowie Aggregationsvorschrift einführen. Die Terminologie sowie die grundlegenden Definitionen sind der Arbeit [**17**] entnommen.

Informell ist ein Algorithmus eine Ansammlung von n deterministischen Automaten. Ein Run (eines Algorithmus) ist eine Ablaufbeschreibung eines konkreten Algorithmus ausgehend von einer Anfangskonfiguration und einem Ausfallszenario. Informell ist ein Problem $\mathbb{P}$ eine Menge von Eigenschaften, die das System erfüllen soll. Wir werden derartige Eigenschaften durch Verwenden von Ausfalldetektoren garantieren.

Sei $\mathbb{D}$ ein Ausfalldetektor, welcher *schwache Vollständigkeit* erfüllt und sei $\mathbb{D}'$ ein Ausfalldetektor, welcher *starke Vollständigkeit* erfüllt. Wir zeigen im weiteren Verlauf des Kapitels, daß *schwache Vollständigkeit* und *starke Vollständigkeit* in unserem Modell äquivalent sind, d.h. jedes Problem, welches mit Hilfe des Ausfalldetektors $\mathbb{D}'$ gelöst werden kann, kann auch mit Hilfe von $\mathbb{D}$ gelöst werden und umgekehrt.

2. Grundlagen

In diesem Kapitel setzen wir die Beschreibung des Modells fort. Im folgenden bezeichnet ein Algorithmus A eine Ansammlung von n deterministischen Automaten, wobei jeder Automat einen Prozeß modelliert.

Die Menge aller Algorithmen bezeichnen wir durch $\mathbb{A}$. Die deterministischen Automaten arbeiten asynchron, es gibt keinen gemeinsamen Takt. Die Automaten werden durch den Empfang von externen Nachrichten getriggert.

Im Modell von Chandra und Toueg, siehe [**17**], erfolgt die Berechnung in Schritten von A. In jedem Schritt kann ein Prozeß alternativ eine der folgenden Aktionen ausführen:
(1) eine Nachricht empfangen,
(2) eine Nachricht verschicken,
(3) einen Zustandswechsel vornehmen, oder
(4) sein Ausfalldetektormodul abfragen.

In unserem Ansatz kann ein Prozeß in einem Schritt:
(1) eine Nachricht empfangen,
(2) eine Nachricht an einen Prozeß verschicken, oder
(3) einen Zustandswechsel vornehmen.

Überwachungsprozesse können in unserem Ansatz zusätzlich in einem Schritt:
(4) den Zustand eines Prozesse ermitteln, der von ihnen überwacht wird.

Im Gegensatz zum Modell von Chandra und Toueg kann in unserem Modell ein Algorithmus den Systemablauf extern beeinflussen, indem er z.B. fälschlicherweise verdächtigte Prozesse herunterfährt bzw. abgestürzte Prozesse hochfährt. Um den Zustandsübergang eines Prozesses p extern zu veranlassen, schickt der Algorithmus eine entsprechende Nachricht an den Prozeß p. Beim Empfang dieser Nachricht vollzieht der Prozeß p den Zustandswechsel.

Sei A ein Algorithmus. Wir bezeichnen den Teil von A, welcher externe Systemablauf-Korrekturen vornimmt und steuert, als *Systemablauf-Korrektor* von A (als Zeichen K_A).

Sei A ein Algorithmus, sei S_i ein interner Systemablauf und sei $\mathbb{D}$ ein Ausfalldetektor, welcher von A verwendet wird. Dann bezeichnen wir durch $K_A(S_i, \mathbb{D})$ die externe Systemablauf-Korrektur des Algorithmus A für den internen Systemablauf S_i.

Der Systemablauf-Korrektor eines Algorithmus A kann verschiede Aktionen durchführen, und zwar:

Recovery: Prozesse bzw. Applikationen können eine vom Systemablauf-Korrektor eingeleitete *Recovery* durchlaufen.

Forced-Shutdown: Prozesse können vom Systemablauf-Korrektor heruntergefahren werden.

Um den Systemablauf-Korrektor einsetzen zu können, verlangen wir die Eigenschaft:

Non-Triviality: Der Algorithmus A darf im allgemeinen durch die Systemablauf-Korrektur nicht eine triviale Lösung liefern.

Die Eigenschaft *Non-Triviality* stellt z.B., wie wir sehen werden, beim *Consensus*-Algorithmus sicher, daß der Korrektor in der Regel nicht alle Prozesse, die am *Consensus* teilnehmen, herunterfährt und somit die triviale Lösung anstrebt.

DEFINITION 24 (**Run eines Algorithmus**). *Ein Run R_A des Algorithmus A unter Verwendung eines verteilten Ausfalldetektors $\mathbb{D}$ ist ein Tupel $R_A :=< \mathcal{S}_i, H_{\mathbb{D}}, I, St >$, wobei gilt:*
$\mathcal{S}_i$ ist ein interner Systemablauf,
$H_{\mathbb{D}} \in \mathbb{D}(\mathcal{S}_i * K_A(S_i, \mathbb{D}))$ ist eine History *des Ausfalldetektors $\mathbb{D}$ für den Systemablauf $\mathcal{S}_i * K_A(S_i, \mathbb{D})$,*
I ist eine initiale Konfiguration und
St ist eine unendliche Folge von Schritten von A.

Die Menge aller Runs eines Algorithmus A bezeichnen wir durch $\mathcal{R}(A)$.■

Informell ist ein **Problem** $\mathbb{P}$ eine Menge von Eigenschaften, die das System erfüllen soll.

DEFINITION 25 (**Lösung eines Problems**). *Ein Algorithmus A löst ein Problem $\mathbb{P}$ unter Benutzung eines Ausfalldetektors $\mathbb{D}$, falls alle Runs von A unter $\mathbb{D}$ die geforderten Eigenschaften von $\mathbb{P}$ erfüllen.*

Sei $\mathbb{C}$ eine Klasse von Ausfalldetektoren.
Ein Algorithmus A löst das Problem $\mathbb{P}$ unter Benutzung von $\mathbb{C}$, falls für alle $\mathbb{D} \in \mathbb{C}$ der Algorithmus A das Problem $\mathbb{P}$ unter Benutzung von $\mathbb{D}$ löst.

Ein Problem $\mathbb{P}$ kann unter Benutzung von $\mathbb{C}$ gelöst werden, falls ein Algorithmus A existiert, welcher $\mathbb{P}$ für die Klasse $\mathbb{C}$ löst.■

Sei v eine Vektor-Variable im Algorithmus A, die eine Komponente für jeden Prozeß enthält. Wir bezeichnen durch v_q die Komponente des Prozesses q von v. Die *History* von v in einem Run R bezeichnen wir durch v^R und setzen $v_q^R(t)$ für den Wert von v_q zum Zeitpunkt t in einem Run R.

Sei A ein Algorithmus. Wir sagen: "der Prozeß p kann mit dem Prozeß q in einem Run R vollständig kommunizieren", falls im Run R zu keinem Zeitpunkt $t \in T$ gilt: $p \in \mathbb{D}_q(t)$ oder $q \in \mathbb{D}_p(t)$.

2.1. Verbesserung der Fehlerdetektion. Seien $\mathbb{D}$ und $\mathbb{D}'$ Ausfalldetektoren und sei $\mathbb{P}$ ein Problem, welches mit Hilfe von $\mathbb{D}'$ gelöst werden kann. Seien $q_1, q_2, ..., q_m$ die Überwachungsprozesse und seien $p_1, p_2, ..., p_n$ die Applikationsprozesse des Systems. Sei $q \in Mon$ beliebig. Sei $w_{i,j} \in SV_P^\infty$ die mit Hilfe von $\mathbb{D}$ erstellte Abschätzung von q_i über den Zustand von p_j. Sei im Gegensatz dazu $w_{i,j}^q \in SV_P^\infty$ die mit Hilfe von $\mathbb{D}$ erstellte Abschätzung von q_i über den Zustand von p_j, die q wahrnimmt. Sei $status^q := \left(w_{i,j}^q\right)_{\substack{i=1,m \\ j=1,n}}$ und sei $status_{q_i}^q := \left(w_{q_i,j}^q\right)_{j=1,n}$ sowie $status_{q_i} := (w_{q_i,j})_{j=1,n}$ für jedes $i = 1, 2, ..., m$. Somit ist $status_{q_i}$ der Statusvektor der Applikationsprozesse, den q_i mit Hilfe von $\mathbb{D}$ erstellt und $status_{q_i}^q$ ist die Sicht über $status_{q_i}$ die q hat.

Sei Z eine Vorschrift, welche die Detektionswerte $\left(w_{i,j}^q\right)_{\substack{i=1,m \\ j=1,n}}$ von $\mathbb{D}$ verwendet, um Detektionswerte $aggreg^q = (v_{q,j})_{j=1,n} \in SV_P^\infty$ im Sinne von $\mathbb{D}'$ zu generieren. Dementsprechend generiert Z einen virtuellen Ausfalldetektor $\mathbb{D}''$. Wir werden in diesem Abschnitt zeigen, daß in diesem Fall $\mathbb{P}$ auch mit Hilfe von $\mathbb{D}$ unter Verwendung von Z lösbar ist.

Vereinfacht dargestellt, verfügen wir über einen weniger präzisen Ausfalldetektor $\mathbb{D}$. Anhand der Information, die die einzelnen Überwachungsprozesse liefern, möchten wir mit Hilfe von Z präzisere Statusinformationen über die Applikationsprozesse gewinnen (in der Güte von $\mathbb{D}'$), so daß wir in die Lage versetzt werden, $\mathbb{P}$ zu lösen. Wir werden im folgenden von der Aggregation der Zustandsinformation bzw. von der Aggregationsvorschrift Z sprechen. Ist die Aggregationsvorschrift unabhängig von der Wahl von $\mathbb{D}$ oder wird $\mathbb{D}$ vorausgesetzt, dann verzichten wir auf das Subskript.

DEFINITION 26 (**Aggregation**). *Sei $q \in Mon$ und sei $p \in App$. Seien m, n sowie $\left(w_{i,j}^{q}\right)_{\substack{i=1,m \\ j=1,n}}$ wie oben definiert. Dann wird die Funktion $aggreg_{p_k}^{q}$ mit Hilfe von Z wie folgt:*

$$\begin{aligned} aggreg_{p}^{q} : \left(SV_P^{\infty}\right)^m &\rightarrow SV_P^{\infty} \\ \left(w_{1,p}^{q}, w_{2,p}^{q}, ..., w_{m,p}^{q}\right) &\longmapsto Z\left(\left(w_{i,p}^{q}\right)_{i=1,m}\right) \end{aligned}$$

definiert.■

Die Aggregationsvorschrift Z verwendet den Ausfalldetektor $\mathbb{D}$, um die Variable $aggreg^q$ für jeden Überwachungsprozeß q zu aktualisieren und generiert somit den virtuellen Ausfalldetektor $\mathbb{D}''$. Die Variable $aggreg^q$, im folgenden *Emulationsvariable* genannt, *emuliert* (bildet nach) die Ausgabe von $\mathbb{D}'_q$.

Sei $R :=< \mathcal{S}, H_{\mathbb{D}}, I, St >$ ein Run.

Die möglichen *Histories* $\left\{\left(aggreg^q\right)^R : q \in Mon\right\}$ in einem Run R (im folgenden $\mathbb{A}\left(H_{\mathbb{D}}\right)$ bezeichnet) sind die Ausfalldetektor-*Histories* des (mit Hilfe von Z simulierten) Ausfalldetektors $\mathbb{D}''$. Eine bestimmte *History* hängt auch vom Laufzeitverhalten der Nachrichten ab, die im Zusammenhang mit der Aggregationsvorschrift Z entstehen und bei q landen. Wir werden im folgenden von einer Aggregations-*History* sprechen, dabei meinen wir eine Ausfalldetekor-*History* von $\mathbb{D}''$.

DEFINITION 27 (**Aggregations-History**). *Die Aggregationsfunktion $\mathbb{A}$ in einem Run $R :=< \mathcal{S}, H_{\mathbb{D}}, I, St >$ ist eine nichtdeterministische Funktion:*

$$\begin{aligned} \mathbb{A} : \mathbb{D}(\mathcal{S}) &\rightarrow \mathbb{D}''(\mathcal{S}) \\ H_{\mathbb{D}} &\longmapsto Z\left(H_{\mathbb{D}}\right) \in \mathbb{A}\left(H_{\mathbb{D}}\right) \end{aligned}$$

Dabei bezeichnet $\mathbb{A}\left(H_{\mathbb{D}}\right) \subseteq \mathbb{D}''(\mathcal{S})$ ***die möglichen Aggregations-Histories*** *des Ausfalldetektors $\mathbb{D}$ im Run R.*■

DEFINITION 28 (**Emulation**). *Die Aggregationsvorschrift Z emuliert $\mathbb{D}'$ mit Hilfe von $\mathbb{D}$ genau dann, wenn für jeden Run $R := < \mathcal{S}, H_{\mathbb{D}}, I, St >$ gilt $\mathbb{A}\left(H_{\mathbb{D}}\right) \subseteq \mathbb{D}'(\mathcal{S})$.*■

Durch die Aggregation wird $\mathbb{D}$ so weit verbessert, daß der von Z auf Basis von $\mathbb{D}$ generierte virtuelle Ausfalldetektor $\mathbb{D}''$die Güte von $\mathbb{D}'$ erreicht.

DEFINITION 29. *Seien $\mathbb{D}$ und $\mathbb{D}'$ Ausfalldetektoren. Gibt es eine Aggregationsvorschrift Z, welche $\mathbb{D}'$ mit Hilfe von $\mathbb{D}$ emuliert, dann schreiben wir "$\mathbb{D} \succeq \mathbb{D}'$" und sagen: "$\mathbb{D}$ ist* mindestens so stark *wie $\mathbb{D}'$" bzw. "$\mathbb{D}$ ist* nicht schwächer *als $\mathbb{D}'$".*

Analog, seien $\mathbb{C}$ und $\mathbb{C}'$ zwei Klassen von Ausfalldetektoren. Gibt es zu jedem $\mathbb{D} \in \mathbb{C}$ ein Ausfalldetektor $\mathbb{D}' \in \mathbb{C}'$, so daß $\mathbb{D} \succeq \mathbb{D}'$, dann schreiben wir "$\mathbb{C} \succeq \mathbb{C}'$" und sagen: "$\mathbb{C}$ ist mindestens so stark *wie $\mathbb{C}'$" bzw. "$\mathbb{C}$ ist* nicht schwächer *als $\mathbb{C}'$".*■

DEFINITION 30 (**Äquivalenz**). *Seien $\mathbb{D}$ und $\mathbb{D}'$ Ausfalldetektoren. Gilt $\mathbb{D} \succeq \mathbb{D}'$ und $\mathbb{D}' \succeq \mathbb{D}$, dann schreiben wir "$\mathbb{D} \cong \mathbb{D}'$" und sagen: "$\mathbb{D}$ und $\mathbb{D}'$ sind* äquivalent*".*

Analog, seien $\mathbb{C}$ und $\mathbb{C}'$ zwei Klassen von Ausfalldetektoren. Gilt $\mathbb{C} \succeq \mathbb{C}'$ und $\mathbb{C}' \succeq \mathbb{C}$, dann schreiben wir "$\mathbb{C} \cong \mathbb{C}'$" und sagen: "$\mathbb{C}$ und $\mathbb{C}'$ sind äquivalent*".*■

Es ist leicht zu zeigen, daß die Relation $\succeq$ transitiv ist.

Sei eine Aggregationsvorschrift Z gegeben. Jedes Problem $\mathbb{P}$, welches mit Hilfe des Ausfalldetektors $\mathbb{D}'$ gelöst werden kann, kann auch mit dem Ausfalldetektor $\mathbb{D}$ gelöst werden. Um dies einzusehen, nehmen wir an, der Algorithmus A', welcher $\mathbb{P}$ löst, braucht den Ausfalldetektor $\mathbb{D}'$, aber nur der Ausfalldetektor $\mathbb{D}$ ist verfügbar.
Wir können aber wie folgt verfahren.
Um $\mathbb{D}'$ zu emulieren, verwenden wir konkurrent zu A' die Aggregationsvorschrift Z , indem wir den Algorithmus A' beim Überwachungsprozeß q wie folgt ändern: q liefert als Zustandsvektor den aktuellen Wert $aggreg^q$, welcher von Z erzeugt wird.

Wir halten fest:

LEMMA 7. *Seien $\mathbb{D}$ und $\mathbb{D}'$ Ausfalldetektoren, sei Z eine Aggregationsvorschrift, welche $\mathbb{D}'$ mit Hilfe von $\mathbb{D}$ emuliert. Jedes Problem $\mathbb{P}$, welches mit $\mathbb{D}'$ gelöst werden kann, kann auch mit dem Ausfalldetektor $\mathbb{D}$ mit Hilfe von Z gelöst werden.*

BEWEIS. Sei $\mathbb{P}$ ein beliebiges Problem, welches mit Hilfe von $\mathbb{D}'$ gelöst werden kann. Somit gibt es einen Algorithmus A', welcher das Problem $\mathbb{P}$ mit $\mathbb{D}'$ löst. Wir werden zeigen, daß es einen Algorithmus A gibt, welches $\mathbb{P}$ mit Hilfe von $\mathbb{D}$ löst.
Sei A die Abwandlung von A', indem jeder Überwachungsprozeß q als Abschätzung den von $aggreg^q$ gelieferten Wert enthält. Sei $Z(H_{\mathbb{D}}) \in \mathbb{A}(H_{\mathbb{D}})$ beliebig und sei $R :=< \mathcal{S}, Z(H_{\mathbb{D}}), I, St >$ gemäß Definition von A ein Run von A. Definitionsgemäß gilt $Z(H_{\mathbb{D}}) \in \mathbb{D}'(\mathcal{S})$. Somit ist R ein Run von A'. Da $\mathbb{P}$ laut Voraussetzung für alle Runs von A' gilt, gilt $\mathbb{P}$ auch für R. Da R beliebig gewählt worden war, gilt die Behauptung. ∎

COROLLARY 1. *Seien $\mathbb{C}$ und $\mathbb{C}'$ zwei Klassen von Ausfalldetektoren. Gilt $\mathbb{C} \succeq \mathbb{C}'$ und ist ein Problem $\mathbb{P}$ unter $\mathbb{C}'$ lösbar, dann ist $\mathbb{P}$ auch unter $\mathbb{C}$ lösbar.*

3. Emulation von *starker Vollständigkeit*

Wir beschreiben im folgenden eine konkrete Emulation Z, die *schwache Vollständigkeit* in *starke Vollständigkeit* überführt.

Wie bekannt, ist $SV_P^\infty := \{\uparrow, \dot{+}, \infty\}$ der Wertebereich der Zustandswerte, die von den Überwachungsprozessen geführt werden können.

Sei m die Anzahl der Überwachungsprozesse. Wir führen im folgenden einen *formalen Verknüpfungsoperator* $\star : (SV_P^\infty)^m \longrightarrow SV_P^\infty$ ein, mit dessen Hilfe die Aggregationsvorschrift Z bestimmt wird. Das Ziel dieses Verknüpfungsoperators besteht darin, einen Applikationsprozeß als abgestürzt zu kennzeichnen, sobald mindestens ein Überwachungsprozeß ihn als abgestürzt einstuft.

DEFINITION 31. *Sei m wie oben die Anzahl der Überwachungsprozesse und sei ein formaler Verknüpfungsoperator $\star : (SV_P^\infty)^m \longrightarrow SV_P^\infty$ wie folgt*

$$(a_i)_{i=1,m} \longmapsto \begin{cases} a_1 \textit{ falls } \forall k \in \{1,2,...,m\} \,.\, a_k = a_1 \\ \dot{+} \textit{ falls } \exists i \in \{1,2,...,m\} \,.\, a_i = \dot{+} \\ \uparrow \textit{ sonst} \end{cases}$$

definiert.■

Sei $last\_status^q$ jeweils der letzte Stand der Matrix-Variable $status^q = \left(w^q_{i,j}\right)_{\substack{i=1,m\\ j=1,n}}$. Sei $l \in \{1, 2, ..., m\}$.

Wir setzen:

$$last\_status^q_{q_l} := \left(w^q_{q_l,j}\right)_{j=1,n}$$

sowie:

$$v_{q,j} := \star\left(w^q_{k,j}\right)_{k=1,m} \qquad j = 1, 2, ..., n$$

bzw.

$$\star last\_status^q := \left(v_{q,j}\right)_{j=1,n}$$

Dabei ist $\left(w^q_{q_l,j}\right)_{j=1,n}$ der jeweils letzte Stand von $status^q$. Somit ist $last\_status^q_{q_l}$ jeweils die letzte Sicht, die q über die von q_l gelieferten Statuswerte hat.

Sei im folgendem $\mathbb{D}$ ein Ausfalldetektor, welcher *schwache Vollständigkeit* erfüllt. Die folgende Aggregationsvorschrift Z im *Pseudo-Code* wiedergegeben, emuliert mit Hilfe von $\mathbb{D}$ die Ausgabe eines starken Ausfalldetektors $\mathbb{D}'$. Analoge Ergebnisse wurden auch von Chandra und Toueg (siehe [**17**]) in ihrem Modell erzielt.

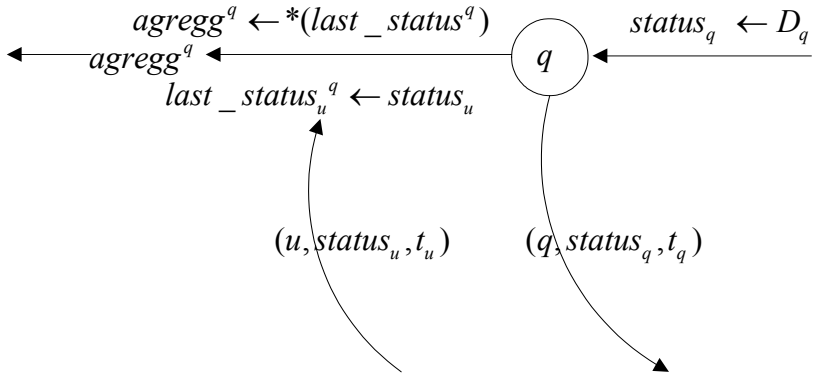


ABBILDUNG 1. **Datenfluß der Überwachungsprozesse**

Das Prädikat $receive\_no\_spurious\_message\,\{u, status_u, t_u\}$ testet im folgenden Algorithmus, ob der Zeitstempel t_u kleiner ist als der zuletzt Empfangene. Ist dies der Fall, dann wird die Statusinformation nicht weiter berücksichtigt.

Wir halten fest:

REMARK 8 (**Out_Of_Order**). *Nachrichten mit veralteter Statusinformation werden ignoriert.*

Jeder Überwachungsprozeß q führt folgende Anweisungsfolge aus (siehe auch Abbildung (1)):

ALGORITHM 1.

```
01  aggreg^q ← ∞;

02  cobegin

|| Task 1:
03  repeat forever{
      /* q ermittelt den Zustand der überwachten Prozesse */
04        status_q ← 𝔻_q;
05        t_q ← clock_q;
06        send (q, status_q, t_q) an alle Überwachungsprozesse
                    des Zustandsdetektormoduls, zu dem q gehört;
          /* t_q wird auf den aktuellen Wert der internen Uhr gesetzt
          und stellt sicher, daß die Nachrichten, die außer der Reihe
          kommen, vom Empfänger ignoriert werden */
07  }

|| Task 2:
08  if receive_no_spurious_message (u, status_u, t_u) {
          /* Nachrichten außer der Reihe werden ignoriert */
09        last_status_u^q ← status_u;
          /* last_status_u^q enthält den letzten, vom
          Prozeß u erhaltenen Statusvektor */
10        aggreg^q ← ⋆(last_status^q) ;
11  }

12        coend
```

Implementierung von Z (Protokolle der Überwachungsprozesse zur Berechnung der Aggregationswerte)

Wir verwenden folgende Abkürzungen:

P1 :=**Emulation von starker Vollständigkeit mit Hilfe von schwacher Vollständigkeit**

♦

P2 :=**Erhalt von kontinuierlich starker Genauigkeit**

♦

P′2 :=**Erhalt von kontinuierlich** pq**-starker Genauigkeit**

♦

Wir werden zeigen, daß die Eigenschaft $P1$ erfüllt wird. Im allgemeinen wird aber die Eigenschaft $P2$ nicht erfüllt.

LEMMA 8. *Z erfüllt die Eigenschaft $P1$.*

BEWEIS. Sei im folgenden $\mathbb{D} \in (W, .)$ und sei $R :=< \mathcal{S}, H_{\mathbb{D}}, I, St >$ ein beliebiger Run. Sei p ein beliebiger Prozeß, welcher abstürzt und dessen Zustand sich nicht mehr ändert.

Wir zeigen:

Wird *schließlich* p von wenigstens einem funktionellen Überwachungsprozeß permanent in $H_{\mathbb{D}}$ im Run R als abgestürzt eingestuft, so wird p *schließlich* von allen funktionellen Überwachungsprozessen in $Z(H_{\mathbb{D}})$ permanent als abgestürzt eingestuft.

Formal:

$$\begin{array}{l} \forall \mathbb{D} \in (W, .) \; . \; \forall R :=< \mathcal{S}, H_{\mathbb{D}}, I, St > \; . \; \forall Z(H_{\mathbb{D}}) \in \mathbb{A}(H_{\mathbb{D}}) \\ \forall p \in App \; . \; \forall t \in T \; . \; perm\_crash(\mathcal{S}, p, t) \implies \\ \left(\begin{array}{l} (\exists q \in Func(\mathcal{S}, p) \; . \; event\_perm\_susp(\mathcal{S}, H_{\mathbb{D}}, q, p, t)) \\ \quad \implies \\ (\forall q \in Func(\mathcal{S}, p) \; . \; event\_perm\_susp(\mathcal{S}, Z(H_{\mathbb{D}}), q, p, t)) \end{array} \right) \end{array}$$

Es gibt gemäß Voraussetzung, da $\mathbb{D}$ *schwache Vollständigkeit* erfüllt, einen Überwachungsprozeß q_0 und einen Zeitpunkt t_0, so daß q_0 ab dem Zeitpunkt t_0 den Prozeß p permanent als abgestürzt einstuft. Es gibt somit aufgrund von Task 1 der Aggregationsvorschrift Z einen Zeitpunkt $t_1 \in T\ .\ (t_0 < t_1)$, so daß q_0 zum Zeitpunkt t_1 eine Nachricht der Form $(q_0, status_{q_0}, t_0)$ an alle Überwachungsprozesse schickt.

Sei q ein beliebiger Überwachungsprozeß, welcher p überwacht. Da der Prozeß q schließlich alle Nachrichten erhält, die an ihn geschickt wurden, existiert ein Zeitpunkt $t_2 \in T\ .\ (t_2 > t_1)$, nachdem q die Nachricht $(q_0, status_{q_0}, t_0)$ erhalten hat. Die Zustandsinformationen werden im Task 2 zusammengefügt, d.h. es wird $\star last\_status^q$ dort gebildet. Da $\forall a_{i \in 1,m} \in (SV_P^\infty)^m$ die Aussage $\star\,(a_i)_{i \in 1,m} = \dot{+}$ gilt, falls $\exists i \in 1, m$ mit $a_i = \dot{+}$, existiert somit ein Zeitpunkt $t_3 \in T\ .\ (t_3 > t_2)$, so daß q den Prozeß p als abgestürzt zum Zeitpunkt t_3 in $Z\,(H_{\mathbb{D}}) := aggreg^R$ einstuft.

