Grundlagen des Autonomen Rechnens

Alexander Paar[1], Gábor Szeder[1], Tom Gelhausen[2], Marc Schanne[2] (Hrsg.)

Hans-Christoph Andersen, Haiko Gaißer, Oliver Hessel,
Michael Kohlbecker, Julia Matt, Oliver Mattes,
Urs Reupke, Andreas Stöckicht, Dominik Strecker

[1] Universität Karlsruhe, Institut für Programmstrukturen und Datenorganisation
(IPD Tichy), Am Fasanengarten 5, 76128 Karlsruhe
{alexpaar, szeder}@ipd.uka.de
http://www.autonomic-computing.org
[2] FZI Forschungszentrum Informatik
Haid-und-Neu-Straße 10-14, 76131 Karlsruhe
{gelhausen, schanne}@fzi.de

Zusammenfassung. Das vegetative Nervensystem (engl. autonomous nervous system) des Menschen kann das, wovon in der IT-Industrie noch geträumt wird. Abhängig von der aktuellen Umgebung und Tätigkeit reguliert das vegetative Nervensystem mandatorische Körperfunktionen wie Herzfrequenz und Atmung. Reflexe, die dem Selbstschutz dienen, werden automatisch ausgelöst. Verletzungen heilen von selbst, ohne dass man seine normalen Tätigkeiten dafür unterbrechen müsste. Im Rahmen des Seminars „Autonomic Computing“ im Sommersemester 2003 am Institut für Programmstrukturen und Datenorganisation der Universität Karlsruhe wurden Grundlagen dieses Autonomen Rechnens besprochen. Als Basis für Selbstkonfiguration und Selbstoptimierung werden in „Kontextbewusstsein: Ein Überblick“ Techniken zur Erfassung des physischen und sozialen Kontexts einer Anwendung erläutert. Die dienstorientierte Architektur und konkrete Implementierungen wie z.B. UPnP, Jini oder Bluetooth werden in „Aktuelle Technologien zur Realisierung dienstorientierter Architekturen“ behandelt. Die Arbeit „Service-Orientierung und das Semantic Web“ beschreibt, wie Semantic Web Technologien zur Beschreibung von Web Services verwendet werden können mit dem Ziel der automatischen Dienstfindung. Danach wird der Begriff „Selbstbewusstsein“ in bezug auf Software anhand zweier komplementärer Forschungsprojekte definiert. Technologien zur Überwachung des Laufzeitverhaltens von Rechnersystemen mit dem Ziel der selbstständigen Optimierung sind Gegenstand der Arbeit „Selbst-Überwachung und Selbst-Optimierung“. Der Artikel „Selbst-Schutz“ fasst die Sicherheitsanforderungen zusammen, die an ein autonomes Computersystem gestellt werden müssen und die Techniken, um solche Anforderungen zu erfüllen. Ansätze aus dem Bereich wiederherstellungsorientiertes- und fehlertolerantes Rechnen werden in „Selbst-Heilung“, „ROC – Recovery Oriented Computing“ und „Recovery Oriented Computing: Modularisierung und Redundanz“ vorgestellt. Alle Ausarbeitungen und Präsentationen sind auch elektronisch auf der diesem Band beiliegenden CD oder unter *www.autonomic-computing.org* verfügbar.

Inhaltsverzeichnis

Kontextbewusstsein: Ein Überblick

Urs Reupke

Zusammenfassung Die Rolle des Kontextbewusstseins für die zukünftige Entwicklung von Computersystemen ist nicht zu unterschätzen. Besonders unter der Doktrin des Autonomic Computing kommt ihm immer größere Bedeutung zu: Die Aufgaben der Selbstkonfiguration und der Selbstoptimierung werden durch Techniken des Kontextbewusstseins erst zu voller Reife gebracht. Nur in einem kontextbewussten System kann der Anwender jederzeit maximale Effizienz erreichen.
Diese Arbeit betrachtet das Forschungsgebiet und zeigt die daraus erwachsenden Möglichkeiten. Schwerpunkt bei der Betrachtung sind die Bereiche des physischen und des sozialen Kontextes.

1 Einleitung

Für uns Menschen ist es selbstverständlich, unsere Umwelt wahrzunehmen und angemessen darauf zu reagieren. Wenn jemand hinter uns einen Raum betritt, wenden wir uns ihm zu. Wir erkennen an seiner Körpersprache, in welcher Stimmung er ist, und stellen uns darauf ein. Abhängig von seiner sozialen Stellung und den Gesten, die seine Worte untermalen, wird diesen eine tiefergehende Bedeutung verliehen. Auch der Veränderung einer Raumeinrichtung können wir uns gleichermassen anpassen.

Ein Schema: Die Veränderung wird erkannt und das System Mensch reagiert darauf. Wir passen uns wie selbstverständlich Veränderungen in unserer Umwelt an, auch wenn diese unerwartet eintreten.

Für Computersysteme ist keiner der beiden Schritte einfach. Gegenwärtige Systeme reagieren ausschliesslich auf Eingaben durch vordefinierte Eingabeschnittstellen, diese Eingaben müssen vorher festgelegten Protokollen folgen. Die wenigsten Systeme sind in der Lage, Veränderungen ähnlich wie wir Menschen wahrzunehmen oder sogar flexibel darauf zu reagieren. Computersystemen diese Fähigkeit zu vermitteln, um durch eine bessere Modellierung der Kommunikation zwischen Menschen die Mensch-Maschine-Interaktion zu erleichtern ist eines der Ziele der Forschung im Bereich des Kontextbewusstseins. Ein weiteres Ziel ist es, durch frühzeitiges Erkennen der Umstände dem System zu ermöglichen, proaktiv, also bevor ein expliziter Befehl gegeben wird, auf die Wünsche des Benutzers einzugehen.

Dem System muss dazu ermöglicht werden, seine Umgebung wahrzunehmen und sich ähnlich einem Lebewesen daran anzupassen. Das System ist sich dann seiner Umwelt, seines Kontextes bewusst.

Ein System muss erkennen, wo es sich befindet und wovon es umgeben wird. Es muss die Position seiner Komponenten kennen, um zu wissen, welche von ihnen für die Anwender von Interesse sind. Darüber hinaus muss das System wissen, in welchem Zustand sich seine Umgebung befindet, um auf Veränderungen in den Umgebungsbedingungen eingehen zu können.

Ein einfaches Beispiel für ein kontextbewusstes System ist das eines Mobiltelefons, das die Lautstärke des Klingeltons erhöht, um auch in einer unruhigen Umgebung bemerkt zu werden, oder einen Vibrationsalarm aktiviert, sobald es in eine Hosentasche gesteckt wird.

Die zweite Richtung der Forschung beschäftigt sich nicht mit der Situation des Systems, sondern mit der des Benutzers. Seine Umgebung wird beobachtet und interpretiert, um frühzeitig mit Informationen oder Hilfestellungen bereit zu stehen und ihm notwendige Arbeit abzunehmen. Als Beispiel für diesen Zweig mag ein Navigationssystem dienen, dass selbstständig um Verkehrsstaus herumführt.

Der folgende Abschnitt definiert zunächst Kontext und Kontextbewusstsein. Anschliessend werden die Begrifflichkeiten erklärt, bevor der dritte Abschnitt auf die einzelnen Teile der Implementierung eingeht. Der vierte Abschnitt behandelt die Modellierung der Kontextinformationen innerhalb des Computersystems, der Fünfte untersucht die Bedeutung des Kontextbewusstseins für die weiteren Bereiche des Autonomic Computing. Abschliessend wird ein Resumé gezogen.

2 Grundlagen

Dieser Abschnitt gibt eine Einführung in die Begriffe und Konzepte von Kontext und Kontextbewusstsein und zeigt die nötigen Schritte, die ein System zum Kontextbewusstsein führen. Zunächst wird der Begriff „Kontext“ definiert, um die Grundlage für die folgenden Betrachtungen zu schaffen. Anschliessend wird die Unterteilung in die Kategorien des elektronischen, physischen und sozialen Kontextes vorgenommen. Der dritte Teil führt den Begriff des Kontextbewusstseins ein und erläutert die nötigen Schritte für die Erschaffung eines kontextbewussten Systems. Schliesslich wird eine Motivation für die Implementierung kontextbewusster Systeme geliefert.

2.1 Was ist Kontext?

Dey und Abowd [1] definieren Kontext als „*any information that can be used to characterize the situation of an entity, where an entity can be a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves.*“

Zum Kontext gehört also jede Information, die nützlich ist, um die Umgebung oder Situation des Systems oder seiner Benutzer genauer zu beschreiben, sei es eine Eigenschaft des Ortes, der anwesenden Personen oder der umgebenden Systeme. Die Definition ist mittlerweile weit verbreitet und akzeptiert [2], [3], [4].

Die Aussage „Der Rechner steht in der Ecke des Konferenzraums 213...“ (Ort) „...in dem gerade eine Besprechung stattfindet...“ (Tätigkeit der Anwesenden) „...und

hat eine Verbindung mit dem Firmennetzwerk.“ (Verbindungen) gibt drei Informationen über den Computer, alle drei sind Teil seines Kontextes.

Nach dieser Definition lässt sich ergänzen: „Alles ist Kontext.“ [5]. In jeder Situation kann jedes beliebige Element genutzt werden, um die Umstände genauer zu beschreiben.

Zu beachten ist jedoch, dass nicht in jeder Situation die gesamte Umgebung beschrieben werden muss, um das Ziel, den Kontext zu ermitteln, zu erreichen. Je nach Anwendungsziel variieren die benötigten Elemente stark.

2.2 Kategorien des Kontext

Bereits im vorangehenden Beispiel wurde der Kontext des Rechners in drei Kategorien eingeteilt. Diese Kategorien, elektronischer Kontext, physischer Kontext und sozialer Kontext und bilden eine weit verbreitete Unterteilung [6], [7].

Andere Unterteilungen des Kontextes [1], [8] gehen genauer auf einzelne Elemente ein, indem zum Beispiel der soziale und physische Kontext weiter unterteilt werden, während man den elektronischen Kontext vernachlässigt. Da die Betrachtung des Kontextes jedoch nicht von einer speziellen Nutzung abhängig gemacht werden sollte, empfiehlt sich dieses Vorgehen nicht.

Deshalb werden im Folgenden die drei Bereiche aus der erstgenannten Taxonomie[1] näher definiert und anhand von Beispielen veranschaulicht.

2.2.1 Elektronischer Kontext. Der elektronische Kontext befasst sich mit der Erreichbarkeit von Systemressourcen und Geräten im Netzwerk. Das System erkennt, welche anderen Systeme es umgeben, welche Geräte angeschlossen sind und wie sie genutzt werden können. Existierende Netzwerkprotokolle wie Bluetooth[9], Firewire [10] und USB[2][11] sind bereits in der Lage, Gerätebeschreibungen zu übermitteln. Weitere Ausarbeitung dieser Fähigkeiten in Kombination mit Entwicklungen im Bereich des Selbst-Bewusstseins werden die Fähigkeiten zur Selbstbeschreibung erweitern und verbleibende Lücken schliessen.

2.2.2 Physischer Kontext. Der physische Kontext des Systems, der auch als „Location Awareness“ oder „Spatial Awareness“ bezeichnet wird, beschreibt die direkt messbaren Aspekte der Umgebung: Er umfasst den Ort von Ressourcen und Peripherie, die in der Umgebung vorherrschenden physikalischen Bedingungen (Lautstärke, Temperatur, Beleuchtung) aber auch Details wie die Gestaltung der Umgebung und eingeschränkte Erreichbarkeit von Peripheriegeräten durch ungünstige Positionierung der Möbelstücke. Schliesslich umfasst der physische Kontext auch die Anwesenheit möglicher Anwender. Der überwiegende Teil der Menge der Eigenschaften dieser Anwender sind jedoch dem sozialen Kontext zuzuordnen.

2.2.3 Sozialer Kontext. Der soziale Kontext schliesslich umfasst die Eigenschaften der anwesenden Personen, die das System umgeben. Dazu gehören, neben offensichtlichen Merkmalen wie der Identität und der gegenwärtigen Aktivität auch Beziehungen zu anderen (nicht zwingend anwesenden) Personen, und, eine

1 Modell zur Segmentierung und Klassifikation der Elemente einer gemeinsamen Obermenge
2 Universal Serial Bus

hinreichend fortgeschrittene Erkennungsmethode vorausgesetzt, Gesundheit und Gemütszustand.

2.2.4 Überschneidungen. Es ist offensichtlich, das es in jedem der drei Gebiete Randbereiche gibt, in denen es nicht eindeutig ist, welcher Kontextkategorie eine Eigenschaft zugeordnet ist; als Beispiel seien hier nur die Position von Endbenutzergeräten (physischer oder elektronischer Kontext?) und die relativen Positionen von Personen zueinander (physischer oder sozialer Kontext?) genannt. Diese Überschneidung ist nicht zu vermeiden, stellt aber für die Umsetzung der Kontextbewusstseins keine weiteren Probleme dar – Daten die einmal erfasst sind, können softwareseitig auch von zwei verschiedenen Systemen interpretiert werden.

2.3 Von Kontext & System zum Kontextbewussten System

Zunächst gilt es, zu klären, was Kontextbewusstsein ausmacht: Von Kontextbewusstsein in Zusammenhang mit einem Computersystem spricht man, sobald das System in der Lage ist, Kontextinformationen aufzunehmen und an Endanwendungen weiterzuleiten. In der Tat genügt es, wenn der Kontext lediglich aufgenommen und ohne weitere Interpretation oder Reaktion an den Benutzer weitergegeben wird.

Um einem System Kontextbewusstsein zu verleihen, sind zwei grundlegende Schritte vonnöten: Zunächst muss das bestehende System um Sensoren und Schnittstellen zur Erfassung der Umgebungsdaten erweitert werden. Dann muss Software erstellt werden, die auf diese Daten zugreift.

Der erste Schritt zum Kontextbewusstsein ist also, dem bestehenden Computersystem durch die Installation von Sensoren Möglichkeiten zu geben, seinen Kontext wahrzunehmen: Von einfachen Thermometern und Photosensoren, die den Zustand des Raumes erkennen, über Bewegungsmelder, Mikrophone oder Kameras, bis hin zu spezialisierten Sensoren wird jede erdenkliche Schnittstelle genutzt, um dem System mehr Daten zur Verfügung zu stellen.

Um optimal auf jede Situation reagieren zu können, muss das System möglichst viele Parameter seiner Umgebung erkennen können, unabhängig davon, ob jedes Datum zu jedem Zeitpunkt benötigt wird.

Doch diese Datenflut allein reicht nicht aus, damit das System seinen Kontext sinnvoll nutzen kann. Der Softwareentwickler muss nun beurteilen, welche Teile des Kontextes für einen gegebenen Zustand seiner Anwendung von Bedeutung sind, damit er die erforderlichen Sensordaten beziehen kann.

Bei der Entwicklung kontextbewusster System muss die Softwareentwicklung mehr als bisher auf die Peripherie eingehen, da die Erfassung der Sensordaten von zentraler Bedeutung ist. Gleichsam muss die Hardwareentwicklung den Anforderungen des Software entgegenkommen, um das System auf die gegebenen Bedingungen masszuschneidern.

2.4 Motivation

Ein kontextbewusstes System bringt sowohl bei der Implementierung als auch bei der Betrachtung der Systembelastung einen deutlichen Mehraufwand mit sich. Bei

stationären Systemen ist mehr Hard- und Software zu installieren, im Fall eines mobilen Geräts werden durch die benötigten Sensoren Gewicht und Größe des Systems negativ beeinflusst.

Darüberhinaus hat ein kontextbewusstes System eine beträchtlich größere Datenmenge zu verarbeiten, als ein herkömmliches Computersystem: Es muss nicht nur auf explizite Eingaben über Maus oder Tastatur reagieren, sondern jede Veränderung in seiner Umgebung registrieren. Während im Falle eines einzelnen Computers oder Mobilgeräts aus diesem Umstand höchstens eine geringere Verarbeitungsgeschwindigkeit resultiert, kann ein Firmennetzwerk durch die erhöhte Last an die Grenzen seiner Kapazität gebracht werden.

Wie soll dieses Mehr an Aufwand in allen Bereichen dennoch zu einem Nutzen führen?

Die gegenwärtig verbreiteten Systeme verlangen vom Benutzer ein hohes Niveau an Aufmerksamkeit, oft konkurrieren sogar mehrere Systeme um die Zeit eines einzelnen Anwenders. Durch die Einführung des Kontextbewusstseins wird es dem System möglich, Teile der einfachsten Aufgaben, die bisher die Zeit des Benutzers in Anspruch genommen haben, selbstständig abzuarbeiten: Ein Telefonanruf muss nicht mehr von Hand durch drei verschiedene Büros durchgestellt werden, bis er seinen Empfänger erreicht. Statt dessen kann das System ermitteln, wo sich die Zielperson befindet, das nächste Telefon ausmachen oder, falls keines verfügbar ist, das Mobiltelefon des Benutzers anwählen, und den Anruf dorthin umleiten.

Das System trägt die Last, um sie vom Benutzer fernzuhalten, und ihm seine Arbeit zu erleichtern. Es stellt nur die Dienste und Informationen zur Verfügung, die gerade benötigt werden, so gut auf den Anwender zugeschnitten wie möglich: Während Baustellenlärm von draussen hereindringt ist Tonwiedergabe auf Zimmerlautstärke praktisch nutzlos, also erhöht das System die Lautstärke oder setzt die Informationen in Text- oder Bildform um.

Weiterhin kann Kontextbewusstsein genutzt werden, um dem Benutzer Informationen über seine Umwelt zu bieten, die ihm andernfalls nicht zur Verfügung ständen: Auf einer Konferenz wird jederzeit der aktuelle Zeitplan auf dem Handheld des Benutzers zugänglich gemacht, Hintergrundinformationen zu Vortragenden und Fragestellern erleichtern die Diskussion.

3 Umsetzung

Dieses Kapitel beschäftigt sich mit der praktischen Umsetzung des Kontextbewussseins. Zu Beginn werden Möglichkeiten zur Erfassung der benötigten Daten beschrieben, danach folgt eine Betrachtung der Modelle zur Auswertung der erfassten Daten. Abschliessend werden die Anwendungsmöglichkeiten untersucht.

3.1 Erfassung der Daten

3.1.1 Elektronischer Kontext. Die Erfassung des elektronischen Kontextes stellt kein großes Problem dar. Die Beschreibung der eigenen Bestandteile, Peripherie und Netzwerkumgebung ist in gegenwärtigen Betriebssystemen bereits enthalten. Durch

Erweiterungen dieser Systeme [12] in Verbindung mit kabellosen Netzwerken wird die Kommunikation untereinander vereinfacht und eine automatische Erkennung der Fähigkeiten anderer Geräte ermöglicht.

3.1.2 Physischer Kontext. Auf den ersten Blick mag der Eindruck entstehen, die Elemente des physischen Kontextes seien die am einfachsten zu beziehenden Kontextinformationen. Mittels einfacher Messinstrumente aus der Physik kann ein Grossteil der relevanten Daten aufgenommen werden. Lediglich eine Schnittstelle zum Anschluss an den Computer muss gegebenenfalls hinzugefügt werden. Dieser Eindruck täuscht.

Sobald der Bereich der Messwerte verlassen wird und es darum geht, Objekte im Raum zu erkennen, gerät die Sensortechnik schnell an ihre Grenzen.

Als Beispiel diene hier die Erkennung der Einrichtung eines Raumes.

Beispiel: Raumeinrichtung. Die einfachste Lösung wäre es, einen Administrator zu beschäftigen, der regelmäßig die gegenwärtige Konfiguration der Einrichtungsstücke aller Räume in eine Datenbank einträgt, doch diese Lösung führt zu unangemessen hohem administrativen Aufwand und beeinträchtigt die Flexibilität.

Eine praktikable Alternative ist es, jedes für das System relevante Objekt, im Idealfall also jedes Objekt in der Anlage, mit einem Transponder auszustatten, aus dessen Ausstrahlung jederzeit Position und Orientierung des zugeordneten Objekts trianguliert werden können. Dieses Verfahren wurde im Haus Olivetti bereits Anfang der 1990er Jahre erfolgreich angewandt und ist seit dem unter dem Namen „(Olivetti) Active Badge“ („Aktiver Ausweis“) bekannt [13].

Die dritte und mit Abstand eleganteste Methode ist die Erkennung des Zustandes eines Raumes aus mit Kameras aufgezeichneten Bildern. Gegenwärtige Bilderkennungssysteme sind jedoch nicht ausgereift genug, um dieses Verfahren erfolgreich anwenden zu können. Schlechte Beleuchtungsverhältnisse, Überdeckung von Kanten (unordentliche Schreibtische) und Verdeckung durch andere Objekte erschweren die Erkennung zusätzlich.

Dieses Verfahren benötigt mehr Rechenzeit. Es ist aber gerade in Gebieten, in denen die Einrichtung häufig verändert wird oder es kein festes Inventar gibt, den anderen beiden Verfahren überlegen.

Beide letztgenannten Optionen eignen sich nicht nur zur Erkennung von Gegenständen, sondern auch zur Erfassung von Personen. Während es jedoch kein Problem bereitet, jeden Tisch mit einem Transponder auszustatten, stellt es für die Mitarbeiter eine zusätzliche Belastung dar, das Gerät jederzeit bei sich zu tragen.

Eine Lösung für dieses Problem findet sich bereits heute im Bereich der „Bewussten Gegenstände“ („Aware Artifacts“) [14]. Dort werden Transponder in Schmuck oder Kleidung integriert – eine Idee, die Datenschützer wegen der erhöhten Überwachungsgefahr ablehnen könnten.

3.1.3 Sozialer Kontext. Die größten Schwierigkeiten bringt die Erkennung des sozialen Kontextes mit sich: Für ein Computersystem ist es gegenwärtig sehr schwierig zu erkennen, welche Tätigkeiten Menschen verrichten – es gibt keine „Tätigkeitssensoren“, die Handlungen eindeutig identifizieren können, statt dessen müssen verschiedene Sensorsignale kombiniert ausgewertet werden, um die soziale Situation zu ermitteln.

