Ausarbeitung zum Vortrag Recovery-oriented Computing

im Rahmen des Seminars **Autonomic Computing**
(SoSe 2009 bei Dr.-Ing. Felix Salfner)

Frank Fuhlbrück Martin Schneider

26. Juli 2011

Im Rahmen des Seminars „Autonomic Computing“ im Sommersemester 2009 haben wir uns mit dem Thema Revorery-oriented Computing befasst. Mit dieser Seminararbeit möchten wir die Idee, die hinter Recovery-oriented Computing (ROC) steht, näher bringen und im ersten Teil in das Thema einführen. Im zweiten und dritten Teil werden wir zwei Ansätze vorstellen und näher erläutern: zum einen das Recursivly-Recoverbal Framework und zum anderen Crash-only Software.

Inhaltsverzeichnis

1 Einführung 3
- 1.1 Begriff 3
- 1.2 Motivation 3
- 1.3 Peres' Law 3
- 1.4 Techniken der ROC 4
- 1.5 Fallstudien der ROC-Techniken 5

2 Recursivly-Recoverbal Framework 5
- 2.1 Einführung 5
- 2.2 Fallbeispiel das Mercury-System 6
- 2.3 Monitoring des Systems und Fehlererkennung 7
- 2.4 Strategien zum Rebooten und Erholung von Fehlern 9
  - 2.4.1 Recover Agent (REC) 9
  - 2.4.2 Reboot Tree 9
- 2.5 Zusammenfassung 13

3 Crash-only-Software 15
- 3.1 Idee 15
- 3.2 Eigenschaften und Anforderungen 15
- 3.3 Effizienz ohne spezielle Techniken 17

4 Schlussbemerkung 18

5 Arbeitsaufteilung 18

6 Abbildungsverzeichnis 18

7 Literatur 19

1 Einführung

1.1 Begriff

Recovery-oriented Computing [Pat02] (ab hier kurz: ROC) bezeichnet ein Gestaltungsprinzip für Softwaresysteme, das auf leichte Wartbarkeit und Wiederherstellung hin optimiert ist. Dieses Prinzip wird mit der Unvermeidbarkeit von Fehlern begründet (s.u.) und bündelt (bereits bekannte) Gestaltungsrichtlinien.

1.2 Motivation

Candea et al. skizzieren in [CCF04, S. 214] die Kosten einer Stunde Downtime. Dabei beziffern sie die Kosten mit bis zu 6 000 000 $ je nach Branche. Beachtet man weiter, dass ein großer Teil der Fehler mittelbar auf Administratoren zurückgeht [Pat02, Abb. 2], so ergibt sich einerseits das Bedürfnis nach automatischen Lösungen und andererseits die Frage nach der Vermeidbarkeit von Fehlern.

1.3 Peres' Law

Als Antwort auf die oben gestellten Bedürfnisse dient den Verfechtern des ROC ein Ausspruch, der Shimon Peres zugeschrieben wird:

> If a problem has no solution, it may not be a problem, but a fact — not to be solved, but to be coped with over time.

Dementsprechend werden Fehler und Fehlverhalten von Programmen als Tatsache postuliert, deren Auswirkungen in der Entwicklung bedacht und dementsprechend revidiert werden müssen. An die Stelle des Verhinderns von Fehlern und insbesondere Abstürzen tritt deren Behandlung. Diesem Mantra wird dann Moore's Law gegenübergestellt, das, obwohl eigentlich als Vorhersage über Transistordichte formuliert, als Stellvertreter der reinen Leistungsorientierung fungiert. Moore's Law wird dabei nicht an sich angegriffen, sondern aus dem Fokus der Betrachtungen geschoben: „[...] we should leave further performance improvement to Moore's law [...]" [CCF04, S.214].

Die Begründung für dieses Vorgehen suchen Candea et al. [CCF04, S.215] u.a. in der Gleichung zur Berechnung der Verfügbarkeit A (engl. availability). Diese gibt den Anteil der Zeit an, in der das System wie vorgesehen funktioniert.

$$A = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}} \tag{1}$$

Dabei gibt MTTF die (arithmetisch) mittlere Zeit an, bis ein Fehler auftritt (engl. mean time to fail) und MTTR die, die es braucht um einen aufgetretenen Fehler zu beheben (mean time to repair). Ihre Summe MTTF + MTTR gibt damit die mittlere Zeit einer Fehlerperiode an und A den Anteil der davon verfügbaren Zeit.

Will man die Verfügbarkeit A erhöhen, so muss entweder der Zähler des Bruchs (also die mittlere Zeit bis zum Fehler) erhöht (der „klassische“ Ansatz) oder der Nenner verringert werden. Dies kann im Wesentlichen nur durch eine Verringerung der MTTR geschehen, was dem Ansatz des ROC entspricht.

1.4 Techniken der ROC

Um das Konzept ROC zu verwirklichen bedienen sich u.a. Patterson et al.[Pat02] sechs bekannter Techniken und deuten diese nach ihrer Funktion für ROC um. Diese Leittechniken („Six techniques that guide ROC“[Pat02, Abschnitt 3]) sind:

Redundanz, die das System im Fehlerfall am Laufen hält und zudem Zeit schafft, den Fehler zu beheben. Läuft das ganze System weiter, so kann eine Wiederherstellung einer replizierten Komponente bereits starten. Die Zeit dafür ist Teil der TTF, da ein kompletter Ausfall (noch) nicht aufgetreten ist.

Partitionierung nutzt in ähnlicher Weise. Teilfunktionalität kann evtl. noch gewährleistet werden, während eine Komponente repariert wird. Zudem sinkt die TTR, da der Aufwand für eine Reparatur eines Teilsystems u.U. erheblich geringer ist, als für die des gesamten Systems.

Fault insertion, also die Möglichkeit Fehlerszenarien gezielt oder zufällig auszulösen, hilft, die Reparaturmechanismen auf ihre Wirksamkeit zu testen und ihren Zeitverbrauch zu optimieren. Dies gilt auch für Reparaturformen, bei denen menschlicher Eingriff nötig ist.