Wir zeigen, daß q den Prozeß p als abgestürzt zu einem beliebigen Zeitpunkt $t \in T\ .\ (t > t_3)$ in $Z\,(H_{\mathbb{D}})$ einstuft.
Da q die Nachrichten, die außer der Reihe kommen, identifizieren kann, siehe auch Remark (8), berücksichtigt q vom Überwachungsprozeß q_0 ab dem Zeitpunkt t_2 nur Nachrichten $(q_0, status_{q_0}, t')$ mit $t' > t_0$. Gemäß Voraussetzung stuft q_0 den Prozeß p nach dem Zeitpunkt t_0 als abgestürzt ein. Somit erhält q von q_0 über den Zustand von p ab dem Zeitpunkt t_1 ausschließlich die Einschätzung $\dot{+}$. Die gleichen Überlegungen wie oben ergeben die Behauptung.
∎

LEMMA 9. *Eigenschaft* $\mathbf{P'2}$ *wird im allgemeinen von* Z *nicht erfüllt.*

Der Grund weshalb im allgemeinen *kontinuierlich pq-starke Genauigkeit* nicht erhalten bleibt, liegt in der Verknüpfung $\star\,(\uparrow, \dot{+})$. Wird diese Verknüpfung als aktiv $(\uparrow)$ definiert, dann werden Genauigkeitseigenschaften erhalten, aber *starke Vollständigkeit* nicht mit Hilfe von *schwacher Vollständigkeit* emuliert. Wir geben nun ein Gegenbeispiel.

BEWEIS. Sei im folgenden $\mathbb{D} \in (W, .)$ beliebig. Wir zeigen, daß es einen Run $R :=< \mathcal{S}, H_{\mathbb{D}}, I, St >$, einen aktiven Applikationsprozeß p sowie einen funktionellen Überwachungsprozeß q gibt, so daß p niemals *permanent* von q in $H_{\mathbb{D}}$ im Run R verdächtigt wird. Unter

diesen Voraussetzungen wird p *permanent* von bezüglich p funktionellen Überwachungsprozessen in $Z\,(H_{\mathbb{D}})$ verdächtigt, so daß $\mathbf{P'2}$ verletzt wird.

Formal:

$$\exists \mathbb{D} \in (W,.)\ .\ \exists R :=< \mathcal{S}, H_{\mathbb{D}}, I, St >\ .\ \exists Z\,(H_{\mathbb{D}}) \in \mathbb{A}\,(H_{\mathbb{D}})\ .$$
$$(\forall p \in App\ .\ \exists q \in Func\,(\mathcal{S}, p)\ .\ \forall t \in T\ .\ \exists t' \geq t\ .$$
$$H_{\mathbb{D}}\,(q,p,t') \neq \infty\ \wedge\ \neg\ false\_susp(\mathcal{S}, H_{\mathbb{D}}, q, p, t'))$$
$$\wedge$$
$$(\exists p \in App\ .\ \forall q' \in Func\,(\mathcal{S}, p)\ .\ \exists t \in T\ .\ \forall t'' \geq t\ .$$
$$Z\,(H_{\mathbb{D}})\,(q',p,t'') = \infty\ \vee\ false\_susp(\mathcal{S}, Z\,(H_{\mathbb{D}})\,, q', p, t''))$$

Sei $p \in perm\_activ\,(\mathcal{S})$ ein beliebiger aktiver Applikationsprozeß, seien $Mon(p) := \{q_1, q_2, ..., q_k\}$ $(k > 1)$ die Überwachungsprozesse, die p überwachen , sei $q \in Mon(p)$ beliebig und sei $R :=< \mathcal{S}, H_{\mathbb{D}}, I, St >$ ein Run von Z, so daß:
(1.1) $H_{\mathbb{D}}\,(q_{2l}, p, t_{2i}) = \dot{+}$ für alle $q_{2l} \in Mon(p)$ und für alle $i \in \mathbb{N}$.
(1.2) $H_{\mathbb{D}}\,(q_{21}, p, t_{2i+1}) = \uparrow$ für alle $q_{2l} \in Mon(p)$ und für alle $i \in \mathbb{N}$.
(1.3) $H_{\mathbb{D}}\,(q_{2l+1}, p, t_{2i}) = \uparrow$ für alle $q_{2l+1} \in Mon(p)$ und für alle $i \in \mathbb{N}$.
(1.4) $H_{\mathbb{D}}\,(q_{21+1}, p, t_{2i+1}) = \dot{+}$ für alle $q_{2l+1} \in Mon(p)$ und für alle $i \in \mathbb{N}$.
(2.0) die Nachricht mit der Information $H_{\mathbb{D}}\,(q_j, p, t_{2i})$ wird zu geraden Zeitpunkten in $last\_status^q$ aufgenommen bzw. die Information $H_{\mathbb{D}}\,(q_j, p, t_{2i+1})$ wird zu ungeraden Zeitpunkten in $last\_status^q$ aufgenommen.

Sei t_1 der Zeitpunkt, zu dem die erste Nachricht mit der Information $H_{\mathbb{D}}\,(q, p, .) = \dot{+}$ den Überwachungsprozeß q erreicht. Gemäß Voraussetzung gilt für die Komponente (p, v) von $aggreg^q := \star\,(last\_status^q)$ für jeden Zeitpunkt $t \in T\ .\ (t > t_1)$ die Aussage $v = \dot{+}$, d.h. $Z\,(H_{\mathbb{D}})\,(q, p, t)$ $\in \{\dot{+}, \infty\}$ für alle $t \in T\ .\ (t \geq t_1)$. Somit wird der aktive Prozeß p von q permanent in $Z(\mathbb{D})$ verdächtigt, d.h. $Z(\mathbb{D})$ erfüllt nicht *kontinuierlich pq-starke Genauigkeit.* ∎

COROLLARY 2. *Eigenschaft* **P2** *wird im allgemeinen von* Z *nicht erfüllt.*

BEWEIS. Es ist zu zeigen, daß es einen $\mathbb{D} \in (W,.)$, einen $R :=< \mathcal{S}, H_{\mathbb{D}}, I, St >$ und einen aktiven Applikationsprozeß p gibt, so daß p niemals *permanent* von bezüglich p funktionellen Überwachungsprozessen in $H_{\mathbb{D}}$ im Run R verdächtigt wird. Unter diesen Voraussetzungen

wird p *permanent* von wenigstens einem bezüglich p funktionellen Überwachungsprozeß in $Z\,(H_{\mathbb{D}})$ verdächtigt, so daß $P2$ verletzt wird.

Die Annahme im Gegenbeispiel im Lemma (9) ist stärker als notwendig, und zwar: p wird niemals permanent von keinem bezüglich p funktionellen Überwachungsprozeß verdächtigt.

Formal:

$$
\begin{array}{l}
\exists \mathbb{D} \in (W, .)\ .\ \exists R :=< \mathcal{S}, H_{\mathbb{D}}, I, St >\ .\ \exists Z\,(H_{\mathbb{D}}) \in \mathbb{A}\,(H_{\mathbb{D}})\ . \\
(\forall p \in App\ .\ \forall q \in Func\,(\mathcal{S}, p)\ .\ \forall t \in T\ .\ \exists t' \geq t\ . \\
\qquad H_{\mathbb{D}}\,(q, p, t') \neq \infty\ \wedge\ \neg\ false\_susp(\mathcal{S}, H_{\mathbb{D}}, q, p, t')) \\
\wedge \\
(\exists p \in App\ .\ \exists q' \in Func\,(\mathcal{S}, p)\ .\ \exists t \in T\ .\ \forall t'' \geq t\ . \\
\qquad Z\,(H_{\mathbb{D}})\,(q', p, t'') = \infty\ \vee\ false\_susp(\mathcal{S}, Z\,(H_{\mathbb{D}})\,, q', p, t''))
\end{array}
$$

Den Beweis kann man analog zum Beweis von **P2** führen. ∎

4. Emulation von *kontinuierlich starker Genauigkeit*

Wir verwenden folgende Abkürzungen für wichtige Eigenschaften:

P3 :=**Emulation von kontinuierlich starker Genauigkeit mit Hilfe von kontinuierlich pq-starker Genauigkeit**

♦

P4 :=**Erhalt von starker Vollständigkeit**

♦

Um *kontinuierlich starke Genauigkeit* mit Hilfe von *kontinuierlich pq-starker Genauigkeit* zu emulieren, führen wir einen neuen Verknüpfungsoperator $\bullet$ ein, mit dem Ziel einen Prozeß p nur dann als abgestürzt zu kennzeichnen, falls alle Überwachungsprozesse p als abgestürzt eingestuft haben.

DEFINITION 32. *Sei m wie üblich die Anzahl der Überwachungsprozesse und sei ein formaler Verknüpfungsoperator $\bullet : (SV_P^\infty)^m \longrightarrow SV_P^\infty$ wie folgt*

$$(a_i)_{i\in 1,m} \longmapsto \begin{cases} a_1 \textit{ falls } \forall k \in \{1,2,...,m\} \ . \ a_k = a_1 \\ \uparrow \textit{ falls } \exists i \in \{1,2,...,m\} \ . \ a_i = \uparrow \\ \dot{+} \textit{ sonst} \end{cases}$$

definiert.■

NOTATION 5. *Sei Z' die Abwandlung von Z, indem der Verknüpfungsoperator $\star$ durch $\bullet$ ersetzt wird.*■

NOTATION 6. *Sei Z'' die Abwandlung von Z', indem zwischen den Zeilen 09 und 10 in der Implementierung von Z' (Algorithmus (1) unter Berücksichtigung der Notation (5)) folgendes* Pseudo-Code*:*

```
01        for h ∈ Mon\{u}{
02            if h ∈ D_q then
03                last_status_h^q ← ∞;
04        }/* falls h abgestürzt oder verdächtigt,
          dann ignoriere den Zustandsvektor von h */
```

eingefügt wird.■

Wir zeigen, daß *kontinuierlich starke Genauigkeit* mit Hilfe von *kontinuierlich pq-starker Genauigkeit* emuliert werden kann. Dabei bleibt *starke Vollständigkeit* erhalten. Wir halten fest:

LEMMA 10. *Z' sowie Z'' erfüllen Eigenschaft* **P3.** *Im allgemeinen wird Eigenschaft* **P4** *von Z' nicht erfüllt, hingegen erfüllt Z'' Eigenschaft* **P4**.

BEWEIS. Wir zeigen zuerst, daß Z' Eigenschaft **P3** erfüllt.
Sei im folgenden $\mathbb{D} \in (., \sim qW)$ und sei $R :=< \mathcal{S}, H_{\mathbb{D}}, I, St >$ ein beliebiger Run. Sei p ein beliebiger aktiver Applikationsprozeß.

Wir zeigen:

Wird p niemals *permanent* von wenigstens einem bezüglich p funktionellen Überwachungsprozeß in $H_{\mathbb{D}}$ im Run R verdächtigt, so wird p von keinem bezüglich p funktionellen Überwachungsprozeß *permanent* in $Z'(H_{\mathbb{D}})$ verdächtigt.

Formal:

$$\forall \mathbb{D} \in (., \sim qW) \; . \; \forall R :=< \mathcal{S}, H_{\mathbb{D}}, I, St > \; . \; \forall Z'(H_{\mathbb{D}}) \in \mathbb{A}(H_{\mathbb{D}}) \; .$$
$$\forall p \in App \; .$$
$$\left(\begin{array}{l} \exists q_0 \in Func(\mathcal{S}, p) \; . \; \forall t \in T \; . \; \exists t' \geq t \; . \\ H_{\mathbb{D}}(q_0, p, t') \neq \infty \; \wedge \; \neg \; false\_susp(\mathcal{S}, H_{\mathbb{D}}, q_0, p, t') \end{array} \right)$$

$$\Longrightarrow$$

$$\left(\begin{array}{l} \forall q \in Func(\mathcal{S}, p) \; . \; \forall t \in T \; . \; \exists t'' \geq t \; . \\ Z'(H_{\mathbb{D}})(q, p, t'') \neq \infty \; \wedge \; \neg \; false\_susp(\mathcal{S}, Z'(H_{\mathbb{D}}), q, p, t'') \end{array} \right)$$

Sei p ein beliebiger aktiver Applikationsprozeß, sei $t_0 \in T$ beliebig. Sei $q_0 \in Func(\mathcal{S}, p)$ ein beliebiger Überwachungsprozeß, welcher p nicht dauernd als abgestürzt einstuft. Sei $t_1 \in T \; . \; (t_1 \geq t_0)$, so daß $H_{\mathbb{D}}(q_0, p, t_1) =\uparrow$. Sei $t_2 \in T \; . \; (t_2 > t_1)$ der Zeitpunkt zu dem die obige Auswertung von q_0 verschickt wird.
Sei $q \in Func(\mathcal{S}, p)$ ein beliebiger Überwachungsprozeß, welcher p überwacht. Gemäß Aggregationsvorschrift Z' existiert ein Zeitpunkt $t_3 \in T \; . \; (t_3 > t_2)$ zu dem q die Information $H_{\mathbb{D}}(q_0, p, t_1) =\uparrow$ in $aggreg_q$ ablegt. Da $\forall a \in SV_P : \bullet(a, \uparrow) =\uparrow$ gilt und $\bullet$ kommutativ ist, gibt es einen Zeitpunkt $t_4 \in T \; . \; (t_4 > t_3)$, so daß der Überwachungsprozeß q den Prozeß p als aktiv in $Z'(H_{\mathbb{D}})$ einstuft.

Falls $\exists h \in Mon \backslash \{u\} \; . \; h \in \mathbb{D}_q$, d.h. falls es einen von u verschiedenen Überwachungsprozeß h gibt, welcher von q verdächtigt wird, dann wird in Z'' die Variable $last\_status_m^q$ auf ∞ zurückgesetzt. Dadurch wird eine Ausführung von Z' simuliert, wo keine Nachricht von h den Prozeß

q erreicht hat. Somit gilt der obige Beweis von Z' Wort für Wort auch für Z''.

Wir zeigen, daß Z' im allgemeinen Eigenschaft **P4** nicht erfüllt. Sei im folgenden $\mathbb{D} \in (S, .)$ beliebig. Wir zeigen, daß es einen Run $R :=< \mathcal{S}, H_{\mathbb{D}}, I, St >$, einen abgestürzten Applikationsprozeß p, welcher nicht mehr hochgefahren wird, d.h. $\exists t' \; . \; \forall t \geq t' \; . \; \mathcal{S}(t, p) = \dot{+}$ sowie einen funktionellen Überwachungsprozeß q gibt, so daß p niemals *permanent* von q in $Z'(H_{\mathbb{D}})$ im Run R verdächtigt wird, obwohl p *permanent* von allen bezüglich p funktionellen Überwachungsprozessen in $H_{\mathbb{D}}$ verdächtigt wird.

Formal:

$$\exists \mathbb{D} \in (S, .) \; . \; \exists R :=< \mathcal{S}, H_{\mathbb{D}}, I, St >\in Z' \; . \; \forall p \in App \; . \; \forall t \in T \; .$$
$$perm\_crash(\mathcal{S}, p, t)$$
$$\wedge(\forall q \in Func(\mathcal{S}, p) \; . \; event\_perm\_susp(\mathcal{S}, H_{\mathbb{D}}, q, p, t))$$
$$\wedge(\exists q \in Func(\mathcal{S}, p) \; . \; \neg event\_perm\_susp(\mathcal{S}, Z'(H_{\mathbb{D}}), q, p, t))$$

Sei p ein Applikationsprozeß, welcher zum Zeitpunkt t_0 abstürzt und nicht wieder hochgefahren wird und sei q_0 ein Überwachungsprozeß, welcher zum Zeitpunkt $t' \in T \; . \; (t' < t_0)$ die Abschätzung $\uparrow$ über den Zustand von p verschickt und nachher sofort abstürzt. Sei $q \in Func(\mathcal{S}, p)$ ein bezüglich p funktioneller Überwachungsprozeß. Da q alle Nachrichten, die an ihn gesendet wurden, erhält, existiert ein Zeitpunkt t'', nachdem q die letzte Nachricht von q_0 erhalten hat. Die üblichen Überlegungen zeigen, daß ein Zeitpunkt $t^{(3)} \in T \; . \; \left(t^{(3)} > t''\right)$ existiert, nachdem q den Prozeß p permanent als aktiv ($\uparrow$) einstuft. Widerspruch.

Wir zeigen nun, daß Z'' Eigenschaft **P4** erfüllt.

Sei im folgenden $\mathbb{D} \in (S, .)$ und sei $R :=< \mathcal{S}, H_{\mathbb{D}}, I, St >$ ein beliebiger Run. Sei p ein beliebiger abgestürzter Applikationsprozeß, welcher nicht mehr hochgefahren wird, d.h. $\exists t' \; . \; \forall t \geq t' \; . \; \mathcal{S}(t, p) = \dot{+}$.

Wir zeigen:

Wird p *schließlich* von allen bezüglich p funktionellen Überwachungsprozessen permanent als abgestürzt in $H_{\mathbb{D}}$ im Run R eingestuft, so wird p *schließlich* von allen bezüglich p funktionellen Überwachungsprozessen permanent als abgestürzt in $Z''(H_{\mathbb{D}})$ eingestuft.
Formal:

$$\begin{array}{l} \forall \mathbb{D} \in (S,.)\ .\ \forall R :=< \mathcal{S}, H_{\mathbb{D}}, I, St >\ .\ \forall Z''(H_{\mathbb{D}}) \in \mathbb{A}(H_{\mathbb{D}})\ . \\ \forall p \in App\ .\ \forall t \in T\ .\ perm\_crash(\mathcal{S}, p, t)\ \Longrightarrow \end{array}$$

$$\left(\begin{array}{l} (\forall q \in Func(\mathcal{S}, p)\ .\ event\_perm\_susp(\mathcal{S}, H_{\mathbb{D}}, q, p, t)) \\ \quad \Longrightarrow \\ (\forall q \in Func(\mathcal{S}, p)\ .\ event\_perm\_susp(\mathcal{S}, Z''(H_{\mathbb{D}}), q, p, t)) \end{array} \right)$$

Sei q_0 ein beliebiger Überwachungsprozeß, welcher funktionell bezüglich p ist. Da p abstürzt und nicht mehr hochgefahren wird, gibt es einen Zeitpunkt $t_1 \in T\ .\ (t_1 > t')$, nachdem q_0 von allen Überwachungsprozessen q nur Nachrichten der Form $status(q, p, t_q) = \dot{+}$ mit $t_q \geq t'$ erhält. Sei m ein Überwachungsprozeß, welcher vor dem Zeitpunkt t_1 abstürzt und nicht mehr hochgefahren wird. Da $\mathbb{D} \in (S,.)$ werden Einschätzungen von m beim Zusammenfügen in $aggreg^{q_0}$ nach einem Zeitpunkt $t_1' \in T\ .\ (t_1' > t_1)$ nicht mehr berücksichtigt (siehe Notation von Z'').

Hieraus folgt, daß es einen Zeitpunkt $t_2 \in T\ .\ (t_2 > t_1)$ gibt, so daß $Z''(H_{\mathbb{D}})(q_0, p, t) \in \{\dot{+}, \infty\}$ für alle $t \in T\ .\ (t \geq t_2)$. Da q funktionell bezüglich p ist, existiert zu jedem Zeitpunkt $t_3 \in T\ .\ (t_3 \geq t_2)$ ein Zeitpunkt $t_4 \in T\ .\ (t_4 \geq t_3)$, so daß q_0 eine Nachricht der Form $status(q_0, p, t_{q_0}) = \dot{+}$ erhält. Die üblichen Überlegungen zeigen, daß es einen Zeitpunkt $t_5 \in T\ .\ (t_5 > t_4)$ gibt, so daß $Z''(H_{\mathbb{D}})(q_0, p, t_5) = \dot{+}$. Da q_0 beliebig gewählt worden war, folgt die Behauptung. ∎

Wir können nun eines der zentralen Sätze dieses Kapitels formulieren.

PROPOSITION 1. *Sei $\mathbb{P}$ ein Problem, welcher unter $(S,.)$ lösbar ist. Dann ist $\mathbb{P}$ auch unter $(W,.)$ lösbar.*

BEWEIS. Die Aussage des Satzes folgt sofort aus Korrolar (1) sowie Eigenschaft $P1$. ∎

5. Emulation von *absoluter (schließlicher) Genauigkeit*

Wir führen den Verknüpfungsoperator $\circledast$ ein, welcher Mehrheitsentscheidungen berücksichtigt. So wird ein Prozeß p als abgestürzt eingestuft, falls eine Mehrheit der Überwachungsprozesse dies auch so sieht, ansonsten wird der Prozeß p als aktiv eingestuft. Sicherlich ist diese Vorgehensweise nur dann sinnvoll, wenn eine aktive Mehrheit oder eine schließlich aktive Mehrheit von Überwachungsprozessen auch existiert. Wir werden zeigen, daß *mehrheitliche Genauigkeit* ausreicht, um *starke Genauigkeit* zu emulieren.

Im zweiten Teil des Abschnittes stellen wir einen Algorithmus vor, wodurch der Zustand von Prozessen sich indirekt bestimmen läßt. Die Idee ist, daß ein Prozeß q den Zustand eines Prozesses p (indirekt) ermittelt, indem q alle nicht verdächtigten und somit erreichbaren Überwachungsprozesse abfragt. Findet q einen Überwachungsprozeß, welcher p nicht verdächtigt, so setzt q seine Abschätzung über den Zustand von p auf aktiv, ansonsten wird p als abgestürzt eingestuft. Wir werden sehen, daß in diesem Fall *absolut q-schwache Genauigkeit* in *absolut starke Genauigkeit* transformiert werden kann, falls keine Menge $M \subset Mon$ mit $M \neq \emptyset$ existiert, so daß M und $Mon \backslash M$ kommunikativ disjunkt sind. Analog kann man zeigen, daß *absolute Pfad-Genauigkeit* in *absolut starke Genauigkeit* transformiert werden kann, dabei wird *starke Vollständigkeit* erhalten.

NOTATION 7. *Sei m wie üblich die Anzahl der Überwachungsprozesse und sei ein formaler Verknüpfungsoperator $\circledast : (SV_P^\infty)^m \longrightarrow SV_P^\infty$ wie folgt:*

$$(a_i)_{i\in 1,m} \longmapsto \begin{cases} \dot{+} \text{ falls } \exists M \in maj\,\{1,2,...,m\} \;.\; \forall k \in M \;.\; a_k = \dot{+} \\ \uparrow \text{ sonst} \end{cases}$$

definiert.■

Wir verwenden folgende Abkürzungen:

P5 :=**Emulation von absolut starker Genauigkeit mit Hilfe von absolut mehrheitlicher Genauigkeit**

♦

P6 :=**Emulation von absolut p-starker Genauigkeit mit Hilfe von absolut p-mehrheitlicher Genauigkeit**

⧫

P7 :=**Emulation von schließlich starker Genauigkeit mit Hilfe von schließlich mehrheitlicher Genauigkeit**

⧫

P8 :=**Emulation von schließlich p-starker Genauigkeit mit Hilfe von schließlich p-mehrheitlicher Genauigkeit**

⧫

P9 :=**Emulation von starker Vollständigkeit mit Hilfe von mehrheitlicher Vollständigkeit**

⧫

P10 :=**Emulation von absolut starker Genauigkeit mit Hilfe von absolut q-schwacher Genauigkeit**

⧫

P11 :=**Emulation von schließlich starker Genauigkeit mit Hilfe von schließlich q-schwacher Genauigkeit**

⧫

P12 :=**Emulation von absolut starker Genauigkeit mit Hilfe von absoluter Pfad-Genauigkeit**

⧫

P13 :=**Emulation von schließlich starker Genauigkeit mit Hilfe von schließlicher Pfad-Genauigkeit**

⧫

NOTATION 8. *Sei $Z^{(3)}$ die Abwandlung von Z, indem der Verknüpfungsoperator $\star$ durch $\circledast$ ersetzt wird.*■

Wir zeigen, daß *absolut p-starke Genauigkeit* mit Hilfe von *absolut p-mehrheitlicher Genauigkeit* emuliert werden kann. Wir halten fest:

LEMMA 11. *Der Algorithmus $Z^{(3)}$ erfüllt Eigenschaft* **P6** *sowie Eigenschaft* **P9**.

BEWEIS. Wir zeigen zuerst, daß $Z^{(3)}$ Eigenschaft **P6** erfüllt.
Sei im folgenden $\mathbb{D} \in (., \Box pM)$, d.h. $\mathbb{D}$ sei ein Ausfalldetektor, welcher *absolut p-mehrheitliche Genauigkeit* erfüllt und sei $R :=< \mathcal{S}, H_{\mathbb{D}}, I, St >$ ein beliebiger Run.

Wir zeigen zuerst:

Wird jeder permanent aktive Prozeß p niemals von einer aktiven Mehrheit von Überwachungsprozessen in $H_{\mathbb{D}}$ im Run R verdächtigt, so wird p von keinem Überwachungsprozeß in $Z^{(3)}(H_{\mathbb{D}})$ verdächtigt.

Formal:

$$\forall \mathbb{D} \in (., \Box pM)\ .\ \forall R :=< \mathcal{S}, H_{\mathbb{D}}, I, St >\ .\ \forall Z^{(3)}(H_{\mathbb{D}}) \in \mathbb{A}(H_{\mathbb{D}})\ .$$
$$\forall p \in App \cap perm\_activ(\mathcal{S})\ .$$
$$\left(\begin{array}{l} \exists M \in act\_maj(Mon(p))\ .\ \forall t \in T\ . \\ \forall q \in M\ .\ \neg\ false\_susp(\mathcal{S}, H_{\mathbb{D}}, q, p, t) \end{array} \right)$$

$$\Longrightarrow$$

$$\left(\begin{array}{l} \forall q \in Mon(p)\ .\ \forall t \in T\ . \\ \neg\ false\_susp(\mathcal{S}, Z^{(3)}(H_{\mathbb{D}}), q, p, t) \end{array} \right)$$

Sei p ein beliebiger aktiver Applikationsprozeß und sei $M \in act\_maj(Mon(p))$ eine Menge von aktiven Überwachungsprozessen, so daß kein Element von M den Prozeß p je verdächtigt. Sei $q \in M$ beliebig. Dann gilt für alle $t \in T$ die Aussage $H_{\mathbb{D}}(q, p, t) = \uparrow$.

Sei $q' \in Func(\mathcal{S}, p)$ ein beliebiger Überwachungsprozeß, welcher p überwacht. Gemäß Algorithmus $Z^{(3)}$ existiert ein Zeitpunkt $t' \in T$ zu dem q' die Zustandsvektoren von allen Überwachungsprozessen $q \in M$ zum ersten Mal erhält. Die Information $H_{\mathbb{D}}(q, p, .) = \uparrow$ wird in $aggreg_{q'}$ abgelegt.