Das System kann erkennen, wieviele Personen anwesend sind und um wen es sich handelt, um dann mittels Datenbankabfragen die Stellung der Betreffenden zueinander ermitteln. Es kann anhand von Mikrofonaufnahmen erkennen, wer spricht und wie intensiv das Gespräch ist, während aus Kameradaten Bewegungsabläufe ausgewertet werden. Mit fortgeschrittener Spracherkennungssoftware wäre es möglich, das Thema des Gesprächs festzustellen, um Rückschlüsse auf benötigte Daten und die Art des Zusammentreffens zu ziehen.

Wiederum können bewusste Gegenstände zur Hilfe herangezogen werden, indem Alltagsobjekte um Sensoren erweitert werden, die ihre gegenwärtige Anwendung an das Computersystem übermitteln können.

Eine weitere Hilfestellung entsteht durch das Einbeziehen von Terminplanern, Raumreservierungen und ähnlichen Datenbanken, die dem System einen Anhalt liefern können, welche Art von Situation und Personenkreis zu erwarten ist, um so die Einschätzung zu vereinfachen.

Durch den Mangel an Möglichkeiten zur direkten Erfassung kann der Eindruck entstehen, dass der soziale Kontext eine untergeordnete Eigenschaft ist. Für den modernen Menschen besitzt jedoch gerade das soziale Umfeld erhöhte Bedeutung, weswegen hier der sozialen Kontext weiterhin als grundlegende Eigenschaft einer Situation gilt.

3.2 Ermittlung des Kontextes

Sobald die sensorisch aufgezeichneten Daten dem System zur Verfügung stehen, beginnt die Aufgabe der Software, die Daten auszuwerten. Dieser Teil informiert über die verschiedenen Modelle und zeigt deren Vor- und Nachteile.

3.2.1 Many on Many-Interpretation. Der erste Ansatz zur Ermittlung des Kontextes ist, jeder Anwendung die Sensoren zur Verfügung zu stellen, die sie für ihre Zwecke benötigt, und ihr die Fähigkeit zu geben, diese Sensoren auszulesen und die Rohdaten als Kontext zu interpretieren.

Da die Anwendungen selbst für die Interpretation der Daten, die ihre Sensoren liefern, verantwortlich sind, kann der einzelne Softwareentwickler sicherstellen, dass die Software in jeder Situation die Daten so behandelt, wie er es beabsichtigt hat.

Diese Methode erzeugt, sofern mehr als eine kontextbewusste Anwendung aktiv ist, viel Verkehr im Netzwerk und viel Last bei den einzelnen Anwendungen, ausserdem wird es redundante Sensoren geben, die für zahlreiche Anwendungen installiert werden. Dadurch kann es sowohl finanzielle als auch räumliche Probleme geben, die die Installation der kontextbewussten Lösung verhindern.

3.2.2 Stand-Alone-Interpretation. Als logischer Schritt zur Vermeidung dieses Problems kann ein Sensorensatz mehreren Anwendungen zur Verfügung gestellt werden. Dazu werden die gesammelten Daten auf einem Server für die Programme zum Abruf bereitgehalten. Diese Variante hat wenig Auswirkung auf die Belastung des Netzwerks, kann aber die Anforderungen an Geld und Platz senken.

Der Vorteil der individuellen Installation bleibt erhalten, jedoch mit der Einschränkung, dass möglicherweise Sensoren, die nur von wenigen Anwendungen benötigt werden, aus Kostengründen nicht installiert werden.

Darüberhinaus muss der Softwareentwickler vorab wissen, welche Sensormodelle zum Einsatz kommen, da die Software deren Schnittstellen gezielt ansprechen können muss.

Ein weiterer Nachteil beider Modelle der Stand-Alone Interpretation kommt zum Tragen, wenn viele kontextbewusste Softwaresysteme parallel eingesetzt werden. Jedes dieser Systeme hat eine eigene Methode der Kontextauswertung und kommt zu eigenen, nicht zwingend übereinstimmenden Ergebnissen. Es kann also passieren, dass zwei Anwendungen durch unglückliche Interpretation gegenläufige Aktionen einleiten anstatt zu kooperieren.

3.2.3 Middleware-Interpretation. Die meisten Mißstände der Stand-Alone-Interpretation behebt das Modell der Middelware-Interpretation: Die Auswertung der Daten wird bereits serverseitig von einem unabhängigen Programm, dem Interpreter, übernommen, das die ermittelten Kontextinformationen (und nicht länger die Rohdaten) über eine Schnittstelle an die verschiedenen kontextbewussten Programme übermittelt. Der einzelne Programmierer muss sich nicht mehr mit den Sensoren und Protokollen, die von diesen unterstützt werden, beschäftigen, sondern muss nur noch die einheitliche Schnittstelle des Servers ansprechen.

Das zugrunde liegende Modell wird von Gellersen *et al.* in [15] eingehender beschrieben, eine praktische Umsetzung erfolgt in [8]: Die einzelnen Sensoren übermitteln dabei einzelne Eigenschaften der aufgezeichneten Daten an Sammelpunkte. Jeder Sammelpunkt erhält einen einzigen Datentyp von einem Sensor, zum Beispiel sammelt ein Punkt die Lautstärkewerte eines Mikrophons, während ein anderer die Frequenzkurven aufzeichnet. Die Sammelpunkte leiten die Daten dann an Interpreter weiter, jedoch wird auch der Rohdatensatz an den Server übermittelt, falls eine Anwendung diese benötigen sollte. Auf einem Server liegt dann das vollständige Modell des Kontextes eines Objekts, und jede Anwendung kann darauf zugreifen. Abbildung 1 zeigt die einzelnen Ebenen des Modells.

Der größte Nachteil dieses Systems ist, dass jede Anwendung für sich entscheidet, wann sie neue Kontextinformationen benötigt, die sie in regelmäßigen Abständen abfragen muss, um dann selbst zu bestimmen, ob die neu empfangenen Informationen eine Änderung für sie bedeuten. Durch dieses Verfahren entsteht sowohl für den Server als auch für die einzelnen Anwendungen viel Last. Das Problem kann durch einen Abonnenten-Dienst, wie ihn das Kanal-Modell vorsieht, behoben werden.

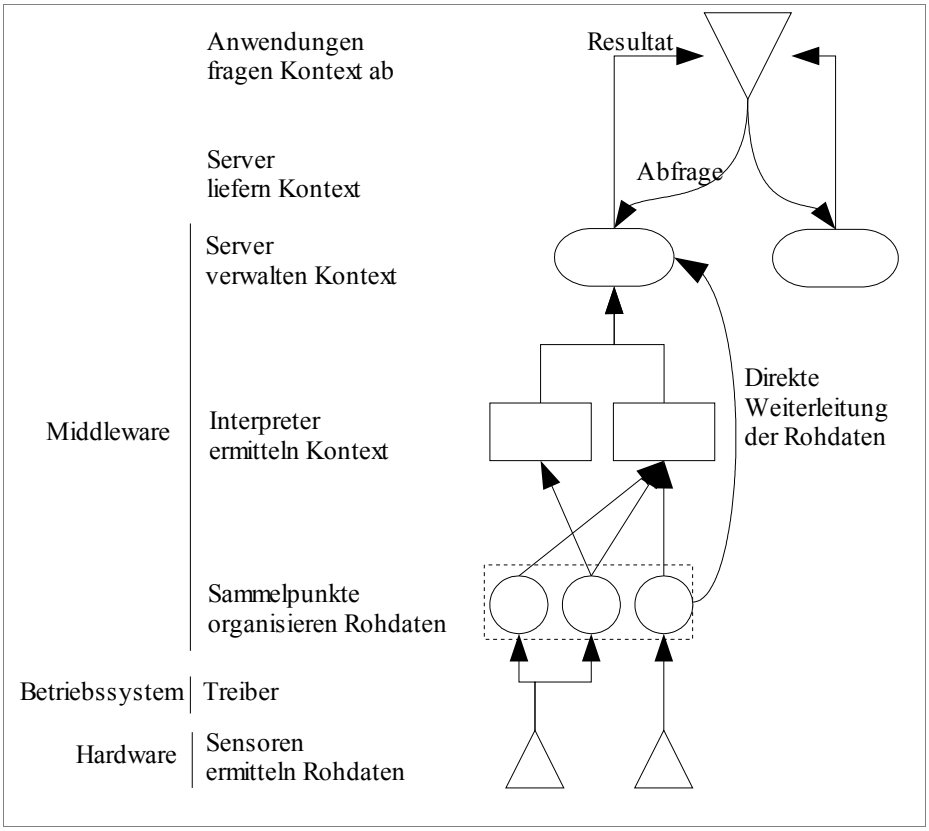


Abbildung 1: Schematische Darstellung des einfachen Middleware-Modells

3.2.4 Das Kanal-Modell. In diesem Modell wird, sobald sich ein neuer Interpreter oder Sammelpunkt am Server anmeldet, ein "Kanal" eröffnet, der für die Informationen dieser Datenquelle exklusiv zur Verfügung steht.

Die Anwendungen werden nun mit einer aktualisierten Kanalliste über die neuen Daten informiert und können sich auf eine Abonnentenliste aufnehmen lassen. Sobald im Kontext des Objekts eine Änderung auftritt wird ein Ereignis ausgelöst, in dessen Folge alle Abonnenten mit dem neuen Kontext versorgt werden und ihrerseits darauf reagieren können.

Dieses Modell stellt geringere Anforderungen an den Server, der so lediglich die Abonnenten-Liste verwalten und die Benachrichtigungen absenden muss. Jede Anwendung muss für sich feststellen, ob eine Reaktion erforderlich ist.

Diese Problematik kann durch das Setzen von Schwellenwerten umgangen werden. Anders als zuvor wird nicht jede Änderung an die Anwendungen übertragen: Anwendungen können bei der Anmeldung für einen Informationstyp Schwellenwert-Parameter angeben, die serverseitig überprüft werden, sobald eine Änderung im entsprechenden Kanal vorliegt. Erst, wenn die von der Anwendung gesetzten Parameter erfüllt sind, wird tatsächlich eine Nachricht ausgesandt. Der Vorteil dieses

Verfahrens ist eine geringere Belastung des Netzwerks. Dafür müssen sowohl auf der Seite des Servers als auch auf Seiten der Anwendung die Umgebungs-Bedingungen überprüft werden, einmal, um das Verhältnis zu den Schwellenwerten festzustellen, einmal, um die angemessene Aktion zu ermitteln.

3.2.5 Kontextzellen. Eine Erweiterung, die mit jeder Version des Middleware-Konzepts anzuwenden ist, bringen Zeidler und Kehr [5] mit ihrem Modell der Kontextzellen ein. In diesem Modell erzeugt jedes Objekt seine eigene Zelle, deren Kontextinformationen erfasst werden können, gleichzeitig aber versucht jede Zelle, sich anderen Zellen unterzuordnen, um Teil eines größeren Kontextes zu werden. Dadurch entsteht eine Hierarchie von Kontextzellen, wobei jeweils der Kontext der höchstrangigen Zellen gleichzeitig Kontext der niederrangigen Zellen ist. Kontextzellen erleichtern die Verknüpfung der Kontextinformationen der Objekte und verringert so das Datenaufkommen für den einzelnen Server.

Durch das Einführen von Referenzen zwischen beliebigen Kontextzellen kann das Modell in seinem Potential verbessert werden, da sich so die Anzahl der möglichen Beziehungen deutlich erhöht [16].

Das folgende Beispiel verdeutlicht die Anwendung von Kontextzellen.

Beispiel: Kontextzellen. Betrachten wir den Mitarbeiter einer Firma, der am Morgen seinen Arbeitsplatz erreicht. Beim Betreten des Gebäudes ordnet sich seine persönliche Kontextzelle der Kontextzelle des Flurs, den er entlanggeht, um sein Büro zu erreichen, unter, während diese wiederum der Kontextzelle des Gebäudes untersteht. Beim Durchschreiten seiner Bürotür verlässt der Mitarbeiter die Zelle des Flurs, und seine Kontextzelle sucht sich eine neue, möglichst niedrig gestellte Zelle, in diesem Fall die des Büros, die ebenfalls dem Gebäude untergeordnet ist. Eine Änderung im Kontext des Gebäudes, beispielsweise ein Wetterumschwung, ist durch diese Baumstruktur leicht als Veränderung des Büro-Kontextes und von dort wiederum als Veränderung des Kontextes des Mitarbeiters zu erkennen, ohne das auf dem Kontext-Server des Mitarbeiters eine Wetter-Information abgelegt sein muss. Die Verbindungen in der Hierarchie sind bidirektional, eine Abfrage eines leitenden Mitarbeiters, wie viele seiner Angestellten sich gegenwärtig in der Anlage befinden, würde nicht direkt aus einer Eigenschaft des Gebäudes beantwortet werden, sondern die Inhalte der einzelnen Räume des Gebäudes überprüfen, die Verbindungen der Räume zu Kontextzellen von Personen überprüfen, und diese wiederum auf ihren Mitarbeiterstatus.

Das Modell der Kontextzelle funktioniert nicht nur im physischen Kontext: Wenn der Mitarbeiter später am Tag an einer Besprechung teilnimmt, wird seine Zelle, sobald das System erkennt, das eine Konferenz stattfindet, Teil der Kontextzelle der Besprechung. Diese könnte wiederum Teil einer größeren Struktur sein, wenn mehrere Filialen des Konzerns ihre zeitgleich stattfindenden Besprechungen mittels Videoübertragung verbunden haben.

3.3 Nutzung des Kontextes

Die interpretierten Daten stehen nun dem System und damit allen kontextsensitiven Anwendungen zur Verfügung. Jede Anwendung kann sich die ermittelten

Kontextinformationen über die eingerichteteten Kanäle und Server übermitteln lassen und sie wie direkte Eingaben auswerten.

Im Laufe der Zeit sind verschiedene Taxonomien vorgeschlagen worden, um die Fähigkeiten von kontextbewussten Anwendungen näher zu beschreiben: Bereits in der ersten Arbeit zu diesem Thema [7], die Schilit 1994 veröffentlichte, waren die Fähigkeiten der Wiedergabe von, Reaktion auf und Rekonfiguration mit Hilfe von Elementen des Kontextes enthalten.

Pascoe [17] ergänzte diese Liste später um eine weitere Fähigkeit: Selbstständige Erweiterung des Kontextes durch Hinzufügen von Informationen. Sein Vorschlag enthält eine Vierteilung der Anwendungsmöglichkeiten: Wiedergabe, Reaktion, Rekonfiguration und Augmentation.

Diese Fähigkeiten werden nachstehend umrissen und Beispiele für ihre praktische Anwendung vorgestellt.

3.3.1 Contextual Sensing (Kontextwiedergabe). Kontextwiedergabe ist die einfachste Möglichkeit, erfassten Kontext zu nutzen. Vom System wird nichts weiter verlangt, als den Kontext in aufbereiteter Form an den Benutzer weiterzugeben. Diese Anwendung ist besonders leicht in Bezug auf den physischen Kontext zu realisieren, und damit vor allem im Bereich des Mobile Computing von hohem Interesse, da sich hier die Umgebung von Nutzer und System häufig und schnell ändert. Einfache Anwendungsbeispiele für Kontextwiedergabe sind eine Positionsermittlung über GPS, oder eine aufbereitete Anzeige der Wetterdaten.

3.3.2 Contextual Adaption (Anpassung an den Kontext). Hier liegt die wahrscheinlich wichtigste Aufgabe, die einem kontextbewussten System gestellt werden kann. Unter „Anpassung an den Kontext“ verstehen wir die Fähigkeit des Systems, eigenständig auf den erfassten Kontext zu reagieren und so dem Benutzer die Notwendigkeit expliziter Befehle abzunehmen. Eine noch nicht implementierte Umsetzung für Anpassung ist ein kontextbewusster Wohnbereich, der ohne weitere Anweisung die bevorzugten Umgebungsbedingungen der anwesenden Bewohner herstellt, z. B. Temperatur, Luftfeuchtigkeit und Beleuchtung.

3.3.3 Contextual Ressource Discovery (Ressourcenfindung). Ein System mit dieser Fähigkeit ist in der Lage, selbstständig seine Umgebung nach Ressourcen und Diensten zu durchsuchen, um diese dem Benutzer zur Verfügung zu stellen oder einzelne Teile des eigenen Systems darauf auszulagern. Diese Fähigkeit benötigt die Unterstützung der Teilbereiche der Selbstbeschreibung und der Selbstoptimierung.

Dey und Abowd [1] betrachten die Ressourcenfindung als einen Teil der Schnittmenge von Kontextwiedergabe und Anpassung, und führen sie nicht als einzelnen Teilbereich auf.

Ein Beispiel für die Anwendung dieser Fähigkeit ist ein mobiles System, das selbstständig den jeweils nächstgelegenen Drucker findet und die Ausgaben auf diesen umlenkt, um Wege zu erparen.

Eine andere Anwendung der Ressourcenfindung ist die Entdeckung von Informationsressourcen. Ein Beispiel hier wäre die Ermittlung der Verfügbarkeit von Parkplätzen bei der Einfahrt in ein Parkhaus, um den Benutzer ohne langes Suchen an einen für seinen Wagen geeigneten Platz zu leiten.

3.3.4 Contextual Augmentation (Kontextabhängige Erweiterung). Die Methoden der kontextabhängigen Erweiterung bilden eine Brücke zwischen der digitalen Welt des Systems und der realen Welt, mit der der Benutzer interagiert. Sie erweitern das System um Möglichkeiten, der Realität Informationen hinzuzufügen. Daher werden Systeme dieser Art oft mit dem Sammelbegriff “Augmented Reality” (“Erweiterte Realität”) bezeichnet.

Die Anwendungen sind vielfältig. Eine einfache Umsetzung ist die mittlerweile verbreitete Idee des elektronischen Reiseführers, der über ein Lokalisierungssystem die Position des Benutzers erkennt und über diese nähere Informationen bereitstellt. Eine weitere Möglichkeit sind „virtuelle Post-It-Zettel“ [18], die über mobile Eingabegeräte an Orten oder Objekten im Raum hinterlassen werden können, um sie später Anderen, die den betreffenden Bereich betreten, anzuzeigen.

Diese Notizen könnten auch vom System selbst generiert werden, zum Beispiel um Informationen über defekte Geräte anzuzeigen oder auf Konferenzen den Teilnehmern die Identifikation ihrer Gesprächspartner zu erleichtern, nachdem diese über Bilderkennungssysteme ermittelt wurde.

Kontextabhängige Erweiterung war im Modell von Schilit noch nicht vorgesehen, sondern wurde erst von Pascoe eingeführt [17].

4 Modellierung des Kontextes

Nach den vorhergehenden Betrachtungen, die sich mit der Entwicklung und Bedeutung des Kontextes für ein Computersystem beschäftigt haben, stellt sich nun die Frage, wie dieser innerhalb des Systems modelliert werden kann und wie er zwischen verschiedenen Stationen übertragen wird.

Die größte Verbreitung haben XML[3]-Modelle erlangt, da sie einfach zu entwickeln und leicht zu erweitern sind, dabei jedoch nur geringe Voraussetzungen haben. Ein weiterer Vorteil ist, dass XML-Beschreibungen für Menschen lesbar sind. Eines dieser Modelle wird im Folgenden beschrieben.

4.1 Ein XML-Modell

Schmidt entwirft in [19] ein einfaches XML-basiertes Modell, in dem Kontextzustände überprüft und behandelt werden können. In diesem Modell werden über prädikatenlogische Ausdrücke Bedingungen an Zustände und Sensordaten formuliert und nach der Auswertung dieser Ausdrücke eine Aktion ausgelöst.

Ein ähnliches Modell wird von Dey & Abowd in [20] vorgeschlagen und für einen Konferenz-Assistenten umgesetzt. Eine weitere Umsetzung der Modellierung des Kontextes über XML findet sich in [21].

Abbildung 2 zeigt den Ablauf in diesem Modell: Zunächst werden die einzelnen Gruppen von Bedingungen auf ihren Wahrheitswert überprüft. Abhängig vom gewünschten Ergebnis kann von einer Gruppe verlangt werden, dass alle ihre Variablen wahr evaluieren (“Alle”-Bedingungen), dass mindestens eine wahr ist (“Eine”-Bedingungen) oder dass keine von ihnen erfüllt ist (“Keine”-Bedingungen). Jede dieser Klassen kann beliebig oft vertreten sein.

3 Extensible Markup Language

Nachdem alle Gruppen durchlaufen sind, wird der gesetzte Trigger geprüft. Trigger haben einen von drei diskreten Werten: Betreten, Erhalten und Verlassen des Zustands.

Beim Betreten und Verlassen kann zusätzlich zum einfachen Trigger noch eine Verzögerungszeit angegeben werden, die bestimmt, wie lange der nach dem Trigger erreichte Zustand erhalten werden muss, um ihn anzuerkennen.

Wenn der Trigger ebenfalls „wahr“ zurückliefert und eine Verzögerungszeit angegeben ist, wird der Zustand der Gruppen nach dem Verstreichen dieser Zeit erneut überprüft und danach die Aktion ausgelöst.

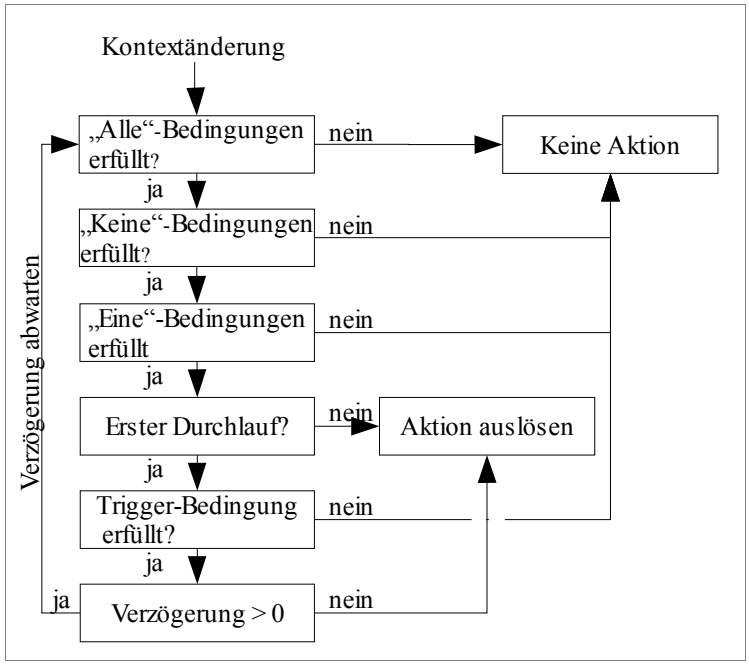


Abbildung 2: Flussdiagramm zur Kontextbehandlung im XML-Modell

Das Modell bietet die Kernelemente der Modellierung von Kontexten, Zustand und Reaktion, und erweitert sie um die Triggerwerte Betreten und Verlassen, um das Skripting von Aktionen zu vereinfachen. Um das Modell einfach zu halten, wurde jedoch über die Möglichkeit der Verzögerungszeit hinaus keine Form von Ereignisbehandlung vorgesehen. Wie auch der Übermittlungsmodus hängt diese von der Implementierung ab.

