Recovery-oriented computing (ROC)

Benjamin Mock

b_mock@informatik.uni-kl.de

Abstract. Seit der Erkenntnis, dass Fehler in Soft- und Hardware sowie Administrierungsfehler unvermeidlich sind, wird nach Wegen gesucht, wie man dennoch die Verfügbarkeit von Systemen weiter verbessern kann. Das Recovery-oriented Computing (ROC) ist ein solcher Weg. In dieser Ausarbeitung wird die Motivation hinter ROC genauer erläutert, ein Vergleich mit existierenden Recovery-Techniken in Datenbankmanagementsystemen vorgenommen und die ROC-Methoden der rekursiven Neustarts, dem Undo auf der Systemebene sowie ein Prototyp für ROC auf der Hardware-Ebene erläutert.

1 Einleitung

Für Fehler in Systemen kann es viele unterschiedliche Gründe geben: Produktionszyklen werden immer kürzer, Systeme werden aus unterschiedlichen Hard- und Software-Komponenten gebaut, deren Zusammenspiel zu Fehlern führen kann, Überlastsituationen überfordern das System oder Administratoren treffen falsche Entscheidungen. Auf Grund des steigenden Preisdrucks wird auf lange Testphasen verzichtet. Der eigentliche Test erfolgt somit erst im laufenden Betrieb. Auf ein Trainingssystem für Administratoren und Benutzer wird aus Kosten- oder Machbarkeitsgründen verzichtet. Somit ist es auch nicht verwunderlich, dass Umfragen zufolge menschliche Bedienfehler die häufigste Ursache für Fehler sind – weit vor Hardware, Software und Überlast. Fast die Hälfte der in diesen Umfragen analysierten Fehler geht auf das Konto von Administratoren. [BBCC+02]

Ein Trend, das autonomic computing, geht in die Richtung, menschliches Einschreiten in Computersysteme zu verringern oder sogar ganz zu eliminieren. Solche Systeme beheben leichte Fehler selbst. Nur zur Korrektur schwerwiegender Fehler werden menschliche Operatoren benötigt. Die so fehlende Übung und Erfahrung sowie zusätzlicher Stress in Fehlersituationen machen es für den Administrator deshalb schwer, diese Fehler zu korrigieren. ROC sieht deshalb vor, den Administrator mit einzubinden. Er soll nicht durch ein Kontrollsystem ersetzt, sondern durch geeignete Werkzeuge unterstützt werden. Ein solches Werkzeug kann beispielsweise ein Trainingssystem sein, das im laufenden Betrieb Fehler simuliert.

Die Ziele des ROC sind die Erhöhung der Verfügbarkeit bzw. der Erreichbarkeit eines Systems und eine Verringerung der total costs of ownership (TCO), die neben den Kosten für den Einkauf des Systems auch die Kosten für den laufenden Betrieb

mit in Betracht ziehen. Die Verfolgung beider Ziele geht Hand in Hand. Durch eine erhöhte Verfügbarkeit fallen weniger Kosten durch Systemausfälle an.

Die Verfügbarkeit ist definiert als der Quotient der Zeitspanne bis zum ersten Fehler (MTTF = mean time to failure) und der Zeitspanne bis zum nächsten Fehler (MTBF = mean time between failures), einschließlich der Zeit, die zum Wiederherstellen des Systems gebraucht wird (MTTR = mean time to repair). Abbildung 1 verdeutlicht den Zusammenhang zwischen MTBF, MTTR und MTTF. Folgende Formel zeigt die Berechnung der Verfügbarkeit eines Systems.

```
Verfügbarkeit = MTTF/MTBF = MTTF / (MTTF + MTTR)        (1)
```

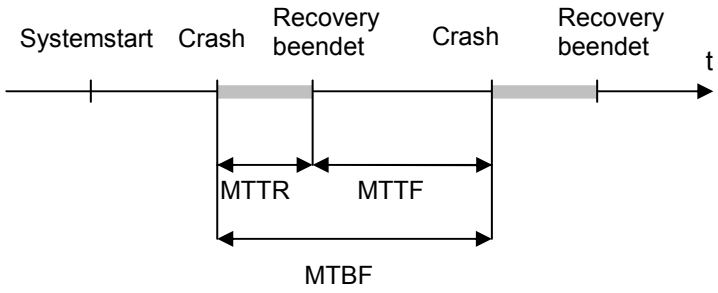

Abb. 1. Schema eines Systemablaufs mit Absturz. *MTTR* ist die Zeit, die zur Reparatur nach einem Absturz benötigt wird. *MTTF* ist die Zeitspanne bis ein Fehler auftritt. Aus diesen beiden Zeitspannen ergibt sich *MTBF*, die Zeitspannen zwischen zwei Fehlern

Der Ansatz von ROC geht in die Richtung, die Zeit für die Wiederherstellung zu verringern. Verringert man die MTTR mit einem bestimmten Faktor hat das dieselben Auswirkungen auf die Verfügbarkeit wie eine Verlängerung der MTTF mit dem gleichen Faktor.

Bisher wurde hauptsächlich Forschung und Entwicklung zur Verbesserung der MTTF betrieben. Wahrscheinlich liegt das auch an der Werbewirksamkeit, die sich für Hersteller daraus ergibt. Allerdings ist die Messung der MTTF sehr schwierig. Heutige Festplatten haben schon eine MTTF von etwa 80 bis 100 Jahren. Jedoch stellt auch dieser Wert nur einen mathematischen Mittelwert dar, der nicht besagt, dass ein Fehler nicht sofort auftreten kann, sondern nur, wie wahrscheinlich er auftritt.

Schon allein aus dem Grund, dass bisher weniger zum Thema MTTR geforscht wurde, ergibt sich ein höheres Potential auf diesem Gebiet. Ein weiterer Vorteil einer geringen MTTR ist, dass unter Umständen ein Fehler für den Benutzer völlig unsichtbar gemacht werden kann. [FoPa05] Ist das System schnell genug wiederhergestellt, merkt der Benutzer nicht, dass überhaupt ein Fehler aufgetreten ist.

Einsatzgebiete für ROC-Techniken sind vor allem Gebiete, in denen eine hohe Verfügbarkeit sowie eine hohe Datensicherheit gefordert werden. Nicht verwundernswert ist es deshalb, dass ROC schon länger in Datenbanksystemen eingesetzt wird. Auch

zeitkritische Systeme, bei denen es auf die genaue Einhaltung von Zeitschranken ankommt, sind für den Einsatz ROC prädestiniert.

Bei jedem Einsatz von ROC muss jedoch eine Trade-off-Entscheidung zwischen Durchsatz und Verfügbarkeit getroffen werden, da durch jede ROC-Technik ein gewisser Overhead entsteht. Doch dieser Overhead wird im Allgemeinen durch viel kürzere Wiederherstellungszeiten amortisiert.

1.1 ROC-Techniken

Die bekannteste und am einfachsten einzusetzende ROC-Technik ist die *Redundanz*. Man baut in ein System zusätzliche Festplatten, Lüfter oder Netzwerkkarten ein, die im Fall eines Ausfalls die eigentliche Arbeit übernehmen. Ein Beispiel dafür ist ein Festplattenverbund mit RAID. Ein anderes Beispiel sind Cluster-Systeme, bei denen jeder Knoten eine eigene Aufgabe übernimmt. Kommt es hier zu einem Ausfall, kann die Aufgabe des vom Ausfall betroffenen Knotens, von den restlichen Knoten mit übernommen werden. Jedoch kommt es durch die Homogenität des Clusters bei gezielten Angriffen zu dem Problem, dass alle Knoten des Clusters dieselben Schwachstellen haben. Fällt also ein Knoten auf Grund eines Angriffs aus, kann es möglich sein, dass durch die Aufgabenverteilung auch die anderen Knoten betroffen sind.