Gemäß Aggregationsvorschrift von $\circledast$ gibt es einen Zeitpunkt $t'' \in T$. $(t'' > t')$, so daß der Überwachungsprozeß q diese Zustandsvektoren auswertet und den Prozeß p als aktiv ($\uparrow$) in $Z^{(3)}(H_{\mathbb{D}})$ einstuft. Da q' von allen $q \in M$ nur die Einschätzung $\uparrow$ über den Zustand von p erhält, gilt $Z^{(3)}(H_{\mathbb{D}})(q', p, t) = \uparrow$ für alle $t \in T$. $(t > t'')$. Da vor dem Zeitpunkt t'' keine aktive Mehrheit von Überwachungsprozessen sich bilden kann, die p als abgestürzt einstuft, gilt insgesamt $Z^{(3)}(H_{\mathbb{D}})(q', p, t) = \uparrow$ für alle $\in T$.

Wir zeigen nun, daß $Z^{(3)}$ Eigenschaft **P9** erfüllt.

Sei im folgenden $\mathbb{D} \in (M, .)$ und sei $R :=< \mathcal{S}, H_{\mathbb{D}}, I, St >$ ein beliebiger Run. Sei p ein beliebiger abgestürzter Applikationsprozeß, welcher nicht mehr hochgefahren wird, d.h. $\exists t'$. $\forall t \geq t'$. $\mathcal{S}(t, p) = \dot{+}$.

Wir zeigen:

Wird p *schließlich* von einer aktiven Mehrheit von Überwachungsprozessen permanent als abgestürzt in $H_{\mathbb{D}}$ im Run R eingestuft, so wird p *schließlich* von allen bezüglich p funktionellen Überwachungsprozessen permanent als abgestürzt in $Z^{(3)}(H_{\mathbb{D}})$ eingestuft.
Formal:

$$\forall \mathbb{D} \in (M, .)\ .\ \forall R :=< \mathcal{S}, H_{\mathbb{D}}, I, St >\ .\ \forall Z^{(3)}(H_{\mathbb{D}}) \in \mathbb{A}(H_{\mathbb{D}})\ .$$
$$\forall p \in App\ .\ \forall t \in T\ .\ perm\_crash(\mathcal{S}, p, t)\ \Longrightarrow$$
$$\left(\begin{array}{l} (\exists M \in act\_maj(Mon(p))\ .\ \forall q \in M\ . \\ \qquad\qquad\qquad\qquad event\_perm\_susp(\mathcal{S}, H_{\mathbb{D}}, q, p, t)) \\ \\ \qquad \Longrightarrow \\ \\ (\forall q \in Func(\mathcal{S}, p)\ .\ event\_perm\_susp(\mathcal{S}, Z^{(3)}(H_{\mathbb{D}}), q, p, t)) \end{array} \right)$$

Sei q' ein beliebiger Überwachungsprozeß, welcher funktionell bezüglich p ist und sei $M \in act\_maj\,(Mon\,(p))$ eine Mehrheit von aktiven Überwachungsprozessen, welche schließlich p als abgestürzt einstuft. Somit existiert ein Zeitpunkt $t' \in T$ zu dem q' die Zustandsvektoren mit $H_{\mathbb{D}}\,(q,p,.) = \dot{+}$ von allen Überwachungsprozessen $q \in M$ zum ersten Mal erhält. Die Information $H_{\mathbb{D}}\,(q,p,.) = \dot{+}$ wird in $aggreg_{q'}$ abgelegt.

Gemäß Aggregationsvorschrift von $\circledast$ gibt es einen Zeitpunkt $t'' \in T$. $(t'' > t')$, so daß jeder Überwachungsprozeß $q \in M$ diese Zustandsvektoren auswertet und den Prozeß p als abgestürzt $(\dot{+})$ in $Z^{(3)}\,(H_{\mathbb{D}})$ einstuft. Da q' von allen $q \in M$ nur die Einschätzung $\dot{+}$ ab dem Zeitpunkt t' über den Zustand von p erhält, gilt $Z^{(3)}\,(H_{\mathbb{D}})\,(q',p,t) = \dot{+}$ für alle $t > t''$.

∎

Wenden wir die obige Emulation für alle aktiven Prozesse an, so erhalten wir:

COROLLARY 3. *Der Algorithmus $Z^{(3)}$ erfüllt Eigenschaft* **P5** *sowie Eigenschaft* **P9**.

Insgesamt haben wir gezeigt, daß wir einen fehlerfreien Ausfalldetektor emulieren können, falls uns gelingt die *Fehlerfreiheit* für eine aktive Mehrheit von Überwachungsprozessen zu garantieren. Wir formulieren die "*schließliche*" Version des obigen Lemmas. Der Beweis ist analog zu führen.

LEMMA 12. *Der Algorithmus $Z^{(3)}$ erfüllt Eigenschaft* **P8** *sowie Eigenschaft* **P9**.

COROLLARY 4. *Der Algorithmus $Z^{(3)}$ erfüllt Eigenschaft* **P7** *sowie Eigenschaft* **P9**.

Wir können nun den ersten Teil des Hauptergebnisses dieses Abschnitts formulieren:

PROPOSITION 2. *Sei $\mathbb{P}$ ein Problem. Ist $\mathbb{P}$ unter:*

a) $(S, \Box pS)$ lösbar. Dann ist $\mathbb{P}$ auch unter $(S, \Box pM)$ sowie $(M, \Box pM)$ lösbar.

b) $(S, \Box S)$ lösbar. Dann ist $\mathbb{P}$ auch unter $(S, \Box M)$ sowie $(M, \Box M)$ lösbar.

c) $(S, \Diamond pS)$ *lösbar. Dann ist* $\mathbb{P}$ *auch unter* $(S, \Diamond pM)$ *sowie* $(M, \Diamond pM)$ *lösbar.*

d) $(S,\Diamond S)$ *lösbar. Dann ist* $\mathbb{P}$ *auch unter* $(S,\Diamond M)$ *sowie* $(M,\Diamond M)$ *lösbar.*

BEWEIS. Die Aussage des Satzes folgt sofort aus Lemma (11), Lemma (12) sowie dessen Korrolare. ∎

Wir kommen nun zum zweiten Teil dieses Abschnittes. Sei $\mathbb{D}$ ein Ausfalldetektor. Wir werden im folgenden einen neuen Ausalldetektor $\mathbb{D}(\mathbf{SEND})$ emulieren.

DEFINITION 33 ($\mathbb{D}(\mathbf{SEND})$). *Sei* p *ein Applikationsprozeß, sei* q *ein Überwachungsprozeß und sei* $\mathbb{D}$ *ein Ausfalldetektor. Sei* $\mathbb{D}(\mathbf{SEND})$ *eine Emulation von* $\mathbb{D}$ *wie folgt:*

$$\begin{aligned} p \notin \mathbb{D}_q(\mathbf{SEND}) \equiv\ & (\exists q_1, q_2, ..., q_k \in Mon \cap perm\_activ(\mathcal{S})\ . \\ & q_i \neq q_j \text{ für } i \neq j\ .\ q_k = q\ .\ (p \notin \mathbb{D}_{q_1}) \wedge \\ & (q_1 \notin \mathbb{D}_{q_2}) \wedge (q_2 \notin \mathbb{D}_{q_3}) \wedge ... \wedge (q_{k-1} \notin \mathbb{D}_{q_k})) \end{aligned}$$

definiert.■

Somit wird p vom Prozeß q in $\mathbb{D}(\mathbf{SEND})$ genau dann nicht verdächtigt, wenn eine Folge $q_1, q_2, ..., q_k$ von verschiedenen aktiven Überwachungsprozessen existiert mit $q_k = q$, so daß für $i \in \{1, 2, ..., k\}$ der Prozeß q_{i+1} den Prozeß q_i nicht verdächtigt und $p \notin \mathbb{D}_{q_1}$.

Wir geben im folgenden eine Implementierung des Ausfalldetektors $\mathbb{D}(\mathbf{SEND})$. Wird p von q verdächtigt, dann schickt q eine Anfrage an alle erreichbaren (d.h. nicht verdächtigten) Prozesse q_c und wartet auf die Antwort. Erhält q_c eine Anfrage, so ermittelt q_c den Zustand von p. Wird p von q_c nicht verdächtigt, so schickt q_c die Abschätzung

aktiv an q, ansonsten wird die Abfrage rekursiv fortgesetzt. Die Variable REQ deutet darauf hin, daß es sich um eine Anfrage handelt. Analog zeigt RES an, daß es sich um eine Antwort auf eine Anfrage handelt. Der Prozeß q_s ist der Sender auf dessen Nachricht q_c wie unten angegeben reagiert. Um schon angesprochene Überwachungsprozesse auszuklammern, werden diese Prozesse in die Variable $List$ aufgenommen. Somit wird die Anfrage REQ an die Prozesse aus der Liste $List$ nicht mehr weitergeleitet. Erhält q keine aktiven Abschätzungen, so gilt $p \in \mathbb{D}_q$ (**SEND**).

Der Nachrichtenaustausch beim Überwachungsprozeß q_c sieht folgendermaßen aus:

```
01  when received (REQ, q_s, p, id_q, List) from q_s then{
02      if p ∉ D_q then
03          send (RES, q_s, p, id_q, ↑) to q_s;
04      else
05          for q_i ∈ {q_1, q_2, ..., q_k} \ List do{
06              send (REQ, q_s, p, id_q, List ∪ {p_c}) to q_i;
07              wait until [ received (RES, q_s, p, id_q, status_p)
                                 from q_i or q_i ∈ D_q;
08              if status_p = ↑ then{
09                  send (RES, q_c, p, id_q, ↑) to q_c;
10                  return;
11              }
12          }
13          send (RES, q_c, p, id_q, ∔) to q_c;
14  }
```

Der Prozeß q startet die Abfrage nach dem Zustandsvektor von p wie folgt:

```
01  send (REQ, q, p, id_q, List = {q}) to q;
02  wait until [ received (RES, q, p, id_q, status_p)
03        if status_p = ↑ then
04              p ∉ 𝔻_q (SEND) ;
05        else
06              p ∈ 𝔻_q (SEND) ;
```

Implementierung von $\mathbb{D}\,(\mathbf{SEND})$ (Protokolle der Überwachungsprozesse zur Ermittlung der Zustandsvektoren)

Wir zeigen, daß der obige Algorithmus (im folgenden als $Z^{(4)}$ bezeichnet) der Definition (33) entspricht.

LEMMA 13. *Sei p ein beliebiger Applikationsprozeß und sei q ein beliebiger Überwachungsprozeß. Führt q die Anweisung "send $(REQ, q, p, id_q, List = \{q\})$ to q;" aus, dann erhält q gemäß Algorithmus $Z^{(4)}$ den Wert $status_p = \uparrow$ genau dann, wenn $p \notin \mathbb{D}_q\,(\mathbf{SEND})$ bzw. erhält q gemäß Algorithmus $Z^{(4)}$ den Wert $status_p = \dotplus$ genau dann, wenn $p \in \mathbb{D}_q\,(\mathbf{SEND})$.*

BEWEIS. Wir nehmen zuerst an, daß q den Wert $status_p = \uparrow$ erhalten hat. Dann gibt es gemäß obigem Algorithmus eine Folge $q_1, q_2, ..., q_i$ von Überwachungsprozessen mit $q_i = q$, so daß $p \notin \mathbb{D}_{q_1}$ und $q_{k-1} \notin \mathbb{D}_{q_k}$ für alle $k \in \{1, 2, ..., k\}$. Somit gilt $p \notin \mathbb{D}_q\,(\mathbf{SEND})$. Hat q den Wert $status_p = \dotplus$ erhalten, so bedeutet dies, daß alle Überwachungsprozesse, die q abgefragt hat, den Prozeß p als abgestürzt eingestuft haben. Somit gibt es keine Folge $q_1, q_2, ..., q_k$ von Überwachungsprozessen mit $p \notin \mathbb{D}_{q_1}$ und $q_{k-1} \notin \mathbb{D}_{q_k}$ für alle $k \in \{1, 2, ..., k\}$, d.h. $p \in \mathbb{D}_q\,(\mathbf{SEND})$. Die Umkehrung folgt sofort. ∎

Wir zeigen im folgenden, daß *absolut q-schwache Genauigkeit* in *absolut starke Genauigkeit* unter Verwendung der Emulation $\mathbb{D}\,(\mathbf{SEND})$

überführt werden kann, falls Überwachungsprozesse indirekt über andere Überwachungsprozesse miteinander kommunizieren können. Dabei bleibt *starke Vollständigkeit* erhalten.

LEMMA 14. *Der Algorithmus* $Z^{(4)}$ *erfüllt Eigenschaft* **P10** *sowie Eigenschaft* **P4**, *falls keine Menge* $M \subset Mon$ *mit* $M \neq \emptyset$ *existiert, so daß* M *und* $Mon \backslash M$ *kommunikativ disjunkt sind.*

BEWEIS. Wir zeigen zuerst, daß $Z^{(4)}$ Eigenschaft **P10** erfüllt.
Sei im folgenden $\mathbb{D} \in (., \Box qW)$, d.h. $\mathbb{D}$ sei ein Ausfalldetektor, welcher *absolut q-schwache Genauigkeit* erfüllt und sei $R :=< \mathcal{S}, H_{\mathbb{D}}, I, St >$ ein beliebiger Run.

Wir zeigen:

Wird jeder permanent aktive Prozeß p niemals von wenigstens einem Überwachungsprozessen in $H_{\mathbb{D}}$ im Run R verdächtigt, so wird p von keinem Überwachungsprozeß in $Z^{(4)}(H_{\mathbb{D}})$ verdächtigt.

Formal:

$$\forall \mathbb{D} \in (., \Box qW) \; . \; \forall R :=< \mathcal{S}, H_{\mathbb{D}}, I, St > \; . \; \forall Z^{(4)}(H_{\mathbb{D}}) \in \mathbb{A}(H_{\mathbb{D}}) \; .$$
$$\forall p \in App \cap perm\_activ(\mathcal{S}) \; .$$
$$(\exists q \in Func(\mathcal{S}, p) \; . \; \forall t \in T \; . \; \neg false\_susp(\mathcal{S}, H_{\mathbb{D}}, q, p, t))$$

$$\Longrightarrow$$

$$\left(\forall q \in Mon(p) \; . \; \forall t \in T \; . \; \neg false\_susp(\mathcal{S}, Z^{(4)}(H_{\mathbb{D}}), q, p, t)\right)$$

Sei p ein beliebiger aktiver Applikationsprozeß und sei $q' \in Func(\mathcal{S}, p)$ ein aktiver Überwachungsprozeß, so daß q' den Prozeß p nie verdächtigt. Sei $q \in Func(\mathcal{S}, p)$ beliebig. Wir werden zeigen, daß q den Prozeß p immer als aktiv einstuft.

Da gemäß Annahme $\{q\}$ und $Func(\mathcal{S}, p) \setminus \{q\}$ nicht kommunikativ disjunkt sind, gibt es einen permanent aktiven Überwachungsprozeß q'', so daß q den Prozeß q'' nie verdächtigt. Da $\{q, q''\}$ und $Func(\mathcal{S}, p) \setminus$

$\{q, q''\}$ nicht kommunikativ disjunt sind, gibt es zu q'' ein Überwachungsprozeß $q^{(3)} \notin \{q, q''\}$, so daß q'' den Prozeß $q^{(3)}$ nicht verdächtigt. Durch vollständige Induktion kann man zeigen, daß eine Folge $q_1, q_2, ..., q_k$ von verschiedenen Überwachungsprozessen existiert, so daß $p \notin \mathbb{D}_{q_1}$und $(q_j \in \mathbb{D}_{q_{j+1}})$ für alle $j \in \{1, 2, ..., k-1\}$. Gemäß Definition (33) bzw. Algorithmus $Z^{(4)}$ gilt $p \notin \mathbb{D}_q$ (**SEND**), d.h. es gilt $Z^{(4)}(H_{\mathbb{D}})(q, p, t) = \uparrow$ für alle $t \in T$.

Wir zeigen nun, daß $Z^{(4)}$ Eigenschaft **P4** erfüllt.
Sei im folgenden $\mathbb{D} \in (S, .)$ und sei $R :=< \mathcal{S}, H_{\mathbb{D}}, I, St >$ ein beliebiger Run. Sei p ein beliebiger abgestürzter Applikationsprozeß, welcher nicht mehr hochgefahren wird, d.h. $\exists t' \,.\, \forall t \geq t' \,.\, \mathcal{S}(t, p) = \dot{+}$.

Wir zeigen:

Wird p *schließlich* von allen bezüglich p funktionellen Überwachungsprozessen permanent als abgestürzt in $H_{\mathbb{D}}$ im Run R eingestuft, so wird p *schließlich* von allen bezüglich p funktionellen Überwachungsprozessen permanent als abgestürzt in $Z^{(4)}(H_{\mathbb{D}})$ eingestuft.
Formal:

$$\forall \mathbb{D} \in (S, .) \,.\, \forall R :=< \mathcal{S}, H_{\mathbb{D}}, I, St > \,.\, \forall Z^{(4)}(H_{\mathbb{D}}) \in \mathbb{A}(H_{\mathbb{D}}) \,.$$
$$\forall p \in App \,.\, \forall t \in T \,.\, perm\_crash(\mathcal{S}, p, t) \implies$$

$$\left(\begin{array}{l} (\forall q \in Func(\mathcal{S}, p) \,.\, event\_perm\_susp(\mathcal{S}, H_{\mathbb{D}}, q, p, t)) \\ \quad \implies \\ (\forall q \in Func(\mathcal{S}, p) \,.\, event\_perm\_susp(\mathcal{S}, Z''(H_{\mathbb{D}}), q, p, t)) \end{array} \right)$$

Sei q ein beliebiger Überwachungsprozeß, welcher funktionell bezüglich p ist. Da p abstürzt und nicht mehr hochgefahren wird, gibt es einen Zeitpunkt $t_1 \in T \,.\, (t_1 > t')$, nachdem alle Überwachungsprozesse den Prozeß p als abgestürzt einstufen. Gemäß Definition (33) bzw. Algorithmus $Z^{(4)}$ gilt $Z^{(4)}(H_{\mathbb{D}})(q, p, t) = \dot{+}$ für alle $t \geq t_1$. Da q beliebig gewählt worden war, folgt die Behauptung.

∎

COROLLARY 5. *Der Algorithmus $Z^{(4)}$ erfüllt Eigenschaft* **P11** *sowie Eigenschaft* **P4**, *falls keine Menge $M \subset Mon$ mit $M \neq \emptyset$ existiert, so daß M und $Mon \backslash M$ kommunikativ disjunkt sind.*

LEMMA 15. *Der Algorithmus $Z^{(4)}$ erfüllt Eigenschaft* **P12** *sowie Eigenschaft* **P4**.

BEWEIS. Der Beweis ist analog zum Beweis des Lemmas (14) zu führen. ∎

COROLLARY 6. *Der Algorithmus $Z^{(4)}$ erfüllt Eigenschaft* **P13** *sowie Eigenschaft* **P4**.

Wir können nun den zweiten Teil des Hauptergebnisses dieses Abschnitts formulieren:

PROPOSITION 3. *Sei $\mathbb{P}$ ein Problem. Ist $\mathbb{P}$ unter:*

a) $(S, \Box S)$ lösbar. Dann ist $\mathbb{P}$ auch unter $(S, \Box qW)$ lösbar, falls keine Menge $M \subset Mon$ mit $M \neq \emptyset$ existiert, so daß M und $Mon \backslash M$ kommunikativ disjunkt sind.

b) $(S, \Box S)$ lösbar. Dann ist $\mathbb{P}$ auch unter $(S, \Box P)$ lösbar.

c) $(S, \Diamond S)$ lösbar. Dann ist $\mathbb{P}$ auch unter $(S, \Diamond qW)$ lösbar, falls keine Menge $M \subset Mon$ mit $M \neq \emptyset$ existiert, so daß M und $Mon \backslash M$ kommunikativ disjunkt sind.

d) $(S, \Diamond S)$ lösbar. Dann ist $\mathbb{P}$ auch unter $(S, \Diamond P)$ lösbar.

KAPITEL 7

Ausgewählte *Agreement*-Probleme

1. Einleitung

In diesem Kapitel untersuchen wir, inwieweit und unter welchen Bedingungen *Agreement*-Probleme wie *Consensus*, *Atomic Broadcast* bzw. das *Non-Blocking Atomic Commitment*-Problem (*NB-AC*-Problem) in unserem Modell lösbar sind. Wir zeigen, daß *Consensus* mit Hilfe von Ausfalldetektoren, die *starke Vollständigkeit* erfüllen, lösbar ist. Verdächtigte Prozesse werden gegebenenfalls heruntergefahren. Aus der Äquivalenz von *schwacher* und *starker Vollständigkeit* können wir folgern, daß *Consensus* auch mit Hilfe von Ausfalldetektoren lösbar ist, die *schwache Vollständigkeit* erfüllen. Genauigkeitseigenschaften sind nicht notwendig, um *Consensus* zu lösen, wohl aber um nichttriviale Ergebnisse mit den Ausfalldetektoren zu erzielen.

Das Kapitel schließt mit einer Untersuchung des *NB-AC*-Problems. Das *NB-AC*-Problem kann im Modell von Chandra und Toueg nicht gelöst werden. Der Grund dafür ist die *Non-Triviality*-Bedingung, die präzises Wissen über Ausfälle verlangt. Dieses Wissen kann nicht von den unzuverlässigen Ausfalldetektoren geliefert werden. Wir zeigen, daß in unserem Modell, in welchem verdächtigte Prozesse gegebenenfalls heruntergefahren werden können, das *Non-Blocking Atomic Commitment*-Problem auf *Consensus* reduzierbar ist, d.h. wann immer *Consensus* lösbar ist, ist auch das *NB-AC*-Problem lösbar.

2. Das *Consensus*- bzw. das *Atomic Broadcast*-Problem

2.1. Beschreibung des *Consensus*. Sei $\mathbb{K} \subseteq App$ eine Teilmenge der Applikationsprozesse. Jeder Prozeß $p \in \mathbb{K}$ startet den *Consensus*-Algorithmus mit einem Anfangswert V_p. Wir sagen: "der Prozeß $p \in \mathbb{K}$ nimmt am Consensus-Algorithmus A teil", falls der Prozeß p den *Consensus*-Algorithmus A startet. In Kurzform werden wir auch sagen: "der Prozeß p nimmt am Consensus teil".

Im *Consensus* haben alle Prozesse, die am Verfahren teilnehmen, einen Anfangswert und müssen eine einheitliche und unwiederrufliche

Entscheidung über einen gemeinsamen Endwert treffen, welcher in Zusammenhang mit den Anfangswerten steht, siehe [**30**].

Wir definieren das *Consensus*-Problem anhand von zwei Prozeduren $propose(V)$ und $decide(V)$. Führt der Prozeß p die Prozedur $propose\,(V)$ aus, dann sagen wir: "der Prozeß p *schlägt* V *vor*", führt der Prozeß p die Prozedur $decide\,(V)$ aus, dann sagen wir: "der Prozeß p *entscheidet* V". Um die Bezeichnung einheitlich und überschaubar zu halten, setzten wir: $propose_p\,(V)$, falls der Prozeß p die Prozedur $propose\,(V)$ ausführt. Das gleiche gilt für $decide_p\,(V)$. Führt der Prozeß p die Prozedur $propose\,(V)$ zum Zeitpunkt t aus, dann setzten wir: $propose_p^t\,(V)$. Das gleiche gilt für $decide_p^t\,(V)$. *Consensus* kann wie folgt spezifiziert werden:

DEFINITION 34 (**Consensus**). *Sei* $\mathbb{K} \subseteq App$ *die Menge der Prozesse, die am* Consensus *teilnehmen, sei* V_p *der Anfangswert des Prozesses* $p \in \mathbb{K}$, *sei* $\mathbb{V}$ *die Menge der Anfangswerte und sei* $\mathcal{S} \in \mathbb{S}$ *ein Systemablauf. Für die Menge* $\mathbb{K}$ *gilt:*

Termination*: Jeder permanent aktive Prozeß entscheidet schließlich, d.h.* $\forall p \in perm\_activ\,(\mathcal{S}) \cap \mathbb{K}\ .\ \exists V \in \mathbb{V}\ .\ decide_p\,(V)$.

Uniform-Integrity*: Jeder Prozeß entscheidet höchstens einmal, d.h.* $\forall p \in \mathbb{K}\ .\ \neg(\exists V, W \in \mathbb{V}\ .\ \exists t, t' \in T\ .\ t \neq t'\ .\ decide_p^t\,(V)\ \wedge decide_p^{t'}\,(W))$.

Agreement*: Alle permanent aktiven Prozesse, die entscheiden, entscheiden gleiche Werte, d.h.* $\forall p_i, p_j \in perm\_activ\,(\mathcal{S}) \cap \mathbb{K}\ .\ [\ \forall V, W \in \mathbb{V}\ .\ decide_{p_i}\,(V)\ \wedge decide_{p_j}\,(W) \Longrightarrow V = W\ ]$.

Uniform-Validity*: Entscheidet ein Prozeß den Wert* V, *dann ist* V *der Anfangswert eines Prozesses, welcher am* Consensus *teilnimmt, d.h.* $\exists p_i \in \mathbb{K}\ .\ \exists V \in \mathbb{V}\ .\ decide_{p_i}\,(V) \Longrightarrow \exists p_j \in \mathbb{K}\ .\ V = V_{p_j}$.■

Bezieht sich die Eigenschaft *Agreement* auf alle Prozesse, die entscheiden, so erhalten wir:

DEFINITION 35 (**Uniform Consensus**). *Wird zusätzlich die Eigenschaft:*

Uniform-Agreement*: Alle Prozesse, die entscheiden, entscheiden gleiche Werte, d.h.* $\forall p_i, p_j \in \mathbb{K}\ .\ [\ \exists V, W \in \mathbb{V}\ .\ decide_{p_i}\,(V)\ .\ decide_{p_j}\,(W) \Longrightarrow V = W\ ]$.

erfüllt, dann sprechen wir vom Uniform Consensus.■

Es ist bekannt, daß Consensus in asynchronen Systemen, in denen auch Absturzfehler zugelassen sind, deterministisch nicht lösbar ist, siehe [**31**] sowie [**22**].

Sei A ein Algorithmus, welcher das *Consensus*-Problem löst. Definitionsgemäß erfüllt A das *Consensus*-Problem**,** falls alle Runs von A die Bedingungen des *Consensus* (siehe Definition (34)) erfüllen. Im Gegensatz zum Modell von Chandra und Toueg können in unserem Modell verdächtige Prozesse heruntergefahren werden, d.h. der Algorithmus A kann einen Prozeß p, welcher am *Consensus* teilnimmt und fälschlicherweise verdächtigt wurde und somit nicht mit den anderen Prozessen kommunizieren kann, herunterfahren. Um triviale Lösungen des *Consensus*-Problem zu vermeiden, wo alle Prozesse heruntergefahren werden, sagen wir: "der Algorithmus A erlaubt eine nicht-triviale Lösung des Consensus-Problems", falls ein Run R von A existiert (siehe Definition (24)), so daß wenigstens ein Prozeß nach dem letzten Schritt von R aktiv ist.