Für diese bietet sich das HTTP[4] an, das als paketorientiertes Protokoll alle Anforderungen erfüllt, die die Übermittlung von Kontextinformationen stellt. Da es auf dem allgegenwärtigen TCP/IP[5] aufbaut, ist ein HTTP-Übermittlungssystem sehr einfach zu implementieren [20].

4 Hypertext Transfer Protocol
5 Transmission Control Protocol/Internet Protocol

Andere Möglichkeiten, diese Übertragung vorzunehmen wären zum Beispiel CORBA[6] oder Java RMI[7], die jedoch einen höheren Implementierungsaufwand (ORB-Interpreter) oder eine im Allgemeinen unerwünschte Bindung an eine spezielle Programmiersprache mit sich bringen.

Ein erläutertes Codesegment verdeutlicht die Vorgehensweise in diesem Modell.

Beispiel XML-Modell:

```
<context_interaction>
     <context>
          <group match='all'>
               persons.present
               sound.volume.speech
          </group>
          <group match='one'>
               movie.running
               speakers.single
          </group>
          <group match='one'>
               schedule.meeting
               schedule.conference
          </group>
     </context>
     <action trigger='enter' time='60'>
          context.social.presentation
     </action>
</context_interaction>
```

Dieser Codeauschnitt lässt zunächst in der “Alle”-Gruppe überprüfen, ob sich in der Umgebung des Benutzers Personen befinden, ob der Lautstärkepegel dem eines Vortrags oder einer Rede entspricht, und ob nur ein einzelner der Anwesenden spricht. Anschliessend überprüft die “Eine”-Gruppe, ob ein Filmwiedergabegerät, etwa ein Beamer, in der Umgebung läuft, oder ob ein einzelner Sprecher einen Vortrag hält. Eine weitere “Eine”-Gruppe überprüft dann den Terminkalender des Benutzers auf ein eingetragenes Meeting oder eine Konferenz.

Wenn alle Gruppen-Bedingungen erfüllt sind, wird der schliesslich der Trigger “Betreteten” ausgelesen, der eine Verzögerungszeit von 60 Sekunden angegeben hat. Wenn die Bedingungen also neu um den Benutzer aufgekommen sind, danach aber für 60 Sekunden stabil anhalten, wird die Aktion ausgelöst: Das System erkennt den sozialen Kontext “Präsentation”.

Dieser Code könnte zum Beispiel in einem Kontext-Interpreter zum Einsatz kommen, der sich der Ergebnisse niedrigstufigerer Interpreter bedient, um eine sichere Aussage über den sozialen Kontext des Benutzers treffen zu können.

Es wird deutlich, dass auch Elemente aus den anderen beiden Kategorien, elektronischer und physischer Kontext, für diese Ermittlung hilfreich sein können, in diesem Fall durch die Anwesenheit weiterer Personen und die Aktivität von Anzeigegeräten.

---

6 Common Object Request Broker Architecture

7 Remote Method Invocation

5 Kontextbewusstsein und Autonomic Computing

Welche Bedeutung hat die Entwicklung der kontextbewussten Systeme nun auf das Paradigma des Autonomic Computing? Dieser Abschnitt betrachtet die anderen Teilgebiete des Autonomic Computing und zeigt die neu eröffneten Möglichkeiten. Die einzelnen Teilgebiete sind in IBM´s “Perspective on the State of Information Technology” [22] beschrieben.

5.1 Selbst-Bewusstsein

Offensichtlich ist die Fähigkeit eines Systems, die eigenen Komponenten zu kennen und bei Bedarf neue anzunehmen eng verknüpft mit dem elektronischen Kontext des Systems. Da für das Selbst-Bewusstsein im Sinne des Autonomic Computing Interaktion mit benachbarten IT-Systeme notwendig ist, werden auch hier Methoden des Kontextbewusstseins benötigt.

5.2 Selbst-Optimierung

Um die Fähigkeit zur Selbst-Optimierung mit größtmöglicher Effizienz einsetzen zu können, benötigt das System jede Datenquelle, die ihm zur Verfügung steht, um seinen gegenwärtigen Zustand zu erkennen. Hier werden hohe Ansprüche an die Subsysteme des physischen Kontextes gestellt, die den Zustand des Systems selbst, sowie den seiner Umgebung zuverlässig erkennen müssen.

5.3 Selbst-Heilung

Da die Selbst-Heilung primär im Inneren des Systems stattfindet, kann der Bereich nur indirekt von Methoden des Kontextbewusstseins profitieren. Kontextbewusstsein kann die Selbstheilung unterstützen, indem zum Beispiel Administratoren ausfindig gemacht werden und die ideale Methode, sie von Problemen zu benachrichtigen, ermittelt wird. Über das Erkennen des gegenwärtigen oder (zum Beispiel über Terminplaner) zukünftigen Kontextes können eher benötigte Bereiche des Systems ermittelt und für eine dringlichere Heilung eingeplant werden.

5.4 Selbst-Schutz

Wie schon im Bereich der Selbst-Heilung kann Kontextbewusstsein neue Möglichkeiten der Kontaktaufnahme ermöglichen. Darüber hinaus ist es in einem kontextbewussten System schwerer, sich falsch zu identifizieren und erweiterte Rechte zu erlangen, da mehr Mittel als nur Benutzername und Passwort, die vergleichsweise leicht beschafft werden können, zur Identifizierung zur Verfügung stehen.

5.5 Unsichtbare Komplexität

Der letzte Punkt in IBM´s “Perspective” lässt sich unter dem Schlagwort “Unsichtbare Komplexität” zusammenfassen. Es wird gefordert, dass ein Computersystem in Zukunft seine Komplexität vor dem Benutzer versteckt, ohne ihn der Möglichkeiten zu berauben, und ihm gleichzeitig durch neue Eingabemöglichkeiten und Erkennung seiner Absichten entgegenkommt.
Diese Eigenschaften, unsichtbare Komplexität und Proaktivität, sind ohne Kontextbewusstsein nicht denkbar. Das System muss mehr als bisher über den Benutzer wissen, frühzeitig seine Absichten erkennen, um proaktiv statt reaktiv agieren zu können.

Gleichzeitig wird die erweiterte Sensorik der Kontexterkennung benötigt, um neue Formen von Benutzerschnittstellen schaffen zu können, die keine direkten Eingaben in Form von Mausklicks und Tastendrücken mehr benötigen, sondern anhand neuer Methoden wie Sprach- oder Gestenerkennung die Wünsche des Benutzers erfüllen können und so intuitiver zu bedienen sind [19].

6 Abschluss und Zusammenfassung

Dieses Papier hat die durch Kontextbewusstsein eröffneten Möglichkeiten und ihre Bedeutung für den Bereich des Autonomic Computing aufgezeigt. Durch Verwirklichung der Ideen zur Schaffung neuer Benutzerschnittstellen und “mitdenkender” Rechner kann dem Benutzer in Zukunft viel unnötige Last abgenommen werden.

Autonomic Computing spielt eine wichtige Rolle auf diesem Weg, und Kontextbewusstsein ist ein zentraler Teilbereich, wie der vorhergehende Abschnitt gezeigt hat. Jeder der Schlüsselpunkte des Autonomic Computing kann von den Fähigkeiten des Systems, Kontext zu erkennen und mit ihm zu interagieren, profitieren, wenn auch teilweise nur indirekt. Einzig zur Unterstützung der Selbstkonfiguration kann, über den Rahmen der bereits durch die Verbesserung des Selbstbewusstseins geschaffenen Möglichkeiten hinaus, nichts beigetragen werden.

Der Computer kann so immer mehr zu dem werden, was er sein sollte: Ein Werkzeug, das den Menschen unterstützt, anstatt einen Grossteil seiner Zeit und Aufmerksamkeit zu verschlingen.

Referenzen

1. Anind K. Dey and Gregory D. Abowd: Towards a Better Understanding of Context and Context-Awareness, CHI Conference on Human Factors in Computing Systems, 2000
   http://www.cc.gatech.edu/fce/contexttoolkit/chiws/Dey.pdf
2. Christopher Lueg: Operationalizing Context in Context-Aware Artifacts: Benefits and Pitfalls, Informing Science Vol. 5, 2002
   http://informingscience.org/Articles/Vol5/v5n2p043-047.pdf
3. Kristof Van Laerhofen and Ozan Cakmakci: What Shall We Teach Our Pants?, Fourth International Symposium on Wearable Computers, 2000
   http://www.comp.lancs.ac.uk/~kristof/old/papers/iswc_2000.pdf

4. Tom Gray, Ramiro Liscano, Barry Wellman, Anabel Quan-Haase, T. Radhakrishnan and Yongseok Choi: Context and Intent in Call Processing, Feature Interactions in Telecommunications and Software Systems, 2003
http://www.chass.utoronto.ca/~wellman/publications/callprocessing/gray-fiw-2b-bw-tg-modified2point4-sixth-draft.pdf
5. Andreas Zeidler and Roger Kehr: Yet another Answer to "Where am I"?, Unpublished, 2000
http://www.dvs1.informatik.tu-darmstadt.de/DVS1/research/infubidis/publications/TR-2000-05-01.pdf
6. Anind K. Dey: Context-Aware Computing: The CyberDesk Project, AAAI Spring Symposium, 1998
http://www.cc.gatech.edu/fce/cyberdesk/pubs/AAAI98/AAAI98.html
7. Bill Schilit, Norman Adams and Roy Want: Context-Aware Computing Applications, IEEE Workshop on Mobile Computing Systems and Applications, 1994
http://seattleweb.intel-research.net/people/ schilit/wmc-94-schilit.pdf
8. Albrecht Schmidt, Michael Beigl and Hans-W. Gellersen: There is more to Context than Location, Computers and Graphics Vol. 23, 1999
http://www.teco.uni-karlsruhe.de/~albrecht/publication/draft_docs/context-is-more-than-location.pdf
9. Bluetooth sig, Inc.: Specification of the Bluetooth System – Wireless connections made easy,,1999
https://www.bluetooth.org/foundry/specification/document/Bluetooth_Core_10_B/en/1/Bluetooth_Core_10_B.pdf
10. Apple Computer Inc.: Firewire 800 Technology Brief,,2003
http://a992.g.akamai.net/7/992/51/93b28da05d3103/www.apple.com/firewire/pdf/FireWire_Tech_Brief-a.pdf
11. USB Group: Universal Serial Bus Specification, Revision 2.0,,2000
http://www.usb.org/developers/docs/usb_20.zip
12. Manuel Román, Christoper K. Hess, Renato Cerqueira, Anand Ranganathan, Roy H. Campbell and Klara Nahrstedt: Gaia: A Middleware Infrastructure to Enable Active Spaces, IEEE Pervasive Computing, 2002
http://choices.cs.uiuc.edu/gaia/papers/GaiaSubmitted3.pdf
13. Roy Want, Andy Hopper, Veronica Falcao and Jonathan Gibbons: The Active Badge Location System,ACM Transactions on Information Systems, vol. 10, 1992
http://www.cs.colorado.edu/~rhan/CSCI_7143_002_Fall_2001/Papers/Want92_ActiveBadge.pdf
14. Jalal Al-Muhtadi, Anand Ranganathan, Roy Campbell and M. Dennis Mickunas: A Flexible, Privacy-Preserving Authentication Framework for Ubiquitous Computing Environments, IEEE Workshop on Mobile Computing Systems and Applications, 1994
http://choices.cs.uiuc.edu/~almuhtad/flex-auth.pdf
15. Hans-W. Gellersen, Albrecht Schmidt and Michael Beigl: Adding Some Smartness to Devices and Everyday Things, IEEE Workshop on Mobile Computing Systems and Applications, 2000
http://www.teco.edu/~michael/publication/gellersen_wmcsa_2000_smart_artifacts.pdf
16. Martin Jonsson: Context Shadow: A Person-Centric Infrastructure for Context Aware Computing, Workshop on Artificial Intelligence in Mobile Systems, 2002
http://www.dsv.su.se/FEEL/zurich/Item_9-Context_Shadow-A_person-centric_Infrastructure_for_context_aware_computing.pdf
17. Jason Pascoe: Adding Generic Contextual Capabilities to Wearable Computers, Second International Symposium on Wearable Computers, 1998
http://c2000.cc.gatech.edu/classes/cs8113c_99_spring/readings/pascoe.pdf
18. Jason Pascoe: The Stick-e Note Architecture: Extending the Interface Beyond the User,
http://www.iuiconf.org/97pdf/1997-002-0040.pdf

19. Albrecht Schmidt: Implicit Human Computer Interaction Through Context, Personal Technologies Vol. 4, 2000
www.teco.uni-karlsruhe.de/~albrecht/publication/ draft_docs/implicit-interaction.pdf
20. Anind K. Dey, Daniel Salber, Masayasu Futakawa and Gregory D. Abowd: An Architecture To Support Context-Aware Applications, GVU Technical Report GIT-GVU-99-23, 1999
ftp://ftp.cc.gatech.edu/pub/gvu/tr/1999/99-23.pdf
21. Daniel Salber and Gregory D. Abowd: The Design and Use of a Generic Context Server, GVU Technical Report GIT-GVU-98-32, 1998
http://www.cc.gatech.edu/fce/contexttoolkit/pubs/pui98.pdf
22. IBM Corp., Autonomic Computing: IBM's Perspective on the State of Information Technology, AGENDA, 2001
http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf

Weitere Quellen

i. Daniel Salber, Anind K. Dey and Gregory D. Abowd: The Context Toolkit:Aiding the Development of Context-Enabled Applications, Workshop of Software Engineering for Wearable and Pervasive Computing, 2000
http://www.cc.gatech.edu/fce/contexttoolkit/pubs/chi99.pdf
ii. Mari Korkea-aho: Context-Aware Applications Survey, Internetworking Seminar (Tik-110.551), Helsinki University of Technology, 2000
http://www.hut.fi/~mkorkeaa/doc/context-aware.html
iii. Barry Brumitt & JJ Cadiz: “Let There Be Light!” Comparing Interfaces for Homes of the Future, ,2000
iv. Asim Smailagic, Daniel P. Siewiorek, Joshua Anhalt, David Kogan and Yang Wang: Location Sensing and Privacy in a Context Aware Computing Environment, Pervasive Computing, 2001
v. Jürgen Bohn, Vlad Coroama, Marc Langheinrich, Friedemann Mattern and Michael Rohs: Disappearing Computers Everwhere – Living in a World of Smart Everyday Objects, Proc. of New Media, Technology and Everyday Life in Europe Conference, 2003
http://www.inf.ethz.ch/personal/bohn/papers/emtel.pdf
vi. Peter J. Brown: The Stick-e Document: A Framework for Creating Context-Aware Applications, Electronic Publishing, 1996
vii. Georgie Institute of Technology: The Aware Home
http://www.cc.gatech.edu/fce/ahri/

Aktuelle Technologien zur Realisierung dienstorientierter Architekturen

Dominik Strecker

Zusammenfassung. Moderne Softwareanwendungen können ihre hohe Komplexität häufig nur durch weltweite Kooperation von spezialisierten Lösungen erreichen. Aus dieser Entwicklung entsteht das Bedürfnis nach einer neuen Softwarearchitektur. Mit Hilfe *dienstorientierter Architekturen* sollen verteilte Systeme zur Grundlage der Softwareentwicklung werden. Anwendungen stellen dazu ihre Funktionalitäten implementierungsunabhängig in Form von *Diensten* zur Verfügung. Mit Hilfe formaler Beschreibungen der Dienste und ihrer Schnittstellen soll sogar autonome Interaktion in beliebigen heterogenen Systemen ermöglicht werden. Verschiedene Technologien versuchen diese Idee zu verwirklichen, legen aber unterschiedliche Einsatzszenarien zu Grunde.
Inhalt dieser Arbeit ist die Darstellung der dienstorientierten Architektur sowie die Präsentation einiger exemplarischer Technologien. Abschließend folgt ein Vergleich dieser Technologien.

1 Einleitung

> *"The real switch in thinking is that we might be able to make these services available to 'silicon-based life forms'."*
>
> James Gosling

Im Prozess der Softwareentwicklung generiert der Entwickler aus einem Problem der realen Welt ein formales, für einen Rechner verständliches Modell. Durch Wiederverwendung und Anpassung alter Modelle und schnell wachsende Leistungspotentiale der Rechner werden diese Modelle immer komplexer. Das menschliche Fassungsvermögen bleibt dagegen begrenzt. Um bei komplexen Softwareprojekten den Überblick behalten zu können, muss die Komplexität deshalb eingedämmt werden. Dazu wird die Implementierung des Modells so aufgeteilt, dass die Teile als „black boxes" ihre Komplexität verstecken (*Geheimnisprinzip*) und nun die kleinste zu betrachtende Einheit bilden (*Modularisierung*). Es bildet sich so eine neue Schicht, die von der darunterliegenden abstrahiert, weniger Teile enthält und dadurch überschaubarer ist [1].

Die erste Abstraktion in der Geschichte der Softwareentwicklung war die Nutzung von *Unterprogrammen*, die sich später zu *Funktionen* entwickelten [2]. Es folgten unter anderem *Objekte* und *Komponenten*. Ein Objekt einer objektorientierten Architektur ist im Idealfall ein formales Abbild genau eines Objektes der realen Welt. Objekte haben Eigenschaften und können Operationen

ausführen. Sie bilden eine geschlossene Einheit und verstecken ihre Implementierungsdetails. Für große Anwendungen sind sie allerdings oft zu feinkörnig. Daher kapseln Komponenten mehrere Objekte zu einem System mit gemeinsamer Aufgabe und einheitlicher Schnittstelle, die dann als (gröbere) Einheit betrachtet werden kann. Dienste – oft wird auch im Deutschen der äquivalente englische Begriff *Services* benutzt – sollen nun wiederum die Probleme von Komponenten beheben (siehe unten Seite 21).

2 Dienstorientierte Architektur

Die Entwicklung hin zu Diensten verlangt eine neue Softwarearchitektur [2][3]. *Dienstorientierte Architekturen* (engl. service-oriented architectures, Abk. SOA) basieren auf der Vereinbarung und Verwendung von Diensten. Die Bezeichnung Dienst sorgt leicht für Verwirrung, weil sie in zwei verschiedenen Bedeutungen benutzt wird. Häufig wird von einem Dienst als Softwarekomponente mit speziellen Eigenschaften gesprochen. Ein Dienst im Sinne der SOA ist aber keine Software, sondern die (Dienst-)Leistung einer Software. Er ist eine exakt spezifizierte Funktionalität, die über eine exakt spezifizierte Schnittstelle angefordert werden kann. Der Dienst abstrahiert daher idealerweise vollkommen von der tatsächlichen technischen Umsetzung. Sämtliche Details der Implementierung bleiben dem Nutzer verborgen.

Dem Komplexitätsproblem begegnet die SOA mit der weiteren Vergröberung ihrer kleinsten Teile. Dienste als Elemente der SOA können auf beliebige Implementierungen aufsetzen – sogar auf andere Dienste. Sie können deshalb selbst beliebig komplex werden.

Das größte und ehrgeizigste Ziel der SOA ist die Ermöglichung autonomer Interaktion zwischen Diensten. Sie sollen in die Lage versetzt werden, ein gegebenes Problem durch selbstverwaltete Kooperation zu lösen. Dazu müssen sie allerdings die Funktionalität anderer Dienste auf semantischer Ebene einschätzen können. Wichtiger Baustein einer SOA ist also die formale Beschreibung der Semantik von Diensten.

Die Komponenten einer SOA werden allesamt als Dienste begriffen. Auch der menschliche Benutzer bildet da keine Ausnahme. Um unterschiedliche Verhaltensweisen zu veranschaulichen, definiert die SOA so genannte Rollen. Jeder Dienst übernimmt bei jeder Interaktion mindestens eine solche Rolle, es können allerdings mehrere Dienste die gleiche Rolle besetzen. Abb. 1 zeigt die Rollen und ihre dazugehörigen Interaktionen in der dienstorientierten Architektur.

Nimmt z. B. ein Dienst einen anderen in Anspruch, so ergibt sich daraus folgende Rollenverteilung: Der erstgenannte Dienst übernimmt die Rolle des *Dienstnachfragers*, der andere ist ein *Dienstanbieter*. Bei bidirektionaler Kommunikation sind beide Dienste sowohl Dienstnachfrager als auch Dienstanbieter. Die genauen Aufgaben der einzelnen Rollen sind für die Struktur der SOA von grundlegender Bedeutung [2].

– **Dienstanbieter**
  Der Dienstanbieter kann seinen Dienst bei einem oder mehreren *Dienstver-*

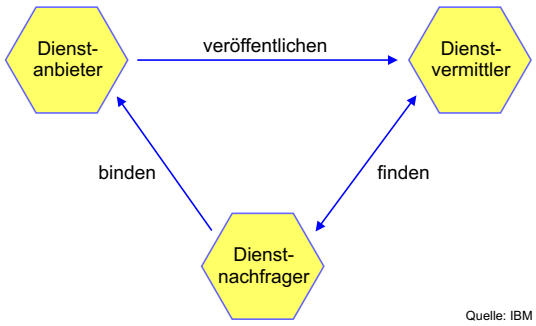


Abb. 1. Rollen und Interaktionen der dienstorientierten Architektur

mittlern veröffentlichen. Dabei werden sowohl formale als auch natürlichsprachliche Beschreibungen des Dienstes sowie seine (eindeutige) Adresse beim Dienstvermittler hinterlegt.