Diagnosehilfen erleichtern den Administratoren das Entdecken eines Fehlers. Sofern sorgfältig implementiert und mit dem System aktualisiert dokumentieren sie den Fehler weit schneller, als ihn ein Administrator ohne sie gefunden hätte und reduzieren so die TTR.

Undo-Funktion ermöglichen den Administratoren, Lösungswege sicher zu testen, ohne neuen bzw. größeren Schaden anzurichten. Patterson et al. schlagen dazu Logs und einen Speicher vor, der neue Daten/Dateien nicht an die Stelle der alten schreibt, sodass diese eine gewissen Zeit lang wieder hergestellt werden können[Pat02].

Orthogonaler Aufbau der Komponenten trägt zur Isolierung der Fehler und Vereinfachung der Teilsysteme da. Dabei soll jede Komponente eine einfache und mit den restlichen Komponenten wenig verzahnte Funktion erfüllen. Andere Komponenten basieren auf diesen, indem sie die Funktion abstrahiert einsetzen. So schreibt z.B. eine Anwendung

ihre Daten in Tabellen einer Datenbank, diese die Tabellen in Dateien, welche wiederum vom Dateisystemtreiber organisiert werden. Quellcode, der für ein korruptes Dateisystem ursächlich ist, ist dann sicher nicht bei der Anwendung zu suchen.

1.5 Fallstudien der ROC-Techniken

Patterson et al. nennen ebenfalls einige Projekte, die aus ihrer Sicht die Nutzung der sechs Techniken im Sinne des ROC belegen und versuchen den Nutzen und den Grad der Umsetzung deutlich zu machen. Besonderes Augenmerk haben sie dabei auf das Mercury-System gelegt [Pat02, Abschnitt 6], da dieses ihrer Meinung nach die Techniken Redundanz, Partitionierung, Fault Insertion umsetzt und einen orthogonalen Aufbau aufweist. Weiteres zu Mercury ist im nächsten Abschnitt zu finden. Weiter wird u.a. auf Pinpoint eingegangen [Pat02, Abschnitt 5], ein automatisches Diagnosesystem zur Fehlererkennung in Java Enterprise Servern, das keine Modifikation der Anwendung benötigt. Entsprechend seines Konzeptes ermöglicht es Fault Insertion und bietet Diagnosehilfen, wobei es sich auch gerade bei replizierten Komponenten sinnvoll auswirkt (Unterscheidung der logisch identischen Komponenten). Dementsprechend lohnt es sich gerade dann, wenn die zu überwachende Anwendung mind. ein weiteres Konzept des ROC umsetzt(Redundanz/Partitionierung). Als Beispiel für Undo-Funktionen wird ein Mailserver mit solcher genannt [Pat02, Abschnitt 7], hier ließen sich aber sicher auch andere Beispiele anführen.

2 Recursivly-Recoverbal Framework

2.1 Einführung

In diesem Abschnitt möchten wir euch das Recursivly - Recoverbal Framework etwas genauer vorstellen. Das Framework hat sich zum Ziel gesetzt die Mean Time to Reocvery (MTTR) für das gesamte System zu reduzieren und somit die Verfügbarkeit des Systems zu steigern. Dies stellt eine Neuerung, im Versuch die Verfügbarkeit eines Systems zu maximieren, dar. Der klassische Ansatz versucht nur die Mean Time to Failure immer weiter zu erhöhen.
Der Begriff „Recursivly-Recoverbal“ erklärt sich daher, dass einzelne Programm Komponenten neugestartet werden um Fehler zu beheben. Die Rekursion kommt hinzu, wenn durch rebooten der Komponente der Fehler immer noch vorhanden ist. Dann werden an Hand von Informationsfluss, Abhängigkeiten und weiteren systemarchitektonischen Informationen weitere Komponenten ermittelt, die auch rebootet werden, bis der Fehler behoben wurde oder das gesamte System neu gestartet wurde.

In diesem Kapitel wollen wir als erstes das Mercury System vorstellen (2.2) was uns im

weiteren als Fallbeispiel für das Recursivly Recoverbal Framework dienen soll. Danach beleuchten wir den den Aufbau und die einzelnen Bestandteile das Frameworks genauer und ihre Funktionen (2.3 -2.4). Abschließend geben wir noch eine Zusammenfassung (2.5).

2.2 Fallbeispiel das Mercury-System

Das Mercury System ist eine Bodenstation zur Kommunikation mit Satelliten der Stanford Universität. Die Bodenstation ist ein Gemeinschaftsprojekt der Gruppe für Recover Orientated Computing (ROC) und dem Space Systems Development Lab (SSDL). Ihr Ziel ist es mit Commercial off the shell (COTS) Produkten eine Infrastruktur aufzubauen die kostengünstiger ist und verlässlicher. Der aktuelle Fokus liegt bei der Steigerung der Verfügbarkeit.
Die Funktion einer Bodenstation ist hierbei das Verfolgen des Satelliten mit einer Antenne, sobald er am Horizont auftaucht, um Telemetrie und Daten zu sammeln. Dies führt dann auch zu folgenden Komponenten im System, um gewünschte Funktionalität zu haben, Funkausrüstung, Flugbahnvorhersage und weitere Kontrollsoftware. Im der Abbildung 1 ist der Aufbau der Mercury Systems schematisch dargestellt. Im oberen Teil