Wir definieren jetzt *Reliable Broadcast,* ein Kommunikationsverfahren, welches wir in den nächsten Algorithmen verwenden werden. Sei p ein Prozeß, welcher eine Nachricht m an eine Menge von Prozessen $p_1, p_2, ..., p_k$ verschickt. *Reliable Broadcast* stellt, vereinfacht dargestellt sicher, daß schließlich entweder alle aktiven Empfänger die Nachricht m erhalten und auswerten können oder keiner von ihnen. Würde der Prozeß p eine Nachricht m sequenziell an alle potentiellen Empfänger verschicken und würde p während dieser Phase abstürzen, so wäre nicht sichergestellt, daß die Nachricht m an alle Empfänger verschickt wurde. Um dies zu verhindern, müssen wir ein komplexeres Protokoll berücksichtigen.

Reliable Broadcast kann formal für eine beliebige Nachricht m mittels zweier Operationen, *R-broadcast* (m) und *R-deliver* (m) definiert werden. *Reliable Broadcast* erfüllt folgende Eigenschaften, siehe [**35**] sowie [**17**, Seite 238].

DEFINITION 36 (**Reliable Broadcast**). *Für jede beliebige Nachricht m gilt:*

Validity: *Führt ein permanent aktiver Prozeß p die Operation R-broadcast (m) aus, dann führt er schließlich die Operation R-deliver (m) aus.*

Agreement: *Führt ein Prozeß die Operation R-deliver (m) aus, dann führen schließlich alle permanent aktiven Prozesse die Operation R-deliver (m) aus.*

Uniform-Integrity: *Jeder Prozeß führt die Operation R-deliver (m) höchstens einmal aus und nur dann, wenn früher ein Prozeß R-broadcast (m) ausgeführt hat.*■

Anschaulich betrachtet verschickt jeder Prozeß p im Rahmen von *Reliable Broadcast*, nachdem er die Nachricht m für das erste Mal empfangen hat, die Nachricht m an alle Prozesse und führt dann *R-deliver* (m) aus. Wir geben hier eine mögliche Implementierung für *Reliable Broadcast* wieder, siehe [**17**].

R-broadcast (m) für den Prozeß p:

```
01      for i := 1 to k do {send m to p_i};
```

Der Empfänger q einer Nachricht m reagiert wie folgt:

```
01      when receive m for the first time{
02          if sender(m) ≠ q then
                  for i := 1 to k do {send m to p_i};
03          R-deliver(m);
04      }
```

Wir zeigen im folgenden, daß der obige Algorithmus die Eigenschaften *Validity*, *Agreement* und *Uniform-Integrity* von *Reliable Broadcast* erfüllt.

BEWEIS.

Validity: Sei p ein permanent aktiver Prozeß, welcher die Operation $R-broadcast\,(m)$ ausführt. Gemäß Modellannahme über zuverlässige Kommunikation (siehe Definition (2) und anschließende Bemerkung) gehen Nachrichten nicht verloren. Somit erhält p schließlich die Nachricht m und führt die Operation $R-deliver\,(m)$ aus.

Agreement: Sei p ein permanent aktiver Prozeß und sei p_i ein Prozeß, welcher die Operation $R-deliver\,(m)$ ausgeführt hat. Dann hat p_i vorher die Nachricht m an alle Prozesse also auch an p verschickt. Folglich führt p schließlich $R-deliver\,(m)$ aus.

Uniform-Integrity: Wir nehmen an, es gäbe einen Prozeß p, welcher die Operation $R-deliver\,(m)$ ausgeführt hat. Gemäß Eigenschaft *No Creation* von *Fair Lossy* Kanälen (siehe Definition (1)) können keine korrupten Nachrichten entstehen, d.h. es gibt einen Prozeß p_i mit $i \in \{1,2,...,k\}$, so daß p_i die Operation $R-broadcast\,(m)$ ausgeführt hat. Da $R-deliver\,(m)$ nur beim ersten Empfang von m ausgeführt wird, wird $R-deliver\,(m)$ höchstens einmal ausgeführt.
∎

2.2. Algorithmus zum Lösen des *Consensus*. Der folgende Algorithmus (FS-*Cons*) löst das *Consensus*-Problem mit jedem Ausfalldetektor, welcher *starke Vollständigkeit* erfüllt. Es werden keine Genauigkeitseigenschaften vorausgesetzt. Verdächtigte Prozesse werden gegebenenfalls heruntergefahren. Die Prozesse, die vom *Consensus*-Algorithmus heruntergefahren wurden, werden als *nicht vertrauenswürdig* eingestuft und werden nicht mehr hochgefahren. Abgestürzte Prozesse werden im Rahmen des *Recovery*-Verfahrens hochgefahren und setzten ihre Tätigkeit von der Stelle fort, wo sie abgestürzt sind. Wie erwähnt, haben Prozesse in unserem Modell Zugang zu einem festen Speicherbereich, wohin sie ihre Zustandsinformationen während eines Absturzes retten können.

Unser Algorithmus zum Lösen des *Consensus* verwendet das Paradigma *rotierender Koordinator* siehe [**54**], [**19**], [**26**] sowie [**12**] und führt mehrere, aber eine endliche Anzahl von asynchronen Runden aus. Dasselbe Paradigma wurde auch von Chandra und Toueg verwendet, um das *Consensus*-Problem für Ausfalldetektoren, die *starke Vollständigkeit* und *schließlich p-starke Genauigkeit* erfüllen, zu lösen. Wie schon im Laufe der Arbeit erwähnt, ist uns keine Methode bekannt,

um *schließlich p-starke Genauigkeit* in praktischen Systemen implementieren zu können.

Wir verwenden Techniken, die auch von Chandra und Toueg eingesetzten wurden, wie z.B. die Überprüfung des Zeitstempels, die Rolle des Koordinators als zentraler Verteiler von Nachrichten sowie das Verschicken von *nack*, falls der Koordinator verdächtigt wird.

Wie üblich bezeichnen wir mit "n" die Anzahl der Prozesse, die am *Consensus* teilnehmen. Chandra und Toueg setzen voraus, daß mindestens $\left\lceil \frac{(n+1)}{2} \right\rceil$ Prozesse, die am *Consensus* teilnehmen, nicht abstürzen. Unser Algorithmus setzt keine Einschränkung bezüglich der Anzahl der aktiven Prozesse voraus. Falls aber die Mehrheit der Prozesse abstürzt, liefert unser Algorithmus nur eine triviale Lösung, indem er die restlichen Prozesse herunterfährt.

Genauigkeitseingeschaften werden von unserem *Consensus*-Algorithmus nicht vorausgesetzt. Ob eine nichttriviale Lösung des Consensus-Problems erzielt wird, hängt von der Güte (Genauigkeitseigenschaften) der Prozesse und der Zahl der abgestürzten Prozesse ab. In unklaren Situationen liefert unser Algorithmus eine triviale Lösung, da verdächtigte Prozesse im Gegensatz zum Algorithmus von Chandra und Toueg unter Umständen heruntergefahren werden.

2.2.1. *Funktionsweise des Algorithmus.* Wir setzen voraus, daß die Prozesse, die am *Consensus* teilnehmen, eine eindeutige Indentifikationsnummer (Id) besitzen mit $Id \in \{1, 2, ..., n\}$. Jeder Prozeß p, welcher am *Consensus* teilnimmt, startet den *Consensus*-Algorithmus mit dem Anfangswert $value_p$. Um die Notation in der folgenden Implementierung überschaubar zu halten, werden die internen Variablen, die dem Prozeß p zugeordnet werden können, durch "p" indiziert. So enthält z.B. die Variable $estimate_p$ den Entscheidungswert von p und entsprechend $estimate_q$ den Entscheidungswert des Prozesses q.

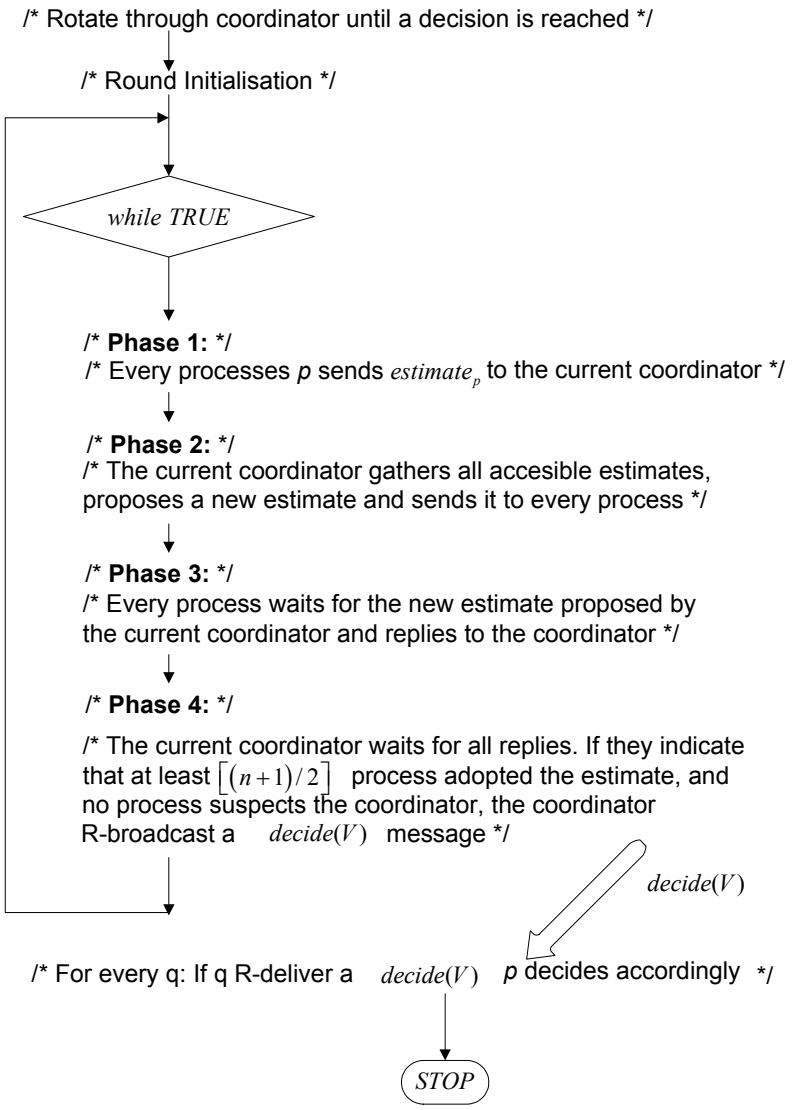
Wir beschreiben jetzt die Funktionsweise des Algorithmus.

Wie erwähnt, wird der Algorithmus mit einem externen Wert $value_p$ gestartet. Die Variable $estimate_p$ enthält jeweils den Entscheidungswert, welcher vom Prozeß p vermuteten wurde. Die Variable r_p dient dazu, um einen logischen Rundenzähler zu implementieren. Die Variable ts_p ist ein Zeitstempel und zeigt an, in welcher Runde der Prozeß p seinen Vorschlag $estimate_p$ das letzte Mal aktualisiert hat. In der Initialisierungsphase wird der Wert von $value_p$ der Variable $estimate_p$ zugewiesen.

```
/* Data Structure */

r_p : integer; /* rounds */
ts_p, t : integer; /* timestamp */
c_{r_p} : integer; /* current coordinator */
estimate_p : string; /* current estimate of p for the proposed value */
msgs_p : array of string;
/* msgs_p[r] is the current set of messages received by p as
            coordinator in the round r */

/* Initialisation */

estimate_p ← value_p;
r_p ← 0;
ts_p ← 0;
```

Datenstruktur und Initialisierung

Jeder Prozeß p durchläuft, solange er nicht entschieden hat, eine Folge von Runden. Jede Runde umfaßt vier Phasen. Die zweite und die vierte Phase werden nur vom aktuellen Koordinator durchlaufen, die erste und die dritte Phase wird von allen Prozessen betreten (auch vom aktuellen Koordinator).



Die vier Phasen des Consensus Algorithmus

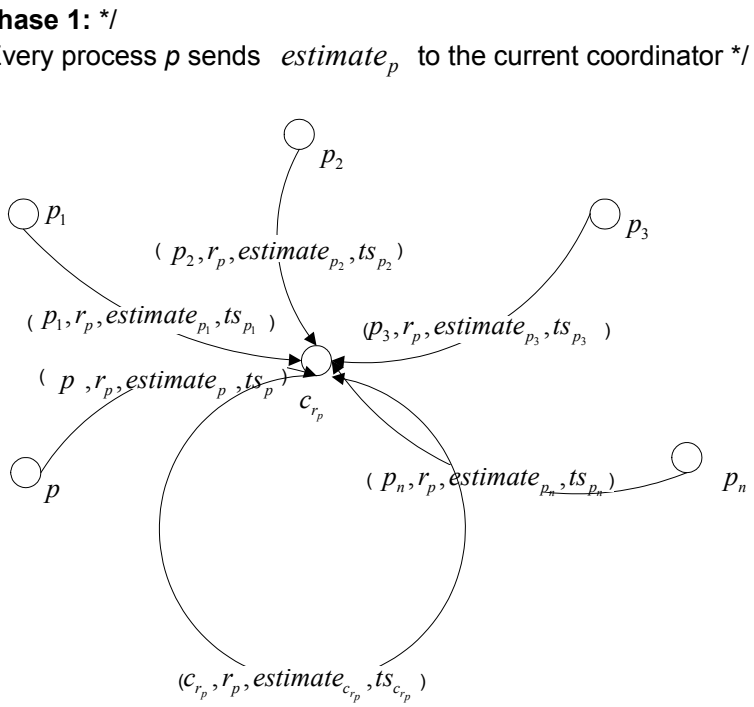
Am Anfang jeder neuen Runde von p wird r_p inkrementiert und der aktuelle Koordinator c_{r_p} für diese Runde festgelegt.

```
/* Round Intialisation */
r_p ← r_p + 1;
c_{r_p} ← (r_p mod n) + 1;   /* choose current coordinator */
```

Initialisierung der Runden

In der **ersten Phase** schicken alle Prozesse der aktuellen Runde r_p ihre Vorschläge $estimate_p$ zum aktuellen Koordinator c_{r_p} mittels der

Anweisung "$send\,(p, r_p, estimate_p, ts_p)\ to\ c_{r_p}$;".



Phase 1. Datenfluß

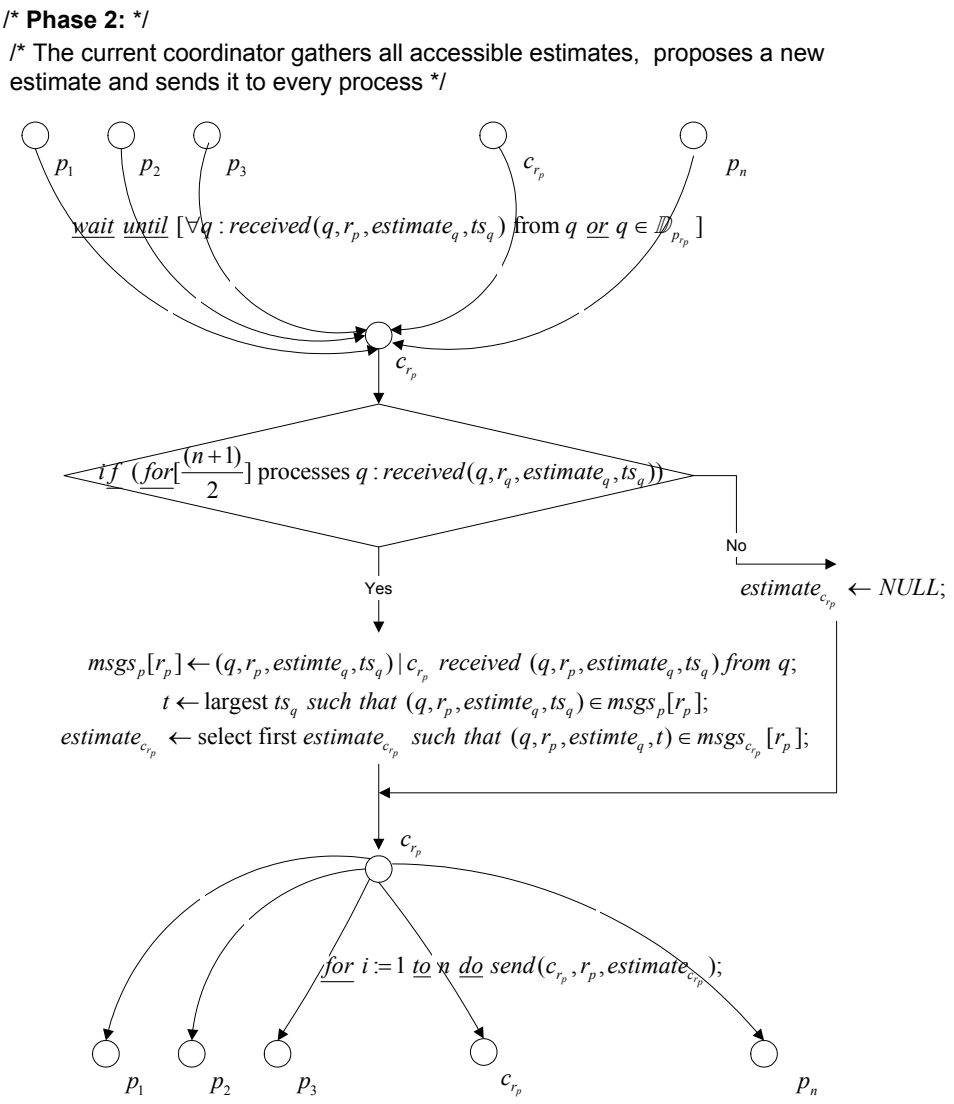
Der Koordinator c_{r_p} wartet in der **Phase 2** der Runde r_p auf das Eintreffen der Vorschläge aller Teilnehmer einschließlich auf seinen eigenen Vorschlag.

Verdächtigt der Koordinator einen Prozeß q, d.h. $q \in \mathbb{D}_{c_{r_p}}$, dann wird auf das Eintreffen des Vorschlages von q nicht mehr gewartet. Nachrichten, die außer der Reihe kommen, d.h. Nachrichten, die nicht in der aktuellen Runde r_p verschickt wurden, werden nicht weiter ausgewertet.

Falls es dem Koordinator c_{r_p} nicht gelingt, die gewünschte Anzahl von Nachrichten ($\left\lceil \frac{(n+1)}{2} \right\rceil$) zu sammeln, wird $estimate_{c_{r_p}}$ auf $NULL$ gesetzt, ansonsten wird der Vorschlag von c_{r_p} aktualisiert, indem $estimate_{c_{r_p}}$ mit einem Vorschlag[1] der Teilnehmer aus der aktuellen Runde ersetzt wird.

[1] dem ersten Vorschlag der Teilnehmer, welcher den größten Zeitstempel hat

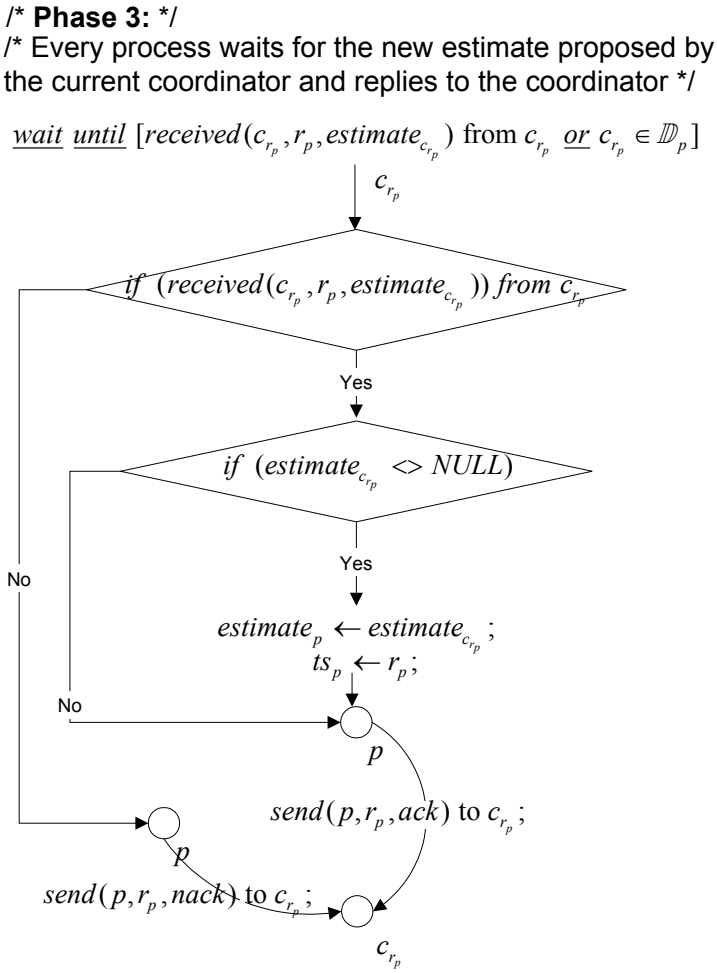
Anschließend wird der Vorschlag des Koordinators c_{r_p} an alle Beteiligten mittels der Anweisung "$send\left(c_{r_p}, r_p, estimate_{c_{r_p}}\right)$ *to all*;" verschickt.



Phase 2. Datenfluß

In der **Phase 3** der Runde r_p warten alle Prozesse auf das Eintreffen des Vorschlages vom Koordinator c_{r_p}, es sei denn, der Koordinator wird verdächtigt.

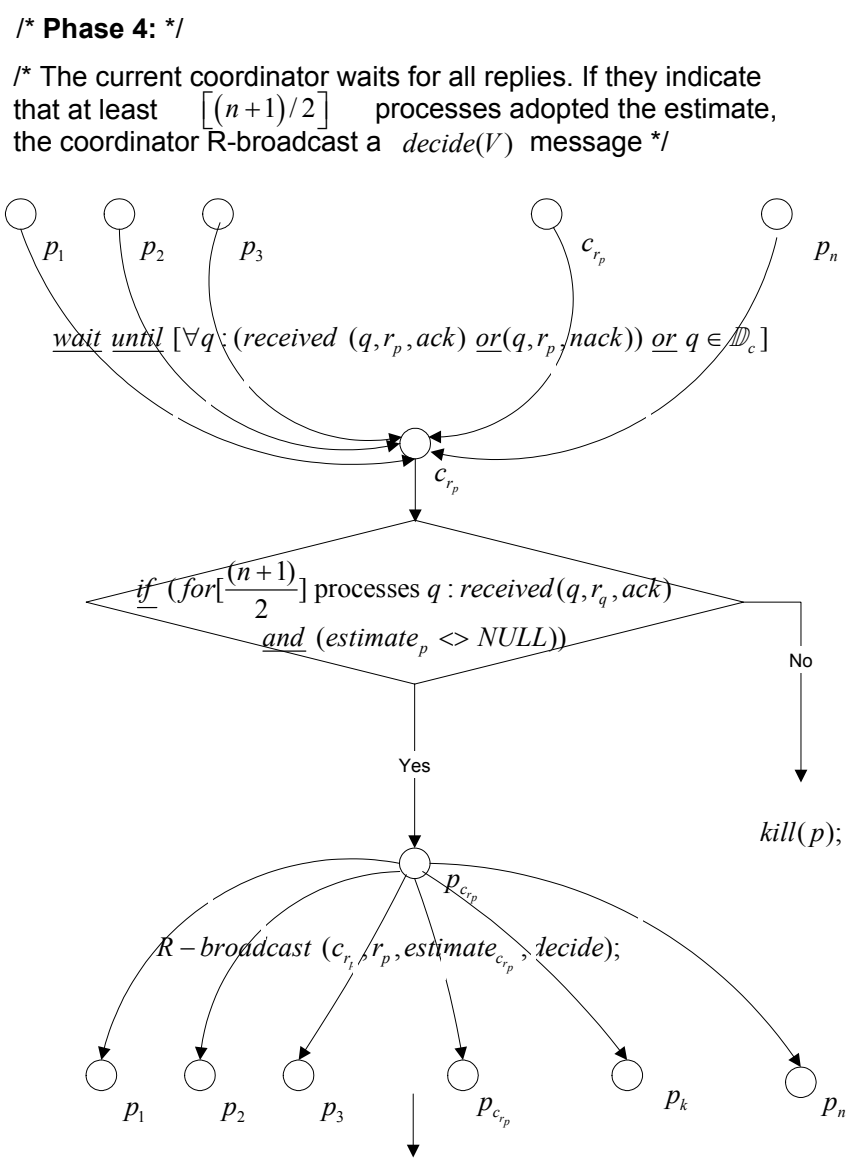
Hat der Prozeß p den Vorschlag des Koordinators aus der Runde r_p erhalten, so prüft der Prozeß p nach, ob die Vorschläge in der Phase 2 vom Koordinator ausgewertet wurden bzw. vom Koordinator berücksichtigt wurden, indem p die Variable $estimate_{c_{r_p}}$ auswertet. Ist dies der Fall, d.h. $estimate_{c_{r_p}} <> NULL$, so macht p den Vorschlag des Koordinators zu eigen, setzt den Zeitstempel ts_p auf r_p und bestätigt den Erhalt des Vorschlages $estimate_{c_{r_p}}$. Verdächtigt der Prozeß p den Koordinator, so schickt er an ihn eine Nachricht der Form $(p, r_p, nack)$ zurück.



Phase 3. Datenfluß

Der Koordinator wartet in der **Phase 4** auf die Bestätigungen aller Teilnehmer, es sei denn, der entsprechende Teilnehmer wird verdächtigt.

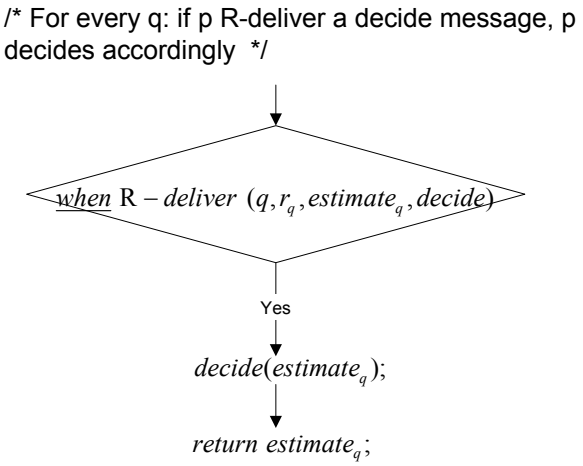
Gelingt es dem Koordinator die gewünschte Anzahl ($\left\lceil \frac{(n+1)}{2} \right\rceil$) von positiven Bestätigungen (ack) zu sammeln und war sein Vorschlag nicht trivial, d.h. $estimate_{c_{r_p}} <> NULL$, so führt der Koordinator die Operation *R-broadcast* für die Nachricht $(p, r_p, estimate_p, decide)$ aus, ansonsten fährt sich der Koordinator selbst herunter.