Ein weiteres Beispiel ist die gegenseitige Überwachung von Komponenten wie bei der ROC-1 Hardware, die in Abschnitt 3.3 noch genauer vorgestellt wird. Hier wird ein Netz aus sich gegenseitig überwachenden Knoten aufgebaut, wobei dieses durchaus heterogen sein kann, also aus verschiedenen Knoten bestehen kann.

Eine weitere ROC-Technik ist die *Modularisierung*. Als Beispiel ist hier die Virtual Machine von Java zu nennen, die ein komplettes System simuliert. Dadurch kann das eigentliche System in unabhängige und eigenständige Komponenten aufgeteilt werden. Somit ist es möglich im Fehlerfall nur die tatsächlich vom Fehler betroffenen Komponenten neu zu starten. Diese Technik wird in Abschnitt 3.1 unter dem Namen recursive Restartability noch einmal genauer durchleuchtet.

1.2 Fehler

Mögliche Fehler teilt man in verschiedene Klassen auf. Zur ersten Klasse gehören *Transaktionsfehler*, die allerdings nur Datenbankmanagementsystemen (DBMS) zugeordnet werden. Ausgelöst werden sie beispielsweise durch Fehler im Anwendungsprogramm oder auf Grund eines Deadlocks. Auch eine durch den Benutzer oder das DBMS abgebrochene Transaktion zählt zu dieser Klasse. Im Normalbetrieb eines DBMS können diese Fehler häufig auftreten. Sie werden durch Zurücksetzen von unvollendeten Transaktionen und dazugehörigen Änderungen behoben.

Alle weiteren Fehlerklassen sind allgemeingültig und können sowohl in einem DBMS als auch in jedem beliebigen anderen System auftreten.

Systemfehler bilden die zweite Fehlerklasse. Die Gründe für solche Fehler sind vielfältig. Sie reichen vom Stromausfall über menschliche Fehlbedienung sowie transienter Hardwarefehler bis hin zu Fehlern in der Systemsoftware. Dort unterscheidet

man noch einmal zwischen reproduzierbaren und nicht reproduzierbaren Fehlern. Die so genannten Heisenbugs sind nicht reproduzierbar und somit nur schwer zu finden. Sie basieren auf dem Zusammenspiel verschiedenster Komponenten und treten vor allem in Überlastsituationen sowie im Mehrbenutzerbetrieb auf. Die so genannten Bohrbugs sind reproduzierbare Fehler. Daher sind sie in moderner Systemsoftware selten, da sie leichter zu eliminieren sind. Die Recovery von Systemfehlern nennt man Crash-Recovery. Bei einem DBMS besteht sie aus dem Wiederholen der abgeschlossenen Transaktionen (Redo) sowie aus einem Zurücksetzen unvollendeter Transaktionen (Undo). Man geht bei dieser Fehlerklasse zwar von einem Verlust der Daten im Hauptspeicher aus, aber nicht von fehlerhaften Daten im Festspeicher.

Plattenfehler stellen die dritte Fehlerklasse dar. Fehler dieser Art haben einen Datenverlust im Festspeicher zur Folge. Durch das so genannte Medien- bzw. Archiv-Recovery können diese behoben werden. Zu den Maßnahmen gehören Spiegelplatten, die eine exakte Kopie der Originalplatte darstellen oder in einem DBMS das vollständige Wiederholen aller Änderungen abgeschlossener Transaktionen auf einer älteren Archivkopie.

Katastrophen wie etwa Erdbeben, Überschwemmungen oder Feuer bilden die vierte Fehlerklasse. Das Katastrophen-Recovery sieht vor, dass alle Änderungen erfolgreich abgeschlossener Transaktionen in einer zweiten, weit genug entfernten Datenbank erfasst werden, die im Katastrophenfall nicht betroffen ist. Hier stellt sich jedoch die Frage, was weit genug entfernt bedeutet, sind doch etwa bei einem Erdbeben sehr große Gebiete betroffen. Oftmals ist die letzte Möglichkeit der Katastrophen-Recovery daher das Reparieren oder neu Aufbauen des ursprünglichen Systems, was jedoch einen Datenverlust zur Folge hat.

1.3 Überblick

Nachdem in Abschnitt 1 bereits auf die verschiedenen Fehlerklassen und ROC-Techniken eingegangen wurde wird in Abschnitt 2 die Anwendung dieser ROC-Techniken in einem DBMS eingegangen.

Abschnitt 3 zeigt dann einige ROC-Baugruppen, die in allgemeinen Systemen eingebaut werden können. Dazu gehören rekursive Neustarts, das Undo auf Systemebene sowie eine Prototyp-Hardware mit dem Namen ROC-1.

In Abschnitt 4 werden dann einige Beispiele vorgestellt in denen diese ROC-Baugruppen bereits in der Praxis verwendet werden. Das prominenteste Beispiel dürfte wohl das Mercury-Satellitenprojekt sein, in dem rekursive Neustarts zum Einsatz kommen.

2 ROC in Datenbankmanagementsystemen

Im Bereich der Datenbankmanagementsysteme (DBMS) hat man früh erkannt, dass nicht nur versucht werden sollte Fehler zu verhindern, sondern auch nach einem Auftritt eines Fehlers das System wieder in einen definierten und Zustand zu bringen, der dem Zustand vor dem Fehler möglichst entsprechen sollte. Daher wurde in diesem

Bereich schon früh der Gedanke des ROC umgesetzt. Einige ROC-Baugruppen bauen auf den Umsetzungen in DBMS auf. Daher werden diese hier kurz vorgestellt.

Transaktionen. Das Transaktionsparadigma verlangt von einem DBMS ACID-Eigenschaften. Atomarität und Persistenz sind dabei für die Wiederherstellung (Recovery) besonders interessant. Da man im Allgemeinen davon ausgeht, dass Fehler nicht zu verhindern sind und zwangsläufig jeder mögliche Fehler irgendwann auftritt ("what can go wrong will go wrong" Murphy's law), muss man geeignete Maßnahmen treffen um das Transaktionsparadigma dennoch zu garantieren. Das setzt voraus, dass während des Normalbetriebs genügend redundante Informationen gespeichert werden, sodass im Fehlerfall durch geeignete Recovery-Maßnahmen der jüngste transaktionskonsistente Zustand wieder hergestellt werden kann.

Logging. Abbildung 2 zeigt ein Schema einer Speicherhierarchie eines DBMS. Um im Fehlerfall Daten wiederherstellen zu können und das Transaktionsparadigma zu erfüllen müssen alle Änderungen protokolliert werden. Da nicht für jede Änderung ein Plattenzugriff erfolgen sollte wird ein Log-Puffer im Hauptspeicher gehalten. Dieser wird spätestens unmittelbar vor dem commit der jeweiligen Transaktion in die temporäre Log-Datei und in das Log-Archiv ausgeschrieben. Die Log-Datei wird bei Transaktions- und Systemfehlern in Verbindung mit der permanenten Datenbank für eine Wiederherstellung verwendet. Kommt es zu einem Plattenfehler, werden die Archiv-Kopie der DB sowie das Archiv-Log verwendet. Das Archiv-Log darf folglich nicht auf derselben Platte wie die temporäre Log-Datei angelegt werden.