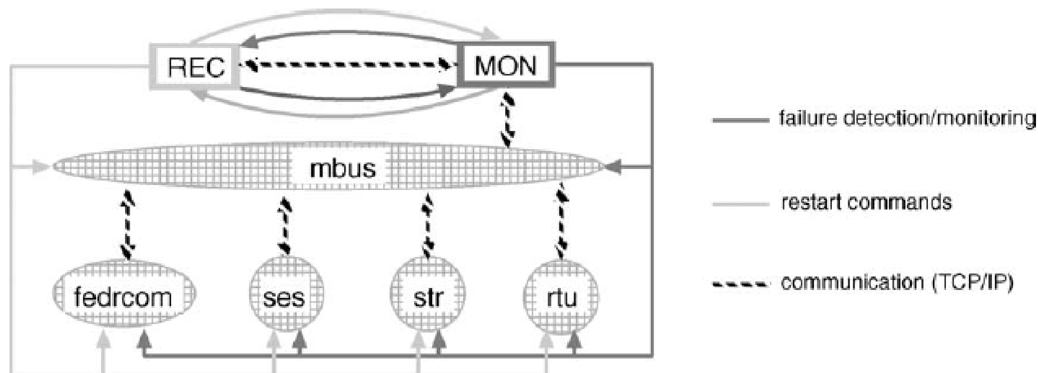

Abbildung 1: Mercury System

der Darstellung sind der Recover Agent (REC) und der Monitor (MON) zu sehen, welche im weiteren Verlauf des Textes noch genauer beleuchtet werden. Kommunikationskanal ist der mbus, dieser überträgt die XML-basierten High-Level-Kommandonachrichten zwischen den einzelnen Komponenten. Fedrcom ist eine bidirektionaler Proxy zwischen den XML-Kommandos und low-level Funkkommandos. Der Satellite estimator (ses) ist für die Berechnung der Satellitenposition, der Funkfrequenz und des Antennenwinkel zuständig. Satellite tracker (str) richtet die Antenne aus, so dass sie der Flugbahn des Satelliten folgt, während eines Überflugs. Als letzte Komponente ist noch der radio tuner (rtu) da,

seine Aufgabe ist es, die immer richtige Frequenz zu halten. Die einzelnen Komponente wurden in Java implementiert.

Ein Überflug, von einem Satelliten, dauert etwa 15 Minuten. Zwischen diesen kann die Bodenstation gewartet werden und die Software gepflegt oder geupdatet werden. Deshalb liegt das Augenmerk auf dem Betrieb beim Überflug eines Satelliten, wo jede Sekunde entscheidend ist. Dies minimiert das Risiko, Daten zu verlieren oder im Extremfall den Satelliten nicht mehr verfolgen zu können.

2.3 Monitoring des Systems und Fehlererkennung

Um Fehler zu beheben, muss man sie erst erkennen! Candea et. al. [CCF04] stellen drei Ebenen der überwachung vor. Jede einzelne mit ihren eigenen Vor- und Nachteilen. Sie sagen, dass keine Ebene für sich genug ist, um alle Arten von Fehlern zu detektieren. Als Schlussfolgerung kann man ziehen, dass im optimalen Fall also eine Kombination aus allen Ebenen für die Fehlererkennung zu wählen ist.

Die einzelnen Eben für die Fehlererkennung sind zum einen Low-Level-Monitoring, Application-specific Monitors und User-activity Monitors. Die Hauptunterscheidungsmerkmale sind ihre Kosten, Laufzeit im Betrieb, Wartbarkeit und Art der detektierten Fehler.
Plattform-Level-Monitoring zeichnet sich durch eine einfache Integration in Systeme aus. Zudem hat es auch sehr geringe Laufzeitkosten. Ein Nachteil ist, dass bei Erweiterung des Systems um neue Komponenten auch das Monitoring angepasst werden muss. Mit dieser Art der Überwachung kann man Maschinen- und Protokolltests machen.
Als zweite Ebene kommt der application-specific-monitor. Dieser kann High-Level-Fehler finden. Zum Beispiel lässt sich ein progress counter implementieren, dieser muss für jede Komponente individuell erstellt werden, welcher den Fortschritt für jede Komponente nach Außen hin sichtbar anzeigt. Dies gibt einen einen sehr guten Eindruck davon, wo der Fehler aufgetreten ist, aber nicht warum. Zudem wird auch deutlich, dass diese Art der Fehlererkennung sehr aufwendig in der Wartung und Anpassung an neue Komponenten ist.
Als letzte Ebene fehlt noch der user-activity-monitor. Mit ihm kann man testen ob das Verhalten nach Außen korrekt ist, zum Beispiel ob eine Webseite erreichbar ist. Dies ließe sich mit einem einfach Aufruf der Webseite in einem Browser überprüfen. Vorteil dieser Art von Monitoring ist die einfache Implementation.

Wie Eingangs schon erwähnt habe alle drei Ebenen der Fehlererkennung unterschiedliche Auswirkungen bei der Ausführung, auf die Laufzeit des Systems. So führt man die Tests zur Fehlererkennung in unterschiedlicher Häufigkeit aus, vom regelmäßigen plattform-level-monitoring über gelegentliches application-specific-monitor zu seltenem

user-activity-monitor.
Im Fallbeispiel des Mercury Systems wurde sich für eine Fehlererkennung auf dem application-level entschieden. Der ursprüngliche Zustand des Systems war so, dass alle Komponenten ohne Zeichen nach außen versagt haben, failed-silent, und man nur durch die ausbleibende Reaktion auf Nachrichten einen Fehler erkennen konnte.
Deshalb wurde entschieden, jede Komponente schickt eine liveness ping über den mbus an MON. So kann bei Ausbleiben des ping festgestellt werden welche Komponente einen Fehler hat und nicht mehr arbeitet.
Die Aufgabe des MON ist somit die Überwachung des liveness pings, der einzelnen Komponenten, und bei ausbleiben das benachrichtigen des REC, der alle weiteren Schritte veranlasst. Das Intervall in dem alle Komponenten ein liveness ping geben müssen, beträgt 1s. Nach Candea et. al. [CCF04, Abschnitt 3.3] hat sich das als guter Kompromiss zwischen Belastung des mbus und Zeitraum bis zur Fehlererkennung herausgestellt. Hier wird deutlich, dass für jedes System eine Stellschraube da ist, an der zwischen Performance und Zeit bis zur Fehlererkennung gewichtet werden kann, um eine optimale Lösung für das eigene System zu erreichen. Als nächstes wäre der REC dran, ihn werden wir im folgenden Kapitel näher erläutern.