Phase 4. Datenfluß

Führt der Prozeß p die Operation *R-deliver* für eine Nachricht der Form $(q, r_q, estimate_q, decide)$ aus[2], so prüft p zuerst, ob er selbst schon entschieden hat.

Ist dies nicht der Fall, so entscheidet p den Wert $estimate_q$ und verläßt den *Consensus*-Algorithmus.



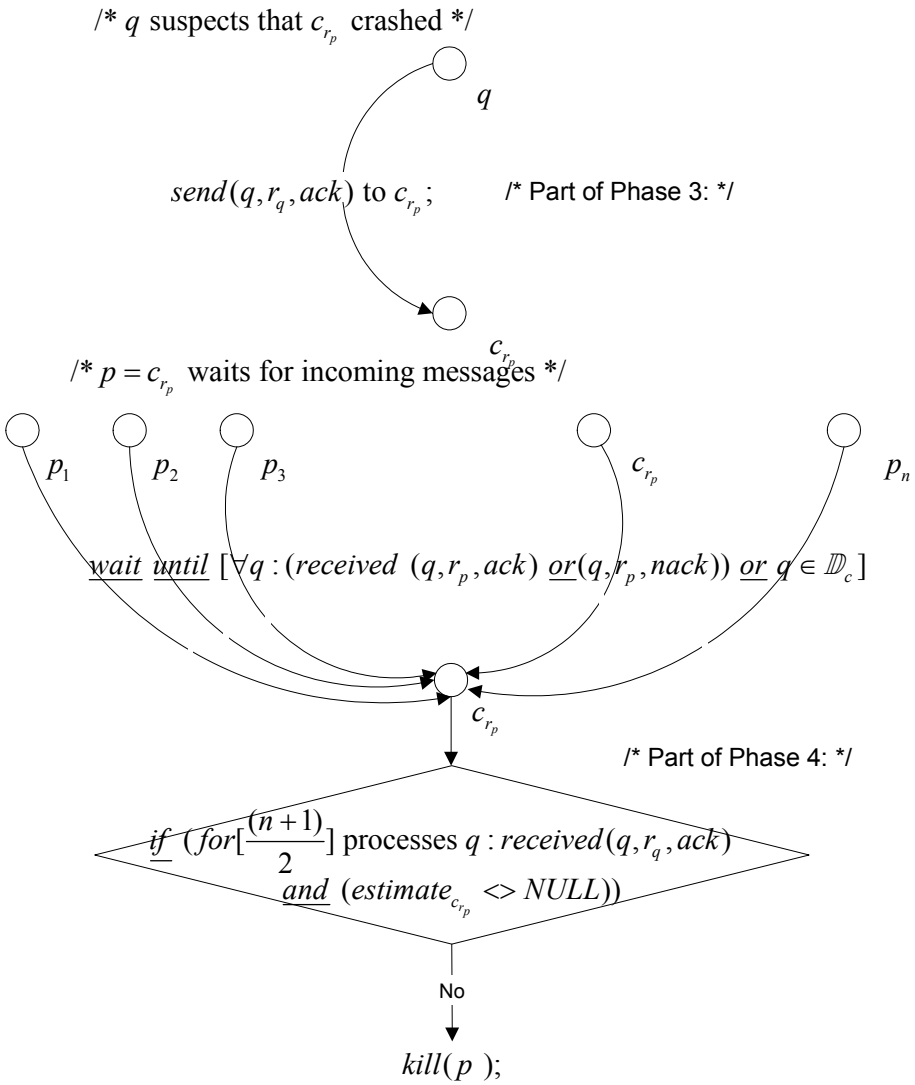
Phase 4. Entscheidung

Der Systemablauf-Korrektor des Algorithmus ist in der Phase 3 bzw. Phase 4 untergebracht.

Jeder Prozeß p schickt am Ende der Phase 3 durch die Anweisung "$send\,(p, r_p, nack)$ to c_{r_p};" ein *nack* an den Koordinator, falls p den Koordinator verdächtigt.

In der Phase 4 wartet, wie bereits erläutert, der Koordinator auf das Eintreffen der Bestätigungen der Teilnehmer. Der Koordinator fährt sich selbst herunter, falls es ihm nicht gelingt wenigstens $\left\lceil \frac{(n+1)}{2} \right\rceil$ positive Bestätigungen zu sammeln oder falls sein Vorschlag trivial war $(estimate_{c_{r_p}} = NULL)$.

[2]die vorher durch die Operation "*R-broadcast* $(p, r_p, estimate_p, decide)$;" ausgelöst wurde



Systemablauf-Korrektor

Zusammenfassend geben wir den Algorithmus im *Pseudo-Code* an. Jeder Prozeß p, welcher am *Consensus* beteiligt ist, führt folgende Anweisungen aus:

```
01 function propose_p (value_p) {

  /* Data structure */

02      r_p : integer; /* rounds */
03      ts_p, t : integer; /* timestamp */
04      c_{r_p} : integer; /* current coordinator */
05      estimate_p : string; /* current estimate of p
                                    for the proposed value */
06      msgs_p : array of string; /* msgs_p[r] is the current
  set of messages received by p as coordinator in the round r */

  /* Initialisation */

07      estimate_p ← value_p;
08      r_p ← 0;
09      ts_p ← 0;

  /* Rotate through coordinator until a decision is reached */

10      while TRUE{

11          r_p ← r_p + 1; /* round initialisation */
12          c_{r_p} ← (r_p mod n ) + 1; /* current coordinator */


  /*
       Phase 1:
  Every process p sends estimate_p to the current coordinator
  */

13          send (p, r_p, estimate_p, ts_p) to c_{r_p};
```

```
/*
Phase 2:
The current coordinator gathers all accessible estimates,
proposes a new estimate and sends it to every process
*/
14      if (p = c_{r_p}) then{

15          wait until [ ∀ q : (received
                (q, r_p, estimate_q, ts_q) from q) or q ∈ D_p];
16          if (for ⌈(n+1)/2⌉ processes q :
                    received (q, r_p, estimate_q, ts_q)) then{
17              msgs_p [r_p] ← (q, r_p, estimate_q, ts_q) |
                    p received (q, r_p, estimate_q, ts_q) from q;
18              t ← largest ts_q such that
                    (q, r_p, estimate_q, ts_q) ∈ msgs_p [r_p] ;
19              estimate_p ← select first estimate_q such that
                        (q, r_p, estimate_q, t) ∈ msgs_p [r_p] ;
20          }else
21              estimate_p ← NULL; /* new iteration */
22          for i := 1 to n do {send (p, r_p, estimate_p) to p_i};
23      }
```

```
/*
Phase 3:
Every process waits for the new estimate proposed by the
current coordinator and replies to the coordinator
*/
24      wait until [received (c_{r_p}, r_p, estimate_{c_{r_p}})
                                        from c_{r_p} or c_{r_p} ∈ D_p];
25      if (received (c_{r_p}, r_p, estimate_{c_{r_p}}) from c_{r_p}) then{
26          if (estimate_{c_{r_p}} <> NULL){
27              estimate_p ← estimate_{c_{r_p}};
28              ts_p ← r_p;
29          }
```

```
30              send (p, r_p, ack) to c_{r_p};
31          }else /* p suspects that c_{r_p} crashed */
32              send (p, r_p, nack) to c_{r_p};

  /*
  Phase 4
  The current coordinator waits for all replies. If they indicate
  that at least ⌈(n+1)/2⌉ processes adopted the estimate, the coor-
  dinator R − broadcast a decide(V) message
  */
33          if (p = c_{r_p}) then{
34              wait until [∀ q : (received
                        (q, r_p, ack) or (q, r_p, nack)) or q ∈ D_p];
35              if (for ⌈(n+1)/2⌉ processes q : received (q, r_p, ack)
                                and (estimate_p <> NULL)) then
36                  R-broadcast (p, r_p, estimate_p, decide);
37              else
38                  kill (p);
39          }

40      }/* while */

41 }
```

```
  /* For every q : if q R-deliver a decide message,
  p decides accordingly */
42          when (R − deliver (q, r_q, estimate_q, decide)) then{
43              decide (estimate_q);
44              return estimate_q;
45          }
```

(2.1) *FS-Cons*-1. Algorithmus zum Lösen des *Consensus*

Wir können den obigen Algorithmus optimieren, indem wir nur dann Prozesse herunterfahren, wenn die Anzahl der aufeinanderfolgenden Verdächtigungen ($curr\_susp_p$) den Schwellwert max_susp_p erreicht hat (im folgenden als $FS\text{-}Cons$-2 bezeichnet).

Um dies zu erreichen, wird der Teil Data Structure sowie Initialisation werden wie folgt ergänzt:

```
01      CONST max_susp_p : integer;
        /* max number of suspicions allowed */
02      curr_susp_p : integer; /* current suspicion */;
```

```
01      curr_susp_p ← 0;
```

Die Anweisung $kill\,(p)$ (Phase 4) wird durch folgende Anweisungsfolge abgeschwächt:

```
01          if (curr_susp_p < max_susp_p) then
02              curr_susp_p ← curr_susp_p + 1;
03          else
04              kill (p) ;
```

Insgesamt haben wir in $FS\text{-}Cons$-2 kurzzeitig auftretende falsche Verdächtigungen aufgefangen, indem verdächtigte Prozesse nicht sofort nach der ersten Verdächtigung heruntergefahren werden.

Sei im folgenden $FS\text{-}Cons$-3 die Änderung von $FS\text{-}Cons$-1, indem die Anweisung $kill\,(p)$ (Phase 4) durch die Platzhalter-Anweisung $Null$ ersetzt wird. Somit werden in $FS\text{-}Cons$-3 verdächtigte Prozesse nicht heruntergefahren.

2.3. Korrektheitsbeweis des Algorithmus. Im folgenden zeigen wir, daß die Eigenschaften *Termination*, *Uniform-Validity*, *Uniform-Agreement* sowie *Uniform-Integrity* des *Consensus* erfüllt werden.

LEMMA 16. *Eine Nachricht m, die vom Prozeß q in der Runde r_q verschickt wurde, wird von einem permanent aktiven Empfänger p in der Runde r_q ausgewertet, oder die Nachricht m wird vom Prozeß p ignoriert.*

BEWEIS. Sei m eine Nachricht, die vom Prozeß q in der Runde r_q verschickt wurde. Gemäß Algorithmus geschieht das Verschicken der Nachricht durch eine Anweisung der Form "$send\,(q, r_q, N, ...)$;". Nach jeder Anweisung der obigen Form folgt im Algorithmus eine Anweisung der Form "$\underline{wait}\ \underline{until}\ [\,\forall\, q : received\,(q, r_p, N, ...)\ from\ q\ \underline{or}\ q \in \mathbb{D}_p]$;". Gemäß Annahme über zuverlässige Kommunikation, erreicht schließlich die Nachricht $m = (q, r_q, N, ...)$ den Empfänger p, es sei denn, p verdächigt q. In diesem letzten Fall wartet p nicht mehr auf das Eintreffen der Nachricht m von q. Somit wertet der Empfänger p in der Runde r_p nur die Nachrichten von q aus, die von q in der Runde $r_q = r_p$ verschickt wurden. ∎

LEMMA 17 (**Uniform Agreement**). *Alle Prozesse, die am Consensus teilnehmen und eventuell entscheiden, entscheiden gleiche Werte.*

BEWEIS. Entscheidet kein Prozeß, so ist das Lemma trivialerweise erfüllt. Wir nehmen an, es gäbe einen Prozeß, welcher entscheidet. Die Eigenschaft *Uniform-Integrity* von *Reliable Broadcast* stellt sicher, daß es Werte $c_r, r, estimate_{c_r}$ gibt, so daß der Koordinator c_r die Anweisung "$R\text{-}broadcast\,(c_r, r, estimate_{c_r}, decide)$;" in der Runde r ausgeführt hat.

Sei r_0 die kleinste Runde, so daß der Koordinator c_{r_0} eine *R-broadcast* Anweisung ausgeführt hat und sei $(c_{r_0}, r_0, estimate_{c_{r_0}}, decide)$ der Wert, welcher durch *Reliable Broadcast* in der Runde r_0, vom Koordinator c_{r_0} in der Phase 4 verschickt wurde.

Wir zeigen durch vollständige Induktion nach der Anzahl der Runden r:

Verschickt der Koordinator $c_{r'}$ die Nachricht $(c_{r'}, r', estimate_{c_{r'}})$ in der Phase 2 der Runde $r' \geq r_0$, dann gilt $estimate_{c_{r'}} = estimate_{c_{r_0}}$ oder

$estimate_{c_{r'}} = NULL$.

Die Aussage gilt trivialerweise für $r' = r_0$. Somit haben in der Phase 3 der Runde r_0 wenigstens $\left\lceil \frac{(n+1)}{2} \right\rceil$ der Prozesse ihre Vorschläge auf $estimate_{c_{r_0}}$ sowie ihren Zeitstempel auf r_0 gesetzt. Kein Prozeß p hat nach Ende der Phase 3 einen eigenen Vorschlag $estimate_p$, der von $estimate_{c_{r_0}}$ verschieden ist bzw. kein Prozeß hat nach Ende der Phase 3 einen Zeitstempel ts_p mit $ts_p \geq r_0$.

Wir nehmen an, die Aussage sei wahr für alle k mit $r_0 < k \leq r$.

Verschickt der Koordinator c_{r+1} die Nachricht $estimate_{c_{r+1}} = NULL$ in der Phase 2 der Runde $r + 1$, so ist schon alles gezeigt. Ansonsten hat der Koordinator wenigstens $\left\lceil \frac{(n+1)}{2} \right\rceil$ Nachrichten der Form $(q, r + 1, estimate_q, ts_q)$ in der Phase 2 der Runde $r + 1$ erhalten. Sei t wie im Algorithmus der größte Zeitstempel ts_q, so daß c_{r+1} die Nachricht $(q, r_q, estimate_q, ts_q)$ erhalten hat und sei $(g, r + 1, estimate_g, t)$ der erste Eintrag in $msgs_{c_{r+1}}(r + 1)$, so daß $(g, r + 1, estimate_g, t) \in msgs_{c_{r+1}}(r + 1)$. Wir zeigen, daß $t \geq r_0$ und $estimate_g = estimate_{c_{r_0}}$.

Gemäß Induktionsvoraussetzung verschickt der jeweilige Koordinator c_k in der Phase 2 der Runde k mit $r_0 < k \leq r$ nur Nachrichten der Form $(c_k, k, estimate_{c_{r_0}})$ oder $(c_k, k, NULL)$. Somit erhält ein Prozeß p in der Runde k in der Phase 3, falls überhaupt dann nur Nachrichten der Form $(c_k, k, estimate_{c_{r_0}})$ oder $(c_k, k, NULL)$. Erhält p eine Nachricht in der Runde k vom lokalen Koordinator, so setzt p seinen eigenen Vorschlag auf $estimate_{c_{r_0}}$ und erhöht seinen Zeitstempel ts_p auf $k > r_0$. Folglich setzten in der Phase 3 der Runden k mit $r_0 \leq k \leq r$ wenigstens $\left\lceil \frac{(n+1)}{2} \right\rceil$ Prozesse ihre Vorschläge auf $estimate_{c_{r_0}}$ und erhöhen den entsprechenden Zeitstempel ts_p auf $ts_p > r_0$ oder lassen den Zeitstempel ts_p unverändert auf $ts_p = r_0$. Kein Prozeß p hat nach Ende der Phase 3 einen eigenen Vorschlag $estimate_p$, welcher von $estimate_{c_{r_0}}$ verschieden ist und $ts_p \geq r_0$.

Dementsprechend verschicken wenigstens $\left\lceil \frac{(n+1)}{2} \right\rceil$ Prozesse p in der Phase 1 der Runde r' den Vorschlag $(p, r', estimate_{c_{r_0}}, ts_p)$, wobei $ts_p \geq r_0$ gilt, an den aktuellen Koordinator $c_{r'}$. Verschickt ein Prozeß p den Vorschlag $(p, r', estimate_p, ts_p)$ mit $estimate_p <> estimate_{c_{r_0}}$ in der Phase 1 der Runde r', so gilt $ts_p \leq r_0$. Folglich verschickt der Koordinator $c_{r'}$ in der Phase 2 der Runde r' die Nachricht $(c_{r'}, r', estimate_{c_{r_0}})$.

Nach dieser Vorbereitung ist die Aussage des Lemmas leicht zu zeigen. Der jeweilige Koordinator verschickt durch *R-broadcast* in der Phase 4**,** falls überhaupt dann seinen eingenen Vorschlag aus der Phase 2. Wie oben gezeigt, ist dieser Vorschlag in jeder Runde $r \geq r_0$ gleich $estimate_{c_{r_0}}$. Folglich verschickt der Koordinator c_r mittels *R-broadcast* in der Phase 4, falls überhaupt dann die Auswertung $estimate_{c_{r_0}}$.

Wir zeigen schließlich, daß jeder Prozeß p, welcher entscheidet, den Wert $estimate_{c_{r_0}}$ entscheidet.
Entscheidet ein Prozeß den Wert $estimate_q$, dann wurde gemäß Eigenschaft *Uniform-Integrity* von *Reliable Broadcast* die Operation "*R-broadcast* $(q, r_q, estimate_q, suspected_q, decide)$;" in der Phase 4 ausgeführt. Gemäß obiger Aussage gilt $estimate_q = estimate_{c_{r_0}}$. Folglich entscheiden alle Prozesse, die am *Consensus* teilnehmen gleiche Werte. ∎

LEMMA 18 (**Termination**). *Jeder permanent aktive Prozeß, welcher am* Consensus *teilnimmt, entscheidet schließlich.*

BEWEIS. Wir zeigen zuerst, daß es keinen permanent aktiven Prozeß gibt, welcher eine Anweisung nicht in endlicher Zeit ausführt.
Wir nehmen an, es gäbe solch einen permanent aktiven Prozeß.
Sei r_0 die kleinste Runde, so daß wenigsten ein Prozeß in dieser Runde eine Anweisung nicht in endlicher Zeit ausführt und sei q solch ein Prozeß.
Die einzigen Stellen, wo q unendlich lange verweilen kann, sind die *wait* Anweisungen der Phasen 2 bis 4. Wir unterscheiden zwei Fälle:

Fall A) Es gibt keinen Prozeß, welcher entscheidet.

Da in der Runde $\max(1, r_0 - 1)$ kein Prozeß unendlich lange wartet, verschicken alle Prozesse p, die nicht abgestürzt sind, in der Phase 1 ihre Vorschläge $estimate_p$ an den aktuellen Koordinator c_{r_0}. Der aktuelle Koordinator c_{r_0} wartet in der Phase 2 auf das Eintreffen aller Vorschläge aus der Phase 1. Der Koordinator c_{r_0} erhält schließlich alle Nachrichten, die in der Runde r_0 Phase 1 verschickt wurden, es sei denn, c_{r_0} verdächtigt (fälschlicherweise) aktive Prozesse. In diesem letzten Fall wird auf das Eintreffen der jeweiligen Nachrichten nicht mehr gewartet. Aufgrund der Eigenschaft *starke Vollständigkeit* verdächtigt c_{r_0} schließlich alle abgestürzten Prozesse. Somit wartet c_{r_0} in

der Phase 2 nicht mehr auf das Eintreffen von Nachrichten von abgestürzten Prozessen. Insgesamt verläßt der Koordinator c_{r_0} die Phase 2.

Fall B) Es gibt einen Prozeß p, welcher entscheidet.

Gemäß Eigenschaft *Agreement* von *Reliable Broadcast* entscheidet auch der Prozeß q schließlich. Hat der Prozeß p seinen Vorschlag $estimate_p$ in der Phase 1 der Runde r_0 an den aktuellen Koordinator c_{r_0} verschickt, so ist schon alles unter *Fall A* gezeigt worden, es sei denn, q entscheidet, bevor er die Phase 2 verläßt. Ansonsten verharrt der Prozeß q so lange in der *wait* Anweisung, bis er eine *decide* Nachricht erhält und entscheidet.

Die gleichen Überlegungen zeigen, daß kein Prozeß in der Phase 3 bzw Phase 4 der Runde r_0 unendlich lange wartet. Widerspruch.

Wir nehmen an, es gäbe einen permanent aktiven Prozeß p, welcher nicht entscheidet. Wir unterscheiden folgende Fälle.

Fall 1) Es gibt wenigstens einen permanent aktiven Prozeß, welcher *R-broadcast* ausführt.

In diesem Fall führt der Prozeß p die Primitive *R-deliver* gemäß Eigenschaft *Validity* und *Agreement* von *Reliable Broadcast* aus und entscheidet schließlich. Widerspruch.

Fall 2) Es gibt keinen Prozeß, welcher *R-broadcast* aufsetzt. Wir unterscheiden mehrere Unterfälle:
a) es gibt eine Runde r', so daß der lokale Koordinator dieser Runde $c_{r'}$ in der Phase 2 wenigstens $\left\lceil \frac{(n+1)}{2} \right\rceil$ Vorschläge erhält. In diesem Fall verschickt der Koordinator $c_{r'}$ seinen Vorschlag $estimate_{c_{r'}}$ an alle Prozesse und wartet auf die Bestätigungen der Prozesse. Falls:
$\mathbf{a}_1$) der Koordinator $c_{r'}$ wenigstens $\left\lceil \frac{(n+1)}{2} \right\rceil$ Bestätigungen (*ack*) erhält, dann führt er *R-broadcast* aus. Wiederspruch.
$\mathbf{a}_2$) der Koordinator weniger als $\left\lceil \frac{(n+1)}{2} \right\rceil$ Bestätigungen erhält, dann erhöht er die Variable $curr\_susp_{c_{r'}}$ in der Phase 4. Falls die Variable

$curr\_susp_{c_{r'}}$ den maximal zulässigen Wert max_susp_p erreicht, dann wird $c_{r'}$ in der Phase 4 heruntergefahren. Insgesamt wird vermieden, daß die Fälle vom Typ a_2 unendlich oft auftreten.

b) Es gibt keine Runde, so daß der lokale Koordinator in der Phase 2 wenigstens $\left\lceil \frac{(n+1)}{2} \right\rceil$ Vorschläge erhält. Sei r eine beliebige Runde. In diesem Fall wird jeweils $curr\_susp_{c_r}$ inkrementiert. Schließlich erreicht $curr\_susp_{c_r}$ den maximalen Wert $max\_susp_{c_r}$ und der Prozeß c_r wird heruntergefahren.

Somit wird vermieden, daß die Fälle vom Typ b) unendlich oft auftreten.

Insgesamt haben wir gezeigt:

Führt kein Prozeß *R-broadcast* aus, dann werden alle Prozesse heruntergefahren.

In diesem Fall ist die Aussage des Lemmas trivialerweise erfüllt. ∎

LEMMA 19 (**Uniform Integrity**). *Jeder Prozeß entscheidet höchstens einmal.*

BEWEIS. Entscheidet der Prozeß p, so bedeutet dies, daß der Prozeß p die Operation *R-deliver* für eine Nachricht der Form $(q, r_q, estimate_q, decide)$ ausführt. Gemäß Eigenschaft *Uniform-Integrity* von *Reliable Broadcast* führt p die Operation *R-deliver* höchstens einmal aus. Somit führt p die nachfolgende Anweisung "$decide\,(estimate_p)$;" höchstens einmal aus und beendet den *Consensus*-Algorithmus. ∎

LEMMA 20 (**Uniform Validity**). *Entscheidet ein Prozeß den Wert V, dann ist V der Anfangswert eines Prozesses, welcher am* Consensus *teilnimmt.*

BEWEIS. Entscheidet der Prozeß p, so führt p eine Anweisung der Form "$decide\,(estimate_q)$;" aus, wobei $estimate_q$ der Vorschlag des Prozesses q ist. Die Variable $estimate_q$ erhält in der Initialisierungsphase den Wert $value_q$, d.h. den Anfangswert von q und wird während der Ausführung des Algorithmus jeweils mit dem Anfangswert eines Prozesses überschrieben. Folglich existiert ein Anfangswert $value_k$, so daß p die Anweisung "$decide\,(value_k)$;" ausführt. ∎

Der folgende Satz garantiert die *Non-Triviality*-Eigenschaft des Systemablauf-Korrektors von *FS-Cons*.

LEMMA 21 (**Non-Triviality Eigenschaft von** $K_{FS-Cons}$). *Kann ein Prozeß p als Koordinator innerhalb höchstens max_susp_p Runden hintereinander in einer Runde r_p mit wenigstens $\left\lceil \frac{(n+1)}{2} \right\rceil$ Prozessen kommunizieren (einschließlich mit sich selbst), so wird p nicht heruntergefahren und entscheidet in der Phase 4 der Runde r_p.*

BEWEIS. Sei p ein beliebiger Prozeß und sei r_p eine Runde, so daß in dieser Runde der Prozeß p Koordinator ist und p mit wenigstens $\left\lceil \frac{(n+1)}{2} \right\rceil$ Prozessen kommunizieren kann (einschließlich mit sich selbst). Dann wählt p in der Phase 2 für $estimate_p$ einen Wert aus dem Wertebereich der möglichen Werte, d.h. $estimate_p <> NULL$ nach der Phase 2. Dementsprechend schicken in der Phase 3 wenigstens $\left\lceil \frac{(n+1)}{2} \right\rceil$ Prozesse q Nachrichten der Form (q, r_p, ack) an den Koordinator p. In der Phase 4 erhält der Koordinator wenigstens $\left\lceil \frac{(n+1)}{2} \right\rceil$ Nachrichten der Form (q, r_p, ack) und entscheidet (d.h. führt "*R-broadcast* $(p, r_p, estimate_p, decide)$;" aus). ∎

THEOREM 2 (**Uniform Consensus**). *Der Algorithmus FS-Cons-1 löst das* Uniform Consensus-*Problem mit Hilfe von Ausfalldetektoren:*

a) die starke Vollständigkeit *erfüllen. Genauigkeitseigenschaften werden nicht vorausgesetzt, verdächtigte Prozesse werden gegebenenfalls heruntergefahren. Abgestürzte Prozesse können hochgefahren werden.*

Gilt die Non-Triviality-*Bedingung des Lemmas (21), so erlaubt der Algorithmus FS-Cons-2 eine nicht-triviale Lösung des Problems. Fälschlicherweise verdächtigte Prozesse werden nicht heruntergefahren.*

b) die starke Vollständigkeit *sowie*

b1) schließliche Pfad-Genauigkeit *oder*

b2) schließlich q-schwache Genauigkeit, *falls Mon nicht schließlich separiert ist.*

erfüllen.

c) die mehrheitliche Vollständigkeit *sowie* schließlich p-mehrheitliche Genauigkeit *erfüllen.*

In Fall b) und c) erlaubt der Algorithmus FS-Cons-3 eine nicht-triviale Lösung des Problems. Fälschlicherweise verdächtigte Prozesse werden nicht heruntergefahren.