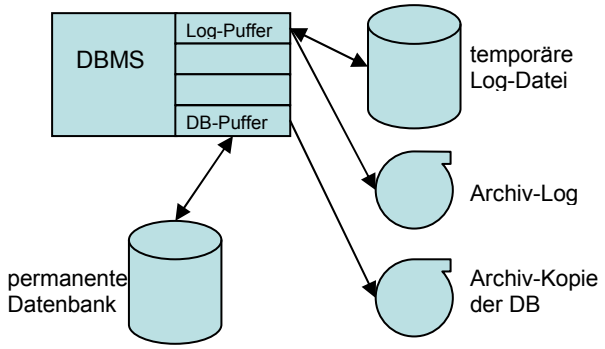

Abb. 2. Speicherhierarchie eines DBMS aus [Härd04]

Nach einem Systemfehler, der einen Hauptspeicherverlust und somit einen Verlust der DB-Puffer zur Folge hat, muss der Wiederherstellungsprozess gestartet werden. Dazu werden die permanente Datenbank und die temporäre Log-Datei verwendet. Der Wiederherstellungsprozess läuft in drei Phasen ab. Die erste Phase, die Analysephase, durchläuft die gesamte temporäre Log-Datei und ermittelt die so genannte Winner- und Loser-Menge von Transaktionen. Zur Winner-Menge gehören beendete Transaktionen, die zum Absturzzeitpunkt noch nicht in die permanente Datenbank eingebracht

wurden. Transaktionen, die zum Zeitpunkt des Fehlers noch kein commit ausgeführt hatten, werden als Loser bezeichnet.

In Phase zwei wird die Historie wiederholt. Alle Änderungen sowohl der Winner als auch der Loser müssen in der Reihenfolge ihrer LSN wiederholt werden.

Phase drei macht die Änderungen der Loser-Transaktionen in umgekehrter Reihenfolge ihrer eigentlichen Ausführung rückgängig. Durch diese Undo- und Redo-Operationen auf der Basis der permanenten Datenbank wird der jüngste transaktionskonsistente Zustand wiederhergestellt.

Man unterscheidet generell zwei Arten von Logging: physisches und logisches. Beim physischen Logging wird entweder der gesamte Inhalt eines Datensatzes oder zumindest die Differenz zu einem früheren Zustand. Das logische Logging protokolliert Änderungen in der Datenmanipulationssprache (DML). Zusätzlich existiert noch eine Kombination aus diesen beiden Protokollierungsverfahren, das physiologische Logging. Dabei bezieht sich jeder Log-Satz auf genau eine Seite. Innerhalb dieser Seite werden die elementaren Operationen protokolliert, also logisches Logging benutzt.

Sicherungspunkte. Nach einem Neustart des DBMS auf Grund eines Fehlers muss genau genommen die komplette Log-Datei seit Beginn des DBMS gelesen und analysiert werden. Für die eigentliche Redo- und Undo-Arbeit ist aber zumeist nur der neuere Teil der Log-Datei interessant. Deshalb existiert die Technik des Aufstellens von Sicherungspunkten oder Checkpoints. Im Normalbetrieb werden diese Sicherungspunkte periodisch erzeugt und zeigen somit der Recovery-Prozedur einen Startpunkt an. Zwar wird durch das Aufstellen von Sicherungspunkten ein zusätzlicher Overhead erzeugt, jedoch ist die Ersparnis im Fehlerfall so groß, dass dieser sich rechnet. Dieses Vorgehen spiegelt genau den Sinn und die Denkweise des recovery oriented computings wieder. Es existieren für diesen Zweck transaktionskonsistente, aktionskonsistente und unscharfe Sicherungspunkte. Ein transaktionskonsistenter Sicherungspunkt bezieht sich immer auf alle Transaktionen. Es wird mit dem Ausschreiben des Sicherungspunktes bis zum Ende aller aktiven Transaktionen gewartet. Weitere Transaktionen müssen bis zum Ende des Sicherungspunktes warten. Ein Wiederherstellungsprozess beginnt beim Ende des letzten Sicherungspunktes.

Für aktionskonsistente Sicherungspunkte werden periodisch alle Transaktionen blockiert und keine weiteren Änderungen zugelassen. Alle bis dahin geänderten Seiten werden in die Datenbank geschrieben. Somit ist zwar die Qualität der Sicherungspunkte nicht so hoch wie im transaktionskonsistenten Fall, die Totzeit des Systems wird jedoch deutlich verringert. Die Qualität der unscharfen Sicherungspunkte ist noch geringer, dafür werden Transaktionen nur sehr kurz unterbrochen oder blockiert und die Transaktionsverarbeitung kann fast normal weiterlaufen.

3 ROC-Bausteine

In DBMS werden ROC-Techniken schon lange verwendet um die Persistenz sowie die Atomarität der getätigten Änderungen zu versichern. Die Einsicht, dass Fehler unver-

meidlich sind lassen mehr und mehr Bemühungen entstehen, ROC auch auf ganze Systeme oder Anwendungen zu beziehen. In diesem Abschnitt werden drei Beispiele aufgeführt, wie ROC-Methoden funktionieren, wie sie sich von ROC in DBMS unterscheiden und wie sie angewendet werden können.

3.1 Rekursive Neustarts

Schon so lange es Computersysteme gibt, gibt es auch die Möglichkeit sie durch einen Neustart wieder in einen definierten Zustand zu bringen. Durch einen Neustart lassen sich Deadlocks auflösen und fehlerhafte Systemzustände, die zum Beispiel durch Heisenbugs verursacht wurden, korrigieren. Die Startphase eines Systems ist sehr gut getestet. Das stellt sicher, dass die Wahrscheinlichkeit, ein System durch einen Neustart in einen definierten Zustand zu bringen, sehr hoch ist. Jedoch sind die meisten Systeme nicht auf Fehlerbehebung durch Neustarts ausgelegt. Sie sollten vor einem Neustart heruntergefahren werden, was im Fehlerfall allerdings nicht machbar ist.

Recursive Restartability ist die Fähigkeit eines Systems, auf verschiedenen Ebenen neu gestartet zu werden. Solche Systeme sind darauf ausgerichtet, neu gestartet zu werden, ohne dass vorher das Betriebssystem heruntergefahren wurde. Durch eine Kombination von Neustarts fehlerhafter Komponenten und periodischer Neustarts fehlerfrei laufender Komponenten zur Verjüngung des Systems wird so eine höhere Verfügbarkeit des Gesamtsystems erreicht. Durch eine feine Granularität wird es ermöglicht das komplette System, Teilsysteme oder nur Komponenten neu zu starten. Von rekursivem Neustarten spricht man, weil das System und seine Komponenten in einer Hierarchie in einem Baum angeordnet sind. Voraussetzung dafür, dass einzelne Komponenten in diesem Baum abgebildet werden können, ist, dass sie voneinander fehlerisoliert sind, also etwa in ihrer eigenen Java Virtual Machine (JVM) ablaufen. Der Neustartbaum bildet Neustart-Abhängigkeiten zwischen den einzelnen Komponenten und dem Gesamtsystem ab. Funktionale Abhängigkeiten und Zustandsabhängigkeiten zwischen den Komponenten sind im Baum nur indirekt vorhanden.

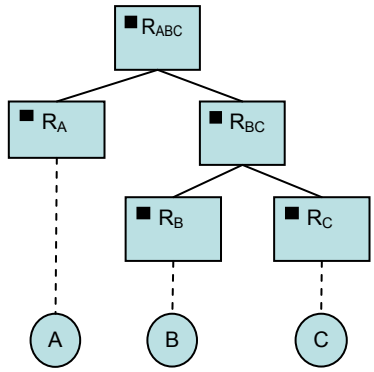

Abb. 3. Restart-Baum eines Systems mit drei *Komponenten* (A,B und C) und fünf *Zellen* (R_{ABC}, R_A, R_{BC}, R_B und R_C)

Abbildung 3 zeigt einen solchen Baum. Die Wurzel stellt das Gesamtsystem dar. Bekommt sie den Befehl zum Neustart wird das gesamte System neu gestartet. Die Rechtecke werden Zellen genannt, die Blätter entsprechen den tatsächlichen Software-Komponenten und werden als Kreise dargestellt. Teilbäume nennt man Neustartgruppen. Nur Zellen können den Befehl zum Neustart erhalten. Sie starten dann ihre Neustartgruppe, für die sie der Vaterknoten sind, neu. Wird beispielsweise Zelle R_{BC} der Befehl zum Neustart gegeben, werden Komponente B und C neu gestartet. So kann das System zuerst versuchen durch den Start einzelner Komponenten einen Fehler zu beheben. Reicht das nicht aus oder treten weitere Fehler auf, kann der Baum nach oben durchlaufen und eine größere Gruppe neu gestartet werden.