- **Dienstvermittler**
  Der Dienstvermittler sammelt Beschreibungen von Diensten, die ihm für gewöhnlich von Dienstanbietern zukommen und bietet Suchmöglichkeiten (*finden*) über seinen Informationen an. Auch weitere Informationen, z. B. über den Dienstanbieter selbst, können hier hinterlegt werden.
- **Dienstnachfrager**
  Der Dienstnachfrager sucht beim Dienstvermittler nach einem benötigten Dienst. Neben der eigentlichen Beschreibung kann er auch nichtfunktionale Parameter wie z. B. Güte oder Preis in die Suche miteinfließen lassen. Kann ihm der Dienstvermittler einen passenden Dienst zuteilen, *bindet* der Dienstnachfrager ihn dynamisch über den Dienstanbieter in seine Ausführung ein.

Die SOA hat vielfältige Vorteile gegenüber Architekturen mit herkömmlichen Komponenten. Lockere Verbindungen der Dienste untereinander, konsequente Kapselung und die Ausrichtung auf das Internet sollen verteilte Systeme zur Grundlage von beliebigen Softwaresystemen machen. Im Einzelnen erscheinen die Neuerungen unspektakulär, ihr Zusammenspiel hat aber dennoch revolutionäre Konsequenzen [3]:

- **Interoperabilität**
  Komponenten haben nur eine – meist proprietäre – Schnittstelle, können also nur über spezielle Protokolle kommunizieren. Vielen Komponenten ist eine direkte Interaktion deshalb unmöglich. Dienste haben dagegen i. A. mehrere oder zumindest standardisierte Schnittstellen. Idealerweise kann auf diese Weise jeder Dienst mit jedem anderen interagieren.
- **Erreichbarkeit im Internet**
  Komponenten werden für gewöhnlich nur auf einer Maschine oder in einem lokalen Netz verwendet. Dienste sind definitionsgemäß per Internet und damit von jedem Ort aus zu erreichen.

– **Dynamische Bindung**
  Komponenten müssen in der Regel bei der Implementierung statisch eingebunden werden. Im Gegensatz dazu fördert die SOA dynamische Kooperation von Diensten zur Laufzeit.
– **Trennung von Funktionalität und Implementierung**
  Komponenten werden auf einer relativ technischen Ebene betrachtet. Die wichtigste Eigenschaft von Diensten ist dementgegen die vollkommene Abstraktion von der technischen Verwirklichung. Ein Dienst kann von verschiedenen Implementierungen, so genannten *Agenten* oder *Instanzen* erbracht werden. Ebenso kann ein Agent auch mehrere Dienste anbieten.
– **Kommunikationsstruktur**
  Ein wichtiger Unterschied in der praktischen Anwendung ist die Art des Datenaustausches. Komponenten werden üblicherweise häufig mit kleinen Datenpaketen aufgerufen. Man spricht auch von einem „chatty interface“ (zu dt. etwa *schwatzhafte Schnittstelle*). Dienste nehmen hingegen i. A. sämtliche zur Erfüllung ihrer Aufgabe benötigten Daten in einer einzelnen Nachricht entgegen und senden das (womöglich umfängliche) Ergebnis in einer weiteren zurück (die Schnittstelle heißt entsprechend „chunky interface“ (zu dt. etwa *klotzende Schnittstelle*).

Die SOA ist eine allgemeine Architektur und lässt viele Möglichkeiten der konkreten Implementierung zu. In den nächsten Abschnitten werden exemplarisch einige auf der SOA basierende Technologien vorgestellt.

3 Jini

Überblick
Jini[4] erweitert die Programmiersprache Java von Sun Microsystems[5]. Auch wenn laut Spezifikation Jini nicht auf räumliche Bereiche begrenzt ist, liegt der Einsatzbereich doch vorwiegend im lokalen Bereich. Speziell mobile Geräte sollen von Jini profitieren. Es soll z. B. einem tragbaren Rechner möglich sein, per Funk ad hoc einen benachbarten Drucker zu nutzen, sobald er sich in die Nähe des Druckers bewegt. Tragbarer Rechner und Drucker werden dann Teil einer so genannten *Jini Community* oder kurz eines *Djinn*.

Architektur
Jinis Architektur[6] lehnt sich stark an die SOA an. Jini ist insofern konsequent als jeder Teilnehmer am Djinn ein Dienst sein muss, unabhängig von der Implementierung in Hard- oder Software. Die Beziehungsstruktur zwischen Diensten kann beliebig komplex werden, vor allem eine transitive Kommunikationsstruktur ist üblich. In diesem Falle fungiert ein Dienstnachfrager auch als Dienstanbieter und leitet an ihn gestellte Anfragen selbst weiter an andere Dienstanbieter. Auf diese Art kann eine Situation entstehen, in der der auslösende Dienst oder Benutzer schließlich nicht mehr wissen kann, wie und wo sein Dienst wirklich ausgeführt wird. Die Dienste arbeiten dann autonom.

Die Aufgaben des Dienstvermittlers der SOA übernimmt ein so genannter *Lookup-Service* (siehe Abb. 2). Er kann je nach Situation auch als *Service Provider* (Dienstanbieter) oder *Client* (Dienstnachfrager) auftreten, so wie generell jeder Dienst mehrere Rollen übernehmen kann. Seine primäre Aufgabe ist die dauerhafte Verwaltung einer Liste der Dienste im Djinn. Diese Liste muss ständig aktualisiert werden, um die verfügbaren Dienste korrekt widerzuspiegeln. Dazu greift der Lookup-Service auf das so genannte *Leasing* zurück. Jeder Dienst muss sich vor Ablauf einer Frist beim Lookup-Service zurückmelden, um in der Liste der verfügbaren Dienste weiter geführt zu werden. Wenn der *Lease* abläuft, wird der Dienst aus der Liste entfernt. Leasing kann zwischen beliebigen Diensten stattfinden, den Prototyp stellt allerdings der Lookup-Service dar. Durch Leasing wird in der Jini Community Selbstheilung möglich gemacht.

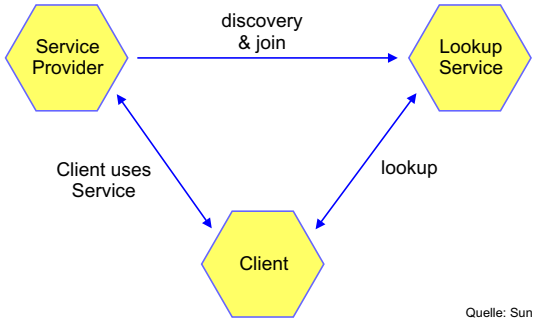


Abb. 2. Rollen und Interaktionen von Jini

Es gibt zwei Möglichkeiten, einen Dienst beim Lookup-Service anzumelden. Entweder veröffentlicht der Service Provider von sich aus seinen Dienst, oder der Lookup-Service bittet die Service Provider, sich zu registrieren. Die Veröffentlichung eines Dienstes teilt sich bei Jini in zwei Teile. Zuerst muss der Service Provider einen Lookup-Service finden. Dazu stellt Jini ein Protokoll namens *discovery* bereit. Der Service Provider weist dabei per Rundruf jeden Lookup-Service im lokalen Netz an, sich zu melden. Anschließend tritt der Service Provider dem Djinn per *join* bei. Umgekehrt funktioniert der so genannte *peer lookup*. Hier fragt der Lookup-Service im lokalen Netz nach Service Providern, die sich registrieren wollen. Der Beitritt zum Djinn verläuft danach analog: Der Service Provider lädt die Schnittstelle des Dienstes, alle zum Aufruf notwendigen Methoden und eventuelle Beschreibungen als gekapseltes Objekt (das *Service Object*) in einen oder mehrere Lookup-Services.

Der wichtigste Schritt ist nun, einem Client zu seinem gewünschten Dienst zu verhelfen. Dazu stellt der Client bei einem Lookup-Service eine spezifizierte Anfrage (*lookup*). Die Spezifikation enthält eine Schablone der Schnittstelle des gesuchten Dienst und gegebenenfalls weitere gewünschte Merkmale. Bei erfolgreicher Beantwortung der Anfrage wird eine Kopie des Service Objects an

den Client weitergeleitet. Gibt es im lokalen Netz keinen Lookup-Service, kann auch der Client per peer lookup versuchen, Service Provider zu finden und deren Service Objects einzubinden.

Der letzte Schritt ist die direkte Kommunikation zwischen Client und Service Provider (*Client uses Service*). Der Client benutzt dazu Remote Method Invocation (RMI), ruft also die Methoden des Service Objects auf. Die weitere Kommunikation findet dann ausschließlich zwischen Service Object und Service Provider statt. Zwischen den beiden Extremen der entfernten Ausführung beim Service Provider und der lokalen Ausführung im Service Object auf dem Client ist jede Kombination möglich. Diese Kombinationen werden als *smart proxies* bezeichnet. Durch die Kapselung der gesamten Kommunikation im Service Object bleiben dem Client die Implementierungsdetails vollständig verborgen. Gleichzeitig ist gewährleistet, dass Service Object und Service Provider korrekt zusammenarbeiten, weil das aktuelle Service Object zeitnah vom Service Provider gestellt wird.

Weitere wichtige Konzepte von Jini bilden *Sicherheitsmaßnahmen*, *Ereignisverarbeitung* und *Transaktionsverwaltung*:

- **Sicherheitsmaßnahmen**
  Neben der Funktion als Fernzugriffsvertreter hat jedes Service Object auch die Möglichkeit als Schutzwand aufzutreten. Dazu entscheidet es anhand einer *Zugriffskontrollliste* (engl. *access control list*) und der *Authentifizierung* des Clients, ob eine Ausführung stattfinden darf (*Autorisierung*). Die Authentifizierung kann dabei geschachtelt erfolgen, der Client übernimmt dann die Authentifizierung wiederum von seinem Client, bzw. schlussendlich dem Benutzer.
- **Ereignisverarbeitung**
  Eine sehr wichtige Funktionalität von verteilten Anwendungen stellen verteilte Ereignisverarbeitungssysteme dar. Jini ermöglicht jedem Dienst, sich bei anderen Diensten in Benachrichtigungslisten einzutragen, um bei Eintritt des Ereignisses benachrichtigt zu werden. Dadurch werden Reaktionen jedes beliebigen Dienstes auf Veränderungen im Djinn ermöglicht.
- **Transaktionsverwaltung**
  Um bei einer Abfolge von Operationen – auch verteilt auf mehrere Dienste – die Konsistenz des Systems zu wahren, stellt Jini eine einfache Transaktionsverwaltung bereit. Durch einen *two-phase commit* kann sichergestellt werden, dass stets entweder alle oder keine der Operationen ausgeführt werden. Jini gibt dazu eine Schnittstelle vor, die von den betroffenen Diensten implementiert werden kann.

Kritik

Das Hauptproblem von Jini ist die Festlegung auf eine proprietäre unterliegende Technologie. Kommunikation zwischen Diensten ist nur mir Java RMI durchführbar. In heterogenen Systemen kann Jini deshalb nur rudimentär eingesetzt werden – eine zentrale Forderung der SOA wird somit nicht umgesetzt.

In punkto Sicherheit bestehen gravierende Mängel. Über Autorisierung hinausgehende Sicherheitsfragen werden nicht beantwortet. So ist z. B. nicht klar,

wie der Service Provider bei einem Lookup-Service authentifiziert wird, damit nicht geheime Daten des Clients an einen gefälschten Service Provider gelangen. Auch hat der Service Provider durch das Service Object beide Seiten der Kommunikation in der Hand. Ihm allein obliegt daher die Wahrung der Vertraulichkeit bei der Kommunikation, geheime Daten des Clients sind also auch durch Verschlüsselung nicht vor ihm zu schützen.

4 Bluetooth

Überblick

Bluetooth[7] ist eine Funktechnologie vor allem für mobile Geräte und wird entwickelt von der Bluetooth Special Interest Group (SIG)[8], einem Zusammenschluss von mehreren hundert Firmen und Privatleuten. Bluetooth-fähige Geräte sind bereits in der Lage über Dienste ad hoc miteinander zu kommunizieren und verteilt zu operieren. Es gibt bisher aber keine einheitliche Definition für Schnittstellen zu Diensten, so dass eine weit gehende Nutzung an proprietären Implementationen scheitert. Um einen Wildwuchs zu verhindern, wurde von der Bluetooth SIG mit dem *Service Discovery Protocol* (SDP) ein Standard vorgeschlagen, der Suche und Beschreibung von Diensten festlegt.

Architektur

Für Bluetooth gelten spezielle Einsatzbedingungen. Zum Einen sind die Dienste bei Bluetooth immer an Hardwarekomponenten (so genannte *Devices*) gebunden und somit nicht selbst mobil. Zum Anderen gibt es keine zentrale Instanz, die einen stationären und dauerhaften Dienstvermittler implementieren könnte. Trotzdem lässt sich die SOA bei Bluetooth wiederfinden.

Bluetooth kennt nur gleichberechtigte Geräte, die sowohl als *Client* als auch *Server* auftreten können (siehe Abb. 3). Client und Server sind Softwareanwendungen, die auf dem jeweiligen Gerät ausgeführt werden. Einzige Einschränkung ist dabei, dass pro Gerät höchstens eine Serveranwendung existiert. Alle Anwendungen auf dem jeweiligen Gerät greifen dann auf den gemeinsamen Server zu. Im Vergleich zur SOA stellen Server und Client den Dienstanbieter bzw. Dienstnachfrager dar. Alle Kommunikation muss wegen der speziellen Einsatzbedingungen zwischen diesen beiden Rollen stattfinden. Es kann keine direkte Kommunikation zwischen Dienstnachfrager und Dienstvermittler geben, wenn dieser beim Dienstanbieter implementiert ist.

Der Dienstvermittler kann in der Architektur von Bluetooth SDP nicht wie bei der SOA ein von den beteiligten Geräten unabhängiger Dritter sein. Dennoch soll die Funktionalität des Dienstvermittlers nicht fehlen, damit autonome Interaktion zwischen beliebigen Diensten stattfinden kann. Die Lösung von Bluetooth SDP ist einfach: Der Dienstanbieter ist selbst ein verteiltes System. Jedes dienstanbietende Gerät führt eine Liste seiner Dienste und übernimmt damit die Rolle eines (kleinen) Dienstvermittlers. Jeder Eintrag dieser *Service Records DB* enthält Informationen zu genau einem Dienst. Diese Informationen werden vorgegebenen oder selbstdefinierten *Service Attributes* zugeteilt. Die Suche nach

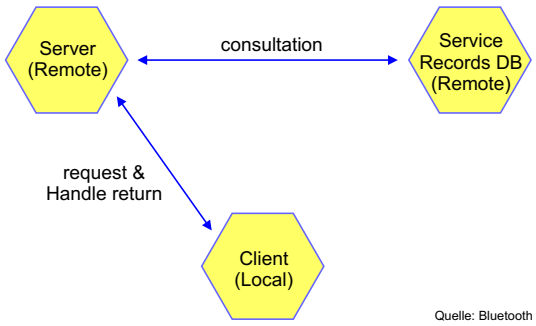


Abb. 3. Rollen und Interaktionen von Bluetooth SDP

einem Dienst vollzieht sich dann in den allermeisten Fällen nach den Werten der Service Attributes. Wichtige Vertreter aus der Menge der Service Attributes zeigt Tabelle 1.

Tabelle 1. Wichtige Service Attributes von Bluetooth SDP

Bezeichner	Beschreibung
ServiceClassIDList	Gibt den Typ des Dienstes als Liste von Klassen an
ServiceID	Referenziert eindeutig eine Instanz des Dienstes
ProtocolDescriptorList	Eine Liste der vom Dienst unterstützten Protokolle
ProviderName	Natürlichsprachlicher Name des Dienstanbieters
ServiceName	Natürlichsprachlicher Name des Dienstes
ServiceDescription	Natürlichsprachliche Beschreibung des Dienstes

Insbesondere die Zuordnung eines Dienstes zu einer oder mehreren Klassen ist für die Auswertung der Service Attributes wichtig. Je nach Klasse haben sie unterschiedliche semantische Bedeutung, stellen also eine Art Schnittstelle zur Beschreibung des Dienstes dar. Um jeden Dienst mindestens einer Klasse zuweisen zu können, wird von Bluetooth SDP eine erweiterbare *Klassenhierarchie* bereitgestellt. Während einfache Dienste sich in die Klassenhierarchie einordnen, können spezialisierte Dienste eine neue *Unterklasse* aus einer existierenden Klasse ableiten. Sie müssen dazu alle Service Attributes der *Oberklasse* übernehmen und zusätzlich neue definieren.

Die Suche nach einem Dienst erfolgt bei Bluetooth durch Filterung. Die Werte einiger Service Attributes werden als *Universally Unique Identifier* (UUID) angegeben. Ein UUID steht stellvertretend für eine spezifizierte Ausprägung des Service Attribute. Bluetooth SDP erlaubt die Suche nur in solchen Service Attributes, deren Wert ein UUID ist. Die Anfrage des Client (*request*) besteht allerdings nur aus einer nichtleeren Menge von UUIDs (dem *Service Search Pattern*). Auf die Übereinstimmung der Zuordnung von Werten zu Service Attri-

butes kommt es nicht an. Der Server prüft anschließend die Anfrage mit seiner Service Record DB gegen (*consultation*). Dabei gilt ein Dienst genau dann als auf das Suchmuster passend, wenn jede einzelne UUID des Service Search Pattern an beliebiger Stelle im zum Dienst gehörigen Service Record vorkommt.

Zu jedem auf die Anfrage passenden Service Record wird ein *Service Record Handle* eingerichtet und an den Client übergeben (*Handle return*). Damit kann er nun weitere Attribute auslesen und den Dienst in Anspruch nehmen.

Kritik

Bluetooth SDP konzentriert sich ausschließlich auf Beschreibung und Suche von Diensten. Die Frage nach der Einbindung wird nicht geregelt, sondern dem Entwickler überlassen. Mit dieser Strategie allein wird wohl keine Vereinheitlichung der Dienst-Landschaft von Bluetooth erreicht werden können. Weiterhin fehlen für verteilte Systeme zentrale Konzepte wie z. B. Ereignisbehandlung oder Transaktionsverwaltung.

Dienste in Bluetooth-Geräten sind nicht mit Sicherheit im Internet zu erreichen. Damit wird eine wichtige Forderung der SOA von Bluetooth nicht erfüllt.

Die Suchstrategie verspricht keine guten Ergebnisse, weil sie in der Praxis wohl zu unflexibel sein dürfte. Gerade für die Autonomie der Dienste ist es ungemein wichtig, eine unscharfe Suche anzubieten. Bluetooth vernachlässigt bei der Suche allerdings, wo die UUID auftaucht und wie oft sie auftaucht. Besonders schwer wiegt vor allem die Tatsache, dass ein Dienst schon dann aus dem Suchmuster herausfällt, wenn nur ein einziger UUID nicht in seinem Service Record vorkommt. Das gilt unabhängig von der Anzahl der zutreffenden UUIDs.

5 Web-Services

Überblick

Web-Services[9] sind ein offener Standard des World Wide Web Consortiums[10] (W3C) zur Vereinfachung der Kommunikation in heterogenen Systemen. Durch Standardisierung der Schnittstellen und Verwendung von *XML* (eXtensible Markup Language) als Datenaustauschformat soll die Integration zwischen beliebigen Diensten möglich sein. Die wichtigste Rolle spielt dabei das Katalogsystem *Universal Description, Discovery and Integration* (UDDI). Es stellt ein Verzeichnis, die so genannte *UDDI-Registry* zur Verfügung, in der Informationen über Web-Services und deren Anbieter vorgehalten werden.

Architektur

Die Web-Services Architektur (WSA) gilt als Prototyp der SOA und wird häufig sogar mit ihr verwechselt. Das mag daran liegen, dass Web-Services mit Abstand die bekannteste Technologie im Bereich der Dienstorientierung sind. Erstaunlicherweise ist die Idee hinter Web-Services nicht neu, sie wurde schon lange durch Technologien wie z. B. *Corba* umgesetzt. Das besondere an Web-Services ist hauptsächlich die Ausrichtung auf XML als interoperablem Datenformat.

Abb. 4 zeigt die Rollenverteilung in der WSA und die Interaktionen zwischen den Komponenten. Jede Kommunikation erfolgt über das *Simple Object*

Access Protocol (SOAP), einer einfachen Datendarstellung in XML. Die Vorgehensweise ist bei WSA und SOA praktisch identisch. Ein *Service Provider* stellt einen Web-Service zur Verfügung und meldet ihn bei einer UDDI-Registry – dem Dienstvermittler – an. Dazu hinterlegt er zumindest eine Beschreibung des Web-Service und seine tatsächliche Adresse. Der Web-Service selbst existiert weiterhin nur beim Service Provider und niemals bei der UDDI-Registry.

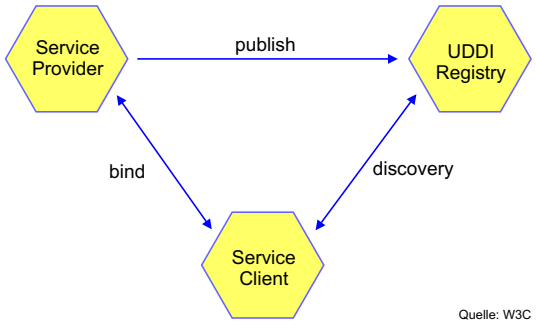


Abb. 4. Rollen und Interaktionen von Web-Services

Ein *Service Client* auf der Suche nach einem passenden Web-Service nimmt entweder direkt Kontakt mit einem Service Provider auf oder wendet sich an die UDDI-Registry. Ersteres ist nur möglich, wenn er Funktion und Adresse des aufgerufenen Web-Services kennt und führt dazu, dass die Beziehung zwischen den beteiligten Diensten statisch festgelegt wird. In allen anderen Fällen versorgt sich der Service Client mit Informationen aus der UDDI-Registry. Anschließend sendet er seine Anfrage per SOAP an den gefundenen Web-Service. Die gesamte Kommunikation verläuft verbindungslos über SOAP-Nachrichten.

UDDI[11] funktioniert wie ein elektronisches Telefonbuch. Service Clients „blättern“ in einem hierarchischen Katalog oder suchen gezielt nach einem Web-Service oder anderen Informationen. Neben Beschreibung und technischer Information zu Web-Services können in UDDI-Registries auch Informationen zu Service Providern abgelegt werden. Zur Trennung der Informationen stellt UDDI die vier Datenstrukturen *businessEntity*, *businessService*, *bindingTemplate* und *tModel* zur Verfügung. Bedeutungen und Beziehungen zwischen den Datenstrukturen zeigt Abb. 5.