Wichtig ist auch noch zu erwähnen, dass der MON den REC überwacht, falls dieser selber crashed. Falls dieser Fall eintritt weiß MON wie er REC rebooten kann. Um die Unabhängigkeit beider Komponenten (MON und REC) zu gewährleisten, kommunizieren beide über separate Kanäle mit einander. In dem betrachteten Fallbeispiel sind es TCP-Verbindungen. Auch der mbus wird als Komponente überwacht. Auch kann der umgekehrte Fall eintreten, dass in MON ein Fehler auftritt. Da MON von REC überwacht wird, wird dies festgestellt und REC kann die Schritte einleiten um MON zu rebooten.
Im Paper [CCF04, Abschnitt 3.3] wird noch ein Szenario aufgeworfen, für die keine Lösungsstrategie geboten wird, nämlich wenn MON und REC gleichzeitig Fehler werfen und sich nicht gegenseitig rebooten können.
Dies Stellt ein Punkt für weitere Forschung dar, um auch für dieses Szenario eine Lösungsstrategie zu entwickeln, welche die Verlässlichkeit und Verfügbarkeit des Gesamtsystems gewährleistet. Man muss bedenken, dass es zwei grundlegende Unterszenarien gibt. Zum ersten das nur MON und REC gecrashed sind und das restliche System unbeeinträchtigt weiter arbeitet. Als zweites Unterszenario ist das neben MON und REC eine oder mehrere weitere Komponenten auch einen Fehler geworfen haben.

Zusammenfassend ist zusagen, dass sich im Fall des Mercury-Systems für eine Fehlererkennung auf dem application-level entschieden wurde und unabhängig von Art der Fehler MON REC informiert um einen reboot der Komponente anzustoßen.

2.4 Strategien zum Rebooten und Erholung von Fehlern

2.4.1 Recover Agent (REC)

Der recover agent (REC) ist dafür zuständig, nach einer Benachrichtigung durch MON die richtigen Komponente(n) für den Reboot zu wählen. Der REC merkt sich welche Komponenten er rebootet hat, um verfolgen zu können, ob der Reboot auch den gewünschten Erfolg gehabt hat und der Fehler behoben wurde. Falls ja, kann mit dem regulären Betrieb weiter gemacht werden, falls aber ein Fortbestand des Fehlers festgestellt wird, muss der REC anhand von Datenfluss oder logischen Verknüpfung der Komponenten, die Nachbarn von der Komponente mit rebooten, welche beim ersten Versuch schon rebootet wurde.
zur weiteren Fehlerbehebung Es werden also rekursiv immer mehr Komponenten durch den REC rebootet bis der Fehler nicht mehr Auftritt oder des gesamte System rebootet wurde und der Fehler immer noch da ist. Dann muss ein Mensch (Administrator) zur weiteren Fehlerbehandlung benachrichtigt werden .
Wie genau die Auswahl der zu rebootenden Komponenten geschieht, das wollen wir im nächsten Abschnitt näher erläutern.

2.4.2 Reboot Tree

Der reboot tree ist eine Darstellung der Abhängigkeiten der einzelnen Komponenten des Systems unter einander. Er stellt eine Form der reboot policy da. Candea et. al. [CCF04, Kap 3.4] stellen zwei unterschiedliche Formen vor. Anfänglich interpretieren sie das System als einen gerichteten Graphen. Die Knoten repräsentieren die Komponenten und die Kanten stellen die fault-propagation-Richtung dar. Mit anderen Worten ausgedrückt, sind die existiert eine Kante zwischen zwei Komponenten wenn sie auch Daten austauschen oder mit einander Kommunizieren. So das über diese Beziehung ein Fehler sich fortsetzen kann. Diesen Graph nennen sie r-map, mit seiner Hilfe können wir nun die Komponenten ausmachen, die im nächsten Schritt mit rebootet werden müssen, falls der Fehler nach dem einmaligen rebooten einer Komponente fortbesteht.
Als zweite Variante, welche auch im Fall vom Mercury System benutzt wird, bauen sie den Baum so auf, dass die MTTR minimiert wird.
Die gewählt Form der reboot policy lässt sich nur anwenden, falls sich das System im laufenden Betrieb nicht verändert. Zu dem kommen noch weitere Rahmenbedingungen und Annahmen hinzu, die Candea et. al. an das System stellen, damit ihr Framework anwendbar ist.

1. Alle Fehler sind durch MON feststellbar und lassen sich durch einen reboot beheben (A_{cure})

2. Der Ausfall einer Komponente durch einen Fehler führt zur zeitweisen Unverfüg-

barkeit des gesamten Systems (A_{entire})

3. Der recovery agent wählt immer die minimal notwendigste Gruppe von Komponenten aus zum rebooten, so das der Fehler behoben wird. (A_{oracle})
4. Microreboot einer Gruppe von Komponente(n) führt nicht zu Fehlern in anderen Komponenten. ($A_{independent}$)

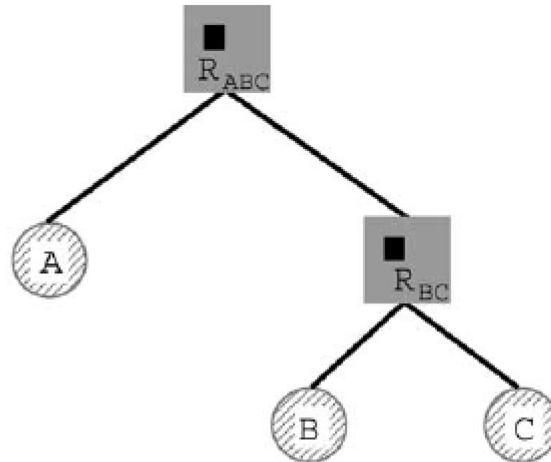

Abbildung 2: einfacher reboot tree

In der nebenstehenden Abbildung ist ein einfacher reboot tree zu sehen. Zur Erklärung: die Kreise(A, B und C) stellen die jeweiligen Komponenten des Systems dar, mit ihrer jeweiligen MTTR und MTTF. Die Quadrate (hier R_{ABC} und R_{BC}) symbolisieren die „reset buttons“welche von REC „gedrückt“ werden können. Das heißt alle Komponenten die unterhalb des Knoten sind werden rebootet.
Im Fall das R_{BC} „gedrückt“wird, werden die Komponenten B und C rebootet. Bei R_{ABC} werden A, B und C rebootet, also das gesamte System, was logisch ist, da die Wurzel des Baumes darstellt.