Der so genannte Recoverer führt die Neustarts aus, entscheidet aber nicht darüber welche Komponente ausgewählt wird. Das ist die Aufgabe des Orakels. Dieses Orakel repräsentiert die Regeln für die Neustarts und wendet diese auf den Baum an. Mit Hilfe der Fehlerdaten entscheidet das Orakel, welche Gruppe neu gestartet werden soll und teilt dem Recoverer seine Entscheidung mit. Dieser startet dann die entsprechende Zelle und somit die zugehörige Gruppe neu. Ist der Fehler dadurch behoben worden, läuft das System in Normalbetrieb weiter. Treten weiterhin Fehler auf, wird der Vaterknoten zum Neustart ausgewählt. Dieser Vorgang geht so lange weiter, bis entweder das System fehlerfrei läuft, oder das Gesamtsystem neu gestartet werden muss.

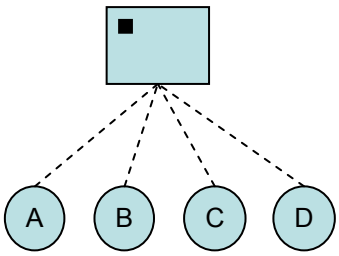

Abb. 4. Restart-Baum eines Systems ohne Möglichkeit zu rekursiven Neustarts. Die *Komponenten A, B, C* und *D* können nur gemeinsam neu gestartet werden

Der Aufbau eines guten Neustartbaums ist nicht einfach. Abbildung 4 zeigt einen Baum, wie er einem üblichen System zugeordnet ist. Nur das Gesamtsystem ist neustartbar. Jedoch ist somit die MTTR des Systems mindestens so groß wie die der schlechtesten MTTR der Komponenten. Es gilt also

```
MTTRsystem >= max(MTTRkomponenten)
```
(**3**)

Nun kann es aber sein, dass eine Komponente, die eine große MTTR hat, auch eine große MTTF hat. Das bedeutet, dass diese Komponente zwar lange braucht bis sie neu gestartet wurde, aber nur selten Fehler in ihr auftreten. Um das auszunutzen wird eine Vergrößerung der Tiefe im Baum angewandt, es werden partielle Neustarts der einzelnen Komponenten ermöglicht.

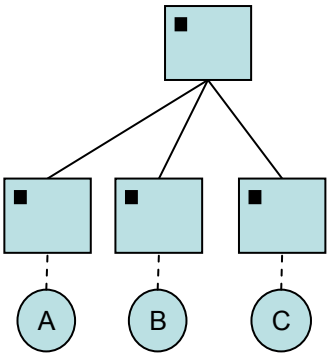

Abb. 5. Restart-Baum eines Systems, bei dem jede Komponente einzeln neu startbar ist

Abbildung 5 zeigt einen solchen Baum. Voraussetzung dafür, dass dieser Baum eine Verbesserung zum vorherigen bietet ist, dass das Orakel perfekt oder zumindest gut funktioniert und die Komponenten tatsächlich fehlerunabhängig sind. Eine weitere Möglichkeit den Baum zu verbessern besteht darin, die Baumtiefe durch Aufteilen von Komponenten zu vergrößern. Hat eine Komponente eine hohe MTTR und besitzt sie keine zu große innere Kopplung, so sollte versucht werden sie in kleinere Komponenten aufzuteilen. Oftmals kommt es nach einer solchen Trennung aber zu Fehlerabhängigkeiten zwischen den neu entstandenen Komponenten. Ein Neustart der einen Komponenten zieht also oft einen Neustart der anderen Komponente nach sich.

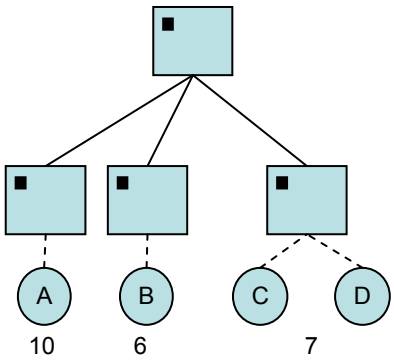

Abb. 6. Restart-Baum bei dem zwei Komponenten zu einer Gruppe zusammengefasst wurden

Schließt man jedoch die neu entstandenen Komponenten wie in Abbildung 6 zu einer Gruppe zusammen und geht davon aus, dass die beiden Komponenten parallel gestartet werden können, so ergibt sich ein Geschwindigkeitsvorteil im Gegensatz zu dem Fall, in dem die Komponenten einzeln gestartet wurden. In diesem Beispiel wurden C und D zu einer Gruppe zusammengefasst. Würde C 7 Zeiteinheiten zum Neustart benötigen und D 3 Zeiteinheiten, so würde ein paralleler Neustart ebenfalls nur 7 Zeiteinheiten benötigen, ein serieller Neustart hingegen 10.

Bisher wurde davon ausgegangen, dass das Orakel perfekt ist und keine Fehler macht. Allerdings ist diese Annahme unrealistisch. Zum einen kann das Orakel einen zu niedrigen Knoten auswählen, dessen Neustart nicht ausreicht um den Fehler zu beheben. Es wird also Zeit für diesen Neustart verschwendet, da anschließend der Elternknoten den Befehl zum Neustart erhält und die Komponente noch einmal mit anderen zusammen neu gestartet werden muss. Zum anderen kann ein zu hoher Knoten ausgewählt werden. Es hätte also gereicht, einen Knoten auf einer tieferen Ebene neu zu starten. Auch hier wurde wieder Zeit verschwendet.

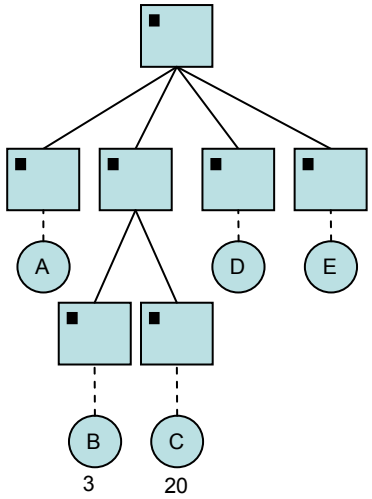

Abb. 7. Restart-Baum vor Aufstieg des Knotens für Komponente C

Eine Möglichkeit, die Auswirkungen solcher Fehler einzudämmen besteht darin, Knoten mit hoher MTTR im Baum aufsteigen zu lassen. Wird in Abbildung 7 der Knoten C neu gestartet und der Fehler dadurch nicht behoben, muss dessen Elternknoten neu gestartet werden. Knoten C wird also noch einmal neu gestartet. Würde man wie in Abbildung 8 Knoten C nach oben schieben, kann dieser nur noch in Verbindung mit Knoten B neu gestartet werden. Für den Neustart dieser Gruppe ist im Gegensatz zu einem Neustart von Knoten C zwar geringfügig mehr Zeit nötig, aber bei einem Orakel, das Fehler macht, ist trotzdem ein Vorteil zu erwarten, weil Knoten C nicht doppelt gestartet werden muss. Das Orakel hat durch diesen Aufstieg eines Knotens weniger Auswahl. Sollte das Orakel perfekt arbeiten und immer den richtigen Knoten zum Neustart auswählen, würde sich daraus ein Nachteil ergeben. Aber auch hier geht man vom Grundgedanken des ROC aus, der besagt, dass Fehler unvermeidlich sind. Daher nimmt man an, dass auch das Orakel fehlerhafte Entscheidungen trifft und auch einmal falsche Knoten neu starten lässt. Dadurch, dass man seine Auswahl der Knoten einschränkt, weil man von teilweise fehlerhaften Entscheidungen ausgeht ergibt sich letztendlich ein Vorteil in der Startgeschwindigkeit gegenüber einem Baum ohne einen solchen Aufstieg eines Knotens.