BEWEIS. Die Eigenschaften *Termination*, *Uniform-Integrity*, *Uniform-Agreement* und *Uniform-Validity* wurden in den vorangehenden Lemmas bewiesen. Der Fall b) folgt gemäß Proposition (2) sowie Proposition (3). ∎

COROLLARY 7. Consensus *ist mit Hilfe von Ausfalldetektoren lösbar, die* schwache Vollständigkeit *erfüllen. Genauigkeitseigenschaften werden nicht vorausgesetzt, verdächtigte Prozesse werden gegebenenfalls heruntergefahren. Abgestürzte Prozesse können hochgefahren werden.*

BEWEIS. Gemäß Eigenschaft $P1$ sind *schwache Vollständigkeit* und *starke Vollständigkeit* äquivalent. Das Ergebnis folg unmittelbar. ∎

2.4. Das Atomic Broadcast-Problem. Wir betrachten jetzt das *Atomic Broadcast*-Problem. Vereinfacht dargestellt, stellt *Atomic Broadcast* sicher, daß alle aktiven Prozesse dieselben Nachrichten in derselben Reihenfolge empfangen.

DEFINITION 37 (**Atomic Broadcast**). *Erfüllt* Reliable Broadcast *zusätzlich die Eigenschaft:*

***Totale Ordnung**: Liefern zwei aktive Prozesse p und q die Nachricht m und m', dann stellt p die Nachricht m vor m' zu genau dann, wenn q die Nachricht m vor m' zustellt.*

dann sprechen wir von Atomic Broadcast.■

Der folgende Satz wurde der Arbeit [**17**, Corollary 7.1.7] entnommen.

THEOREM 3 (**Chandra und Toueg**). Consensus *und* Atomic Broadcast *sind in asynchronen Systemen äquivalent.*

Consensus kann leicht auf *Atomic Broadcast* zurückgeführt werden. Die folgenden Überlegungen gehen auf Dolev, Dwork und Stockmeyer zurück, siehe [**22**]. Um einen Wert vorzuschlagen, wird *Atomic Broadcast* verwendet. Um einen Wert zu entscheiden, nimmt jeder Prozeß

den Wert der ersten Nachricht, die er weiterleitet (*R-deliver*). Aufgrund der Eigenschaft *Totale Ordnung* von *Atomic Broadcast*, stellen alle aktiven Prozesse dieselbe erste Nachricht zu. Demzufolge wählen alle Prozesse, die am *Consensus* beteiligt sind, denselben Wert und die Eigenschaft *Agreement* von *Consensus* ist erfüllt. Die anderen Eigenschaften von *Consensus* sind auch leicht nachzuprüfen.

Die andere Richtung des obigen Satzes ist tiefliegend, siehe [**17**, Theorem 7.1.6].

REMARK 9. *Theorem (3) ist unabhänging von der Wahl der Ausfalldetektoren, es ist auch in asynchronen Systemen wahr, in denen entweder* Consensus *oder* Atomic Broadcast *implementierbar ist.*■

Als Folgerung von Korrolar (7) und Satz (3) erhalten wir:

COROLLARY 8. Atomic Broadcast *ist mit Hilfe von Ausfalldetektoren lösbar, die* schwache Vollständigkeit *erfüllen. Genauigkeitseigenschaften werden nicht vorausgesetzt, verdächtigte Prozesse werden gegebenenfalls heruntergefahren.*

3. Das *NB-AC*-Problem

3.1. Beschreibung des *NB-AC*. Folgende Beschreibung des *Non-Blocking Atomic Commitment*-Problems (*NB-AC*-Problems) beruht auf der Arbeit von Rachid Guerraoi, siehe [**32**].

Um die Atomizität von Transaktionsausfällen in einem verteilten (Datenbank-) System zu gewährleisten, muß ein *Agreement*-Problem zwischen einer Menge von teilnehmenden Prozessen gelöst werden. Dieses Problem, auch *Atomic Commitment* genannt, verlangt von allen Teilnehmern, daß sie sich auf den Ausgang einer Transaktion einigen, entweder Commit oder Abort.

Man sagt, ein Teilnehmer *AC-entscheidet* Commit bzw. *AC-entscheidet* Abort. Wie ein Teilnehmer *AC*-entscheidet, hängt von den Vorschlägen (Yes oder No) der einzelnen Teilnehmer ab. Ein Teilnehmer *AC*-entscheidet Commit nur dann, wenn alle Teilnehmer Yes vorgeschlagen haben.

Um triviale Lösungen zu vermeiden, wo alle Teilnehmer immer Abort entscheiden, ist es naheliegend zu verlangen, daß Commit entschieden werden sollte, falls alle Teilnehmer Yes vorschlagen und kein Teilnehmer abstürzt.

Wird zusätzlich zum *Atomic Commitment* verlangt, daß jeder permanent aktive Teilnehmer schließlich eine Entscheidung trifft trotz Ausfälle der anderen Teilnehmer, so wird das neue Problem *Non-Blocking Atomic Commitment* (*NB-AC*) genannt. Das Lösen dieses Problems setzt die Teilnehmer in die Lage, Ressoursen freizugeben (z.B. *Locks*), ohne auf die *Recovery* der abgestürzten Teilnehmer zu warten.

Das sogenannte *FLP*-Unmöglichkeitsergebnis, welches besagt, daß es unmöglich ist eine nicht-triviale Einigung (*agreement*) in einem asynchronen System, in dem Abstürze zugelassen sind, zu erzielen, bezieht sich auch auf das *NB-AC*-Problem, siehe [**31**].

DEFINITION 38 (**Non-Blocking Atomic Commitment**). *Das* NB-AC*-Problem ist durch folgende vier Bedingungen angegeben:*

***Uniform-Agreement**: Alle Teilnehmer AC-entscheiden gleiche Werte.*

***Uniform-Validity**: AC-entscheidet ein Teilnehmer* Commit*, dann haben alle Teilnehmer Yes beschlossen.*

***Termination**: Jeder permanent aktive Teilnehmer AC-entscheidet schließlich.*

***Non-Triviality**: Beschließen alle Teilnehmer Yes und gibt es keine Ausfälle, dann AC-entscheidet jeder permanent aktive Teilnehmer schließlich* Commit.■

Es ist eine interessante Frage, ob im Modell von Chandra und Toueg das *NB-AC*-Problem mit Hilfe von unzuverlässigen Ausfalldetektoren lösbar ist. Guerraoui hat gezeigt, daß dies für bestimmte Ausfalldetektor-Klassen wie $(S, \Box pS)$, $(S, \Diamond S)$, $(S, \Diamond pS)$, $(W, \Diamond S)$ bzw. $(W, \Diamond pS)$ nicht der Fall ist (siehe [**32**, Theorem 1, sowie Corrolary 1]) und dies ist nicht überraschend, da das *NB-AC*-Problem stärker eingestuft wurde als *Consensus*.

Die Schwierigkeit, um *NB-AC* zu lösen, betrifft die *Non-Triviality*-Bedingung. Diese Bedingung, üblich nur um triviale Lösungen zu vermeiden, verlangt präzises Wissen über Ausfälle, die nicht von den unzuverlässigen Ausfalldetektoren geliefert werden können.

Wird die *Non-Triviality*-Bedingung des *NB-AC*-Problems abgeschwächt (Commit muß entschieden werden, falls alle Teilnehmer Yes beschließen und kein Teilnehmer ist je verdächtigt), dann kann das neue Problem, auch *Non-Blocking Weak Atomic Commitment* (*NB-WAC*) genannt, gelöst werden. Das *NB-WAC*-Problem ist in asynchronen Systemen, die mit unzuverlässigen Ausfalldetektoren versehen sind (im Sinne von Chandra und Toueg) und in welchen Prozeßabstürze zugelassen sind, auf *Consensus* reduzierbar, d.h. wann immer *Consensus* lösbar ist, ist das *NB-WAC*-Problem auch lösbar. Somit ist das *NB-WAC*-Problem mit Ausfalldetektoren der Klasse $(S, \Box pS)$ lösbar, falls wenigstens ein Teilnehmer aktiv ist bzw. mit Ausfalldetektoren der Klasse $(S, \Diamond pS)$ lösbar, falls die Mehrheit der Teilnehmer aktiv ist.

Wir zeigen, daß in unserem Ansatz, in dem verdächtigte Prozesse gegebenenfalls heruntergefahren werden können, das *Non-Blocking Atomic Commitment*-Problem mit Hilfe von unzuverlässigen Ausfalldetektoren auf *Consensus* reduzierbar ist, d.h. wenn immer *Consensus* lösbar ist, ist auch das *NB-AC*-Problem lösbar.

Ein direkter Vergleich mit den Ergebnissen von Delporte-Gallet, Hadzilacos, Fauconnier, Kouznetsov, Guerraoui und Toueg aus [**21**] ist nicht möglich, da der Ausfalldekter $(\Psi, \mathcal{FS})$, welcher das *NB-AC*-Problem löst, den Ausfall von Prozessen signalisiert. Dies ist mit den unzuverlässigen Ausfalldetektoren in unserem Modell nicht möglich.

3.2. Algorithmus zum Lösen des *NB-AC*-Problems.

3.2.1. *Funktionsweise des Algorithmus.* Der folgende Algorithmus *FS-NB-AC* zum Lösen des *NB-AC*-Problems verwendet explizit einen Algorithmus (im folgenden als $UniformConsensus$ bezeichnet), welches das *Uniform Consensus*-Problem löst wie z.B. *FS-Cons* aus diesem Kapitel. Wir geben den Algorithmus im Run wieder, welcher vom Prozeß p gestartet wurde.

Sei n die Anzahl der Teilnehmer, die am *NB-AC*-Problem teilnehmen. Der Algorithmus *FS-NB-AC* verwendet die Variablen: $votum_p$, $consens_p$, $suspicion_p$ und $outcome_p$.

Die Variable $votum_p$ enthält den Anfangswert des Prozesses p, d.h. den Wert womit p den Algorithmus startet und dieser Wert ist entweder Yes oder No. Analog enthält die Variable $consens_p$ den Vorschlag von p für die Prozedur $UniformConsensus$. Verdächtigt p einen

Prozeß q, von dem er eine Nachricht erwartet, so setzt er die Variable $suspicion_p$ auf $TRUE$, ansonsten enthält diese Variable den Wert $FALSE$. Die Variable $outcome_p$ enthält den (durch den *Consensus*-Algorithmus) *AC*-entschiedenen Wert (Zeile 23).

Jeder Prozeß p, welcher am *Atomic Commitment* teilnimmt, startet den Algorithmus, indem er die Variable $consens_p$ auf Commit setzt und an alle Teilnehmer sein Votum $votum_p$ verschickt. Nachdem p sein Votum verschickt hat, wartet p auf die entsprechenden Voten der anderen Teilnehmer einschließlich auf sein Eigenes. Erhält p vom Prozeß q den Wert No, dann setzt er die Variable $consens_p$ auf Abort. Wurde q verdächtigt und hat p kein No erhalten, dann wird q von dieser Verdächtigung in Kenntnis gesetzt, indem p ein *nack* an q verschickt, ansonsten bestätigt p den Erhalt des Votums von q durch ein *ack*.

Hat p von wenigstens einem Prozeß q eine Nachricht $(q, votum_q)$ mit $votum_q = No$ erhalten, dann wird $UniformConsensus(consens_p)$ aufgerufen und p beendet den Algorithmus, ansonsten wartet p auf alle positiven bzw. negativen Bestätigungen der Teilnehmer, es sei denn, der Teilnehmer wird verdächtigt. Erhält p von einem beliebigen Teilnehmer q ein *nack* oder wurde q vom Ausfalldetektormodul von p verdächtigt, dann wird p heruntergefahren. Wurden Prozesse verdächtigt, d.h. $suspicion_p = TRUE$ dann wird $consens_p$ auf Abort gesetzt. Anschließend wird $UniformConsensus(consens_p)$ aufgerufen und p beendet den Algorithmus.

3.2.2. *Algorithmus.* Jeder Prozeß p, welcher am *Atomic Commitment* teilnimmt, startet folgende Prozedur:

```
01 function AtomicCommitment (votum_p) {

02      consens_p :=Commit;
03      suspicion_p := FALSE;

04      for q := 1 to n do
05          send (p, votum_p) to q;

06      wait until [ ∀ q : received (q, votum_q) or q ∈ 𝔻_p];
07      if (∃q . ¬ received (q, votum_q)) then
08          suspicion_p := TRUE;
09      if (∃q . votum_q = No) then
10          consens_p :=Abort;

11      for q := 1 to n do
12          if ((consens_p =Commit) and (¬ received (q, votum_q))
                                                          then
13              send (p, nack) to q;
14          else
15              send (p, ack) to q;

16      if (consens_p = Commit) then{
17          wait until [ ∀q : received (q, ack) or received (q, nack)
                                                  or q ∈ 𝔻_p];
18          if (∃q . ¬ received (q, ack)) then
19              kill (p) ;
20          if (suspicion_p) then
21              consens_p :=Abort;
22      }

23      outcome_p := UniformConsensus (consens_p) ;

24      return outcome_p;

25 }
```

FS-NB-AC. Algorithmus zum Lösen des *NB-AC*-Problems

3.2.3. *Korrektheitsbeweis.* Im folgenden zeigen wir, daß die Eigenschaften *Termination*, *Uniform-Validity*, *Uniform-Agreement* sowie *Non-Triviality* des *Non-Blocking Atomic Commitments* erfüllt werden.

LEMMA 22 (**Termination**). *Jeder permanent aktive Prozeß AC-entscheidet schließlich.*

BEWEIS. Wir zeigen: Jeder permanent aktive Prozeß p führt die Anweisung $outcome_p := UniformConsensus\,(consens_p)$ (Zeile 23) aus und verläßt den Algorithmus mit der Anweisung $return\ outcome_p$ (Zeile 24).

Die einzigen Stellen, wo p unendlich lange warten würde, sind die zwei *wait* Anweisungen (Zeile 06 und Zeile 17) sowie die Ausführung von $UniformConsensus$. Da jede gesendete Nachricht schließlich den Empfänger erreicht, erhält p schließlich alle Nachrichten, die an ihn geschickt wurden.

Stürzt ein Prozeß q ab, so wird dies durch den Ausfalldetektor erkannt (*starkte Vollständigkeit*). Somit wartet p nie unendlich lange in den *wait* Anweisungen und startet $UniformConsensus$, es sei denn, p wird heruntergefahren. Gemäß Eigenschaft *Termination* von *Consensus* *AC*-entscheidet schließlich jeder permanent aktive Prozeß. ∎

LEMMA 23 (**Uniform Agreement**). *Je zwei Teilnehmer AC-entscheiden nie verschiedene Werte.*

BEWEIS. Jeder Prozeß, welcher überhaupt *AC*-entscheidet, führt die Anweisung "$outcome_p := UniformConsensus\,(consens_p)$;" (Zeile 23) aus, siehe auch Beweis zu Lemma 22. Gemäß der Eingeschaft *Uniform-Agreement* des *Consensus* entscheiden zwei Teilnehmer (aktiv oder abgestürzt) nie verschiedene Werte. Somit *AC*-entscheiden zwei Teilnehmer nie verschieden. ∎

LEMMA 24 (**Uniform Validity**). *AC-entscheidet ein Teilnehmer* Commit, *dann haben alle Teilnehmer* Yes *beschlossen.*

BEWEIS. *AC*-entscheidet ein Teilnehmer Commit, dann wurde $outcome_p$ nach einem Aufruf von $UniformConsensus$ auf Commit gesetzt. Gemäß Eigenschaft *Uniform-Validity* von *Consensus* hat ein

Prozeß q den Wert Commit vorgeschlagen. Somit hat q die Anweisung "$outcome_q := UniformConsensus\,(consens_q)$;" mit dem Parameter $consens_q$ =Commit ausgeführt, d.h. q hat von allen Teilnehmern das Votum Yes erhalten. Somit haben alle Teilnehmer Yes beschlossen. ∎

LEMMA 25 (**Non-Triviality**). *Beschließen alle Teilnehmer* Yes *und gibt es keine Ausfälle, dann AC-entscheidet jeder permanent aktive Teilnehmer schließlich* Commit.

BEWEIS. Gibt es keine Ausfälle, dann gibt es gemäß obigem Algorithmus (Zeile 18) auch keine (falschen) Verdächtigungen. Schlagen alle Prozesse Yes vor, dann schlägt jeder Prozeß im *Consensus*-Algorithmus Commit vor. Gemäß Eigenschaft *Uniform-Validity* des *Consensus* AC-entscheidet schließlich jeder permanent aktive Teilnehmer. ∎

Der folgende Satz garantiert die *Non-Triviality*-Eigenschaft des Systemablauf-Korrektors von $FS\text{-}NB\text{-}AC$.

LEMMA 26 (**Non-Triviality von** $K_{FS-NB-AC-1}$). *Kann ein Prozeß p mit allen anderen Prozessen, die am Algorithmus FS-NB-AC teilnehmen, kommunizieren oder ist sein Votum* No*, dann wird p vom Systemablauf-Korrektor von FS-NB-AC nicht heruntergefahren.*

BEWEIS. Die Aussage des Satzes erfolgt sofort gemäß if Bedingung des Algorithmus (Zeile 18). ∎

REMARK 10 (**Forced Crash**). *Haben alle Teilnehmer* Commit *vorgeschlagen, dann führt jede Verdächtigung im Rahmen des Algorithmus FS-NB-AC zum Herunterfahren des verdächtigten Prozesses.*■

THEOREM 4. *Das* Non-Blocking Atomic Commitment*-Problem kann auf* Consensus *reduziert werden unter Zuhilfenahme von Ausfalldetektoren,*

a) die starke Vollständigkeit *erfüllen. Genauigkeitseigenschaften werden nicht vorausgesetzt, verdächtigte Prozesse werden gegebenenfalls heruntergefahren. Abgestürzte Prozesse können hochgefahren werden.*

Gilt die Non-Triviality-Bedingung des Lemmas (21), so erlaubt der Algorithmus FS-Cons-2 eine nicht-triviale Lösung des Problems. Fälschlicherweise verdächtigte Prozesse werden nicht heruntergefahren.

b) die starke Vollständigkeit *sowie*

b1) schließliche Pfad-Genauigkeit *oder*

b2) schließlich q-schwache Genauigkeit, *falls Mon nicht schließlich separiert ist.*

erfüllen.

c) die mehrheitliche Vollständigkeit *sowie* schließlich mehrheitliche Genauigkeit *erfüllen.*

In Fall b) und c) erlaubt der Algorithmus FS-Cons-3 sowie der Algorithmus FS-NB-AC eine nicht-triviale Lösung des Problems. Fälschlicherweise verdächtigte Prozesse werden nicht heruntergefahren.

BEWEIS. Die Eigenschaften *Termination*, *Uniform-Agreement*, *Uniform-Validity* und *Non-Triviality* wurden in den vorangehenden Lemmas bewiesen. Die Gültigkeit des *Consensus* ist Inhalt des Theorems (2). Der Fall b) folgt gemäß Proposition (2) sowie Proposition (3). ∎

Da in unserem Ansatz *schwache Vollständigkeit* und *starke Vollständigkeit* äquivalent sind und *Consensus* lösbar ist, siehe Theorem 2 sowie Corollary 7, erhalten wir:

COROLLARY 9. *Das* Non-Blocking Atomic Commitment-*Problem kann mit Hilfe von Ausfalldetektoren, die* schwache Vollständigkeit *erfüllen, gelöst werden. Genauigkeitseigenschaften werden nicht vorausgesetzt, verdächtigte Prozesse werden gegebenenfalls heruntergefahren. Abgestürzte Prozesse können hochgefahren werden.*

KAPITEL 8

Hochverfügbarkeit von Applikationen

In diesem Kapitel untersuchen wir, inwieweit Ausfalldetektoren zur Erhöhung der Verfügbarkeit von Applikationen beitragen können. Dabei stützten wir uns auf die Erkenntnisse der vorausgehenden Kapitel und zeigen, daß *Consensus* die Güte unserer Algorithmen wesentlich erhöhen kann.

Zuerst untersuchen wir zwei Implementierungsmethoden von Ausfalldetektoren, und zwar:

a) *Timeout*-Mechanismen sowie

b) Auswertung der *Prozeß-ID* aus der Statustabelle der Prozesse.

Anschließend gehen wir näher auf die Unterschiede zu den Ausfalldetektoren im Modell von Chandra und Toueg ein und vergleichen unser Konzept mit kommerziellen Überwachungsprogrammen. Unser Konzept sieht eine applikationsfeine Überwachung des Systems vor, da im allgemeinen wichtig ist, in welcher Reihenfolge die Prozesse einer Anwendung gestartet werden. Die gängigen Überwachungstools wie *Hawk* von Tibco oder *Isis Availability Manager* von Stratus melden nur den Ausfall von Prozessen, eine Überwachung von Applikationen ist nicht vorgesehen.

Anschließend stellen wir drei Strategien zur Einleitung einer *Recovery* vor. Diese Strategien sehen auch eine sinnvolle Verteilung bzw. Synchronisation des Zustandsvektors zwischen den einzelnen Entscheidungsprozessen vor. Zu den ersten zwei Strategien unterbreiten wir auch Algorithmen. Die erste Strategie verwendet einen variablen Koordinator und liefert in praktischen Systemen zufriedenstellende Ergebnisse, ohne *Consensus* zu erfüllen. Die zweite Verteilungsstrategie verwendet implizit einen *Consensus*-Algorithmus, um eine Einigung unter den Entscheidungsprozessen über den jeweils aktuellen Zustandsvektor zu erzielen. Die dritte Verteilungsstrategie verwendet einen festen Koordinator, um *Recovery* einzuleiten. Dadurch wird *Consensus* implizit

erzielt.

Abschließend veranschaulichen wir anhand eines einfachen Beispiels unsere *Recovery*-Strategie. Der Zustand einer Applikation A wird aus den Informationen erstellt:

a) die der Ausfalldetektor über die Prozesse liefert, die von A gestartet wurden bzw.

b) die die Entscheidungsprozesse beim Herunter- oder Hochfahren einzelner Applikationen generieren.

1. Ausfalldetektoren und Hochverfügbarkeit

1.1. Implementierungsmethoden von Ausfalldetektoren.

Viele Untersuchungen über Ausfalldetektoren schlagen *Timeout*-Mechanismen als mögliche Implementierung vor. Jeder Anwendungsprozeß p schickt in regelmäßigen Abständen eine "*I am alive*" Nachricht an seine Überwachungsprozesse. Bleibt die Nachricht "*I am alive*" aus, dann wird p als abgestürzt eingestuft und ein entsprechender Eintrag bzw. Änderung wird jeweils im (von den Überwachungsprozessen geführten) Zustandsvektor vorgenommen. Erhalten die Überwachungsprozesse später die Nachricht "*I am alive*" von p, dann ersetzen sie jeweils den Eintrag "p ist abgestürzt" durch "p ist aktiv".

In einem asynchronem System wird durch dieses Schema kein Ausfalldetektor im Sinne von Chandra und Toueg implementiert, denn die *schließlich p-starke Genauigkeit* wird in diesem Modell nicht erfüllt. Chandra und Toueg schlagen vor, in praktischen Systemen die *Timeouts* entsprechend höher zu setzen, so daß es keine *Timeouts* für wenigstens einen aktiven Prozeß gibt. Wie dies sichergestellt wird, ist in den Arbeiten von Chandra und Toueg nicht näher erläutert.

Allerdings kann in praktischen Systemen also auch in unserem Ansatz, in denen kurzzeitige Prozessorbelastungen auftreten, die *kontinuierlich starke Genauigkeit* erfüllt werden.

Die uns bekannten kommerziellen Ausfalldetektoren verwenden die *Prozeß-ID* aus der Statustabelle der Prozesse, um die Applikationsprozesse zu überwachen. Hängende Prozesse können mit dieser Methode

nicht erfaßt werden. Eine Methode, um hängende Prozesse zu identifizieren ohne *Timeout*-Mechanismen zu verwenden, wäre die Prozesse auf extreme CPU-Tätigkeit zu untersuchen.

1.2. Unvollkommene Ausfalldetektoren. Wie in der Einführung beschrieben, lassen wir zu, daß jeder Überwachungsprozeß q Fehler machen kann, indem er fälschlicherweise:

a) den Zustand von aktiven Prozessen als abgestürzt einstuft, oder

b) abgestürzte Prozesse nicht erkennt.

Dieses Fehlverhalten von Ausfalldetektoren hängt mit der Implementierungsmethode zusammen. Geschieht die Überwachung über *Timeouts*, so hängt das erste Fehlverhalten von Ausfalldetektoren mit der Tatsache zusammen, daß es unmöglich ist zwischen abgestürzten Prozessen und Prozessen mit sehr langen Antwortzeiten zu unterscheiden. Das zweite Fehlverhalten von Ausfalldetektoren, d.h. die Möglichkeit, daß abgestürzte Prozesse unerkannt bleiben, hängt mit den unvollkommenen Informationen aus der Statustabelle zusammen. Wie bekannt, werden hängende Prozesse nicht aus der Statustabelle entfernt bzw. als abgestürzt gekennzeichnet.

1.3. Informationen über abgestürzte Prozesse. Im Modell von Chandra und Toueg wird die Liste der abgestürzten Prozesse von jedem Überwachungsprozeß ergänzt und dementsprechend weitergegeben, d.h. wird ein Prozeß p einmal von einem Überwachungsprozeß als abgestürzt eingestuft, dann wird p weiter als abgestürzt in der aktuellen Runde des *Consensus*-Algorithmus behandelt, obwohl es Überwachungsprozesse geben kann, die p als aktiv erkannt haben (**Worst Case Szenario**).
Wird die Ausfallerfassung mit Hilfe von *Timeouts* implementiert, so spricht die Praxis eher gegen eine solche Ergänzung der Liste der abgestürzten Prozesse. Wird ein Prozeß p als aktiv von einem Überwachungsprozeß erkannt, dann ist es naheliegend p als aktiv einzustufen, obwohl es Überwachungsprozesse gegeben haben konnte, die p als abgestürzt gesehen haben (**Best Case Szenario**).
Wie erwähnt, lassen wir zu, daß das System alle (aktiven) Prozesse herunterfährt, die fälschlicherweise als abgestürzt eingestuft wurden.

1.4. Vergleich mit kommerziellen Überwachungsprogrammen. Die größte Unzulänglichkeit der gängigen kommerziellen Überwachungsprogrammen besteht darin, daß sie nicht Applikationen sondern nur Prozesse überwachen (wie z.B. *Hawk* von Tibco oder *Isis Availability Manager* von Stratus). Somit können zwar abgestürzte Prozesse neu gestartet werden bzw. ein *Recovery*-Skript aktiviert werden, es kann aber keine koordinierte Aktion eingeleitet werden, welche die verschiedenen Prozeßabstürze berücksichtigt und eine globale Strategie erarbeitet.

Stürzen fast gleichzeitig mehrere Prozesse ab, dann würde mit diesen Tools eine unkontrollierte, fast gleichzeitig gestartete *Recovery* für jeden abgestürzten Prozeß angestoßen. Obwohl noch mit sehr viel Mühe und Aufwand lokale Applikationen sich überwachen lassen und einfache *Recovery*-Mechanismen implementierbar sind, ist koordiniertes *Recovery* über Rechnergrenzen hinweg mit den obengenannten Tools nicht mehr sinnvoll möglich.