Nun stellt sich die Frage, wie man auf diesen Baum für das jeweilige System kommt. Die Antwort ist einfach, ein iteratives Verfahren, was aber sehr gute Kenntnisse über die System Architektur und den Datenfluss benötigt.
Vorab aber noch zum theoretischen Hintergrund. Für einen Teilbaum (wie R_{BC}) gilt :

- $MTTF_{R_{BC}} = min(MTTF_B, MTTF_C)$
- $MTTR_{R_{BC}} = max(MTTR_B, MTTR_C)$.

Verallgemeinert gilt dann:

- $MTTF_{R_{1-n}} = min(MTTF_1, ..., MTTF_n)$
- $MTTR_{R_{1-n}} = max(MTTR_1, ..., MTTR_n)$.

Wenn wir uns nun mit diesem Wissen den ursprünglichen reboot tree vom Mercury System anschauen, sehen wir das nur das gesamte System rebootet werden kann. Das

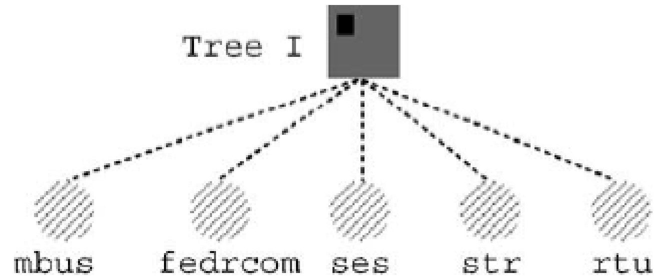

Abbildung 3: reboot tree I

hat zur Folge für das gesamte System gilt:

- $MTTF = min(MTTF_{mbus}, MTTF_{fedrcom}, MTTF_{ses}, MTTF_{str}, MTTF_{rtu})$
- $MTTR = max(MTTR_{mbus}, MTTR_{fedrcom}, MTTR_{ses}, MTTR_{str}, MTTR_{rtu})$.

Was bei stark unterschiedlichen MTTR zu einem unverhältnismäßigen Zeitverschwendung führt. Somit ist der erste Schritt zu ermöglichen jede Komponente einzeln zu rebooten. Candea et. al. sprechen hier von „simple depth augmentation“. Mit diesem Schritt

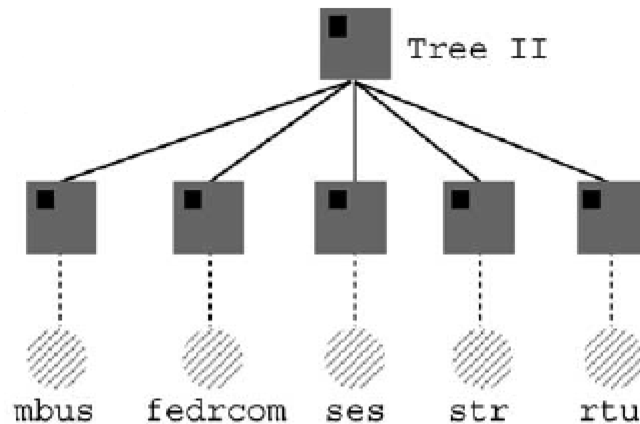

Abbildung 4: reboot tree II

ändern wir zwar nichts an der MTTF für das gesamte Mercury-System, aber nun haben wir die Möglichkeit jede Komponente einzeln zu rebooten. Dies ist besonders bei stark unterschiedlichen MTTR vorteilhaft. Was im Fall des Mercury-Systems zutrifft, wie die nachfolgende Tabelle zeigt. Die Werte sind die aus 100 Messungen entstanden. Die leichte Verbesserung in der MTTR erklärt sich daraus, das es einfacher ist eine Komponente zu rebooten als das gesamte System. Somit erhalten wir durch diese kleine Veränderung

Komponente	mbus	ses	str	rtu	fedrcom
$MTTR^{I}$	24,75	24,75	24,75	24,75	24,75
$MTTR^{II}$	5,73	9,5	9,76	5,59	20,93

Tabelle 1: Zeit bis zur Fehlererkunnung und reboot Zeit (in s)

schon eine Verbesserung um fast 4 Sekunden.

Für den nächsten Schritt benötigt man Systemkenntnisse, er heißt subtree depth augmentation. Hier sind die einzelnen Komponenten im Fokus der Betrachtung. ähnlich zur simple depth augmentation wird hier geschaut, ob Komponenten sich weiter unterteilen lassen und Subkomponenten mit stark unterschiedlicher MTTR existieren, um für beide einen neuen, separaten „reboot button“einzufügen.
In Mercury-System ist fedrcom so eine Komponente. Sie unterteilt sich zum einen in pbcom, bildet einen Serialenport auf einen TCP-Socket ab, und fedr, front-end driver-radio was sich mit pbcom über TCP verbindet. Wie angesprochen sind bei Subkomponenten nicht nur in ihr MTTR (pbcom: über 21 s; fedr: unter 6 s) stark verschieden sonder auch in ihrer MTTF ($MTTF_{fedr} << MTTF_{pbcom}$). In Worten ausgedrückt heißt das, dass man eher Fehler mit einem kurzen microreboot von fedr beheben kann als mit dem reboot von fedrcom. So gelangen wir zu: Der nächste Schritt stellt mehr oder weniger das Komplement zum subtree depth augmentation da. Consolidating dependent nodes, Ziel ist es Knoten von Komponenten zusammen zu fassen, wenn Fehler in einer Komponente A mit dem Fehler in Komponente B korrellieren.
Diese Beschreibung trifft auf die Komponenten ses und str zu. Ursache dafür ist, dass beide Komponenten beim Start sich miteinander synchronisieren und wenn eine von beiden rebootet wird, führt das unweigerlich zu einer Asynchronisierung beider Komponenten und zum reboot der zweiten Komponente. Deshalb ist es sinnvoll beide Komponenten zusammen zu legen, um auch nicht $A_{independent}$ zu verletzen.
Für die MTTR hat dies einen positiven Effekt, da wir von $MTTR_{ses} + MTTR_{str}$ zu $max(MTTR_{ses}, MTTR_{ses})$ gewechselt sind. In Zahlen ausgedrückt kommen wir von ursprünglich 9,50s und 9,76s, nun auf 6,25s und 6,11s. Der letzte Schritt der noch aussteht ist das promoting high-MTTR nodes. Dieser Schritt wird nötig, da Candea et. al. sich von der A_{oracle} Annahme verabschieden, sprich das der REC immer die kleinst mögli-