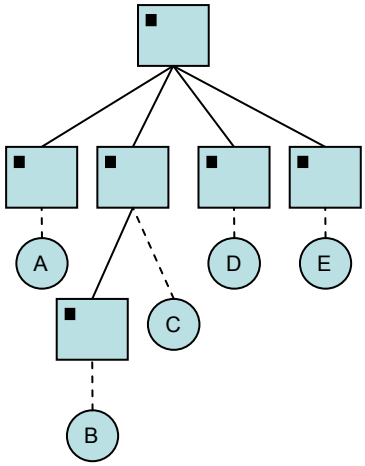

Abb. 8. verbesserter Restart-Baum nach Aufstieg von Knoten für Komponente C

Durch die feine Granularität ist es nötig, viele interne Schnittstellen zwischen den einzelnen Komponenten zu schaffen. Dadurch wird der Softwareentwicklungsprozess aufwendiger und somit auch teurer. Zeitgarantien, wie sie bei Echtzeitsystemen gefordert werden, können bei Systemen mit Recursive Restartability nicht garantiert werden. Das alles macht den Einsatz von solchen Systemen schwierig. Für die richtigen Anwendungen mit der richtigen Implementierung kann es aber durchaus eine erhöhte Verfügbarkeit des Systems mit sich bringen. Manche Internetportale oder Suchmaschinenbetreiber wenden schon heute Neustarts an, um ihre Systeme zu verjüngen und so eine hohe Verfügbarkeit zu erreichen. Generell sind Internet-Cluster und verteilte Systeme gut geeignet für den Einsatz rekursiver Neustarts.

3.2 Undo auf Systemebene

Wie bereits angesprochen haben Administratorfehler einen großen Anteil an den Fehlern, die zu Ausfällen führen und so die Verfügbarkeit eines Systems verringern. Bisher ist es nicht gelungen bei großen Systemen ohne Administratoren auszukommen. Zwar besteht die Möglichkeit, die Zahl der Eingriffe, die von Menschen in ein System gemacht werden müssen, zu verringern, indem man leichte Fehler automatisch beheben lässt. Das führt jedoch zur so genannten Automatisierungsironie: Die Administratoren verlieren Einblick und Verständnis in und für das System durch die fehlende Praxis mit kleineren Problemen. Gerade bei schwierigen Problemen, wo Administratoren dann doch zum Einsatz kommen, können sie durch Stress und fehlendes Training überfordert sein.

Ein besserer Weg als der Versuch Administrationsfehler völlig auszuschließen ist es, das System so zu entwickeln, dass es auf die menschlichen Schwächen zugeschnitten ist. [BrPa03] Es sollte Fehler tolerieren und eventuell kompensieren können und die Versuchs-und-Irrtums-Methode unterstützen, weil die dem menschlichen Verhalten am nächsten kommt. Die große menschliche Stärke gegenüber Computern ist es, auch durch Fehler und Irrtümer zum richtigen Ergebnis zu kommen. Dazu ist jedoch

eine rückwirkende Reparatur, ein Undo nötig. Der Administrator muss das System nach einem Fehler auf einen vorangegangenen Zustand zurücksetzen können. Somit wird es möglich unerwartete Situationen zu erforschen und Alternativen zu testen. Außerdem wird der Stress für den Administrator gesenkt. Begeht er einen Fehler, muss das keinen langen Systemausfall nach sich ziehen. Das System kann in einen lauffähigen Zustand zurückgesetzt werden.

Undo wird in Datenbanksystemen schon lange eingesetzt. Genauso ist es Bestandteil jeder Textverarbeitung, Tabellenkalkulation und Bildbearbeitungssoftware. In diesen Fällen wird jedoch das Undo nur auf Anwendungsebene ausgeführt. Mögliche Auswirkungen einer Aktion auf das Betriebssystem werden also durch ein Undo nicht rückgängig gemacht. Ein solches Undo auf Anwendungsebene ist also zum Zurücksetzen von Upgrades oder Patches einer Software unbrauchbar. Genauso kann eine Hackerattacke oder ein Virus nicht einfach zurückgesetzt werden Hierzu ist ein Undo auf Systemebene notwendig.

Technik. Die Technik die dazu angewendet wird ist der in einem DBMS sehr ähnlich: Logging auf einem nicht überschreibbaren Speicher. Das Undo auf Systemebene wird in drei Schritte unterteilt. Man spricht dabei von den "Three R's". Sie stehen für rewind, repair und replay.

Im ersten, dem Rewind-Schritt wird das vollständige System auf einen früheren Zustand zurückgesetzt. Ist also ein Fehler aufgetreten, wird ein Zustand vor dem Fehler zum aktuellen Zustand gemacht. Betroffen vom Zurücksetzen ist nicht nur die Anwendung, sondern auch das Betriebssystem sowie vorhandene Benutzerdaten. Der Rewind-Schritt ist das eigentliche Undo. Er läuft als einzelne Operation ab und stellt einen älteren physischen Zustand wieder her. Dadurch, dass ein physischer Zustand verwendet wird, müssen keine Annahmen über den aktuellen Zustand getroffen werden. Das macht diesen Schritt sehr flexibel und gut anwendbar.

Der zweite Schritt ist der Repair-Schritt. Hier wird dem Administrator die Möglichkeit gegeben beliebige Änderungen am System durchzuführen. Er kann beispielsweise einen Patch für ein Anwendungsprogramm einspielen, eine fehlgeschlagene Operation wiederholen oder eine Operation, die sich als falsch herausgestellt hat, unterlassen.

Im Replay-Schritt, dem letzten Schritt, werden alle Aktionen der Endbenutzer wiederholt um den Zustand ihrer Daten wieder herzustellen. Dabei werden alle Änderungen am System, die während dem Repair-Schritt gemacht wurden, beachtet und integriert. Dieser Schritt muss logisch ablaufen, damit es überhaupt möglich ist Änderungen der Benutzer in den aktuellen Zustand zu integrieren.

Um eine gewisse Flexibilität des Repair-Schrittes zu gewährleisten, sollten keine konkreten Aktionen eines Endnutzers, sondern besser die Absicht aller seiner Aktionen geloggt werden. Durch das Verwenden eines Protokolls und dessen festgelegtem Aktionssatz ist das möglich. Beispiele hierfür sind SMTP und IMAP zum Senden von E-Mails. Allerdings müsste genauso die Absicht einer Aktion im Repair-Schritt selbst geloggt werden. Da dies aber zumeist nicht möglich ist und der Repair-Schritt nicht durch ein Protokoll auf bestimmte Aktionen eingeschränkt werden soll, wird auf ein automatisches Wiederholen von Reparaturen verzichtet.