– **businessEntity**
  Die Beschreibungen der Service Provider werden als businessEntities abgelegt. Neben allgemeinen Informationen, wie z. B. Kontaktadressen (auch als „white pages“ (zu dt. „Weisse Seiten“) bezeichnet) wird der Service Provider in einer hierarchischen Taxonomie eingeordnet. Dieses „Branchenverzeichnis“ wird auch als „yellow pages“ (zu dt. „Gelbe Seiten“) bezeichnet.

– **businessService**
  Die gleichen Informationen wie in businessEntities, jetzt aber bezogen auf Dienste, werden in businessServices aufbewahrt.
– **bindingTemplate**
  Das einem Web-Service zugeordnete bindingTemplate besitzt Informationen darüber, wie und wo man auf den Web-Service zugreifen kann. BindingTemplates bilden zusammen mit tModels die so genannten „green pages“ (zu dt. „Grüne Seiten“), also sämtliche technischen Informationen.
– **tModel**
  Technische Informationen zur Verwendung eines Web-Service sind in tModels angegeben. Das wichtigste ist die syntaktische Schnittstellendefinition. Sie wird i. A. mit der Web Service Definition Language[12] (WSDL) festgelegt. In WSDL kann sowohl die abstrakte Funktionalität eines Web-Services als auch seine konkrete Schnittstelle dargestellt werden.

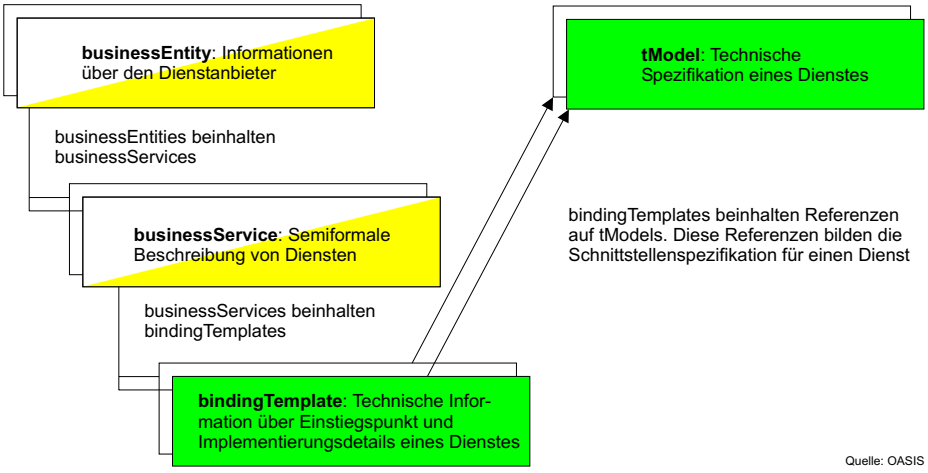


Abb. 5. Zentrale Datenstrukturen von UDDI

Vor allem Sicherheitsaspekte von Web-Services wurden ausgiebig diskutiert. OASIS[13] (Organization for the Advancement of Structured Information Standards), eine große Gemeinde von Firmen und Privatleuten, verabschiedete die Web Services Security Language (WS-Security). Sie schafft ein mächtiges Rahmenwerk, in das beliebige Sicherheitsmechanismen integriert werden können.

Kritik

Web-Services haben sich in der Praxis trotz hoher Akzeptanz noch nicht in erwartetem Maße durchgesetzt. Dafür ist wohl die Flut von Hilfsstandards verschiedenster Gremien verantwortlich. Das W3C, OASIS und andere haben inzwischen schon mehr als 30 Standards verabschiedet, deren Spezifikationen zum

Teil nur wenige Seiten umfassen. Zwischen diesen Standards gibt es häufig Überschneidungen und damit Konkurrenz, die zu Unsicherheit bei den Anwendern führt.

6 Universal Plug and Play

Überblick

Das Universal Plug and Play (UPnP) Forum[14] bündelt die Interessen von über 500 Firmen, größtenteils Soft- und Hardwarehersteller aus Nordamerika und Asien. Es forciert mit UPnP einen Standard zum konfigurationsfreien Betrieb von Systemen aller Art[15] – vom programmierbaren Thermostat bis zum Funknetz im Sportstadion. UPnP sieht in erster Linie die Vermittlung zwischen einem menschlichen Benutzer und einem technischen Dienst vor. Autonome Interaktion direkt zwischen Diensten ist kein ausdrückliches Ziel, auch wenn die Architektur diese Möglichkeit zulässt.

Architektur

UPnP[16] basiert auf den drei Komponenten *Device*, *Service* und *Control Point* (siehe Abb. 6). Devices (zu dt. *Geräte*) sind Hardwarekomponenten, die an einem UPnP-Netzwerk teilnehmen können. Bedingung dafür ist die Unterstützung der von UPnP eingesetzten Protokolle, allen voran des *Internet Protocol* (IP). Devices sind Behälter für alle anderen Komponenten eines UPnP-Netzwerkes, Services und Control Points müssen also stets Teil eines Device sein. Im Extremfall kann ein Device auch nur eine einzelne Komponente beinhalten, wie z. B. den Control Point am linken Rand von Abb. 6. Der Logik der Architektur folgend, ist dieser Control Point dennoch in einem Device gekapselt.

Ein Device kann neben Services und Control Points auch weitere Devices enthalten. In diesem Fall spricht man von dem Device der obersten Ebene als *Root Device* und bezeichnet die ihm untergeordneten als *Embedded Devices*. Mit Hilfe dieser Strukturierung lassen sich mehrere Devices zu einer funktionalen Einheit zusammenfassen, ohne gleichzeitig die Einzelteile anpassen zu müssen. Ein Beispiel für eine solche Kombination sind Fernseher und Videorecorder in einem Gerät.

Control Points entsprechen der Umsetzung der Dienstvermittler aus der SOA (siehe Abb. 7). Sie können in einem Device entweder exklusiv oder zusammen mit Services implementiert sein. Ihre primäre Aufgabe ist die Vermittlung der vorhandenen Devices bzw. ihrer Services an den Benutzer (*User*). Die Aufnahme eines Services in die Verwaltung des Control Point kann in beiden Richtungen geschehen. Control Points können das UPnP-Netzwerk nach Services durchsuchen (*search*). Alternativ kann ein Service sich bei einem oder mehreren Control Points anpreisen (*advertisement*). Anders als in der SOA ist bei UPnP nicht der Dienstnachfrager, sondern der Dienstvermittler für den unmittelbaren Aufruf von Diensten zuständig. Den Impuls dafür erhält er aber für gewöhnlich vom User. Control Points bilden damit insofern zentrale Instanzen, als jede Kommunikation zumindest über einen Control Point ablaufen muss.

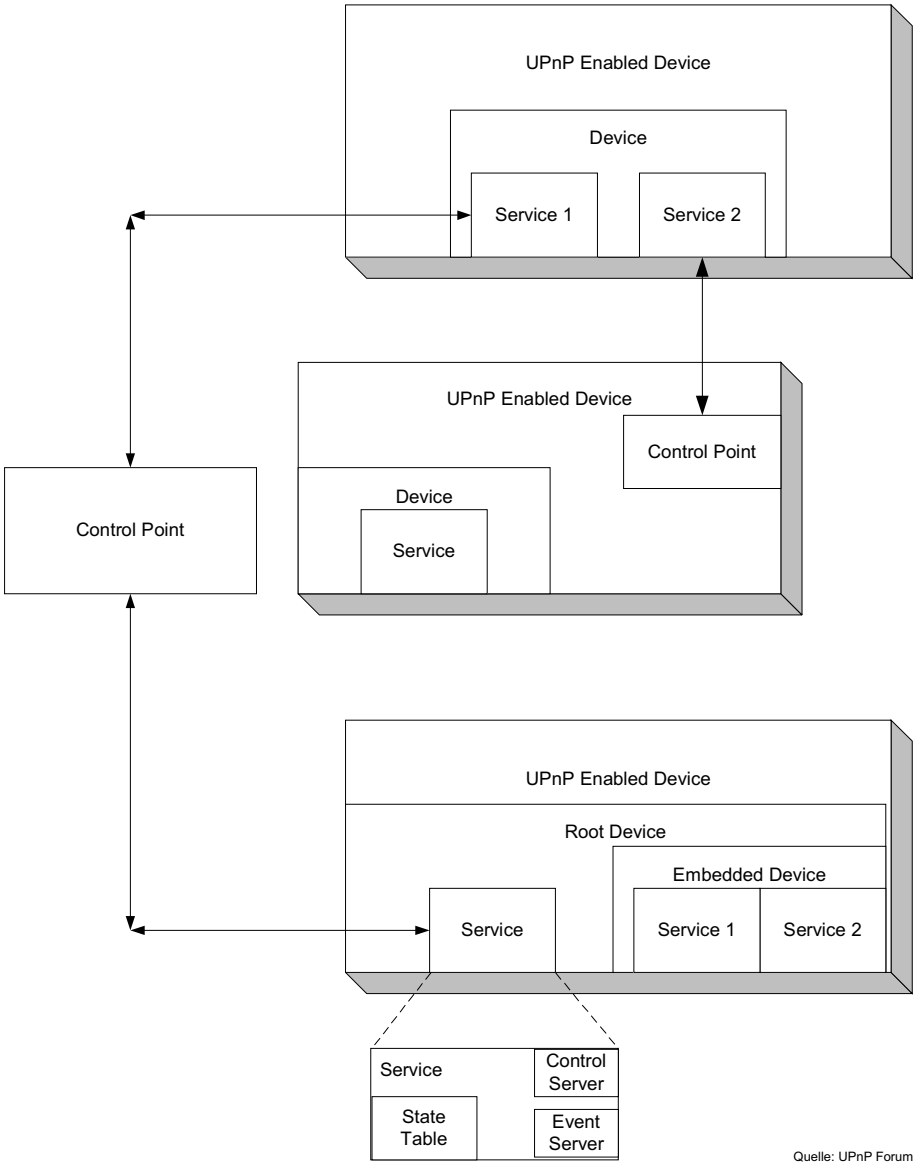


Abb. 6. Struktur der Komponenten von UPnP

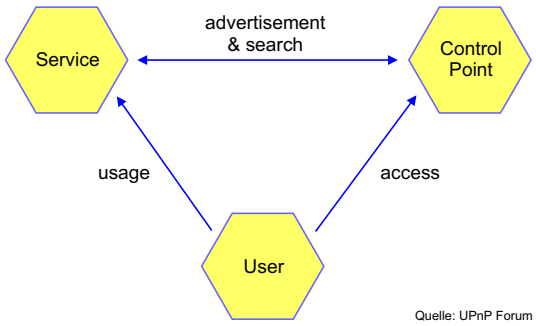


Abb. 7. Rollen und Interaktionen von UPnP

Die eigentliche Funktionalität wird bei UPnP von den Services zur Verfügung gestellt. Sie bilden die kleinsten logischen Einheiten in der Architektur von UPnP und sind als Zustandsautomaten mit einigen Erweiterungen implementiert. Jeder Service besitzt eine aktuelle Zustandstabelle (*State Table*), einen *Control Server* und einen *Event Server*. Der Control Server nimmt Anfragen von Control Points entgegen, führt sie in Abhängigkeit des aktuellen Zustands aus, aktualisiert die Zustandstabelle und liefert gegebenenfalls Ergebnisse zurück. Interessierte Control Points können einen Service zudem abonnieren. Aufgabe des Event Server ist dann die Benachrichtigung der Abonnenten bei jeder Zustandsänderung.

Als Dienstnachfrager tauchen in der Spezifikation von UPnP nur menschliche Benutzer auf. Die unmittelbare Nutzung (*usage*) eines Services ist – per Zugriff auf Control Points (*access*) – in allen Einsatzszenarien dem Menschen überlassen. Es ist aber ebenso denkbar, dass ein Service in Kooperation mit einem Control Point bei Eintritt eines Zustandes von sich aus handelt.

Kritik

Die Technologie hinter UPnP ermöglicht zwar Autonomie, aber in der Praxis wird sie nicht verwendet. Dadurch engt sich UPnP bei den Einsatzmöglichkeiten ohne ersichtlichen Grund enorm ein. In allen Einsatzszenarien wird ausschließlich die Konfigurationsfreiheit betont, autonome Interaktion ohne direkte Beteiligung eines menschlichen Benutzers kommt dagegen nicht vor. Weil UPnP keine komplexen Kommunikationsstrukturen kennt, werden auch keine Transaktionen unterstützt.

7 Vergleich der Technologien

SOA steht Pate für verschiedenste Technologien. Unterschiedliche Implementierungen reichen von der eher technischen Sichtweise von Bluetooth bis hin zu den abstrakten Web-Services. In der Mitte dazwischen bewegen sich UPnP und Jini. Die Einsatzszenarien haben dabei großen Einfluss auf die Gestaltung der einzelnen Architektur und geben zwingende Rahmenbedingung vor.

Eine objektive Beurteilung der Technologien anhand ausgewählter Kriterien ist wegen der großen Unterschiede schwierig. Im folgenden werden deshalb Vor- und Nachteile der einzelnen Technologien skizziert.

– **Jini**

  - **Vorteile**
    * Jini verfolgt konsequent die klare Trennung zwischen allen beteiligten Diensten. Es wird ausschließlich über öffentliche Schnittstellen kommuniziert.
    * Autonomie ist eine klare Zielsetzung und wird von Jini grundlegend unterstützt.
    * Ereignisbehandlung und Transaktionsverwaltung werden berücksichtigt.

  - **Nachteile**
    * Die verwendete Technologie ist nicht standardisiert. Bei jeder Kommunikation wird ausschließlich Java RMI benutzt. Die SOA sieht aber proprietäre Technologien ausdrücklich nicht vor. Interaktion von Djinns mit Netzen auf Basis anderer Technologien wird dadurch praktisch unmöglich gemacht.
    * Zentrale Sicherheitsfragen wie z. B. Vertraulichkeit oder Integrität der Kommunikation werden nicht geregelt. Jini wird so nur in vertrauenswürdigen Umgebungen zum Einsatz kommen können.

– **Bluetooth**

  - **Vorteile**
    * Bluetooth ist verhältnismäßig einfach zu implementieren. Es existieren Programmbibliotheken für alle gängigen Programmiersprachen (z. B. für C++ und Java).
    * Die Technologie ist weit verbreitet. Es gibt bereits sehr viele konkrete Anwendungen auf Bluetooth-Geräten.

  - **Nachteile**
    * Es gibt keine Regelung zur Einbindung von Diensten. Es wird nur die Beschreibung und Suche standardisiert.
    * Es fehlen für verteilte Systeme zentrale Konzepte wie z. B. Ereignisbehandlung oder Transaktionsverwaltung.
    * Bluetooth ermöglicht keine unscharfe Suche.

– **Web-Services**

  - **Vorteile**
    * Interoperabilität steht bei Web-Services an höchster Stelle. Durch die Verwendung von XML kann gewährleistet werden, dass fast jede Technologie mit einfachen Mitteln Kompatibilität zu Web-Services erreichen kann.

    * Web-Services sind immer und von überall aus zu erreichen. Schon der Name deutet an, dass Web-Services per Definition im Internet zur Verfügung stehen.
    * Mit UDDI kann die dynamische Bindung von Web-Services verwirklicht werden. Die semantische Kategorisierung von UDDI erlaubt ihnen, von sich aus mit anderen Web-Services Kontakt aufzunehmen.

  - **Nachteile**
    * Unklare Zuständigkeiten verhindern konsequente Standardisierung. Mehrere Gremien entwickeln unabhängig voneinander ähnliche Standards.
    * Konkurrenz zwischen Standards sorgt für Verwirrung bei den Anwendern. Viele Standards zeigen Überlappungen oder werden sogar doppelt definiert (z. B. die Web Service Choreography Language (WS-Choreography) vom W3C und das Web Service Choreography Interface (WSCI) von Sun und anderen).
    * Die Regelungswut nimmt viel Zeit in Anspruch und schränkt die Anwender ein. Selbst für winzige Bereiche werden Standards aufgestellt.

– **UPnP**

  - **Vorteile**
    * Die Konzentration auf Alltagsgegenstände eröffnet UPnP ein breites Anwendungsspektrum.
    * Hard- und Softwarehersteller ziehen gemeinsam an einem Strang, um UPnP in der Praxis durchzusetzen.

  - **Nachteile**
    * Die Möglichkeit der Autonomie wird nicht genutzt.
    * UPnP kennt keine Transaktionsverwaltung.
    * Es gibt keine klare Trennung zwischen Device, Service und Control Point. Sind z. B. Service und Control Point im selben Device untergebracht, wird die strikte Trennung zwischen ihnen aufgehoben.

Zusammenfassung und Ausblick

Je nach Einsatzzweck kann die eine oder andere Technologie die beste Wahl darstellen. Für verteilte Anwendungen im Internet eignen sich Web-Services am besten. Für ad hoc Kommunikation in lokalen Umgebungen sollte man auf Bluetooth oder Jini setzen. UPnP verhält sich ähnlich wie Bluetooth, konzentriert sich aber vor allem auf Alltagsgegenstände.

Die dienstorientierte Architektur liefert eine mächtige Ausgangsbasis zur Realisierung unterschiedlichster Technologien.

Literatur

1. B. Milewski: C++ In Action. Kapitel 6: About Software. www.relisoft.com 1997. *http://www.relisoft.com/book/proj/1software.html* – Abruf am 04.07.2003
2. S. Burbeck: The Tao of e-business services. IBM Software Group 2000. *ftp://www6.software.ibm.com/software/developer/library/ws-tao.pdf* – Abruf am 17.06.2003.
3. M. Stevens: Service-Oriented Architecture Introduction. www.developer.com 2002. *http://www.developer.com/design/article.php/1010451/* – Abruf am 17.06.2003.
4. Jini™ Technology Core Platform Specification Version 2.0. Sun Microsystems, Inc. 2003. *http://wwws.sun.com/software/jini/specs/core2_0.pdf* – Abruf am 08.07.2003.
5. Sun Microsystems, Inc. *http://www.sun.com*
6. Jini™ Architecture Specification Version 2.0. Sun Microsystems, Inc. 2003. *http://wwws.sun.com/software/jini/specs/jini2_0.pdf* – Abruf am 17.06.2003.
7. Specification of the Bluetooth System Version 1.1. Bluetooth SIG, Inc. 2001. *https://www.bluetooth.org/foundry/specification/document/Bluetooth_V1.1_Core_Specifications.pdf* – Abruf am 17.06.2003.
8. Bluetooth Special Interest Group. *http://www.bluetooth.org*
9. Web Services Architecture Working Draft Version 20030514. World Wide Web Consortium 2003. *http://www.w3.org/TR/2003/WD-ws-arch-20030514/* – Abruf am 17.06.2003.
10. World Wide Web Consortium. *http://www.w3c.org*
11. UDDI Version 3.0 Technical Committee Specification. OASIS 2002. *http://uddi.org/pubs/uddi-v3.00-published-20020719.pdf* – Abruf am 04.07.2003.
12. Web Services Description Language Version 1.2. World Wide Web Consortium 2003. *http://www.w3.org/TR/wsdl12/* – Abruf am 11.07.2003.
13. Organization for the Advancement of Structured Information Standards. *http://www.oasis-open.org*
14. Univeral Plug and Play Forum. *http://www.upnp.org*
15. Understanding Universal Plug and Play. Microsoft Corporation 2000. *http://www.upnp.org/download/UPNP_UnderstandingUPNP.doc* – Abruf am 01.07.2003.
16. UPnP™ Device Architecture 1.0. UPnP Forum 2001. *http://www.upnp.org/download/Clean%20UPnPDA101-20030506.doc* – Abruf am 01.07.2003.

Service-Orientierung und das Semantic Web

Hans-Christoph Andersen

Zusammenfassung. Die Organisation, Suche und Pflege von Daten im weltweiten Medium Internet wirft angesichts des bis auf weiteres ungebremst fortschreitenden Wachstums (mittlerweile gibt es deutlich mehr als eine Milliarde Webseiten) immer größere Probleme auf. Ein Großteil der Webseiten wurde ausschließlich für Menschen entwickelt, weshalb Maschinen diese trotz effizienter Such- und Rankingmechanismen nicht „verstehen“ können. Dieses Problem soll durch eine Erweiterung des heutigen World Wide Web, das Semantic Web, gelöst werden. Die vorliegende Arbeit gibt einen Einblick in die Techniken, Probleme und Anwendungsfälle des Semantic Web, wobei der Schwerpunkt auf die Vorstellung von Beschreibungssprachen, ihren Konzepten und Anwendungsfällen gelegt wurde.

1 Einleitung

Für das Zurechtfinden im mittlerweile unüberschaubar gewachsenen Internet sind heutzutage leistungsfähige Suchmaschinen wie Google oder Altavista unabdingbar. Aber auch diese offenbaren mitunter evidente Schwächen: Innerhalb sehr kurzer Zeit nach Eingabe des gesuchten Begriffes geben sie eine unter Umständen sehr lange Liste von Webseiten aus, in denen dieser Begriff vorkommt. In der Regel ist ein Großteil dieser Seiten für den Anwender unbrauchbar, was seine Intention angeht, sodass er "von Hand" Wichtiges von Unwichtigem trennen muss. Das liegt daran, dass die klassische Software die Webseiten zwar sehr schnell lesen, aber nicht "verstehen", d.h. semantisch interpretieren kann, sodass keine gezielten, dem jeweiligen Problem angemessenen Anfragen möglich sind.

Aus diesem Grund entstand die Vision des *Semantic Web* als einer Erweiterung des gegenwärtigen World Wide Web. Hier soll den Daten eine wohldefinierte, für Maschinen verarbeitbare Bedeutung gegeben werden. Dieses zukunftsorientierte Konzept und seine Techniken sind Gegenstand des vorliegenden Textes.

Eingangs wird mit Ontologien ein zentraler Begriff dieses Themas erklärt. Daraufhin werden mit RDF, DAML+OIL und OWL drei wichtige, sich teilweise noch im Aufbau befindliche Beschreibungssprachen vorgestellt, mit deren Hilfe das Hauptziel des Semantic Web, das automatisierte Finden, Aufrufen und Ausführen von Web-Diensten sowie ihre Komposition und Interoperation, realisiert werden kann. Im Anschluss daran werden konkrete Beispiele und Probleme anhand einer existierenden

Ontologie (DAML-S) beschrieben sowie kurz einige aktuelle Anwendungsfälle der vorgestellten Techniken behandelt.