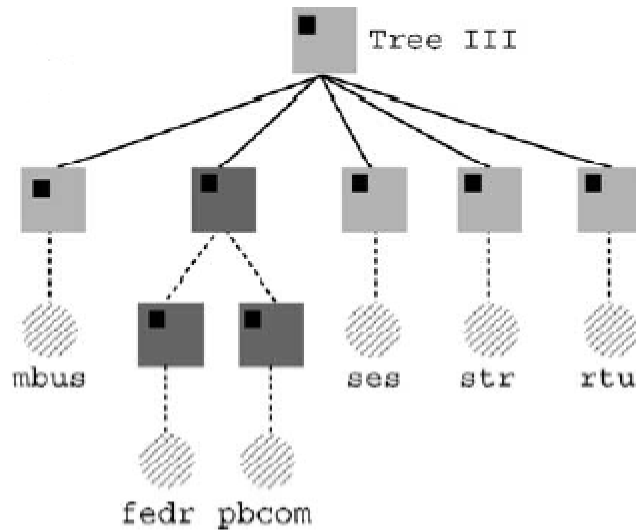

Abbildung 5: reboot tree 3

che Gruppe an Komponenten wählt, für den Reboot, so das der Fehler behoben wird [CCF04, Abschnitt 3.5.4]. Durch den nicht perfekten REC kann es passieren, dass entweder zu wenige Komponenten oder zu viele rebootet werden. Was im Falle von fedr und pbcom zu einer möglichen Erhöhung der MTTR führt, laut Candea et. al. stieg die MTTR mit einen nicht perfekten REC und dem reboot tree IV auf über 29 s an.
Daher schlagen sie vor das man pbcom eine Ebene höher schiebt so das man folgenden Baum erhält: Abschließend hier noch die MTTR der einzelnen Komponenten für die 5 reboot trees, um die Wirkung der Schritte aufzuzeigen:

Baum	REC	mbus	ses	str	rtu	fedr	pbcon	fedrcom
I	perfekt	24,75	24,75	24,75	24,75	-	-	24,75
II	perfekt	5,73	9,5	9,76	5,59	-	-	20,93
III	perfekt	5,73	9,5	9,76	5,59	5,76	21,24	-
IV	perfekt	5,73	6,25	6,11	5,59	5,76	21,24	-
IV	fehlerhaft	5,73	6,25	6,11	5,59	5,76	29,19	-
V	fehlerhaft	5,73	6,25	6,11	5,59	5,76	21,63	-

2.5 Zusammenfassung

Wie wir in dem Kapitel gesehen haben, kann man mit ein paar Annahmen wie A_{cure}, A_{entire} und $A_{independent}$ an das System, mit Hilfe des vorgestellten Frameworks und mit wenig Aufwand die MTTR des Systems verbessern und somit die Verfügbarkeit steigern.

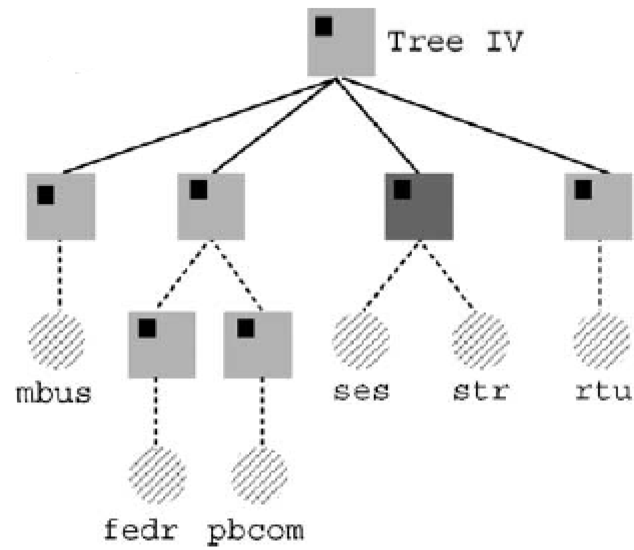

Abbildung 6: reboot tree IV

Das Hauptaugenmerk lag dabei auf der reboot policy was im Fallbeispiel der reboot tree ist. Dieser wurde über mehrere Schritte zu seiner endgültigen Form entwickelt, mit vier verschieden Schritten.