2. Beschreibung der Überwachungsstrategie

2.1. Aufbau der Überwachung. Die lokalen Überwachungsprozesse ermitteln den Status der überwachten Anwendungsprozesse, den sogenannten Zustandsvektor und leiten diese Information an die Entscheidungsprozesse weiter.

Sei m ein beliebiger Überwachungsprozeß, sei $App(m)$ wie üblich die Menge der Prozesse, die von m überwacht werden und sei V_m der aktuelle Zustandsvektor. Die Prozedur $send\_status\_vector(\underline{in}\ m)$ startet m, ermittelt den aktuellen Zustandsvektor V_m und schickt diesen Wert an alle Entscheidungsprozesse. Das Einsammeln von Zustandsinformationen wird nach einer vordefinierten Zeit δ_m fortgesetzt.

```
01 procedure send_status_vector(in m) {
 /* m is the current monitoring process */

      /* Data structure */
02      V_m : array of char; /* status vector buffer */

03      repeat
          /* reset the status vector buffer */
04          V_m ← (∞, ∞, ..., ∞) ;
05          ∀p ∈ App(m) : V_m ← status_p;
06          for q ∈ Decp do      send (V_m) to q;
07          sleep(δ_Mon); /* wait a predefined time */
08      forever
09 }
```

Um den Zusammenbruch der Überwachung durch Wegfall der Entscheidungsprozesse zu vermeiden, werden die Entscheidungsprozesse repliziert.

Nach einem bestimmten Auswahlverfahren wird zwischen den Entscheidungsprozessen ein *Koordinator* gewählt. Die Rolle des Koordinators besteht darin anhand der Zustandsmatrix den vermeintlichen Status der überwachten Prozesse zu ermitteln und gegebenenfalls eine

Recovery einzuleiten. Die Entscheidungsprozesse tauschen untereinander den aktuellen Zustandsvektor aus, um die Konsistenz der jeweiligen Statusmatrix zu gewährleisten.

Wir unterscheiden in dieser Arbeit drei Strategien zur Wahl des Koordinators, und zwar DST1, DST2 und DST3.

DST1:

Diese Strategie sieht einen Koordinator nur dann vor, falls eine Mehrheit der Entscheidungsprozesse sich auf diesen Koordinator einigen kann. Kommt keine Mehrheit zusammen, dann wird kein Koordinator gewählt. Alle Entscheidungsprozesse wissen, ob sie Koordinator sind oder nicht, ohne umbedingt den aktuellen Koordinator zu kennen. Der Koordinator ergänzt die Zustandsmatrix mit den aktuellen Werten des Zustandsvektors und leitet eventuell eine *Recovery* ein. Die Entscheidungsprozesse, die an der Wahl des Koordinators beteiligt waren, erhalten den Zustandsvektor des aktuellen Koordinators und verwenden diesen Zustandsvektor, um ihre Zustandsmatrix zu ergänzen.

Wir erläutern zuerst detailliert die Verteilungstrategie DST1 und geben nachher den Algorithmus im *Pseudo-Code* an.

Beschreibung des Algorithmus:

Nach dem Start aktiviert jeder Entscheidungsprozeß p ein *Thread* um die Zustandsvektoren von den Überwachungsprozessen zu empfangen. Sei n die Anzahl der Prozesse, die am Algorithmus teilnehmen.

Der Koordinator wird wie folgt bestimmt:
Jeder Entscheidungsprozeß p schickt seine Prozeßnummer und seinen Zustandsvektor V_p an alle Entscheidungprozesse und wartet anschließend auf das Eintreffen der entsprechenden Information von den anderen Teilnehmern, es sei denn, der Teilnehmer ist verdächtigt. Anschließend wird aus den erhaltenen Antworten bei jedem Prozeß p die kleinste Prozeßnummer $p_{\min}$ ermittelt, falls es dem Prozeß p gelungen ist mit einer Mehrheit (d.h. wenigstens $\left\lceil \frac{n+1}{2} \right\rceil$) von Teilnehmern zu kommunizieren, ansonsten wird $p_{\min}$ auf $\emptyset$ gesetzt.

Sei $abs\_maj\,(n,(q_1,q_2,...,q_k))$ gleich q, falls wenigstens $\left[\frac{n+1}{2}\right]$ der Prozesse aus $\{q_1,q_2,...,q_k\}$ gleich q sind, ansonsten $\emptyset$.

Dasselbe Verfahren wird wiederholt, indem jeder Prozeß jetzt die Nachricht $(p_{\min}, V_{p_{\min}})$ verschickt. Am Ende dieses Nachrichtenaustausches ist bei jedem Prozeß p_{abs_maj} entweder $\emptyset$ oder enthält denselben von $\emptyset$ verschiedenen Wert. Der Prozeß p mit $p = p_{abs\_maj}$ ist der Koordinator, es sei denn, p_{abs_maj} ist bei jedem Prozeß gleich $\emptyset$. In diesem letzten Fall wird kein Koordinator gewählt. Der Koordinator ergänzt durch $V_{p_{\min}}$ die Zustandsmatrix $SM_{p_{\min}}$ und wertet sie aus für eine Entscheidung über *Recovery*. Um transiente Zustände zu vermeiden, wird eine *Recovery* nur dann eingeleitet, falls keine neuen Abstürze in einem bestimmten Zeitintervall erfolgen. In der letzten Phase leitet der Koordinator nach Auswertung der Statusmatrix eventuell eine *Recovery* ein.

Alle anderen Entscheidungsprozesse p ergänzen ihre Zustandsmatrix SM_p durch $V_{p_{\min}}$, falls $p_{\min}$ ungleich $\emptyset$ ist, ansonsten durch V_p. Durch dieses Verfahren wird sichergestellt, daß bei allen Prozessen, die miteinander kommunizieren können und eine Mehrheit bilden können, derselbe Wert $V_{p_{\min}}$ zur Ergänzung der Zustandsmatrix SM_p verwendet wird. Diese Form der Einigung, ohne *Consensus* zu erfüllen, ist für praktische Systeme zufriedenstellend.

Sei p ein beliebiger Entscheidungsprozeß.

Wir geben im folgenden den Algorithmus für den Prozeß p im *Pseudo-Code* an.

```
01  procedure rec_round_DST1(in p){

  /* Data structure */
02      V_p : array of char; /* status vector buffer */

  /* send the own status vectors to all participants */
03      for q := 1 to n do        send (p, V_p) to q;
04      wait until [ ∀ q : received (q, V_q) or q ∈ 𝔻_p];
05      if (for at least ⌈(n+1)/2⌉ processes q : received (q, V_q) then
06          p_min := min (p_q | received (q, V_q)) ;
07      else
08          p_min := ∅;


  /* choose the coordinator */
09      for q := 1 to n do        send (p_min, V_p) to q;
10      wait until [ ∀ q : received (q, V_q) or q ∈ 𝔻_p];
11      p_abs_maj := abs_maj (n, (q ≠ ∅ | received (q, V_q))) ;

12      if (p = p_abs_maj) then{ /* current coordinator */

13          SM_p ← V_{p_abs_maj};
14          Compute_Status_Vector_Matrix;
15          Recovery;

16      }else{/* no coordinator */

17          if (p_abs_maj = ∅) then
18              SM_p ← V_p;
19          else
20              SM_p ← V_{p_abs_maj};

21      }

22 }
```

DST2:

Diese Strategie verwendet einen *Consensus*-Algorithmus, wie er z.B. in den vorigen Kapiteln beschrieben wurde. Vereinfach dargestellt schlägt jeder Entscheidungsprozeß einen Wert = (Prozeßnummer, Zustandsvektor) vor und nach Beenden des *Consensus*-Algorithmus erhält jeder Entscheidungsprozeß denselben entschiedenen und verbindlichen Wert.

Wir erläutern zuerst die Verteilungstrategie DST2 und geben nachher den Algorithmus im *Pseudo-Code* an.

Beschreibung des Algorithmus:

Nach dem Start aktiviert jeder Entscheidungsprozeß p ein *Thread* um die Zustandsvektoren von den Überwachungsprozessen zu empfangen. Am Anfang wird durch einen *Consensus*-Algorithmus die Prozeßnummer p sowie der aktuelle Zustandsvektor V_p an alle Entscheidungsprozesse verschickt und eine eindeutige Einigung über einen Wert (c, V_c) aus der Menge der verschickten Werte getroffen. Bei jedem Entscheidungsprozeß p wird die Zustandsmatrix SM_p durch denselben Wert V_c ergänzt. Der Entscheidungsprozeß c übernimmt die Rolle des Koordinators und leitet, falls er inzwischen nicht abgestürzt ist, eventuell eine *Recovery* ein, im Rahmen dessen die abgestürzten Prozesse bzw. Applikationen hochgefahren werden.

Wir geben im folgenden den Algorithmus für den Prozeß p im *Pseudo-Code* an.

```
01 procedure rec_round_DST2(in p){

  /* Data structure */
02      V_p : array of char; /* status vector buffer */

  /* start Consensus */
03      (c, V_c) ← Consensus (p, V_p) ;
04      SM_p ← V_c;

05      if (p = c) then{ /* current coordinator */
06           Compute_Status_Vector_Matrix;
07           Recovery;
08      }

09 }
```

DST3:

Diese Verteilungsstrategie verwendet einen festen Koordinator. Der Koordinator kann in der Regel nicht abgewählt werden. Stürzt der Koordinator ab, so wird nach einem bestimmten Auswahlkriterium ein neuer Koordinator gewählt.

Die Entscheidungsgsprozesse leiten, wie bei den anderen Strategien, den Statusvektor zum Koordinator weiter. In der nächsten Phase wird ein neuer korrigierter Statusvektor erstellt, die für alle Entscheidungsprozesse verbindlich ist (*Consensus*).

3. *Recovery*-Strategie

3.1. Grundlagen. Wir führen zuerst zwei Relationen ein. Die erste Relation gibt die Reihenfolge an, in der einzelne Applikationen gestartet werden und die zweite Relation gibt die Abhängigkeit nach einem Absturz einer der Applikationen wieder.

DEFINITION 39 (**Relation** *R-st*). *Seien A und B Applikationen. Wir sagen: "*die Applikationen A und B stehen in der Relation *R-st* zueinander"*, als Zeichen $A \sqsupset B$, falls durch Hochfahren der Applikation A auch B hochgefahren wird.*

Seien $B_1, B_2, ..., B_k$ Applikationen.

Wir setzen: $A \sqsupset \{B_1, B_2, ..., B_k\}$ für $(A \sqsupset B_1) \wedge (A \sqsupset B_2) \wedge ... \wedge (A \sqsupset B_k)$ ■

DEFINITION 40 (**Relation** *R-rec*). *Seien A und B Applikationen. Wir sagen: "*die Applikationen A und B stehen in der Relation *R-rec* zueinander"*, als Zeichen $A \gg B$, falls $A \sqsupset B$ und falls das Abstürzen von B den Ausfall der Applikation A verursacht.*

Formal:

$$A \gg B \iff (A \sqsupset B) \ .$$
$$(\exists t_0 \in T \ . \ \mathcal{S}(t_0, B) = \dot{+}) \implies \exists t' > t_0 \ . \ \mathcal{S}(t', A) = \dot{+}$$

Seien $B_1, B_2, ..., B_k$ Applikationen.

Wir setzen: $A \gg \{B_1, B_2, ..., B_k\}$ für $(A \gg B_1) \wedge (A \gg B_2) \wedge ... \wedge (A \gg B_k)$ ■

Wie leicht ersichtlich, ist die Relation $\sqsupset$ ist transitiv, d.h. aus $A \sqsupset B$ und $B \sqsupset C$ folgt unmittelbar $A \sqsupset C$.
Analoges gilt für $\gg$.

Wie bekannt, ist $SV := \{\uparrow, \downarrow, \dot{+}, \nearrow, \searrow, \emptyset, \infty\}$ der Wertebereich der

Zustandswerte, die für eine Applikation von den Entscheidungsprozessen ermitteln werden können.

Berücksichtigen wir **replizierte** Überwachungsprozesse und erhalten die Entscheidungsprozesse verschiedene Zustandswerte, so kann eine Verknüpfung der einzelnen Zustandswerte mit Hilfe des (wie anschließend erläutert) auf SV erweiterten Operators $\star$ erfolgen (siehe auch Definition (31)). Die Idee der Erweiterung auf SV besteht darin, beim *Recovery*-Vorgang für die Verknüpfung den plausibelsten Wert zu setzen. So wird z.B. $\star(\uparrow, \searrow)$ als $\searrow$ gesetzt, da man davon ausgeht, daß ein Entscheidungsprozeß schon die *Recovery* eingeleitet hat.

DEFINITION 41. *Sei* $\star : SV \times SV \rightarrow SV$ *tabellarisch wie folgt definiert:*

$$\begin{array}{rrcl}
\forall a \in SV: & \star(a, \dot{+}): & = & \star(\dot{+}, a) := \dot{+} \\
\forall a \in SV \setminus \{\dot{+}\}: & \star(a, \infty): & = & \star(\infty, a) := a \\
\forall a \in SV \setminus \{\dot{+}, \infty\}: & \star(a, \emptyset): & = & \star(\emptyset, a) := \emptyset \\
& \star(\uparrow, \nearrow): & = & \star(\nearrow, \uparrow) := \uparrow \\
& \star(\uparrow, \downarrow): & = & \star(\downarrow, \uparrow) := \downarrow \\
& \star(\uparrow, \searrow): & = & \star(\searrow, \uparrow) := \searrow \\
& \star(\downarrow, \nearrow): & = & \star(\nearrow, \downarrow) := \nearrow \\
& \star(\downarrow, \searrow): & = & \star(\searrow, \downarrow) := \downarrow \\
& \star(\nearrow, \searrow): & = & \star(\searrow, \nearrow) := \nearrow
\end{array}$$

Wir erweitern die Definition von $\star$*, indem wir für alle* $k > 2$ *setzen:*

$$\forall a_{_{i \in \mathbb{N}_k}} \in SV : \star(a_1, a_2, ..., a_k) := \star(\star(a_1, a_2, ..., a_{k-1}), a_k)$$

sowie der Vollständigkeit wegen:

$$\forall a \in SV : \star a := a$$

■

Es können beim Festlegen der Verknüpfungsvorschrift auch andere Strategien verwendet werden.

3.2. Beispiel. Wir betrachten im folgenden ein willkürlich gewähltes Beispiel. Zuerst stellen wir den Applikationsbaum als Relation *R-st* dar:

$A \sqsupset \{p_1, A_1, A_2\}$
$A_1 \sqsupset \{p_1^1, A_1^1, A_2^1\}$
$A_2 \sqsupset \{p_1^2, p_2^2\}$
$A_3 \sqsupset \{p_1^3, p_2^3\}$
$A_4 \sqsupset \{p_1^4, p_1^4\}$
$A_5 \sqsupset \{p_1^5, p_1^5\}$
$A_1^1 \sqsupset \{p_1^{1,1}, p_2^{1,1}, p_3^{1,1}, A_1^{1,1}\}$
$A_2^1 \sqsupset \{p_1^{1,2}, p_2^{1,2}, p_3^{1,2}\}$
$A_1^{1,1} \sqsupset \{p_1^{1,1,1}, p_2^{1,1,1}, p_3^{1,1,1}\}$

nachher alternativ als Graphik.

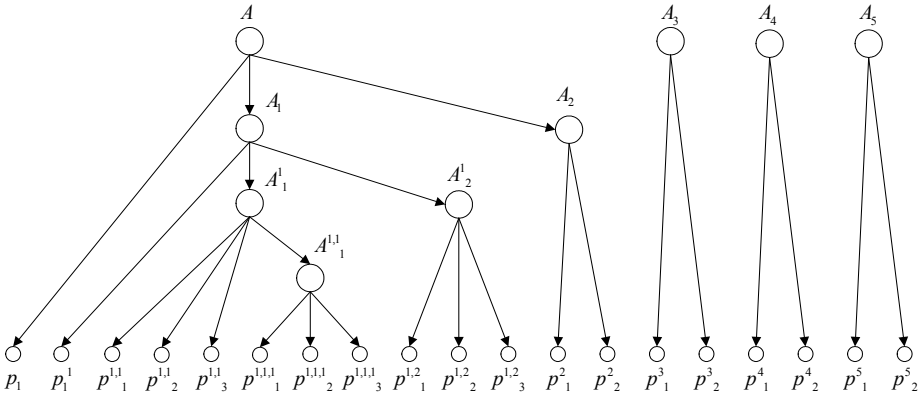


ABBILDUNG 1. Baum-Struktur

Wie ersichtlich, sind die Applikationen A, A_3, A_4, A_5 jeweils von den anderen Applikationen unabhängig. Wir können bei den obigen Verknüpfungen die Applikationen durch die Prozesse, die diese Applikationen starten, ersetzen. Die obigen Verknüpfungen siehen dann folgendermaßen aus:

$A \sqsupset \{p_1, p_1^1, p_1^{1,1}, p_2^{1,1}, p_3^{1,1}, p_1^{1,1,1}, p_2^{1,1,1}, p_3^{1,1,1}, p_1^{1,2}, p_2^{1,2}, p_3^{1,2}, p_1^2, p_2^2\}$
$A_1 \sqsupset \{p_1^1, p_1^{1,1}, p_2^{1,1}, p_3^{1,1}, p_1^{1,1,1}, p_2^{1,1,1}, p_3^{1,1,1}, p_1^{1,2}, p_2^{1,2}, p_3^{1,2}\}$
$A_2 \sqsupset \{p_1^2, p_2^2\}$
$A_3 \sqsupset \{p_1^3, p_2^3\}$
$A_4 \sqsupset \{p_1^4, p_2^4\}$

$A_5 \sqsupset \{p_1^5, p_1^5\}$
$A_1^1 \sqsupset \left\{p_1^{1,1}, p_2^{1,1}, p_3^{1,1}, p_1^{1,1,1}, p_2^{1,1,1}, p_3^{1,1,1}\right\}$
$A_2^1 \sqsupset \left\{p_1^{1,2}, p_2^{1,2}, p_3^{1,2}\right\}$
$A_1^{1,1} \sqsupset \left\{p_1^{1,1,1}, p_2^{1,1,1}, p_3^{1,1,1}\right\}.$

Es ist leicht zu erkennen, welche Applikationen jeweils welche Prozesse starten.

In unserem Beispiel bestehen folgende *R-rec* Abhängigkeiten (Absturz einer Applikation verursacht den Ausfall der übergeordneten Applikationen):

$A \gg \{A_1, A_2\}$
$A_1 \gg \{p_1^1, A_1^1\}$
$A_3 \gg \{p_2^3\}$
$A_1^1 \gg \left\{p_1^{1,1}, p_2^{1,1}, A_1^{1,1}\right\}$
$A_1^{1,1} \gg \left\{p_1^{1,1,1}, p_2^{1,1,1}\right\}$

und falls wir die Applikationen auflösen, dann gilt:

$A \gg \left\{p_1^1, p_1^{1,1}, p_2^{1,1}, p_1^{1,1,1}, p_2^{1,1,1}\right\}$
$A_3 \gg \{p_2^3\}.$

Gibt es zu einem Prozeß p keine Applikation B, so daß $B \gg p$, so wird p nach einem Absturz im Rahmen einer *Recovery* selbstständig hochgefahren.

Insgesamt haben wir eindeutig einer Gruppe von Prozessen eine Applikation zugeordnet, dessen *Shutdown*- bzw. *Reboot*-Skript im Rahmen einer *Recovery* gestartet werden sollte, falls eine Teilmenge dieser Prozesse abstürzt.

Wir nehmen an, die Applikationen sind auf fünf Rechnern $R_1, R_2, ..., R_5$ verteilt, so daß die Applikation A_i auf dem Rechner R_i läuft ($1 \leq i \leq 5$). Auf jedem Rechner R_i läuft ein Überwachungsprozeß m_i, welcher die lokalen Prozesse auf dem Rechner R_i überwacht.

Die Statusinformationen werden an die folgenden drei Entscheidungsprozesse d_1, d_2 und d_3 verschickt. Die Entscheidungsprozesse

sammeln die Zustandsinformationen von den einzelnen Überwachungsprozessen ein, tauschen untereinander den Zustandsvektor aus und leiten eine *Recovery* ein, falls sich das System in einem stabilen (also nicht transienten) Zustand befinden.

Die Überwachungsprozesse können nur zwischen den Zuständen $\uparrow$ und $\dot{+}$ unterscheiden. In unserem Beispiel sendet der Überwachungsprozeß m_1 den Zustandsvektor:

p_1	p_1^1	p_1^{11}	p_2^{11}	p_3^{11}	p_1^{12}	p_2^{12}	p_3^{12}	p_1^{111}	p_2^{111}	p_3^{111}
$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\dot{+}$	$\uparrow$

, wobei der besseren Übersichtlichkeit halber der Zustandsvektor tabellarisch dargestellt ist. Insgesamt erhalten die Entscheidungsprozesse folgenden Zustandsvektor.

p_1	p_1^1	p_1^{11}	p_2^{11}	p_3^{11}	p_1^{12}	p_2^{12}	p_3^{12}	p_1^{111}	p_2^{111}	p_3^{111}	p_1^2	p_2^2	p_1^3	p_2^3
$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\dot{+}$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\dot{+}$

p_1^4	p_2^4	p_1^5	p_2^5
$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$

Anhand dieses Vektors und den Informationen, die mit der vom Koordinator gestarteten *Recovery* zusammenhängen, wird der Zustandsvektor der Applikationen wie folgt:

A	A_1	A_2	A_3	A_4	A_5	A_1^1	A_2^1	A_{11}^1
$\dot{+}$	$\dot{+}$	$\uparrow$	$\dot{+}$	$\uparrow$	$\uparrow$	$\dot{+}$	$\uparrow$	$\dot{+}$

erstellt. Somit kann man aus den obigen Tabellen eindeutig die Applikation(en) identifizieren, die abgestürzt sind und für welche eine *Recovery* sinnvoll ist. In unserem Beispiel sind es die Applikationen A und A_3. Die Applikationen A_1, A_2, A_1^1 und A_{11}^1 hängen von der Applikation A ab. Bevor eine *Recovery* für eine Applikation eingeleitet wird, wird der Status der entsprechenden Applikation auf $\searrow$ gesetzt.

Ist es wichtig, in welcher Reihenfolge die Applikationen gestartet werden, bzw. ist es wichtig, ob eine Applikation hochgefahren werden sollte, bevor die *Recovery* einer anderen Applikation gestartet werden

kann, so muß dies im *Recovery*-Algorithmus hinterlegt sein. Wir nehmen an, die Applikation A_1 muß seine *Recovery*-Phase beendet haben, bevor das Hochfahren der Applikation A_2 beginnen kann.

Wir zeigen anhand des Zustandsvektors, wie eine *Recovery* in diesem Fall ausgeführt werden kann. Um die Darstellungsweise einfach zu halten, nehmen wir an, die Zustände sind jeweils stabil, d.h. sie bleiben über mehrere Auswertungen konstant. In unserem Fall wird der Statusvektor des Koordinators wie folgt:

A	A_1	A_2	A_3	A_4	A_5	A_{11}	A_{12}	A_{111}
$\searrow$	$\searrow$	$\searrow$	$\searrow$	$\uparrow$	$\uparrow$	$\searrow$	$\searrow$	$\searrow$

gesetzt. Die *Recovery*-Phase wird eingeleitet, indem die entsprechenden Applikationen heruntergefahren werden. Die zweite Phase (Hochfahren der Applikationen) wird eingeleitet, falls die entsprechenden Applikationen vollständig heruntergefahren wurden.

In unserem Beispiel sind die Applikationen A und A_3 voneinander unabhängig, d.h. sie können voneinander unabhängig herunter und hochgefahren werden.

Erhält der jeweilige Koordinator ein Zustandsvektor der folgenden Form:

p_1	p_1^1	p_1^{11}	p_2^{11}	p_1^{12}	p_2^{12}	p_3^{12}	p_1^{111}	p_2^{111}	p_3^{111}	p_1^2	p_2^2	p_1^3	p_2^3
$\dot{+}$	$\dot{+}$	$\dot{+}$	$\dot{+}$	$\dot{+}$	$\dot{+}$	$\dot{+}$	$\dot{+}$	$\dot{+}$	$\dot{+}$	$\dot{+}$	$\dot{+}$	$\dot{+}$	$\dot{+}$

, dann kann das Hochfahren der Applikation A_3 bzw. der Applikation A erfolgen. Der Zustandsvektor der Entscheidungsprozesse wird wie folgt:

A	A_1	A_2	A_3	A_4	A_5	A_{11}	A_{12}	A_{111}
$\nearrow$	$\nearrow$	$\downarrow$	$\nearrow$	$\uparrow$	$\uparrow$	$\nearrow$	$\nearrow$	$\nearrow$

ergänzt. Das Hochfahren der Applikation A_2 wird eingeleitet, falls die Applikation A_1 vollständig hochgefahren wurde, d.h. der jeweilige Koordinator erwartet einen Zustandsvektors der Form:

p_1	p_1^1	p_1^{11}	p_2^{11}	p_1^{12}	p_2^{12}	p_3^{12}	p_1^{111}	p_2^{111}	p_3^{111}	p_1^2	p_2^2	p_1^3	p_2^3
$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\dot{+}$	$\dot{+}$	$\uparrow$	$\uparrow$

bevor A_2 hochgefahren wird. Die Zustände der Applikationen werden wie folgt:

A	A_1	A_2	A_3	A_4	A_5	A_{11}	A_{12}	A_{111}
$\nearrow$	$\uparrow$	$\nearrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$

gesetzt. Der neue Zustandsvektor wird jeweils an alle Entscheidungsprozesse im Rahmen der Überwachungsstrategie verschickt. Wenn alle Prozesse der Applikation A hochkommen, wird der Status von A auf $\uparrow$ gesetzt.

REMARK 11. *Man beachte, daß wir in diesem Beispiel davon ausgegangen sind, daß dieselbe Applikation nicht in kurzen Zeitabständen ausfällt. Unter Umständen wird dann die* Recovery *der Applikation neu gestartet, bevor die vorausgehende* Recovery *vollständig beendet wurde.*

KAPITEL 9

Ausblick

In diesem Kapitel behandeln wir Möglichkeiten der Weiterentwicklung unserer Ansätze.

Wir führen ST-Eigenschaften ein. Die ST-Eigenschaften beziehen sich auf die Status-Tabelle der Prozesse, um Aussagen über den Zustand der Prozesse zu erhalten.

Es stellt sich in natürlicher Weise die Frage, welche Eigenschaften einschließlich ST-Eigenschaften ein Ausfalldetektor erfüllen muß, damit in unserem Ansatz *Consensus* bzw. mit *Consensus* verwandte Probleme lösbar sind.

Eine andere offene Frage in unserem Ansatz ist der Zusammenhang zwischen dem *Highly Available Leader Election*-Problem und dem *Consensus*. Das *Highly Available Leader Election*-Problem verlangt, vereinfacht dargestellt, daß schließlich ein Prozeß die Rolle des Koordinators übernimmt und dies von allen Prozessen akzeptiert wird.

1. ST-Eigenschaften

Im folgenden ergänzen wir unseren Ansatz und führen ST-Eigenschaften ein.