2 Eine kurze Übersicht

Mit Unterstützung des W3C (WWW Consortium) [2] wird das Problem des Semantic Web derzeit mit starkem Einsatz in der Forschung angegangen. Die Lösungsvorschläge bestehen im Kern darin, die im WWW zur Verfügung stehenden Informationen mit einer für Maschinen verarbeitbaren Semantik zu versehen. Speziellen wissensbasierten Software-Agenten soll dadurch der intelligente Zugriff auf diese Informationen ermöglicht werden, um sie gemäß den Anforderungen des Benutzers zu verarbeiten.

Die Kern-Technologie zur Repräsentation des "semantischen Wissens" bilden sogenannte *Ontologien* (formale, explizite, konsensfähige und wiederverwertbare Konzeptualisierungen). Um Ontologien zu konstruieren, wurden verschiedene auf XML basierende Sprachen definiert; die prominentesten Vertreter sind RDF [4], DAML und DAML+OIL [7] sowie OWL [8] (siehe unten). Damit wurden bereits konkrete Ontologien aus verschiedenen Domänen erstellt (z.B. DAML Library, DAML-S). Auf all diese Aspekte wird im Folgenden näher eingegangen.

3 Ontologien

In der Philosophie ist die Ontologie die Lehre vom Sein bzw. von den Ordnungs-, Begriffs- und Wesensbestimmungen des Seienden. Im Kontext der Repräsentation von Wissen, speziell des Semantic Web wird dadurch hingegen eine Spezifikaton einer *Konzeptualisierung* bezeichnet. Eine Konzeptualisierung ist eine abstrakte, vereinfachte Sicht und Beschreibung eines bestimmten Systems, seiner Komponenten und ihrer Beziehungen untereinander. Die Menge der Komponenten sowie ihre relevanten Beziehungen werden auf repräsentative Terme abgebildet, mit denen wissensbasierte Programme (Software-Agenten) arbeiten können. Eine Ontologie (eines Programmes) wird also durch die Definition einer Menge von repräsentativen Termen beschrieben. Die Namen der Entitäten im betrachteten System (z.B. Klassen, Relationen, Funktionen oder andere Objekte) werden auf diese Weise mit für Menschen verständlichen Informationen assoziiert: was die Namen bedeuten oder nach welchen formalen Axiomen die Terme interpretiert und benutzt werden müssen.

In der Theorie ist jedes wissensbasierte System (explizit oder implizit) an bestimmte Konzeptualisierungen, also auch Ontologien gebunden. Genauer besteht eine Bindung eines Software-Agenten an eine gegebene Ontologie genau dann, wenn seine beobachtbaren Aktionen mit den Definitionen in der Ontologie konsistent sind. Für eine Menge von Software-Agenten, die miteinander auf einem bestimmten Gebiet kommunizieren sollen, werden sinnvollerweise gemeinsame Ontologien benutzt.

Jeder Agent besitzt eine Benutzerschnittstelle zur Eingabe von Anfragen und (logischen) Zusicherungen sowie eine Schnittstelle zur Kommunikation und Interaktion mit anderen Agenten. Praktischerweise sollte eine gemeinsame Ontologie

das Vokbular definieren, mit dem Anfragen und Zusicherungen unter den Agenten ausgetauscht werden.

4 RDF

RDF (*Resource Description Framework*) [5] ist eine grundlegende Technik (Sprache) zur Beschreibung und zum Austausch von Daten im Semantic Web. Seine auf XML (Extensible Markup Language) basierende Spezifikation wurde erstmals 1999 vom W3C vorgelegt.

RDF kann für eine Reihe von Anwendungszwecken eingesetzt werden, z.B.:

- Finden von Daten und Ressourcen für die Bereitstellung von besseren Suchmaschinenkapazitäten,
- Beschreibung des Inhaltes und der inneren Beziehungen der Objekte einer bestimmten Website, Webpage oder einer digitalen Bibliothek,
- Erleichterung von Wissensteilung und Wissensaustausch durch Software-Agenten.

Um die Definition von Metadaten zu erleichtern, hat RDF (wie auch die darauf aufbauenden Beschreibungssprachen) ein Klassensystem, ähnlich vielen objektorientierten Programmier- und Modellierungssystemen. Eine für einen bestimmten Zweck oder Bereich generierte Sammlung von Klassen wird *Schema* genannt.

In dieser Ausarbeitung wird häufig der Begriff *URI* auftauchen. Ein URI (Uniform Resource Identifier) ist eine Zeichenfolge zur Kennzeichnung einer Internet-Ressource. Es ist ein übergeordneter Begriff, der sich in die Unterbegriffe URL (Uniform Resource Locator) und URN (Uniform Resource Name) aufgliedern lässt. Ausführliche Informationen über dieses Konzept finden sich in [4].

Im Folgenden wird zwischen dem *RDF-Modell*, der *RDF-Syntax* und dem RDF-*Schema* unterschieden.

4.1 RDF-Modell

Das RDF-Datenmodell ist eine syntax-neutrale Repräsentation von RDF-Ausdrücken (Statements). Es wird benutzt, um Statements auf (Bedeutungs-)Äquivalenz zu überprüfen. Zwei RDF-Ausdrücke sind genau dann äquivalent, wenn ihre Repräsentation im Datenmodell dieselbe ist. Diese Definition von Äquivalenz lässt einige syntaktische Variationen von Ausdrücken zu, ohne dass deren Bedeutung verändert wird.

Das Basisdatenmodell besteht aus drei Objekttypen:

- *Ressourcen*: Alle "Dinge", die von RDF "beschrieben" werden, werden Ressourcen genannt. Eine Ressource kann z.B. eine Webpage, ein Teil einer

Webpage (etwa ein spezielles HTML- oder XML-Element innerhalb des Dokument-Quelltextes) oder auch eine gesamte Website sein, aber auch ein Objekt, das nicht direkt über das Netz zugänglich ist, wie z.B. ein gedrucktes Buch. Ressourcen werden durch URIs und optionale Anker-IDs gekennzeichnet.

- *Merkmale (Properties)*: Ein Merkmal ist ein spezielles Charakteristikum, ein Attribut oder eine Relation, um eine Ressource zu beschreiben. Jedes Merkmal hat eine spezifische Bedeutung, für jedes Merkmal gibt es bestimmte Werte, die es annehmen kann, Ressourcen, die es beschreiben kann, sowie Relationen mit anderen Merkmalen. Ein Merkmal kann als binäre Relation angesehen werden, da es zwei Entitäten (zwei Objekte oder ein Objekt und einen Datenwert) miteinander in Beziehung setzt.

- *Statements*: Informationen werden in RDF in sogenannten Statements abgelegt. Hierbei handelt es sich um Tripel aus Subjekt, Prädikat und Objekt. Das Subjekt ist eine bestimmte Ressource, das Prädikat ein Merkmal dieser Ressource und das Objekt der Wert, den das Merkmal im jeweiligen Statement annimmt. Dieser Wert kann verschiedene Formen haben: eine weitere Ressource (durch einen URI gekennzeichnet) oder ein Literal: ein einfacher String bzw. ein anderer primitiver, in XML definierter Datentyp.

Beispiel. Wir betrachten den folgenden einfachen Satz:

Peter Mustermann ist der Leiter des Unternehmens Mustermann AG.

In RDF interpretiert enthält diese Aussage eine Ressource, nämlich das Unternehmen *Mustermann AG* (dieses sei durch den URI *http://www.mustermann-ag.com* repräsentiert), ein Merkmal (*Leiter*) sowie einen String (*Peter Mustermann*) als Wert des Merkmales.

Damit ist dieser Satz gemäß einem RDF-Statement wie folgt zu unterteilen:

- Subjekt (Ressource): *http://www.mustermann-ag.com*
- Prädikat (Merkmal, Eigenschaft): *Leiter*
- Objekt (Literal, String): *"Peter Mustermann"*

Diese Beziehungen werden für gewöhnlich durch *gerichtete Graphen* modelliert. Die Knoten stellen die Ressourcen dar, die Kanten die Merkmale. Knoten mit URIs werden als Ovale gezeichnet, solche mit (String-)Literalen als Rechtecke. Daher wird unser Beispielsatz mit folgendem Diagramm repräsentiert:

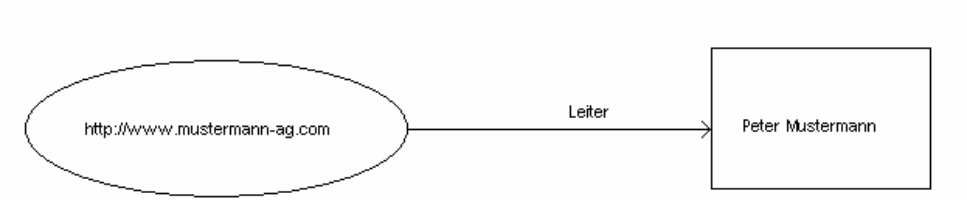


Abb. 1: Einfacher RDF-Graph

Wichtig ist hierbei die Richtung des Pfeiles. Er beginnt immer beim Subjekt und zeigt auf das Objekt des jeweiligen Satzes. Um die Bezeichnungen *Subjekt* und *Objekt* mit denen der gesprochenen Sprache konsistenter zu machen, kann man obigen Satz auch folgendermaßen lesen:

Das Unternehmen Mustermann AG (http://www.mustermann-ag.com) hat den Leiter Peter Mustermann.

Oder allgemein:

<Subjekt> HAT <Prädikat> <Objekt>

Betrachten wir nun den Fall, dass wir mehr über den Leiter des Unternehmens sagen wollen. Dazu wäre etwa folgender (sprachlich sicherlich unglücklich formulierte, weil auf die RDF-Konditionen zugeschnittene) Satz geeignet:

Das Individuum, dessen Name Peter Mustermann ist und das den Wohnort Musterstadt hat, ist Leiter des Unternehmens Mustermann AG (http://www.mustermann-ag.com).

Die Intention dieses Satzes ist, den Wert des Merkmales *Leiter* zu einer strukturierten Entität zu machen. So eine Entität wird in RDF durch eine weitere Ressource repräsentiert. Der obige Satz gibt dieser Ressource keinen Namen, sie ist anonym. Im entsprechenden Diagramm wird sie daher durch ein leeres Oval dargestellt:

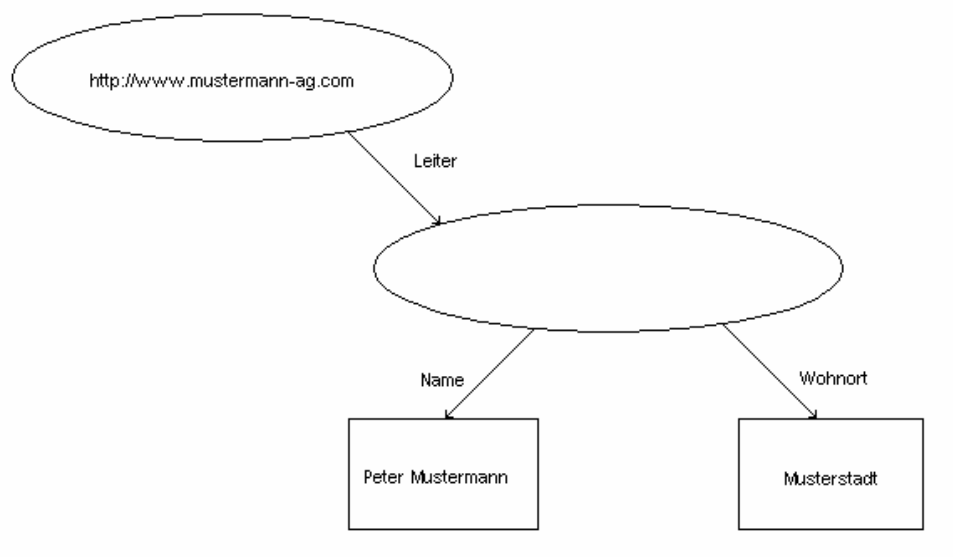


Abb. 2: Ein Merkmal mit einem strukturierten Wert ohne Kennung

Nach der obigen "HAT-Schreibweise" kann das Diagramm so gelesen werden:

Das Unternehmen Mustermann AG (http://www.mustermann-ag.com) hat den Leiter 'irgendetwas' und 'irgendetwas' hat den Namen Peter Mustermann und den Wohnort Musterstadt.

Die strukturierte Entität des letzten Beispieles kann mit einer eindeutigen Kennung versehen werden, deren Wahl vom Designer der Anwendungsdatenbank getroffen wird. Um das Beispiel fortzuführen, betrachten wir nun den Fall, dass eine Mitarbeiter-ID für eine *Person*-Ressource als Kennung benutzt wird. Die URIs, die für jeden Mitarbeiter als eindeutige Schlüssel fungieren (definiert von der jeweiligen Firma), hätten dann eine Form wie etwa *http://www.mustermann-ag.com/MitarbeiterID/101*. Nun können wir zwei Sätze betrachten:

Das Individuum, dem die Mitarbeiter-ID 101 zugeordnet ist, hat den Namen Peter Mustermann und den Wohnort Musterstadt. Das Unternehmen Mustermann AG (http://www.mustermann-ag.com) wird von diesem Individuum geleitet.

Das entsprechende RDF-Modell sieht folgendermaßen aus:

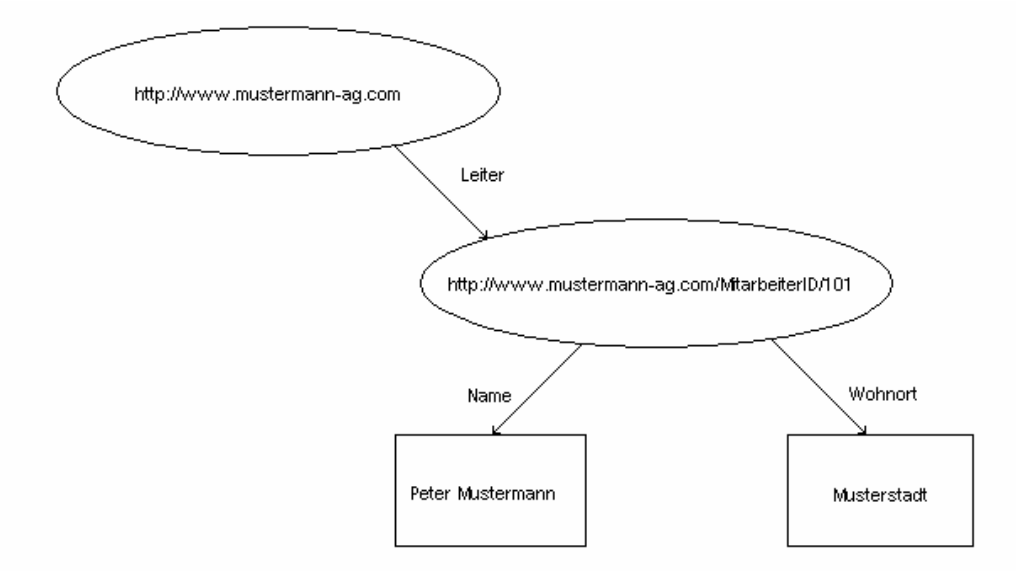


Abb. 3: Strukturierter Wert mit Kennung

Dieses Diagramm unterscheidet sich vom vorherigen (Abb. 2) nur dadurch, dass die vorher anonyme Ressource nun einen URI erhalten hat. Aus der Sicht einer weiteren Applikation besteht in der Regel kein Unterschied, ob die obigen Statements in einem einzigen oder in zwei Sätzen formuliert werden. RDF unterstützt allerdings eine derartige Unterscheidung. Zum einen können über die als Prädikat verwendeten Ressourcen (Merkmale) wiederum Aussagen getroffen werden (siehe Beispiel oben). Zum anderen bilden in RDF Statements selbst Ressourcen, auf die mit weiteren Statements verwiesen werden kann. Diese Technik der Aussagen über Aussagen wird als *Reification* bezeichnet.

4.2 RDF-Syntax

Das RDF-Modell ist unabhängig von speziellen Darstellungsformen. Am meisten verbreitet ist die Repräsentation in XML. Weiterhin gibt es eine kürzere Syntax in Form der von Tim Berners-Lee entworfenen *Notation 3* [6]. Für die Speicherung von RDF in Datenbanken und Datenstrukturen gibt es verschiedene Konzepte, da ein reines Ablegen in einer Tabelle nicht sehr effektiv ist.

Ein isoliertes RDF-Statement kommt vergleichsweise sehr selten vor; in den meisten Fällen werden mehrere Merkmale einer Ressource zusammengefasst. Die RDF-XML-Syntax wurde entworfen, um diese Gruppenbildung effizient zu unterstützen: Beliebig viele Statements, die sich auf dieselbe Ressource beziehen, können innerhalb eines *Description*-Elementes zusammengefasst werden. Dieses Element besitzt ein *about*-Attribut, in dem die jeweilige Ressource angegeben wird. Falls diese Ressource noch keine Kennung hat, kann ein *Description*-Element alternativ ein ID-Attribut benutzen. Die Grundsyntax von RDF besitzt folgende Form (in BNF):

```
RDF           ::= ['<rdf:RDF>'] Description*
                      ['</rdf:RDF>']
Description   ::= '<rdf:Description' idAboutAttr? '>'
                        propertyElt*
idAboutAttr   ::= idAttr | aboutAttr
aboutAttr     ::= 'about="' URI-reference '"'
idAtt         ::= 'ID="' IDsymbol '"'
propertyElt   ::='<' propName '>' value '</' propName '>'
              | '<' propName resourceAttr '/>'
propName      ::= Qname
value         ::= Description | String
resourceAttr  ::= 'resource="' URI-reference '"'
Qname         ::= [ NSprefix ':' ] Name
URI-reference ::= String, interpretiert durch URI
IDsymbol      ::= (beliebiges legales XML-Namenssymbol)
Name          ::= (beliebiges legales XML-Namenssymbol)
NSprefix      ::= (beliebige legale XML-
                   Namensbereichspräfix)
String        ::= (beliebiger XML-Text ohne die Zeichen
                     "<", ">" und "&")
```

Das *RDF*-Element stellt eine simple "Hülle" dar, die den Bereich in einem XML-Dokument kennzeichnet, der als Instanz eines RDF-Modelles interpretiert werden soll.

Ein *Description*-Element enthält, wie oben bereits erklärt, Statements über eine bestimmte Ressource; für diese stellt es eine Identifikation dar. Wenn das *about*-Attribut bei *Description* spezifiziert ist, beziehen sich die enthaltenen Statements auf die diesem Attribut entsprechende Ressource. Der Wert des *about*-Attributes wird als URI-Referenz interpretiert. Enthält das *Description*-Element kein *about*-Attribut, repräsentiert es eine neue Ressource. Diese kann ein Stellvertreter für irgendeine andere physikalische Ressource sein, die keinen erkennbaren URI hat. Hat das jeweilige *Description*-Element ein spezifiziertes *ID*-Attribut, so stellt dessen Wert die Anker-ID dieser Ressource dar.

In einem *propertyElt*-Element spezifiziert das *resource*-Attribut eine weitere Ressource als Wert dieses Merkmales, d.h. das Objekt des Statements ist eine Ressource, die durch einen URI oder ein Literal identifiziert wird.

Beispiel (Fortsetzung). Der eingangs behandelte Satz

Peter Mustermann ist der Leiter des Unternehmens Mustermann AG.

wird in RDF/XML folgendermaßen repräsentiert:

```
<rdf:RDF>
```

```
  <rdf:Description
           about="http://www.mustermann-ag.com">
  <s:Leiter>Peter Mustermann</s:Leiter>
  </rdf:Description>
</rdf:RDF>
```

Das Präfix *s* bezieht sich auf einen bestimmten Namensbereich, der vom Autor dieses RDF-Ausdruckes gewählt und in einer XML-Namensbereichsdeklaration definiert wurde – etwa folgendermaßen:

```
xmlns:s="http://description.org/schema/"
```

So eine Deklaration wird typischerweise als ein XML-Attribut in das `rdf:RDF`-Element eingebunden. Der benutzte URI ist eine global eindeutige Kennung für das Schema, das der Autor für die Definition des Leiter-Merkmales benutzt. Auch andere Schemata können ein Merkmal namens *Leiter* definieren; die beiden Merkmale werden dann durch die jeweiligen Kennungen unterschieden.

Das komplette XML-Dokument, das obigen RDF-Ausdruck enthält, sieht so aus:

```
<?xml version="1.0"?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:s="http://description.org/schema/">
 <rdf:Description about="http://www.mustermann-ag.com">
    <s:Leiter>Peter Mustermann</s:Leiter>
  </rdf:Description>
</rdf:RDF>
```

4.3 Schemata und Namensbereiche

Wenn man einen Satz in natürlicher Sprache schreibt, haben die einzelnen Wörter eine bestimmte Bedeutung, die entscheidend für das richtige Verständnis der Aussage ist, d.h. der Schreiber und der Leser (oder allgemeiner: Sender und Empfänger) müssen beide dieselbe Bedeutung in der Aussage erkennen, andernfalls schlägt die Kommunikation fehl. Ebenso verhält es sich bei RDF-Statements, da diese ja insbesondere zur Interkommunikation zwischen heterogenen Software-Systemen dienen. In einem globalen Medium wie dem World Wide Web reicht es daher nicht aus, sich auf ein gemeinsames "kulturelles" Verständnis von Begriffen wie z.B. "die Eigenschaft, Leiter eines Unternehmens zu sein" zu stützen, sondern die Bedeutungen müssen so einheitlich und präzise wie möglich definiert werden.

"Bedeutungen" werden in RDF durch eine Referenz auf ein Schema ausgedrückt. Ein Schema – eine vordefinierte Sammlung von Klassen – kann als eine Art Wörterbuch verstanden werden, das die Terme definiert, die in RDF-Statements benutzt werden. In ihm werden sowohl Definitionen als auch Restriktionen für den Gebrauch von Merkmalen dokumentiert.