Eine weitere Herausforderung ist die externe Konsistenz. Wird beispielsweise von einem E-Mail System veranlasst eine Mail, die der Benutzer vorher in seinem Posteingang gesehen hat, durch eine Reparatur des Spamfilters in den Spamordner zu verschieben, entsteht so eine Inkonsistenz zwischen den beiden Zuständen. Da es für solche Probleme keine vollständige Lösung gibt, wird versucht mit der externen Inkonsistenz auszukommen, anstatt sie zu eliminieren. Der einfachste Ansatz dazu sieht vor, sie völlig zu ignorieren. Es bleibt also dem Endnutzer überlassen mit ihr zurechtzukommen. Dem Benutzer wird höchstens eine Hilfestellung angeboten um ihn dabei zu unterstützen, indem man ihm das Problem erklärt. Wird ein Zustand verändert, der vorher von keinem externen Benutzer gesehen wurde ist diese Änderung unproblematisch. Erstrebenswert ist es, dass möglich alle Änderungen so ablaufen.

Ein Undo sollte verschiedene Granularitäten anbieten. Möchte ein Endbenutzer ein Undo durchführen wird es nicht nötig sein, dieses auf das gesamte System zu beziehen. Deswegen reicht für ihn ein Undo aus, das sich auf den Zustand seiner Daten bezieht. Für den Administrator hingegen ist ein Undo vonnöten, das den gesamten Systemzustand mit allen Daten bearbeitet. Bei einem Cluster ist es zusätzlich aus Performanzgründen sinnvoll sowohl ein Undo für die einzelnen Knoten sowie ein Undo für das gesamte Cluster anzubieten. Durch diese verschiedenen Granularitäten wird allerdings ein Regelwerk nötig, das entscheidet welches Undo Vorrang hat. So sollte ein Administrator-Undo Vorrang vor einem Benutzer-Undo haben und dessen Änderungen mit rückgängig machen.

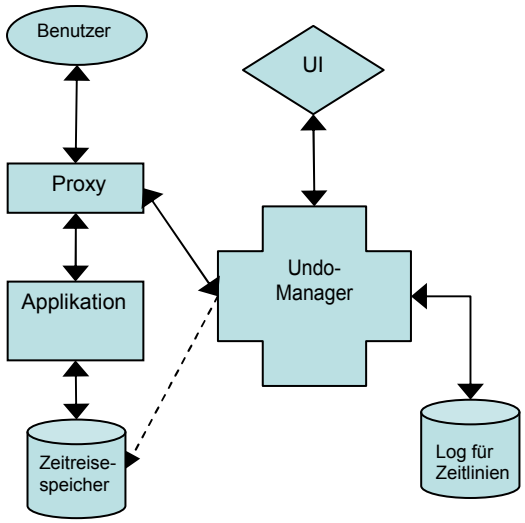

Abb. 9. Interaktion des *Undo-Managers* mit anderen Komponenten, die für ein Undo auf Systemebene notwendig sind [BrPa03]

Abbildung 9 zeigt das Schema eines Systems in dem ein Undo auf Systemebene möglich ist. Der Undo-Proxy hört alle Benutzeroperationen mit und lässt diese protokollie-

ren. Über ihn ist es auch möglich, für den Replay-Schritt eigene Operationen einzuspeisen. Der Proxy schränkt zwar die Flexibilität ein, da jedoch die Kommunikation im Internet sowieso nur über einige wenige Protokolle wie HTTP oder SMTP abläuft, bringt er auch Vorteile mit sich. Der Proxy kann für Standardprotokolle wieder verwendet werden. Er muss also nur einmal geschrieben werden. Dadurch wird die Fehleranfälligkeit geringer, weil davon ausgegangen werden kann, dass der Proxy gut entwickelt und ständig verbessert wird. Des Weiteren kann die Dienstapplikation selbst beliebig geändert werden, solange nur das Protokoll gleich bleibt.

Im Zeitreisespeicher werden periodisch Zustandsschnappschüsse des Systems gespeichert. So wird ein physisches Wiederherstellen eines beliebigen gespeicherten Zustands ermöglicht. Dieser Zeitpunkt kann ein vergangener sein, um etwa ein Rollback nach einem Fehler durchzuführen, oder ein zukünftiger, um ein Zurücksetzen rückgängig zu machen.

Im Log werden ähnlich wie in einem DBMS alle Änderungen durch den Benutzer protokolliert. Als interne Repräsentation der Zeit sowie zur Kennzeichnung der Änderungen wird auch hier jeweils eine Log Sequence Number (LSN) vergeben. Durch diese LSN kann dann nach einem Undo entschieden werden, welche Benutzeraktionen wiederholt werden müssen um den korrekten Zustand wiederherzustellen.

Das Herzstück bildet der Undo-Manager. Er koordiniert den Proxy sowie den Zeitreisespeicher. Er kommuniziert mit dem Proxy und speichert die von ihm erhaltenen Benutzeraktionen im Log. Im Fall eines Rollbacks speist der Undo-Manager im Replay-Schritt die benötigten Änderungen über den Proxy in das System ein. Auch der Zeitreisespeicher wird vom Undo-Manager koordiniert.

Beim Undo auf Systemebene wird die Wiederherstellung der Historie unter der Berücksichtigung und Integration der Änderungen im Repair-Schritt vollzogen. Dies ist der Hauptunterschied zu einem Undo in einem transaktionalen DBMS, wo Transaktionen nach ihrem commit dauerhaft gespeichert sind und niemals ein anderes Ergebnis annehmen.

3.3 Hardwareunterstützung für ROC: Der ROC-1 Prototyp

Die Annahme, dass erheblich bessere Ergebnisse in Bezug auf Verfügbarkeit und laufende Kosten eines Systems erreicht werden können, wenn ROC-Techniken von Grund auf in Software und Hardware integriert werden, hat zur Entwicklung der ROC-1 Hardware geführt. [BBH+02]

Daten. Der Prototyp dieser Hardware besteht aus 64 Knoten, welche Bausteine genannt werden. Jeder dieser Bausteine hat einen 266MHz-Pentium-II-Mobile-Prozessor, eine 18GB-SCSI-Festplatte und 256MB fehlerkorrigierendes DRAM. Des Weiteren sind in jedem Baustein vier redundante 100Mb/s-Netzwerkkarten eingebaut über die die Bausteine zu einem systemweiten Ethernet verbunden werden. Ein 18MHz-Motorola-Diagnoseprozessor wird ebenfalls verbaut. Diese Prozessoren werden durch ein privates Diagnosenetzwerk verbunden.

Alle 64 Bausteine sind untereinander verbunden. Es existieren 16 First-Level-Switches, zu denen jeder der vier Netzwerkkarten der Bausteine verbunden ist. Diese

First-Level-Switches sind über einen Gigabit Ethernet Uplink mit einem von zwei Gigabit-Switches verbunden, die beide alleine den Verkehr des gesamten Systems routen können. Somit hat der ROC-1-Prototyp eine Gesamtspeicherkapazität von 1,2TB und passt in drei Racks.

Der Hauptunterschied zu sonst üblichen Clustern ist das Verhältnis von Festplatte zu CPU. Die ROC-1 Hardware sieht pro Festplatte einen Prozessor vor. Bei einem normalen Cluster werden mehrere Festplatten von einem Prozessor verwaltet. Durch die dadurch entstehende geringere Last für den Prozessor ist es möglich, billigere und stromsparendere Prozessoren zu verwenden. Ein weiterer Unterschied ist der Zusammenschluss mit einem Diagnosesystem. Hier werden nur sehr gut geprüfte und zuverlässige Prozessoren eingesetzt, da sonst womöglich ein Diagnosesystem für das Diagnosesystem notwendig wäre.

Diagnose. Die Diagnoseprozessoren erhalten ihre Daten aus Sensoren, die an jedem Baustein und der Rückwand des Racks angebracht sind. In jedem Baustein findet man drei Thermometer, vier Stromüberwacher, einen Sensor, der den Versuch eines Ausbaus des Bausteins anzeigt, einen Batteriewarner für den SRAM des Diagnoseprozessors und einen Beschleunigungsmesser, um unnatürliche Schwingungen an Festplatten festzustellen, die zu einem Defekt der Platte führen könnten. Zusätzlich zu den Sensoren in den einzelnen Bausteinen gibt es an jedem Regal mit acht Bausteinen noch drei Thermometer, einen Stillstandsdetektor für die Lüfter und einen Feuchtigkeitssensor. Die Stromversorgung von Festplatte, CPU und Netzwerkkarten der Bausteine wird vom Diagnoseprozessor geregelt.