Wir sagen: "ein Prozeß p ist **st-abgestürzt**", falls der Prozeß p nicht mehr in der Statustabelle der Prozesse (ST) geführt wird oder dort den Zustand beendet ($\curvearrowright$) hat, wir sagen: "ein Prozeß p ist **st-aktiv**", falls p in der Statustabelle geführt ist und dort den Zustand aktiv ($\uparrow$) hat.

ST-schwache Vollständigkeit verlangt, daß schließlich jeder abgestürzte Prozeß p von wenigstens einem Prozeß q, welcher p st-überwacht auch st-verdächtigt wird. Analog erfüllt ein System **ST-starke Vollständigkeit**, falls schließlich jeder abgestürzte Prozeß p von allen

Prozessen, die p überwachen, st-verdächtigt wird. **ST-starke Genauigkeit** verlangt, daß kein Prozeß st-verdächtigt wird, bevor er abstürzt.

ST-schwache Vollständigkeit ist eine Eigenschaft, die in praktischen Systemem nicht erfüllt werden kann, da hängende Prozesse nicht aus der Status-Tabelle der Prozesse entfernt werden. Somit ist es in praktischen Systemen nur sichergestellt, daß st-verdächtigte Prozesse abgestürzt sind. *ST-starke Genauigkeit* wird in praktischen Systemen erfüllt, da Prozesse, die aus der Status-Tabelle entfernt wurden, nicht mehr aktiv sind.

Durch Einführen von ST-Eigenschaften kann der Status der Prozesse in praktischen Systemen bis auf die hängenden Prozesse korrekt ermittelt werden. Über hängende Prozesse liefern uns die ST-Eigenschaften keine Informationen. Somit sind ST-Eigenschaften von Ausfalldetektoren allein nicht ausreichend, um ein Überwachungssytem darauf aufzusetzen. Allerdings können ST-Eigenschaften, um die schon behandelten bzw. um neue Vollständigkeits- und Genauigkeitseigenschaften ergänzt werden.

Es stellt sich in natürlicher Weise die Frage, welche zusätzlichen Eigenschaften ein Ausfalldetektor erfüllen muß, um den *Consensus* bzw. mit dem *Consensus* verwandte Probleme zu lösen.

2. *Highly Available Leader Election*

Das *Highly Available Leader Election*-Problem verlangt von den Teilnehmern, daß zu jedem Zeitpunkt t_0 ein Zeitpunkt $t > t_0$ existiert, so daß ein hochverfügbarer Koordinator zum Zeitpunkt t existiert.

Es stellt sich die Frage, wie das *Highly Available Leader Election*-Problem und *Consensus* zusammenhängen, d.h. welche (minimalen) Eigenschaften ein Ausfalldetektor erfüllen sollte, damit das *Highly Available Leader Election*-Problem gelöst werden kann bzw. unter welchen Bedingungen das obige Problem direkt gelöst werden kann, ohne *Consensus* zu verwenden.

Wir haben die Existenz eines (hochverfügbaren) Koordinators bei der Überwachungsstrategie $DST3$ ohne weitere Begründung vorausgesetzt.

KAPITEL 10

Anhang

1. Ausgewählte Symbole und Abkürzungen

in der Reihenfolge ihres Erscheinens ab Kapitel 2

Formale	Beschreibung des Systems
Aufbau des Systems	
Π	Menge der Prozesse in einem System
p	Prozeß
App	Teilmenge der Applikationsprozesse (von Π)
Mon	Teilmenge der Überwachungsprozesse (von Π)
Dec	Teilmenge der Entscheidungsprozesse (von Π)
Mon	Teilmenge der Überwachungsprozesse (von Π)
$Mon\,(p)$	Teilmenge von Mon, die p überwachen
$App\,(q)$	Teilmenge von App, von q überwacht
T	Bereich der Uhrschläge
Zustandswerte	
SV_P	Zustände, die ein Prozeß tatsächlich annimt
SV_A	Zustände, die eine Appl. tatsächlich annimt
SV	Wertebereich der Zustände
$\uparrow$	aktiver Prozeß
$\dot{+}$	abgestürzter Prozeß
$\mp$	als nicht vertrauenswürdig eingestuft
$\downarrow$	Prozeß wurde heruntergefahren
$\nearrow$	Hochfahren gestartet
$\searrow$	Herunterfahren gestartet
$\emptyset$	kein vernünftiges Zusammenfügen möglich
∞	kein Wert ermittelt
Systemablauf	
$\mathcal{S}$	Systemablauf
$\mathbb{S}$	Menge aller Systemabläufe
t	Zeitpunkt

$activ(\mathcal{S})_{t_0,t_c}$	Menge der aktiven Prozesse zw. t_0 u. t_c
$rec(\mathcal{S})_{t_0,t_c}$	Menge der Prozesse in *Rec.* zw. t_0 u. t_c
$crashed(\mathcal{S})_{t_0,t_c}$	Menge der abgest. Prozesse zw. t_0 u. t_c
$perm\_activ(\mathcal{S})$	Menge der aktiven Prozesse zw. 0 u. ∞
$activ(\mathcal{S})_{t_0}$	Menge der aktiven Prozesse zw. t_0 u. ∞
$Func(\mathcal{S}, p)$	Menge der funktionellen Überw. bzg. p

Erm. v. Zuständen

$\mathbb{D}$	Ausfalldetektor
H	Ausfalldetektor-*History*
$\mathbb{D}(\mathcal{S})$	Ausfalldetektor-*Histories* von $\mathcal{S}$
$\mathbb{H}(\mathbb{S})$	Menge der Ausfalldetektor-*Histories*
$perm\_crash$	Prozeß ist permanent abgestürzt
$event\_perm\_susp$	Prozeß ist schließlich perm. verdächtigt
$false\_susp$	Aktiver Prozeß wird verdächtigt
$correct\_failure\_\det ec$	Abgestürzter Prozeß wird verdächtigt

Stand der Forschung

Ausfalldetektoren

P	A-Klasse mit *st. Vollst.* u. *st. Genauig.*
Q	A-Klasse mit *schw. Vollst.* u. *st. Genauig.*
$(S, \Box pS)$	A-Klasse mit *st. Vollst.* u. *abs. p-st. Genauig.*
W	A-Klasse mit *schw. Vollst.* u. *schw. Genauig.*
$(S, \Diamond S)$	A-Klasse mit *st. Vollst.* u. *schl. st. Genauig.*
$(W, \Diamond S)$	A-Klasse mit *schw. Vollst.* u. *schl st. Genauig.*
$(S, \Diamond pS)$	A-Klasse mit *st. Vollst.* u. *schl. p-st.* Gen.
$<> W$	A-Klasse mit *schw. Vollst.* u. *schl schw. Gen.*
f	Anzahl der abgestürzten Prozesse
$<> W\,(om)$	A-Klasse in Message Omission Failure Umg.
CCP	Collective Consistency Protocol
$<> S_e$	A-Klasse, wertet die *Epoch*-Zahl aus
$<> S_u$	A-Klasse, wertet die *Epoch*-Zahl aus
$\perp$	Anfangswert im Alg. von Chandra u. Toueg

Zuverlässige Kommunikation

$C_{p,q}$	Kanal mit Nachricht von p nach q
$m_{p,q}$	Nachricht von p nach q
Sen	Menge der Sender
$Mes_{p,q}$	Menge der Nachrichten der Form $m_{p,q}$
$s_{p,q}$	Sequenznummer zu $m_{p,q}$
$Seq_{p,q}$	Menge der Sequenznummern der Form $s_{p,q}$
$OutBuf_{p,q}$	Puffer für Nachricht beim Sender
$OutSeqN_{p,q}$	Puffer für die Sequenznummer von $m_{p,q}$
$AckSeqN_{p,q}$	Sequenznr. von $m_{p,q}$, letzt. bestätig. von p
$InBuf_{p,q}$	Puffer für Nachricht beim Sender
$InSeqN_{p,q}$	Sequenznr. von $m_{p,q}$, letzter Empfang bei q
$C^m_{p,q}$	Kanal, dem $m_{p,q}$ übergeben wurde
$succ\_send_{p,q}(m_{p,q})$	$m_{p,q}$ erfolgreich an den Kanal übergeben
$\perp$	Initialisierungswert in den Puffern

Ausgewählte Agreement-Probleme

Algorithmus	
A	Algorithmus
R	Run eines Algorithmus
I	Initiale Konfiguration
St	Eine unendliche Folge von Schritten
$\mathbb{P}$	Problem
v	Variable im Algorithmus
v_q	Kopie des Prozesses p von v
v^R	*History* von v im Run R
$v_q^R(t)$	Wert von v_q z. Zeitpunkt t im Run R
$\mathbb{D}_p$	lokales Ausfalldetektormodul von p
Reduktion	
Z	Aggregationsvorschrift
$aggreg_q$	emuliert die Ausgabe des Prozesses q
$aggreg^R$	emuliert die Ausgabe des Runs R
$\mathbb{D} \succeq \mathbb{D}'$	$\mathbb{D}'$ ist auf $\mathbb{D}$ reduzierbar
$\mathbb{D} \cong \mathbb{D}'$	$\mathbb{D}'$ und $\mathbb{D}$ sind äquivalent
schw. Voll– >	**st. Voll**
$status_q$	von q ermittelter Zustand der überw. Prozesse
$\star$	Verknüpfungsoperator
$\bullet$	Verknüpfungsoperator
$P1$	Überführung von *schw. Vollst.* in *st. Vollst.*
$P2$	Erhalt von *kont. st. Genauigkeit*
$P'2$	Erhalt von *kont. pq-st. Genauigkeit*
$P3$	Überführung v. *kont. pq-st. Gen.* in *kont. st. Gen.*
$P4$	Erhalt von *starker Vollst.*

Consensus Alg.

$estimate_p$	vermuteter Entscheidungswert von p
$state_p$	Status von p (decided/undecided)
r_p	aktuelle Runde von p
ts_p	Runde in der $estimate_p$ aktualisiert wurde
c_{r_p}	Koordinator in der Runde r_p
$q \in \mathbb{D}_p$	q wird von p verdächtigt
$suspected_p$	Liste der Prozesse, die von p verdächtigt werden
$msgs_p[r_p]$	von p erhaltene Nachricht in der Runde r_p
ack	Nachricht wurde erhalten
$nack$	Nachricht wurde nicht erhalten

Atomic Commit.

$NB\text{-}AC$	*Non-Blocking Atomic Commitment*
$NB\text{-}WAC$	*Non-Blocking Weak Atomic Commitment*

2. Lebenslauf

geboren am 11. April 1955 in Borsec/Rumänien

Schulbildung

1970 bis 1974	naturwissenschafliches Gymnasium in Sathmar/Rumänien
Juni 1974	Abitur

Hochschulausbildung

1974 bis 1980	Technische Universität "Traian Vuia" in Temeschburg/Rumänien
Juni 1980	Abschluß Dipl.-Ing. Maschinenbau
Okt. 1982	Aufnahme eines Studiums der Mathematik an der Fernuniversität Hagen
ab März 1990	Möglichkeit zur Ausreise in die B.R. Deutschland um die Prüfungen/Praktika abzulegen
Juni 1994	Abschluß Dipl-Math.

Berufliche Tätigkeit

Sept. 1980 bis Juni 1990	als Ing. in Sathmar/Rumänien
Sept. 1990 bis Juni. 1994	stud./wiss. Mitarb. an der Fernuniversität Hagen
Juni 1994 bis Nov. 1994	Fortbildung in Informatik/Praktikum bei Siemens
seit Jan. 1995	Rechenzentrum von Siemens/Infineon in Dresden

Literaturverzeichnis

[1] Adleno, A.; Andreasson, S. -A.; ed. Yiksel, O. "Fault tolerance aspects in computerizes systems" 7th Mediterranean Electrotechnical Conference. Proceedings (Cat. No. 94CM3388-6) vol.3 IEEE: New York, NY, USA, 1994. p.1024-8 vol. 3, 3 vol. xvi+1348 pp., 24 refs. Conference: Antalya, Turkey, 12-14 April 1994.

[2] Afek, Y.; Attiya, H.; Fekete, A. D.; Fischer, M. ; Lynch, N.; Mansour, Y; Wang, D. and Zuck, L; "Reliable communication over unreliable channels" Journal of the ACM, 41(6):1267–1297, 1994.

[3] Aguilera, M. K.; Chen, W.; Toueg, S.; "Failure Detection and Consensus in the Crash-Recovery Model." Distributed Computing, Volume 13, Number 2, 2000 pp. 99-125

[4] Aguilera, M. K.; Chen, W.; Toueg, S.; "Heartbeat: A Timeout-Free Failure Detector for Quiescent Reliable Communication" Marios Mavronicolas, Philippas Tsigas (Eds.): Distributed Algorithms, 11th International Workshop, WDAG '97, Saarbrücken, Germany, September 24-26, 1997, Proceedings. Lecture Notes in Computer Science, Vol. 1320, Springer, 1997, ISBN 3-540-63575-0 pp. 126-140

[5] Aguilera, M. K.; Chen, W.; Toueg, S.; "Using the Heartbeat Failure Detector for Quiescent Reliable Communication and Consensus in Partitionable Networks" Theoretical Computer Science, Volume 220, Number 1, June 1999 pp.3-30

[6] Aguilera, M. K.; Chen, W.; Toueg, S.; "On the Weakest Failure Detector for Quiescent Reliable Communication" Technical Report 97-1640, Department of Computer Science, Cornell University, July 1997.

[7] Aguilera, M. K.; Chen, W.; Toueg, S.; "On Quiescent Reliable Communication" SIAM Journal on Computing, Volume 29, Number 6, 2000 pp. 2040-2073

[8] L. Alvisi, D. Malkhi, L. Pierce, and M. Reiter. "Fault detection for Byzantine quorum systems (extended abstract) " Proceedings of the 7th IFIP International Working Conference on Dependable Computing for Critical Applications, January 1999

[9] Basu, A.; Charron-Bost, B.; Toueg, S. "Simulating Reliable Links with Unreliable Links in the Presence of Process Crashes." WDAG 1996: 105-122

[10] Basu, A.; Charron-Bost, B.; Toueg, S. "Solving problems in the presence of process crashes and lossy links." Technical Report TR96-1609, Cornell University

[11] Benn, E.Z.; Rosenberg, J.; "Fault tolerance techniques and their applicability to persistent systems" Australian Computer Science Communications James Cook Univ: 1997. vol.19, no.1, p.277-86, 27 refs. Conference: Sydney, NSW,

Australia, 5-7 Feb. 1997 SICI: 0157-3055(1997)19:1L.277:FTTT;1-6 CODEN: ACSCDD ISSN 0157-3055 Conference paper (English)

[12] Berman, P.; Garay, J. A.; and Perry, K. J.; "Towards optimal distributed consensus ". In Proceedings of the Thirtieth Symposium on Foundations of Computer Science (Oct. 1989), pp. 410-415. IEEE Computer Society Press.

[13] Birman, K.P. ;Cooper, R. ;Joseph,T.A. ;Kane, K. and Schmuck, F.B. "Isis. A Distributed Programming Environment" Cornell University, 1990

[14] Carreira, J.; Costa, D.; Silva, J.G.; "Fault tolerance for Windows applications" BYTE (International Edition) (1997) vol 22, no.2, p 51-2. 0 refs.

[15] Chandra, T. D.; Hadzilacos, V.; Toueg, S.; "The weakest failure detector for solving consensus " Journal of the ACM, 43(4):685-722, July 1996.

[16] Chandra, T. D. and Toueg, S. 1990. "Time and message efficient reliable broadcasts ". In Proceedings of the Fourth International Workshop on Distributed Algorithms (Sept. 1990), pp. 289-300. Springer Verlag.

[17] Chandra, T. D.; Toueg, S.; "Unreliable Failure Detectors for Reliable Distributed Systems" I.B. M. Thomas J. Watson Research Center, Hawthorne, New York Journal of the ACM, Vol. 43, No. 2, March 1996. pp. 225-267.

[18] Chandra, T. D., Hadzilacos, V., Toueg, S and Charron-Bost, B. "Impossibility of group membership in asynchronous systems " Technical Report 95-1533, Computer Science Department, Cornell University, Ithaca, 1995.

[19] Chang, J. and Maxemchuk, N. "Reliable broadcast protocols ", ACM Transactions on Computer Systems 2,3 (Aug.), 1984 pp. 251-273.

[20] Cristian, F. and Schmuck F. "Agreeing on processor-group membership in asynchronous distributed systems. Technical Report CSE95-428. UCSD, 1995.

[21] Delporte-Gallet, C.;Hadzilacos, V.; Fauconnier, H.; Kouznetsov, P.; Guerraoui, R.; Toueg, S. "Certain Fundamental Problems in Distributed Computing" PODC'04, July 25-28, 2004 St. Johns, Newfoundland, Canada. Copyright 2004 ACM 1-58113-802-4/04/0007

[22] Dolev, D.;Dwork, C.; Stockmeyer, L.; "On the minimal synchronism needed for distributed consensus" J. ACM 34, 1(Jan.), 1987. pp. 77-97.

[23] Dolev, D.; Malki, D.(Comput. Sci. Inst. , Hebrew Univ., Jerusalem, Israel) Theory and Practice in Distributed Systems. International Workshop. Selected Papers Editor Birman, K.P.; Mattern, F.; Schiper, A. Berlin, Germany: Springer-Verlag, 1995. p.83-98 ISBN: 3-540-60042-6

[24] Dolev, D.: Keidar, I.: Friedman, R.: Malkhi D.: "Failure Detectors in Omission Failure Environments" Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing, Santa Barbara, California, USA, August 21-24,1997. ACM, ISBN 0-89791-952-1 286

[25] Dwork, C. ;Ho, C. T.;Strong, R.; "Collective Consistency" I.B.M. Almaden Research Center Distributed Algorithms. 10th International Workshop, WDAG '96. Proceedings Springer-Verlag: Berlin, Germany, 1996. p.234-50, viii+379 pp., 27 refs.

[26] Dwork, C., Lynch, N. A., and Stockmeyer, L. "Consensus in the presence of partial synchrony" Journal of the ACM 35, 2(April), 1988 pp. 288-323.

[27] Fetzer, Ch. and Cristian, F. "On the Possibility of Consensus in Asynchronous Systems" In Proceedings of the 1995 Pacific Rim Int'l Symp. on Fault-Tolerant Systems, pages 86-91, Newport Beach, CA, Dec 1995.

[28] Fetzer, Ch. and Cristian, F. "Fail-Aware Failure Detectors" The 15th Symposium on Reliable Distributed Systems, October 23-25, 1996, Niagara-on-the-Lake, Ontario, Canada, Proceedings. IEEE Computer Society, 1996, ISBN 0-8186-8178-0 pp. 200-209

[29] Fetzer, Ch. and Cristian, F. "Fail-awareness in timed asynchronous systems ". Proceedings of the Fifteenth Annual ACM Symposium on Princiles of Distributed Computing, Philadelphia, Pennsylvania, USA, May 23-26, 1996.ACM, ISBN 0-89791-800-2 pp.314-321

[30] Fischer, M. J. 1983. "The consensus problem in unreliable distributed systems (a brief survey) ". Technical Report 273 (June), Department of Computer Science, Yale University.

[31] Fischer, M. J.; Lynch, N. A.; Paterson, M. S.; "Impossibility of distributed consensus with one faulty process" J. ACM 32, 2(Apr.), 1985. pp. 374-382.

[32] Guerraoui R. "Revisiting the relationship between non-blocking atomic commitment and consensus" Jean-Michel Hélary, Michel Raynal (Eds.): Distributed Algorithms, 9th International Workshop, WDAG '95, Le Mont-Saint-Michel, France, September 13-15, 1995, Proceedings. Lecture Notes in Computer Science, Vol. 972, Springer, 1995, ISBN 3-540-60274-7 pp. 87-100

[33] Guerraoui, R.; Schiper, A.; ed. Strohmeier, A. "Fault-tolerance by replication in distributed systems" Reliable Software Technologies - Ada-Europe '96. 1996 Ada-Europe International Conference on Reliable Software Technologies. Proceedings Springer-Verlag: Berlin, Germany, 1996. p.38-57, xi+511 pp., 35 refs. Conference: Montreaux, Switzerland, 10-14 June 1996 ISBN 3-540-61317-X Conference paper (English)

[34] Guerraoui R.; Oliveira, Schiper, A. "Stubborn communication channels." EPFL, Computer Science Department, Technical Report, December 1996 Paper

[35] Hadzilacos, V., and Toueg, S. "A modular approach to fault-tolerant broadcasts and related problems" Tech. Rep. 94-1425 (May), Computer Science Department, Cornell University, Ithaca, N.

[36] Landis, S.; Maffeis, S.; "Building reliable distributed systems with CORBA" Theory and Practice of Object Systems (1997) vol.3, no.1, p.31-43. 15 refs. CODEN: TPOSF3 ISSN 1074-3227 Journal paper (English)

[37] Landis S. and Maffeis S., "Building Reliable Distributed Systems with CORBA," Theory and Practice of Object Systems, New York: John Wiley, Apr. 1997.

[38] Landis, S.; Stento, R.; "CORBA with fault tolerance" Object Magazine (1995) vol.5, no.7, p.62-6. 0 refs. ISSN 1055-3614 Journal paper (English)

[39] Maffeis, S.: "Adding Group Communication and Fault-Tolerance to CORBA," in Proceddings of the Conference on Object-Oriented Technologies,(Monterey, CA), USENIX, June 1995.

[40] Maffeis, S. and Schmidt, D. C., "Constructing Reliable Distributed Communication Systems with CORBA, "IEEE Communications Magazine, vol. 14, February 1997.

[41] Maheshwari, P.; Ouyang, J.; ed Mensgen, D. "Supporting fault tolerance in heterogeneous distributed applications" Proceedings. Sixth Heterogeneous Computing Workshop (MCW '97) IEEE Comput. Soc. Press: Los Alamitos, CA, USA, 1997. p.195-207, ix+235 pp., 21 refs. Conference: Geneva, Switzerland, April 1997

[42] Malkhi, D., Reiter M. K. "Survivable Consensus Objects" The Seventeenth Symposium on Reliable Distributed Systems, October 20-22, 1998, West Lafayette, Indiana, USA, Proceedings. IEEE Computer Society, 1998, ISBN 0-8186-9218-9, online proceedings pp. 271-279

[43] Mong-Yi Tzeng; Kai-Yeung Siu; "Message-optimal protocols for fault-tolerant broadcast/multicast in distributed systems with crash failures" IEEE Transactions on Computers (1995) vol. 44, no.2, p. 346-52. 6 refs.

[44] Manimaran, G.; Murthy, C. S. R.; "A new study for fault-tolerant real-time dynamic scheduling algorithms" Proceedings. 3rd International Conference on High Performance Computing (96TB100074) IEEE Comput. Soc. Press: Los Alamitos, CA, USA, 1996. p.289-94, xvi+476 pp., 9 refs. Conference: Trivandrum, India, 19-22 Dec. 1996

[45] Muang, Y.; Kintala, C.; "Software fault tolerance: Technologies and experience" Digest of Papers FTCS-23 The Twenty-Third International Symposium on Fault-Tolerant Computing IEEE Comput. Soc. Press: Los Alamitos, CA, USA, 1993. p. 2-9, xxii+685 pp., 25 refs. Conference: Toulouse, France, 22-24 June 1993.

[46] Narasimhan, P.; Moser, L.E.; Melliar-Smith, P.M.; "Replica consistency of CORBA objects in partitionable distributed systems" Distributed Systems Engineering (1997) vol.4, no.3, p.139-50. 19 refs. CODEN: DSENEK ISSN 0967-1846 Journal paper (English)

[47] Narasimhan, P.; Moser, L. E. ; Melliar-Smith, P. M.; "Exploiting the Internet Inter-ORB Protocol Interface to provide CORBA with fault tolerance" Proceedings fo the Third USENIX Conference on Object-Oriented Technologies and Systems (COOTS) USENIX Assoc; Berkeley, CA, USA, 1997. p. 81-90, 248 pp., 18 refs. Conference: Portland, OR, USA, 16-20 June 1997

[48] Narasimhan, P.; Moser, L. E. ; Melliar-Smith, P. M.; "The interception approach to reliable distributed CORBA objects" Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems (COOTS) USENIX Assoc; Berkeley, CA, USA, 1997. p.245-8, 248 pp., 7 refs. Conference: Portland, OR, USA, 16-20 June 1997

[49] Nick, J.M.; King, G.M.; Jen-Yao Chung, N.S.; Ching-Shan Peng; "Parallel Sysplex: a scalable, highly available, high performance commercial system" Journal of Parallel and Distributed Computing (1997) vol. 43, no.2, p. 179-189. 15 refs.

[50] G.D.Parrington, S.K.Shrivastava,S.M.Wheater and M.C.Little, "The Design and Implementation of Arjuna", USENIX Computing Systems Journal, Vol 8,No 3,1995

[51] Oliveira, R.; Guerraoui R.; Schiper, A. "K-Stubborn Channels" EPFL, Computer Science Department, Technical Report, 1997 Paper

[52] Oliveira, R.; Guerraoui R.; Schiper, A. "Consensus in the Crash-Recover Model" EPFL, Computer Science Department, TR-97/239, 1997 Paper

[53] Prisco, De R., Malkhi, D., Reiter, M. K. "On K-set Consensus Problems in Asynchronous Systems" In Proceedings of the 18th ACM Symposium on Principles of Distributed Computing, pages 257-265, May 1999.

[54] Reischuk, R. 1985. "A New Solution for the Byzantine Generals problem" In Information and Control Vol. 64, Nos. 1-3, January/February/March 1985 Academic Press, New York and London.

[55] Ricciardi, A.; Birman, K. P.; "Using process groups to implement failure detection in asynchronous environments." Proceedings of the 10th ACM Symposium on Principles of Distributed Computing (Montreal, Que., Canada, Aug. 19-21). 1991. ACM, New York, pp. 341-354.

[56] Rooke, Chris, "UNIX Review", (Nov 1995) Vol.13, No.12,pp.35-42.ISSN:0742-3136

[57] Schmidt D. C. and Cleeland C., "Applying Patterns to Develop Extensible and Maintainable ORB Middleware," Communication of the ACM, Special Issue on Software Maintenance, Vol. 40, No. 12, December 1997.

[58] Schmidt, D. C. ; Levine, D. L. and Mungee S. "The Design and Performance of Real-Time Object Request Brokers," Computer Communications " volume 21, pages 294-324, April 1998

[59] Shokri, E.; Hecht, M.; Crane, P.; Dussdault, J.; Kim, K. M.; "An approach for adaptive fault-tolerance in object-oriented open distributed systems" Proceedings. Third International Workshop on Object-oriented Real-Time Dependable Systems (Cat. No. 97TB 100132) IEEE Comput. Soc: Los Alamitos, CA, USA, 1997. p.298-305, x+356 pp., 14 refs. Conference: Newport Beach, CA, USA, 5-7 Feb. 1997.

[60] The Orbix Journal - Q2 1998 IONA Technologies PLC.

[61] Trowbridge, Dave "Is high availability good enough for you? "Computer Technology Review, (Dec 1993) Vol. 13, No. 14, p.8,10