Verschiedene Schemata können voneinander unabhängige – und möglicherweise konkurrierende – Definitionen desselben Termes bereitstellen. Um daraus entstehende Konflikte zu vermeiden, benutzt RDF das XML-Konzept der Namensbereiche, d.h. der spezifische Gebrauch eines Wortes wird an das Schema gebunden, in dem die gewünschte Definition zu finden ist. In RDF muss jedes Prädikat in einem Statement mit genau einem Namensbereich bzw. Schema identifiziert werden.

5 DAML+OIL

DAML+OIL [7] steht für *DARPA Agent Markup Language* + *Ontology Inference Layer*. Es ist eine Beschreibungssprache für Web-Ressourcen und baut auf Standards wie RDF und RDF-Schema auf, indem es deren Vokabular und Modellierungsprimitive erweitert.

DAML+OIL schreibt für bestimmte RDF-Tripel, die das erweiterte Vokabular benutzen, eine spezifische Bedeutung vor. Im Folgenden wird auf die Syntax und Semantik einführend eingegangen.

Eine Ontologie in DAML+OIL besteht aus einer Vielzahl von Komponenten, von denen einige optional sind, andere wiederholt auftreten können.

5.1 Header

Eine Ontologie in DAML+OIL besitzt eine optionale Anzahl von Headern, gefolgt von ebenfalls beliebig vielen Klassenelementen, Klassenausdrücken, Merkmalselementen und Instanzen.

Ein Beispiel für einen Ontologie-Header:

```
<Ontology rdf:about="">
  <versionInfo>$Id: NOTE-daml+oil-reference
20011218.html,v 1.6 2001/12/18 22:12:09 connolly Exp
                     $</versionInfo>
  <rdfs:comment>Eine Beispiel-Ontologie</rdfs:comment>
  <imports
    rdf:resource="http://www.w3.org/2001/10/daml+oil/">
</Ontology>
```

Das `daml:versionInfo`-Element enthält einen String, der Informationen über die benutzte Version gibt.

Jedes `daml:imports`-Statement referenziert eine weitere DAML+OIL-Ontologie, die Definitionen enthält, die in der aktuellen Ressource Verwendung finden. Jede Referenz besteht aus einem URI, der genau spezifiziert, woher die Ontologie importiert werden muss.

Die Imports-Statements sind transitiv, d.h. wenn die Ontologien A, B, und C gegeben sind, gilt: *A importiert B und B importiert C* impliziert *A importiert C*. Importieren sich zwei Ontologien gegenseitig, werden sie als äquivalent angesehen.

5.2 Klassenelemente

Um Objekte und Beziehungen zu beschreiben, ist es nützlich, einige Klassen als Basistypen zu definieren. Wenn wir beispielsweise in einer Ontologie die Eigenschaften bestimmter Tiere beschreiben wollen, so müssen wir zunächst einen geeigneten Typ *Tier* definieren. Um dies zu tun, benutzen wir ein Klassenelement:

```
<daml:Class rdf:ID="Tier">
</daml:Class>
```

Mit diesen zwei Programmzeilen wird zugesichert, dass eine Klasse mit dem Namen *Tier* existiert, sonst wird nichts über diese Klasse ausgesagt. Ein Klassenelement enthält üblicherweise nähere Informationen über die dieser Klasse zugehörigen Objekte.

Es soll zwei Typen von Tieren geben, *Maennchen* und *Weibchen*:

```
<daml:Class rdf:ID="Maennchen">
  <rdfs:subClassOf rdf:resource="#Tier"/>
</daml:Class>
```

Das *subClassOf*-Element sichert zu, dass sein Subjekt (*Maennchen*) eine Unterklasse seines Objektes (der Ressource, die durch `#Tier` gekennzeichnet wird) ist.

```
<daml:Class rdf:ID="Weibchen">
  <rdfs:subClassOf rdf:resource="#Tier"/>
  <daml:disjointWith rdf:resource="#Maennchen"/>
</daml:Class>
```

Ein Tier kann (in dieser Ontologie) nur entweder männlich oder weiblich sein, nicht beides gleichzeitig. Dies wird durch das `disjointWith`-Element zugesichert.
Ein Klassenelement enthält also die Definition einer Objektklasse. Eine formale Erklärung der wichtigsten Elemente zur Beschreibung einer Klasse findet sich im Anhang.

5.3 Klassenausdrücke

Als ein Klassenausdruck werden in diesem Text bezeichnet:

- ein Klassenname (ein URI) oder
- eine Aufzählung, eingeschlossen in die Tags `<daml:Class>` bzw. `</daml:Class>`, oder
- eine Merkmals-Restriktion oder
- eine boolesche Kombination von Klassenausdrücken, eingeschlossen in die Tags `<rdfs:Class>` bzw. `</rdfs:Class>`.

Jeder Klassenausdruck bezieht sich auf eine benannte Klasse, entweder explizit als URI oder je nach Typ des Ausdruckes implizit, indem er entweder eine anonyme Klasse oder die Klasse, die genau die aufgezählten Elemente enthält, oder die Klasse aller Instanzen, die der Merkmals-Restriktion genügen, oder die Klasse, die die boolesche Kombination solcher Ausdrücke erfüllt, definiert.

Es gibt zwei Klassennamen, die bereits vordefiniert sind, nämlich `daml:Thing` und `daml:Nothing`. Jedes Objekt ist Mitglied der Klasse `daml:Thing`, kein Objekt Mitglied von `daml:Nothing`. Folglich ist jede Klasse eine Unterklasse von `daml:Thing`, und `daml:Nothing` ist eine Unterklasse jeder Klasse.
Aufzählungen. Eine Aufzählung wird mit einem `daml:oneOf`-Element realisiert, das eine Liste der Objekte enthält, die seine Instanzen darstellen. Mithilfe dieser Technik kann man eine Klasse definieren, die explizit ihre Elemente (Instanzen) angibt. Damit enthält so eine Klasse genau diese aufgezählten Elemente, nicht mehr und nicht weniger. Ein (selbsterklärendes) Beispiel:

```
<daml:oneOf parseType="daml:collection">
  <daml:Thing rdf:about="#Eurasien"/>
  <daml:Thing rdf:about="#Afrika"/>
  <daml:Thing rdf:about="#Nordamerika"/>
  <daml:Thing rdf:about="#Suedamerika"/>
  <daml:Thing rdf:about="#Australien"/>
  <daml:Thing rdf:about="#Antarktis"/>
</oneOf>
```

Merkmals-Restriktionen. Eine Merkmals-Restriktion ist ein spezieller Klassenausdruck. Sie definiert implizit eine anonyme Klasse, nämlich die Klasse aller Objekte, die der Restriktion genügen. Es gibt zwei Typen von Restriktionen. Die erste Art, *ObjectRestriction*, behandelt Objektmerkmale, also solche, die Beziehungen zwischen Objekten beschreiben. Die zweite Art, *DatatypeRestriction*, behandelt Datentypmerkmale, also solche, die Beziehungen zwischen Objekten und Datentypwerten beschreiben. Beide Typen von Restriktionen benutzen dieselbe Syntax mit der Unterscheidung, ob ein Klassenelement oder eine Datentyp-Referenz benutzt wird.

Als Beispiel definieren wir eine Klasse *Person* als Unterklasse von *Tier*:

```
<daml:Class rdf:ID="Person">
  <rdfs:subClassOf rdf:resource="#Tier"/>
```

Im Folgenden wenden wir eine Restriktion bezüglich des Merkmales *hatElternteil* (das im Abschnitt 5.4 noch definiert wird) an, nämlich die, dass die Mutter oder der Vater einer Person wieder eine Person sein muss:

```
<rdfs:subClassOf>
  <daml:Restriction>
    <daml:onProperty rdf:resource="#hatElternteil"/>
    <daml:toClass rdf:resource="#Person"/>
  </daml:Restriction>
</rdfs:subClassOf>
```

Hierdurch wird die anonyme Klasse aller "Dinge" definiert, deren Mutter oder Vater eine Person ist, und die Klasse *Person* wird als eine Unterklasse dieser anonymen Klasse deklariert. Also muss jede Person der Restriktion genügen.

Merkmals-Restriktionen können eine Vielzahl von Beschreibungselementen enthalten. Die wichtigsten davon werden im Anhang erklärt.

Boolesche Kombinationen von Klassenausdrücken. Eine boolesche Kombination von Klassenausdrücken kann aus folgenden Elementen bestehen:

- ein `daml:intersectionOf`-Element, das eine Liste von Klassenausdrücken enthält – es definiert die Klasse, die genau die Objekte enthält, die allen Klassenausdrücken der Liste genügen; es ist analog zur logischen Konjunktion,

- ein `daml:unionOf`-Element, das eine Liste von Klassenausdrücken enthält – es definiert die Klasse, die genau die Objekte enthält, die (mindestens) einem der Klassenausdrücke in der Liste genügen; es ist analog zur logischen Disjunktion,

- ein `daml:complementOf`-Element, das einen einzelnen Klassenausdruck enthält – es definiert die Klasse, die genau die Objekte enthält, die nicht zum Klassenausdruck gehören; es ist analog zur logischen Negation, aber nur auf Objekte beschränkt.

Mithilfe dieser Elemente kann durch Verschachtelung jede beliebige boolesche Kombination beschrieben werden.

5.4 Merkmals-Elemente

Ein Merkmal kann auch als binäre Relation verstanden werden. In DAML+OIL werden Merkmale in zwei Arten unterteilt: solche, die Objekte mit anderen Objekten in Beziehung setzen (`daml:ObjectProperty`), und solche, die Objekte mit Werten von Datentypen verbinden (`daml:DatatypeProperty`).

Wir betrachten exemplarisch die Relation *hatElternteil*, die zwei Tiere miteinander in Beziehung setzt:

```
<daml:ObjectProperty rdf:ID="hatElternteil">
```

Auf diese Weise wird – analog zu Klassen – festgelegt, dass ein Merkmal mit dem genannten Namen existiert.

```
<rdfs:domain rdf:resource="#Tier"/>
```

Mit diesem Element wird festgelegt, dass das Merkmal *hatElternteil* nur auf Tiere zutreffen darf.

```
 <rdfs:range rdf:resource="#Tier"/>
```

Ähnlich zum `rdfs:domain`-Element legt das `rdfs:range`-Element fest, dass der Wert des Merkmales *hatElternteil* nur zur Klasse *Tier* gehören darf. Beide Elemente zusammen sichern also zu, dass das Merkmal zwei Tiere miteinander in Beziehung setzt.

Am Ende darf natürlich die Schließung des `daml:ObjectProperty`-Elementes nicht fehlen:

```
</daml:ObjectProperty>
```

Analog zu Unterklassen können wir Untermerkmale definieren. Das Merkmal *hatVater* wird im Folgenden als Untermerkmal von *hatElternteil* definiert. Weiterhin wird festgelegt, dass der Vater eines Tieres zur Klasse *Maennchen* gehören muss.

```
<daml:ObjectProperty rdf:ID="hatVater">
  <rdfs:subPropertyOf rdf:resource="#hatElternteil"/>
  <rdfs:range rdf:resource="#Maennchen"/>
</daml:ObjectProperty>
```

Will man z.B. das Alter eines Gegenstandes oder Lebewesens als Merkmal definieren, so benutzt man den Typ *DatatypeProperty*. Die folgende Definition legt fest, dass das Alter eine nichtnegative ganze Zahl ist und dass jedes Individuum genau ein Alter hat (*UniqueProperty*). Für die benutzten Datentypen benutzen wir den XML-Datentyp *nonNegativeInteger*:

```
<daml:DatatypeProperty rdf:ID="Alter">
  <rdf:type
```

```
      rdf:resource="http://www.w3.org/2001/10/daml+oil#
                    UniqueProperty"/>
  <rdfs:range
      rdf:resource=http://www.w3.org/2000/10/XMLSchema#
                  nonNegativeInteger"/>
</daml:DatatypeProperty>
```

Die Elemente `daml:ObjectProperty` und `daml:DatatypeProperty` spezifizieren den generellen Merkmalstyp `rdf:Property`. Ein `rdf:Property`-Element kann eine Vielzahl von Beschreibungselementen enthalten; eine Erklärung der wichtigsten davon findet sich im Anhang.

5.5 Instanzen

Sowohl Instanzen von Klassen (also Objekte) als auch solche von Merkmalen (also Paare von Objekten) werden in RDF- bzw. RDF-Schema-Syntax geschrieben. Ein Beispiel in gängiger Notation:

```
<Kontinent rdf:ID="Asien"/>

<rdf:Description rdf:ID="Asien">
  <rdf:type>
    <rdfs:Class rdf:about="#Kontinent"/>
  </rdf:type>
</rdf:Description>

<rdf:Description rdf:ID="Indien">
  <is_part_of rdf:rescource="#Asien"/>
</rdf:Description>
```

6 OWL

OWL (Web Ontology Language) [8] ist ebenfalls eine Beschreibungssprache für Web-Ressourcen bzw. zur Erstellung von Ontologien. Das W3C legte hiermit einen Vorschlag für eine auf RDF-Schema aufbauende Sprache vor, die Ausdrucksmöglichkeiten von DAML und OIL in RDF vereinigt. OWL erweitert ebenfalls das Vokabular von RDF und ist von DAML+OIL abgeleitet. Da es noch sehr neu und bisher nur als überarbeiteter Entwurf anzusehen ist, gibt es bisher keine relevanten Anwendungsfälle.

Es gibt zwei spezifische Untermengen dieser Sprache. *OWL Lite* [8] wurde entworfen, um Benutzer mit einem einfachen, aber funktionalen Werkzeug auszustatten, das ihnen den Einstieg in OWL erleichtert und eine einfache Implementierung ermöglicht.

OWL DL (OWL Description Logic) [8] beinhaltet bereits den vollständigen OWL-Wortschatz, der aber unter einer Anzahl einfacher Beschränkungen interpretiert wird.

Die vollständige Sprache wird zur Unterscheidung von den beiden Untermengen *OWL Full* [8] genannt. Sie lässt freies Vermischen von OWL mit RDF Schema zu und erzwingt keine strikte Trennung von Klassen, Merkmalen, Individuen und Datenwerten.

Eine OWL-Ontologie ist als RDF-Graph darstellbar, der wiederum als eine Menge von RDF-Tripeln zu verstehen ist und in vielen verschiedenen syntaktischen Formen geschrieben werden kann. Allerdings wird die Bedeutung einer OWL-Ontologie von diesem Graph eindeutig vorgeschrieben.

Ein OWL-Dokument besteht aus optionalen Ontologie-Headern (normalerweise höchstens einem) und einer beliebigen Anzahl von Klassenaxiomen, Merkmalsaxiomen und Fakten über Individuen.

6.1 Ontologie-Header

Ein Header wird mit einer Instanz der Klasse `owl:Ontology` repräsentiert. Der Aufbau ist analog zu dem eines DAML+OIL-Headers. Ein Beispielheader (analog zu DAML+OIL, daher ohne Erklärungen):

```
<owl:Ontology rdf:about="">
 <owl:versionInfo>v 1.17 2003/02/26 12:56:51
                  mdean</owl:versionInfo>
 <rdfs:comment>Eine Beispiel-Ontologie</rdfs:comment>
 <owl:imports rdf:resource="http://www.example.org/foo"/>
</owl:Ontology>
```

6.2 Klassenaxiome

Axiome kann man in diesem Zusammenhang auch als Definitionen ansehen. Sie sind demnach (ohne Prämissen gültige) Aussagen über Klassen bzw. Merkmale. Eine Klasse bzw. ein Merkmal wird durch diese Aussagen eindeutig gekennzeichnet und definiert.

Die einfachste Form eines Klassenaxiomes ist eine bloße Zusicherung der Existenz einer Klasse, indem `owl:Class` mit einem Klassenidentifizierer benutzt wird. Beispielsweise deklariert das folgende Klassenaxiom die URI-Referenz `#Mensch` als den Namen einer OWL-Klasse:

```
<owl:Class rdf:ID="Mensch"/>
```

Dies ist korrekter OWL-Code, gibt aber kaum Information über die Klasse *Mensch* preis. Klassenaxiome enthalten typischerweise zusätzliche Komponenten, die

notwendige und/oder hinreichende Charakteristika einer Klasse angeben. Analog zu DAML+OIL gibt es hierfür die Sprachkonstrukte `rdfs:subClassOf`, `owl:equivalentClass` und `owl:disjointWith`. *subClassOf* und *disjointWith* wurden im Rahmen des Kapitels 5 bereits behandelt, außerdem werden alle drei im Anhang formal erklärt. Zu jedem folgt ein (selbsterklärendes) Beispiel:

```
<owl:Class rdf:ID="Oper">
  <rdfs:subClassOf rdf:resource="#Musikwerk"/>
</owl:Class>

<owl:Class rdf:about="#Deutscher_Bundeskanzler">
  <owl:equivalentClass
        rdf:resource="#Deutscher_Regierungschef"/>
</owl:Class>

<owl:Class rdf:about="Mann">
  <owl:disjointWith rdf:resource="#Frau"/>
</owl:Class>
```

6.3 Merkmalsaxiome

Analog zu den Klassenaxiomen definiert ein Merkmalsaxiom Charakteristika eines Merkmales. In der einfachsten Form sichert es nur die Existenz eines Merkmales zu:

```
<owl:ObjectProperty rdf:ID="hatElternteil"/>
```

Auch Merkmalsaxiome definieren meistens zusätzliche Eigenschaften. OWL unterstützt folgende Konstrukte:

- `rdfs:subPropertyOf`, `rdfs:domain` und `rdfs:range`
- `owl:equivalentProperty` und `owl:inverseOf`
- `owl:FunctionalProperty` und `owl:InverseFunctionalProperty`
- `owl:SymmetricProperty` und `owl:Transitive Property`

Einige dieser Konstrukte sind leicht anhand eines Beispieles zu verstehen, andere werden im Folgenden kurz erklärt:

`rdfs:subPropertyOf`: Hier genügt ein Beispiel:

```
<owl:ObjectProperty rdf:ID="hatMutter">
  <rdfs:subPropertyOf rdf:resource="#hatElternteil"/>
</owl:ObjectProperty>
```

`rdfs:domain`: Man kann für ein Merkmal (verschiedene) `rdfs:domain`-Axiome definieren. Syntaktisch ist `rdfs:domain` selbst ein eingebautes Merkmal, das ein Merkmal (also eine Instanz der Klasse `rdf:Property`) mit einer Klassenbeschreibung verbindet. Ein `rdfs:domain`-Axiom sichert zu, dass die Werte solcher Statements zu der dieser Beschreibung entsprechenden Klasse gehören müssen. Dieses Axiom wurde bereits in Abschnitt 5.4 behandelt, eine formale Erläuterung findet sich im Anhang. Es ist auch erlaubt, mehrere `rdfs:domain`-Axiome in einem Merkmal zusammenzufassen. Diese Zusammenfassung ist als Konjunktion zu interpretieren: sie beschränkt den zulässigen Bereich des Merkmales auf die Individuen, die zur Schnittmenge der Klassenbeschreibungen gehören.

`rdfs:range`: Ein `rdfs:range`-Axiom verbindet ein Merkmal entweder mit einer Klassenbeschreibung oder einem Datenbereich. Es sichert zu, dass die Werte dieses Merkmales zu der dieser Beschreibung entsprechenden Klasse oder zu Datenwerten des spezifizierten Datenbereiches gehören müssen. Auch dieses Axiom wurde bereits in Abschnitt 5.4 behandelt und wird im Anhang formal erläutert. Ebenso wie bei `rdfs:domain` ist es erlaubt, mehrere `rdfs:range`-Axiome in einem Merkmal zusammenzufassen, auch hier ist dies als Konjunktion zu interpretieren.

`owl:equivalentProperty`: Dieses Konstrukt kann benutzt werden, um festzulegen, dass zwei Merkmale dieselben Werte haben.

`owl:inverseOf`: Hier genügt ein Beispiel:

```
<owl:ObjectProperty rdf:ID="hatKind">
  <owl:inverseOf rdf:resource="#hatElternteil"/>
</owl:ObjectProperty>
```

`owl:FunctionalProperty`: Ein funktionales Merkmal ist ein Merkmal, das nur einen (eindeutigen) Wert y für jede Instanz x haben kann, d.h. es kann keine zwei verschiedenen Werte y_1 und y_2 geben, sodass die Paare (x,y_1) und (x,y_2) beide Instanzen dieses Merkmales sind. `owl:FunctionalProperty` wird von OWL als eine spezielle Unterklasse der RDF-Klasse `rdf:Property` definiert. Ein Beispiel:

```
<owl: ObjectProperty rdf:ID="Ehemann">
  <rdf:type rdf:resource="&owl;FunctionalProperty"/>
  <rdfs:domain rdf:resource="#Frau"/>
  <rdfs:range rdf:resource="#Mann"/>
</owl:ObjectProperty>
<owl:FunctionalProperty rdf:about="#Ehemann"/>
```

`owl:InverseFunctionalProperty`: Bei einem solchen Merkmal wird durch das Objekt eines Statements das Subjekt eindeutig festgelegt. Formal: Wenn P als `owl:InverseFunctionalProperty` deklariert wurde, ist damit festgelegt, dass ein Wert y nur der Wert einer einzigen Instanz x sein kann, d.h. es kann keine zwei

verschiedenen Instanzen x_1 und x_2 geben, sodass beide Paare (x_1,y) und (x_2,y) Instanzen von P sind. Beispiel:

```
<owl:InverseFunctionalProperty
                rdf:ID="leiblicheMutterVon">
  <rdfs:domain rdf:resource="#Frau"/>
  <rdfs:range rdf:resource="#Mensch"/>
</owl:InverseFunctionalProperty>
```

`owl:SymmetricProperty`: Ein symmetrisches Merkmal P hat die Eigenschaft, dass für jedes Paar (x,y), das Instanz von P ist, auch (y,x) eine Instanz von P ist.

`owl:TransitiveProperty`: Ein transitives Merkmal P hat die Eigenschaft, dass für alle Werte x,y,z gilt: Sind sowohl (x,y) als auch (y,z) Instanzen von P, dann ist auch (x,z) Instanz von P.