Fehler. Durch die Möglichkeit, die Stromversorgung zu einem beliebigen Baustein abzubrechen, können Ausfälle eines Bausteins simuliert werden. Des Weiteren kann der Diagnoseprozessor über eine zusätzliche Hardware Fehler in SCSI und Speicherbus einbringen. So können Bit-Dreher simuliert werden. Auch simulierte Fehler in bei der Ethernetübertragung und beim Empfang sind mit dieser Hardware möglich.

Aber nicht nur auf eine erhöhte Verfügbarkeit wurde beim Entwurf der ROC-1 Hardware geachtet, sondern auch auf die Kosten für den Betrieb. Da bei Server-Plätzen zumeist nach Anzahl der Racks und dem Stromverbrauch abgerechnet wird, wurde versucht beides zu minimieren. Acht Bausteine passen in ein Standard-3U-Rack. Dadurch wird nur wenig Platz benötigt. Durch die geringe benötigte Prozessorleistung bedingt durch ein Verhältnis von Festplatte zu CPU von 1:1 verbrauchen die Prozessoren wenig Strom. Außerdem wurden mobile Prozessoren eingesetzt, was den Strombedarf zusätzlich noch einmal senkt. In üblichen Servern werden mehrere Festplatten pro CPU eingesetzt, was zur Folge hat, dass leistungsstarke Prozessoren verwendet werden müssen, die einen hohen Stromverbrauch haben. Die höheren Kosten für die zusätzlichen CPUs werden durch Einsparung von Verkabelung, Kühlung und Halterung der nicht verbauten weiteren Festplatten amortisiert.

ROC-Techniken. Vier ROC-Techniken wurden in dir ROC-1-Hardware integriert. Es handelt sich um Redundanz und Isolation, Selbsttests und Verifikation im laufenden Betrieb, Unterstützung zur Problemdiagnose und der Verbesserung der menschlichen Interaktion mit dem System.

Durch die *Redundanz* wird ein single point of failure umgangen. Netzwerkkarten, Netzwerk-Switches, die Stromversorgung sowie die Lüfter sind in einem Baustein mehrfach vorhanden und können im Fehlerfall die Arbeit des ausgefallenen Bauteils übernehmen. So kommt es beispielsweise durch einen Ausfall einer Netzwerkkarte nicht zu einem Ausfall des ganzen Bausteins, weil eine der drei anderen Karten die Arbeit übernehmen kann.

Die *logische Isolation* ist Aufgabe der Software und wird durch verschiedene virtuelle Maschinen für die Software-Komponenten erreicht. Die *physische Isolation* hingegen ist Aufgabe der Hardware. Bei ROC-1 wird sie durch den Diagnoseprozessor gewährleistet, der die Stromzufuhr zu CPU, Netzwerkkarte und Festplatte kontrolliert. Dadurch ist es möglich, durch wegnehmen der Stromzufuhr zu der Netzwerkkarte eines Bausteins diesen vom Rest des Netzes zu isolieren.

Redundanz und Isolation helfen, den aufgetretenen Fehler auf den Verursacherknoten einzuschränken. Reparatur und Diagnose werden so im laufenden Betrieb möglich, ohne dass der Rest der Bausteine betroffen ist. Eine Skalierung durch Erweitern des Netzes wird somit ebenfalls ermöglicht. Das Training für Administratoren mit der tatsächlichen Hard- und Software-Konfiguration ist ein weiterer Verdienst der Isolation.

Selbsttest und Verifikation im laufenden Betrieb sind auch erst durch Isolation möglich. Es gibt drei Typen von Tests. Der erste Ist ein Korrektheitstest. Hier wird eine Eingabe mit bekanntem erwartetem Ergebnis in das System eingespeist. Das tatsächliche Ergebnis wird dann mit dem erwarteten verglichen. Bei einem Robustheitstest werden zufällige Eingaben erzeugt. Von einer Software wird dann überprüft, ob sich das System angemessen verhält und nicht etwa einfach ohne Fehlermeldung abstürzt. Beim dritten Typ eines Tests werden Fehler erzeugt, die einen Eingriff eines Administrators nach sich ziehen. Dadurch werden Administratoren trainiert und die Unterstützung des Administrators durch das System überprüft, indem getestet wird, ob der Fehler innerhalb einer bestimmten Zeitspanne behoben werden konnte.

Bei der Unterstützung zur Problemdiagnose kommen die Sensoren an den Bausteinen und den Racks zum Einsatz. Die Software dazu ist allerdings noch in der Entwicklung. Generell gibt es aber zwei Ansätze, die man sich dafür vorstellen kann. Die erste sieht vor, den Administrator vom Fehler zu benachrichtigen. Die zweite sieht eine automatische Reaktion auf einen Fehler vor. So könnte man sich zum Beispiel vorstellen, dass bei einer Überhitzung eines Knotens automatisch der Strom für diesen Baustein abgeschaltet wird, um die restlichen Bausteine im Rack zu schützen. Realistisch für eine Umsetzung ist jedoch die Umsetzung einer Kombination von beiden Ansätzen.

Eine *Verbesserung der menschlichen Interaktion mit dem System* wird vor allem durch Einfachheit versucht zu erreichen. Die kleinste austauschbare Einheit im ROC-1-System ist ein Baustein. Das System ist komplett homogen aufgebaut. Jeder Austausch eines Bausteins gleicht also dem Anderen. Des Weiteren gibt es innerhalb der

Racks keine Kabel, was einen Austausch erleichtert und Fehler durch falsche Verkabelung verhindert.

Ein weiterer Ansatz zur Verbesserung der Interaktion zwischen Administrator und dem ROC-1-System ist das Training, das beim Punkt Isolation bereits angesprochen wurde.

4 ROC in der Praxis

Obwohl wenn man sich durch ROC relativ große Verbesserungen in der Verfügbarkeit von Systemen verspricht, kommen ROC-Bausteine bisher nur in wenigen Systemen in der Praxis vor. Hervorzuheben ist in diesem Zusammenhang das Mercury-Satelliten-System, das auf rekursive Neustarts zur Verminderung der MTTR setzt. [BBC+02]

Abbildung 10 zeigt einen Vergleich von rekursiven Neustarts und Undo auf Systemebene in einem Kosten/Nutzen-Graphen. Daraus wird ersichtlich, dass rekursive Neustarts zwar günstig sind, aber auch weniger Fehler beheben wie das teurere Undo auf Systemebene.