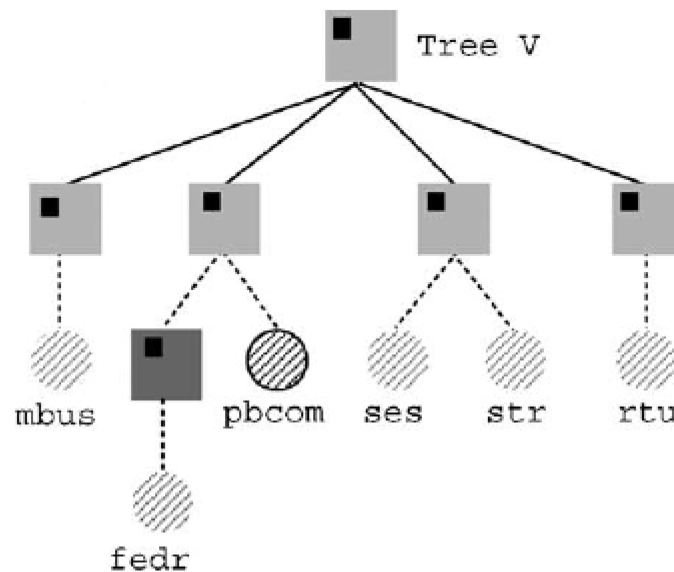

Abbildung 7: reboot tree V

3 Crash-only-Software

3.1 Idee

Im Allgemeinen wird ein Crash, also das unplanmäßige und unkontrollierte Beenden von Software, als Ausnahme- oder gar Katastrophenfall gesehen. Dementsprechend werden Mechanismen entwickelt, um das System am Laufen zu halten (z.B. Ausnahmebehandlung) oder wenigstens **kontolliert zu Beenden**. Die Wiederherstellung beim Start bleibt also auch beim Auftreten kleinerer Fehler der Sonderfall und wird entsprechend selten eingesetzt und optimiert.

Crash-only-Software verlagert die **Problembehandlung in die Startphase**. Dem System wird die Möglichkeit eines kontrollierten Beendens weitgehend genommen, sodass z.B. auch im Wartungsfall die Wiederherstellung aktiv wird: „There is only one way to stop such software— by crashing it—and only one way to bring it up—by initiating recovery“ [CCF04, S. 233]. Damit das funktioniert muss beim Aufbau der Komponenten umgedacht werden, was auch das eigentliche Ziel von Crash-only Software zu sein scheint.

3.2 Eigenschaften und Anforderungen

Dieses Umdenken manifestiert sich in den Anforderungen an Crash-only-Software und den durch sie gewonnenen Eigenschaften. Ein Vorteil ist das simple Fehlermodell (z.B. lediglich Fehler ja/nein): „[. . .] components only need to know how to recover from one

type of failure“ [CCF04, S. 235]. Wenn das System selbst auf den kompliziertesten auftretenden Fehler korrekt reagiert (z.B. plötzlicher Stromausfall mitten in einer Serie von Operationen), dann kann auch ein simplerer Fehler so behandelt werden. Zudem kann ein nachträgliches Feststellen der exakten Fehlerursache nicht garantiert werden, sodass auch ein Zwang zu solch simplen Fehlermodellen besteht.

Ein unkontrolliertes Abbrechen verlangt, dass der aktuelle Zustand nicht an beliebigen Stellen im System gespeichert ist, es also mit nur wenigen Ausnahmen aus **zustandslosen** Komponenten besteht (Anwendungen nur als Programme und Datenbanken als spezialisierte Speicher[?CCF03, 235]). Weiterhin wird der Einsatz von **abstrahierten Datenstrukturen** gefördert. Liest die Anwendung z.B. einen Record aus einer Datenbank, existiert dieser bei der Wiederherstellung oder er existiert nicht. Um eventuell in einem Cache befindliche Daten, die nur z.T. auf den Festspeicher geschrieben wurden kann und muss sich die Anwendung nicht kümmern.

Ist Crash-only-Software aus mehreren Komponenten aufgebaut, so müssen diese **voneinander abgeschirmt** sein. Das kann z.B. bedeuten, dass Ressourcen wenn überhaupt, dann nur im gemeinsamen nicht flüchtigen Speicher (s.o.) geteilt werden. Jeglicher Zusammenhang der Komponenten bedingte wieder einen Zustand des Gesamtsystems. Bsp.: Teilen sich zwei Komponenten einen Festspeicherbereich, muss festgehalten werden, wer aktuell schreiben darf. Diese Information muss mit wiederhergestellt werden und verkompliziert so den Prozess. Selbstverständlich kann das Gesamtsystem nicht crash-only sein, wenn es die einzelnen Komponenten nicht auch sind.

Eine weitere Anforderung trägt dazu bei, die wiederherzustellende Information zu minimieren: **Timeouts für Ressourcen**[CCF04, S.237]. Statt auf den Halter einer permanent allozierten Ressource zu warten, wird die Ressource in kurzen Zeitfenstern neu zugeteilt. Beim Wiederherstellen muss so keine spezielle Komponente (i.A. die letzte) den Zugriff erhalten, sondern dieser kann ggf. zufällig zugeteilt werden. Stürzt nur die haltende Komponente ab, so verfällt ihr Anspruch und nur sie allein muss neu gestartet werden.

Der Umgang mit Nachrichten ist sehr ähnlich. Wird ein Verfallszeitstempel (Time-to-live, TTL) und ein Marker für die Idempotenz der Nachricht (idempotent: mehrmaliges Senden ruft selben Zustand beim Empfänger hervor wie einmaliges Senden) mitgesendet, kann entschieden werden, ob

- die Nachricht noch aktuell ist (TTL) und
- es sicher ist, sie erneut zu senden (idempotent).

Diese Anforderung wird durch die **Restart-Retry-Architekur**(nach [CCF04, Abschnitte 4.3 **und 4.2**]) erfüllt:

Schritte im Fehlerfall:

1. Komponente `A` stellt Anfrage M an `B`.

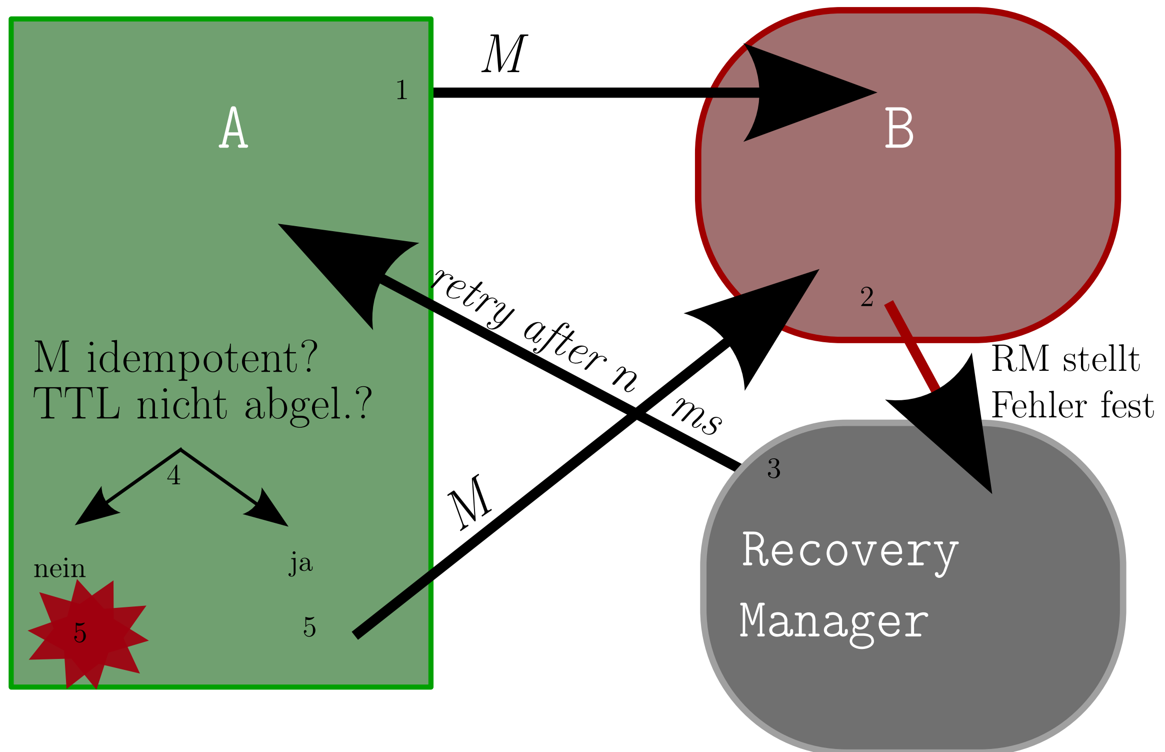

Abbildung 8: Ablauf einer Restart-Retry-Architektur

2. Recovery-Manager stellt Ausfall fest
3. Recovery-Manager sendet *Retry after n ms* an `A`.
4. Ist M idempotent und TTL(M) nicht abgel.?
5. ja ⇒ `A` fragt nach n ms erneut.
   nein ⇒ `A` bricht ab und sendet Fehler weiter.

Der Recovery-Manager ist eine spezielle Komponente im Retry-Restart-System, deren einziger Zweck es ist, den Ausfall von Komponenten festzustellen, diese neu zu starten und abhängige Komponenten zu benachrichtigen.

3.3 Effizienz ohne spezielle Techniken

Die Machbarkeit von ROC-Techniken bzw. des Crash-Only-Ansatzes wird deutlich, wenn bei aktuellen Systemen ohne durch ROC beeinflusstes Design die Zeiten für einen sauberen Neustart (Herunterfahren und Neustarten) und einen Absturz vergleicht. Da die Zeiten für die Reparatur recht unterschiedlich ausfallen können, wurden die Ergebnisse mehrerer Tests gemittelt (Daten aus [CF03, S.67]):

System	sauberer Neustart	Absturz & Neustart
RedHat 8 (mit ext3fs)	104s	75s
JBoss 3.0 application server	47s	39s
Windows XP	61s	48s

Bei allen Systemen dauert ein sauberer Neustart länger (bei RedHat gar mehr als $\frac{1}{3}$). Dabei darf allerdings nicht vergessen werden, dass diese nicht auf ROC optimierten Systeme nicht alle möglichen Fehler beim Start auch abklären bzw. beheben (Partitionen werden z.B. je nach Einstellung auch mit leichten Fehlern gemountet (Freispeichersumme, Inodezahlen für Directories und Files)[mount, Option error für ext2/3]). Zudem wurde in [CF03] keine Aussage gemacht, ob sich dabei das Ausfallrisiko erhöhte oder nicht.

4 Schlussbemerkung

Recovery-oriented Computing ist keine Revolution auf Grund seiner Techniken, wohl aber ein Betrachtungswechsel. Den eingesetzten Techniken wird teilweise eine neue Begründung geliefert und sie werden als Gesamtkonzept gebündelt. Weiterhin wird der seit langem bekannten Erkenntnis Rechnung getragen, dass Wartung von Softwaresystemen der größte Kosten- und Zeitfaktor ist. Viele kaum zu verhindernde u.a. hardwarebedingte Probleme werden durch ROC besser mit aufgegriffen, als dies durch die Idee von perfekter, fehlerfreier Software möglich wäre. Das darf jedoch nicht dazu führen, dass zur Verringerung von Entwicklungskosten vermeidbare Fehler mit der Begründung im System verbleiben, sie seien durch ROC-Konzepte eh unschädlich.

5 Arbeitsaufteilung

Kapitel	**Bearbeitung durch**
Einleitung	Frank Fuhlbrück/ Martin Schneider
Einführung	Frank Fuhlbrück
Recursivly-Recoverbal Framework	Martin Schneider
Crash-only Software	Frank Fuhlbrück
Schlussbemerkung	Frank Fuhlbrück/ Martin Schneider

6 Abbildungsverzeichnis

- Abbildungen 1-7: aus [CCF04, S.222,224f,227,229f]
- Abbildung 8: Frank Fuhlbrück

7 Literatur

[Pat02] D.A. Patterson et al., *Recovery-oriented Computing: Motivation, Definition, Techniques and Case Studies*, Berkeley Computer Science (2002).

[CKKF06] G. Candea, E. Kiciman, S. Kawamoto, and A. Fox, *Autonomous recovery in componentized Internet applications*, Cluster Computing **9** (2006), 175–190.

[CCF04] G Candea, J Cutler, and A Fox, *Improving Availability with Recursive Microreboots: A Soft-State System Case Study*, Performance Evaluation (2004), 213–248.

[CF03] G Candea and A Fox, *Crash-Only Software*, Proceedings of the 9th Workshop on Hot Topics in Operating Systems **(HotOS-IX)** (2003).

[mount] *Manpage zu mount*, http://linux.die.net/man/8/mount (2011).