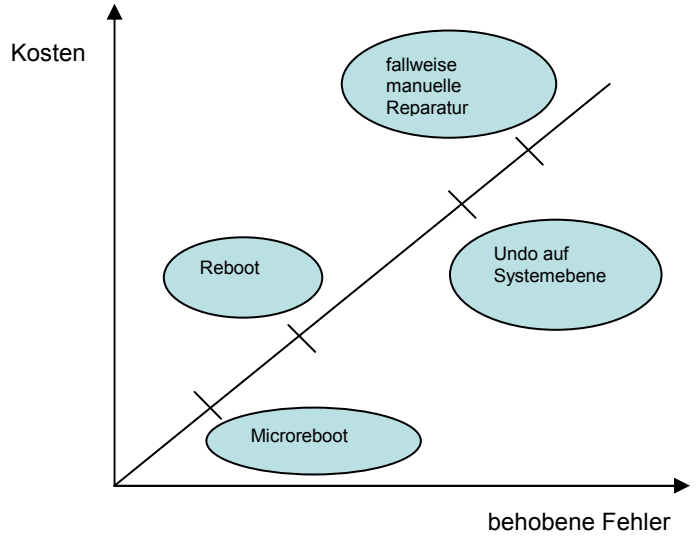

Abb. 10. Kosten/Nutzen-Graph für verschiedene Recovery-Methoden aus [BCFP04]. Die billigste Art und Weise Fehler zu beheben sind *Microreboots*, also Reboots von einzelnen Komponenten, wie sie bei den rekursiven Neustarts vorkommen. Allerdings ist die Anzahl der Fehler die dadurch behoben werden können nicht sehr hoch. Den größeren Nutzen, allerdings auch größere Kosten bringt das Undo auf Systemebene, welches nur geringfügig weniger Fehler beheben kann wie eine *fallweise manuelle Reparatur*

Rekursive Neustarts. Die Mercury-Bodenstation ermöglicht die Kommunikation mit zwei Satelliten. Nur wenn die Satelliten in einem bestimmten Winkel zur Station stehen und die Antennen richtig ausgerichtet sind, sowie das System verfügbar ist, ist eine Datenübertragung möglich. Diese Zeitspanne ist allerdings relativ gering.
Beim Aufbau des Mercury-Systems wurde ein untypischer Weg eingeschlagen. Obwohl es sich um ein zeitkritisches System handelt, wurden günstige, serienmäßige Komponenten verwendet, die hauptsächlich in Java implementiert wurden. Daher war es möglich die Technik der rekursiven Neustarts auf das System anzuwenden. Durch die Umstrukturierung des Systems und der Anwendung dieses ROC-Prinzips wurde eine Verbesserung der MTTR fast um das sechsfache erreicht. Da nicht die gesamte Zeitspanne zur Übertragung benötigt wird, sondern auch ein Teil ausreicht, um alle Daten des Satelliten zur Station zu senden, wurde so nicht nur eine quantitative Verbesserung erzielt, sondern auch eine qualitative: Die Station schafft es in fast 100% der kritischen Zeitspannen die Daten zu empfangen. [BBC+02]

Komponenten:	msgbone	fedr	pbcom	ise/istr	istu
MTTF	1 Monat	10min	1 Monat	5h	5h
MTTR vorher in s	28,9	28,9	28,9	28,9	28,9
MTTR nachher in s	4,7	5	21,9	6,1	5,8

(**1**)

Tabelle 1 zeigt die MTTR des Systems vor Integration der rekursiven Neustarts der einzelnen Komponenten und danach. Deutliche Verbesserungen sind zu erwarten, falls das Orakel es tatsächlich schafft die richtige Komponente, die für den Fehler verantwortlich ist neu zu starten.

Aber nicht nur im Mercury-System werden rekursive Neustarts verwendet, auch Keno, ein neuartiges Lottosystem von Lotto Hessen, welches vom Fraunhofer Institut mit einem Rechnersystem ausgestattet wurde, setzt auf rekursive Neustarts um eine möglichst hohe Verfügbarkeit gewährleisten zu können. [Spie04]

Undo auf Systemebene. Zwar wurde das Undo auf Systemebene bisher nur in Testsystemen eingesetzt, dennoch lassen sich, wenn auch wenige, Ergebnisse daraus ableiten. Es wurde beispielsweise in einem E-Mail-System der Berkeley Universität mit 1270 Benutzern eingesetzt. Um die Protokolle zu speichern ist ungefähr 1GB zusätzlicher Speicherbedarf pro Tag nötig. Mit drei 120GB-Festplatten kann also ungefähr der Jahresbedarf an zusätzlichem Speicher für dieses System gedeckt werden, was Kosten von etwa 240€ entspricht. Wie teuer es aber ist, das eigentliche Undo in die Anwendung zu integrieren, wird aus der Literatur nicht ersichtlich.

Der Gewinn an Benutzerfreundlichkeit für die Administratoren und die daraus resultierende verbesserte Verfügbarkeit des Systems lässt sich leider schwer messen. Allerdings ist die Anzahl an Fehlern, die durch ein Undo auf Systemebene behoben werden kann, recht hoch.

5 Zusammenfassung

Methoden, Techniken und Prinzipien des ROC werden schon lange in DBMS angewendet vor allem um die Persistenz der Daten zu garantieren. Wendet man ROC auf ganze Systeme an, lässt sich so die Verfügbarkeit dieser verbessern. Außerdem können die Administrierung erleichtert sowie die TCO gesenkt werden. Sind die Recovery-Methoden gut genug können auftretende Fehler sogar ganz vor dem Benutzer versteckt werden.

Durch die Einsicht, dass Fehler, seien sie bedingt durch Administratoren, Bugs in Software, Hardwarefehler oder unflexiblem Verhalten in Überlastsituationen, unvermeidlich sind, wird die Entwicklung von ROC-Methoden verstärkt vorangetrieben.

Es gibt bereits heute Systeme die auf ROC aufbauen. Das Mercury-Satelliten-Projekt gehört beispielsweise dazu. Hier werden rekursive Neustarts der Komponenten verwendet um im Fehlerfall die kurze Zeitspanne, die die Bodenstation Kontakt zum Satelliten hat, nicht durch einen langwierigen Neustart des gesamten Systems zu verlieren.

Der ROC-1 Prototyp verbessert die Verfügbarkeit schon auf der Hardwareebene durch eine Integration der ROC-Prinzipien in die Hardware. Weiterhin soll er im Betrieb günstiger als ein übliches Internet-Cluster sein.

In Zukunft werden ROC-Prinzipien wohl auch in „normalen“ Systemen anzutreffen sein, weil ihnen ein recht großes Verbesserungspotenzial beizumessen ist.

Quellen

BBC+02 Patterson, D. A.; Brown a.; Broadwell P.; Candea G.; Chen M.; Cutler J; Enriquez P.; Fox A.; Kiciman E.; Merzbacher M.; Oppenheimer D.; Sastry N.; Tetzlaff W.; Traupman J.; Treuhaft N.: Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. UC Berkeley Computer Science Technical Report; 2002

BBH+02 Brown A.; Beck J.; Hettena D.; Kuroda J.; Treuhaft N.; Oppenheimer D.; PattersonD.; Yelick K.: ROC-1: Hardware Support for Recovery-Oriented Computing. IEEE Transactions on Computers; 2002

BCFP04 Brown A. B.; Candea G.; Fox A.; Patterson D.: Recovery-Oriented Computing:Building Multitier Dependability, IEEE Computer Society, 2004

BrPa01 Brown, A.; Patterson D.: Embracing Failure: A Case for Recovery-Oriented Computing (ROC). 2001 High Performance Transaction Processing Symposium

BrPa02 Brown, A.; Patterson D.: Rewind, Repair, Replay: Three R's to Dependability; ACM SIGOPS European Workshop; 2002

BrPa03 Brown, A; Patterson D.: Undo for Operators: Building an Undoable E-mail Store. USENIX Annual Technical Conference; 2003

CaFo01	Candea G.; Fox A.: Recursive Restartability: Turning the Reboot Sledgehammer into a Scalpel, 8th Workshop on Hot Topics in Operating, 2001
CCD+02	Candea G.; Cutler J.; Doshi R.; Fox A.; Garg P.; Gowda R.;: Reducing Recovery Time in a Small Recursively Restartable System. International Conference on Dependable Systems and Networks, 2002
FoPa05	Fox, A.; Patterson D.: Approaches to Recovery-Oriented Computing. IEEE Internet Computing; 2005
Härd04	Härder, T.: Logging und Recovery, Vorlesung Datenbankanwendung WS04/05
Spie04	o.V.: Würdevoller Verfall; Der Spiegel. Juli 2004