6.4 Axiome über Individuen (Fakten)

Individualaxiome (auch Fakten genannt) sind Statements über Individuen, die die Zugehörigkeit zu einer Klasse und Aussagen über relevante Merkmale angeben. Als Beispiel betrachten wir die folgende Menge von Statements über eine Instanz der Klasse *Oper*:

```
<Oper rdf:ID="Tosca">
  <hatKomponist rdf:resource="#Giacomo_Puccini"/>
  <hatLibrettist rdf:resource="#Victorien_Sardou"/>
  <hatLibrettist rdf:resource="#Giuseppe_Giacosa"/>
  <hatLibrettist rdf:resource="#Luigi_Illica"/>
  <PremiereDatumrdf:datatype="&xsd;date">1900-01-
                  14</PremiereDatum>
  <PremiereOrt rdf:resource="#Rom"/>
  <AnzahlDerAkte
   rdf:datatype="&xsd;positiveInteger">3</AnzahlDerAkte>
</Oper>
```

7 Eine konkrete Ontologie: DAML-S

DAML-S (*DAML for Services*) [9] ist eine in DAML+OIL geschriebene Sammlung von drei einzelnen Ontologien für Web-Dienste, die Dienstprovider mit einer Menge von Sprachkonstrukten für die Beschreibung der Merkmale und Fähigkeiten ihrer Web-Dienste in eindeutiger und maschineninterpretierbarer Form versorgt. Damit wird die Automatisierung der Service-Aufgaben (das Finden, Aufrufen und Ausführen automatischer Dienste sowie ihre Komposition und Interoperation) erleichtert. Man spricht auch kurz von der DAML-S-Ontologie (statt von einer Sammlung von Ontologien) und von Subontologien, wenn man die drei einzelnen

Ontologien betrachtet. Die drei Subontologien spezifizieren, was ein Web-Dienst tut (*Profile*), wie er funktioniert (*Process*) und wie er implementiert ist (*Grounding*).

Im Folgenden nehmen wir das Beispiel eines fiktiven Buchverkaufsservice, der von dem Webservice-Provider Congo Inc. angeboten wird. Congo bietet eine Reihe von Programmen im Internet an, die folgende selbsterklärende Namen haben: *LocateBook*, *PutInCart*, *SignIn*, *CreateAcct*, *CreateProfile*, *LoadProfile*, *SpecifyDeliveryDetails* und *FinalizeBuy*. Wir werden unser Augenmerk kurz auf die Erstellung einer Untermenge dieser Dienste in DAML-S richten.

Der erste Schritt der Erstellung eines Web-Dienstes besteht darin, ein Programm zu beschreiben, das diesen Dienst realisiert.

7.1 Definition der Programme als Prozesse

Der Buchverkaufsservice ist eine Sammlung aus kleineren Congo-Programmen, von denen jedes im Web zugänglich ist. Zuerst muss jedes dieser einzelnen Programme beschrieben werden, danach ihre Komposition, die letztendlich von der "Außenwelt" als einheitlicher Dienst wahrgenommen wird.

Definition der einzelnen Programme als Prozesse. Im Prozessmodell von DAML-S ist jedes Programm entweder ein atomarer oder ein zusammengesetzter Prozess. Der Vollständigkeit halber unterstützt das Modell zusätzlich die Notation eines einfachen Prozesses (*simple process*), die benutzt wird, um eine abstrakte Sicht eines atomaren oder zusammengesetzten Prozesses zu beschreiben:

```
<daml:Class rdf:ID="Process">
  <rdfs:comment> Die allgemeinste Klasse von Prozessen
                    </rdfs:comment>
  <daml:unionOf rdf:parseType="daml:collection">
    <daml:Class rdf:about="#AtomicProcess"/>
    <daml:Class rdf:about="#SimpleProcess"/>
    <daml:Class rdf:about="#CompositeProcess"/>
  </daml:unionOf>
</daml:Class>
```

Ein Beispiel für einen atomaren Prozess ist das *LocateBook*-Programm, das als Eingabe den Namen eines Buches erhält und die Beschreibung dieses Buches sowie seinen Preis ausgibt, soweit es im Katalog vorhanden ist. Der einfachste Weg, *LocateBook* als einen atomaren Prozess zu deklarieren, sieht so aus:

```
<daml:Class rdf:ID="LocateBook">
  <rdfs:subClassOf
         rdf:resource="&process;#AtomicProcess"/>
```

```
</daml:Class>
```

Mit jedem Prozess wird eine Menge von Merkmalen (oder auch Parametern) assoziiert. Zwei solcher Parameter sind *input* und *output*:

```
<rdf:Property rdf:ID="parameter">
  <rdfs:domain rdf:resource="#Process"/>
  <rdfs:range
rdf:resource="http://www.daml.org/2001/03/daml+oil#Thing"
/>
</rdf:Property>

<rdf:Property rdf:ID="input">
  <rdfs:subPropertyOf rdf:resource="#parameter"/>
</rdf:Property>
```

Die Eingabe für *LocateBook* ist üblicherweise der Name eines Buches:

```
<rdf:Property rdf:ID="bookName">
  <rdfs:subPropertyOf rdf:resource="&process;#input"/>
  <rdfs:domain rdf:resource="#LocateBook"/>
  <rdfs:range rdf:resource="&xsd;#string"/>
</rdf:Property>
```

Eingaben können erforderlich oder optional sein. Dagegen hängen Ausgaben generell von Bedingungen ab. Wenn man z.B. ein Buch im Congo-Katalog sucht, kann die Ausgabe eine detaillierte Beschreibung dieses Buches sein, wenn es im Katalog vorhanden ist, andernfalls etwas wie *„Sorry, das Buch ist nicht im Katalog vorhanden.“* Also definiert man eine Klasse *ConditionalOutput*, die sowohl die Bedingung als auch die an diese Bedingung gebundene Ausgabe beschreibt:

```
<rdf:Property rdf:ID="output">
  <rdfs:domain rdf:resource="#parameter"/>
  <rdfs:range rdf:resource="#ConditionalOutput"/>
</rdf:Property>

<daml:Class rdf:ID="ConditionalOutput">
  <daml:subClassOf
rdf:resource="http://www.daml.org/2001/03/daml+oil#Thing"
/>
</daml:Class>

<rdf:Property rdf:ID="coCondition">
  <rdfs:domain rdf:resource="#ConditionalOutput"/>
  <rdfs:range rdf:resource="#Condition"/>
</rdf:Property>

<rdf:Property rdf:ID="coOutput">
  <rdfs:domain rdf:resource="#ConditionalOutput"/>
```

```
  <rdfs:range
rdf:resource="http://www.daml.org/2001/03/daml+oil#Thing"
/>
</rdf:Property>
```

Durch die Bestimmung von Ein- und Ausgaben können die Programme für den Aufruf automatischer Web-Dienste benutzt werden. Für die Fähigkeit der automatischen Zusammenarbeit verschiedener Dienste müssen zusätzlich die "Seiteneffekte" eines jeden Programmes beschrieben werden, sofern welche existieren. In diesem Kontext können wir annehmen, dass Dienste die Merkmale *precondition* (Vorbedingung) und *effect* (Effekt) haben, die analog zu *input* und *output* beschrieben werden.

Vorbedingungen müssen erfüllt sein, damit ein Software-Agent einen Dienst ausführt. Viele Web-Services, die als Programme eingebettet sind, haben nur die Vorbedingung, dass die Eingabeparameter bekannt sein müssen.

Effekte sind wie Ausgaben optional. Konditionale Effekte charakterisieren die physikalischen Seiteneffekte von Service-Programmen. So ein Effekt kann z.B. für den *FinalizeBuy*-Dienst so aussehen: *Own(bookName), when cleared(creditCardNumber)*.

Definition von Kompositionen von Programmen als Kompositionen von Prozessen. Zusammengesetzte Prozesse werden durch Kontrollstrukturen wie z.B. *sequence*, *if-then-else*, *while*, *fork* usw. gekennzeichnet, die die Reihenfolge und die konditionale Ausführung der Prozesse in der Komposition lenken. Eine einfache Definition eines zusammengesetzten Prozesses kann so aussehen:

```
<daml:Class rdf:ID="CompositeProcess">
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Class rdf:about="#Process"/>
    <daml:Restriction daml:minCardinality="1">
      <onProperty rdf:resource="#composedOf"/>
    </daml:Restriction>
  </daml:intersectionOf>
</daml:Class>

<rdf:Property rdf:ID="composedOf">
  <rdfs:domain rdf:resource="#CompositeProcess"/>
  <rdfs:range rdf:resource="#ControlConstruct"/>
</rdf:Property>
```

Der ideale Vorgang beim Benutzen von CongoBuy wird durch zwei Hauptschritte gekennzeichnet: Suchen und Lokalisieren des Buches sowie, wenn vorhanden, der Kauf. Während die Lokalisierung als ein atomarer Prozess angesehen wird (s.o.), umfasst der Kauf eines Buches eine Folge von Subprozessen (andere atomare oder

auch selbst zusammengesetzte Prozesse), die z.B. eine Zahlungsmethode oder die Details der Lieferung (Adresse usw.) spezifizieren.

7.2 Anzeige der zur Verfügung gestellten Dienste

Die Beschreibungsmöglichkeiten von DAML-S bzw. DAML+OIL können für die Generierung einer Registratur von Diensten benutzt werden, um für Suchmaschinen bessere Eigenschaften zur Erstellung eines Index bzw. zum automatischen Finden geeigneter Web-Services zu erhalten.

Nun betrachten wir die Konstruktion eines DAML-S-Profiles für CongoBuy. Ein Dienstprofil ist eine Instanz der Klasse *Profile*, die in der Ontologie *profile* definiert ist. Mit dem `rdf:ID`-Attribut wird der Instanz eine ID übergeben, sodass sie von anderen Ontologien referenziert werden kann:

```
<profile:profile
rdf:ID="Profile_Congo_BookBuying_Service">
```

Die erste Menge von Informationen, die das Dienstprofil bereitstellt, besteht aus Beschreibungen des Dienstes und Informationen über den Provider:

isPresentedBy setzt das Profil mit dem Dienst, das es beschreibt, in Verbindung – in diesem Fall *Congo_BookBuying_Service*.

```
<service:isPresentedBy>
  <service:Service
    rdf:resource="&congo;#Congo_BookBuying_Service"/>
</service:isPresentedBy>
```

serviceName ist eine Kennung des Dienstes.

```
<profile:serviceName> Congo_BookBuying_Agent
</profile:serviceName>
```

textDescription ist eine für Menschen verständliche Beschreibung des Dienstes.

```
<profile:textDescription>
      Dieser Web-Dienst ermoeglicht es dem Benutzer,
      Buecher online zu suchen, zu bestellen und zu
      kaufen.
</profile:textDescription>
```

providedBy beschreibt den Provider des Dienstes.

```
<profile:providedBy>
  <profile:ServiceProvider rdf:ID="CongoBuy">
```

```
    <profile:name>CongoBuy</profile:name>
    <profile:phone>412 268 8780 </profile:phone>
    <profile:fax>412 268 5569 </profile:fax>
    <profile:email>Bravo@Bravoair.com</profile:email>
    <profile:physicalAddress>
      somewhere 2,
      OnWeb,
      Montana 52321,
      USA
    </profile:physicalAddress>
    <profile:webURL>
            http://www.daml.org/services/daml-
                    s/0.9/CongoBuy.html
    </profile:webURL>
  </profile:ServiceProvider>
</profile:providedBy>
```

Die zweite Menge von Informationen besteht aus zusätzlichen Attributen des Dienstes, z.B. Qualitätsgarantien oder Dienstrestriktionen.

geographicRadius spezifiziert, ob es eine Begrenzung der Dienstverteilung gibt. Beispielsweise legt man durch die Restriktion von *geographicRadius* auf "United States" fest, dass der Dienst Benutzern außerhalb der USA nicht angeboten wird.

```
<profile:geographicRadius
rdf:resource="&country;#UnitedStates"/>
```

qualityRating spezifiziert die Qualität des Dienstes nach vordefinierten Richtlinien.

```
<profile:qualityRating
rdf:resource="&concepts;#qualityRatingGood"/>
```

Die dritte Informationsmenge des Profiles enthält Attribute, die die Schlüsselelemente des Prozesses beschreiben, den dieses Profil charakterisiert. Schlüsselelemente sind Eingabe- und Ausgabeparameter sowie evtl. Beschränkungen im Gebrauch des Dienstes, z.B. Vorbedingungen und bedingte (Seiten-)Effekte des Dienstes. Die Art der Beschreibung von Ein- und Ausgabeparametern unterscheidet sich etwas von der im Prozessmodell, da nicht direkt die Prozessaktionen beschrieben werden.

Jede Eingabe erfordert einen Namen, eine Beschränkung der Informationen, die eingegeben werden, und eine Referenz auf die benutzte Eingabe des Prozess-Modelles.

```
<input>
  <profile:ParameterDescription>
    <profile:parameterName rdf:resource="bookTitle"/>
    <profile:restrictionTo rdf:resource="&xsd;#string"/>
    <profile:refersTo
```

```
        rdf:resource="&congo;#congoBuyBookName"/>
  </profile:ParameterDescription>
</input>
```

Ausgaben werden ähnlich wie Eingaben repräsentiert.

```
<output>
  <profile:ParameterDescription>
    <profile:parameterName rdf:resource="EReceipt"/>
    <profile:restrictedTo
         rdf:resource="&congoProcess;#EReceipt"/>
    <profile:refersTo
         rdf:resource="&congo;#congoBuyReceipt"/>
  </profile:ParameterDescription>
</output>
```

Dies war eine kurze (und natürlich unvollständige) Betrachtung der Konzeptuierung und Erstellung eines (fiktiven) Web-Service. Sie sollte als eine grobe Einführung in die generelle Vorgehensweise des Gebrauchs von Ontologien verstanden werden.

7.3 Generelle Beobachtungen

Die Stärke von DAML-S (wie auch anderer ontologischer Beschreibungssprachen) liegt darin, dass es einen Dienst mit einer semantischen, maschineninterpretierbaren Beschreibung versieht. Wie zu Beginn geschrieben, wird dadurch das dynamische Finden und Benutzen von Diensten wesentlich erleichtert bzw. überhaupt erst ermöglicht. Weitere nützliche Informationen können dadurch bereitgestellt werden, dass der Dienst anhand von Umgebungskonzepten beschrieben wird, die in einer Umgebungs-Ontologie Ausdruck finden. Dies ist z.B. für das Finden von passenden Ergänzungen wichtig, etwa, wenn der Benutzer alternative Begriffe bzw. Dienste sucht oder benutzt. Die Umgebungsontologie kann Relationen benutzen wie *SameClassAs*, um Synonyme zu identifizieren, oder *SubClassOf* für Ober- und Unterbegriffe.

Allerdings hat DAML-S auch einige Unzulänglichkeiten. Die für eine universell einsetzbare Beschreibungssprache sehr wichtige Flexibilität führt an einigen Stellen zu Unklarheiten. Das bedeutet, dass das zugrundeliegende konzeptuelle Modell ungenau ist. Beispielsweise gibt es keine klare Korrespondenz zwischen den DAML-S-Konzepten und denen des Software Engineering, was insofern Schwierigkeiten bereitet, als viele der (potentiellen) Benutzer von DAML-S Softwareentwickler sind. Außerdem sind die Verbindungen zwischen den einzelnen konzeptuellen Modellen unklar; die Beschreibungen mancher Merkmale und Beziehungen in der DAML-S-Dokumentation sind unzureichend. Inkonsistenzen entstehen bei Verbindungen zwischen Profil- und Prozessmodell, da nicht erzwungen ist, dass Parameter eines

bestimmten Types im Profil sich nur auf Parameter desselben Types im Prozess beziehen können. Mehr Informationen über diese Unzulänglichkeiten sowie Erfahrungsberichte von Entwicklern finden sich in [10].

Die Schwachpunkte zu beheben wird ein Hauptvorhaben der DAML-S-Koalition für die nächsten Versionen sein.

8 Konkrete Anwendungen im Semantic Web

Die vorgestellten Techniken finden in einem begrenzten Umfang bereits heute Verwendung. Im Folgenden werden einige wichtige Anwendungen und Anwendungsbereiche kurz vorgestellt.

8.1 Datenbankintegration

Informationssysteme großer Unternehmen umfassen häufig einige heterogene Datenbanken, die von verschiedenen Gruppen entwickelt wurden, und zwar in der Regel zu unterschiedlichen Zeiten und für unterschiedliche Zwecke. In vielen Fällen ist es wünschenswert bzw. erforderlich, Anfragen an diese Datenbanken sowie Informationen, die aus der Bearbeitung solcher Anfragen gewonnen werden, zu vereinen. Dies erfordert eine einheitliche Übersetzung von Termen (Namen und Kennungen von Beziehungen, Kategorien, Objekten usw.), d.h. eine Integration von semantischer Interpretation.

Als ein Beispiel hierfür ist die Boeing Corporation [11] zu nennen. Das Unternehmen benutzt für die Integration seiner Datenbanken (z.B. Flugzeugwartungs- und Flugzeugentwurfsdatenbanken) RDF als ein gemeinsames Datenmodell, weiterhin eine Anfragesprache, die auf RDF aufbaut, um die Zielanfragen zu formulieren, sowie eine Vermittlungsarchitektur, um die Anfragen sowohl syntaktisch (z.B. in SQL) als auch semantisch (in das Datenmodell der Zieldatenbank) zu übersetzen.

8.2 Finanzportale

In vielen Fällen sind die Finanzen einer Firma oder auch einer Einzelperson auf mehrere Organisationen verteilt (Banken, Aktieninvestitionen usw.). Jede dieser Organisationen bietet häufig einen web-basierten Zugang zu den Konten, der allerdings in der Regel für jede Organisation anders ist. Ein Finanzportal sammelt hingegen alle relevanten Informationen von den verschiedenen Online-Quellen und integriert sie in eine gemeinsame Sicht, um das finanzielle Management zu vereinfachen. Für die Erstellung solcher Portale bietet sich RDF an. Es findet bereits Anwendung im wohl führenden Finanzportal Spaniens, *GetSee* von iSoco [12].

8.3 Mozilla

Der Browser Mozilla [13] benutzt intern ebenfalls RDF als ein gemeinsames Format, um die vielen verschiedenen Datenressourcen, die dargestellt werden können (Bookmarks, History, Suchergebnisse, Dateisysteme, FTP, E-Mail-Header usw.), zu repräsentieren.

8.4 B2B-Märkte

Verschiedene Formen von B2B-Plattformen wurden bisher entwickelt, um es Leuten, die an bestimmten Produkten oder Diensten interessiert sind, zu ermöglichen, online passende Anbieter zu finden, Geschäfte zu vereinbaren und Transaktionen durchzuführen. Sowohl Werbung der Anbieter als auch Anforderungen der Käufer erforden komplexe Beschreibungen der Komponenten.

Für eine Verbesserung der gegenwärtigen Bedingungen wird daran gearbeitet, DAML+OIL als Beschreibungssprache für die Angebote und Forderungen zu benutzen, sodass intelligente Agenten dazu befähigt werden, passende Käufer und Verkäufer zusammenzuführen.

8.5 E-Business

Große Softwareunternehmen haben mit der Entwicklung von Plattformen begonnen, die die Automation von Web-Diensten unterstützen. Als Beispiele sind hierbei unter anderem zu nennen: *e-speak* von Hewlett Packard [14], Microsofts Werkzeuge *.NET* und *BizTalk* [15], *Application Framework for E-Business* von IBM [16] sowie *ONE (Open Network Environment)* von Sun [17].

9 Zusammenfassung, Ausblick

In dieser Arbeit wurden Ontologien sowie einige grundlegende Techniken zur Erstellung solcher Wissensrepräsentationen (RDF, DAML+OIL, OWL) einführend vorgestellt sowie anhand eines konkreten Beispieles (DAML-S) die gängigen Konzepte beleuchtet. Auch wurde in diesem Zusammenhang auf einige noch vorhandene Probleme mit konkreten Ontologien hingewiesen und ein Einblick in bereits existierende Anwendungsfälle gegeben.

Die gegenwärtigen Techniken und Problematiken erfordern noch viel Erweiterungs- und Forschungsarbeit. Zur Zeit wird das Semantic Web massiv in Forschung und Entwicklung angegangen, insbesondere vom World Wide Web Consortium. Gegenwärtig werden bereits gebräuchliche Technologien durch die Verwendung von semantischer Information verbessert.

Dennoch werden wegen der Dimension dieses Vorhabens in näherer Zukunft wohl keine einschneidenden Veränderungen im Netz der Netze eintreten. Anwendungen, in denen durch semantische Interpretation wesentliche Verbesserungen des Nutzens erreicht werden könnten, sind z.B. Suchmaschinen (wie eingangs erwähnt) und E-Learning.

Gleichwohl bleibt die keinesfalls unrealistische Vision einer vernetzten Welt, in der Maschinen nicht mehr nur zum passiven Informationsaustausch ohne Eigeninterpretation dienen, sondern aktiv am täglichen Handlungsprozess teilnehmen und das Leben der Menschen wesentlich erleichtern.

Anhang

Im Anhang werden einige wichtige DAML+OIL-Elemente erklärt. Die meisten dieser Elemente finden auch in OWL Verwendung (dann mit der Präfix `owl` statt `daml`).

Klassenelemente:

Ein Klassenelement (`daml:Class`) enthält die Definition einer Objektklasse. Es bezieht sich auf einen Klassennamen (einen URI; im folgenden sei C diese Klasse) und besteht aus einer optionalen Anzahl von

- `rdfs:subClassOf`-Elementen (wobei jedes einen Klassenausdruck enthält) – jedes solche Element sichert zu, dass C eine Unterklasse der Klasse ist, auf die sich das Element bezieht,

- `daml:disjointWith`-Elementen (wobei ebenfalls jedes einen Klassenausdruck enthält) – jedes solche Element sichert zu, dass C und der Klassenausdruck im Element disjunkt sind, d.h. C und die andere Klasse dürfen keine gemeinsamen Instanzen haben,

- `daml:disjointUnionOf`-Elementen (wobei jedes eine Liste von Klassenausdrücken enthält) – jedes solche Element sichert zu,dass alle Klassen, die von den Klassenausdrücken des `disjointUnionOf`-Elementes definiert werden, paarweise disjunkt sind (d.h. in der Liste existiert kein Objekt, das Instanz von mehr als einem Klassenausdruck ist),

- `daml:sameClassAs`-Elementen (wobei jedes einen Klassenausdruck enthält) – jedes solche Element sichert zu, dass C äquivalent zum Klassenausdruck im Element ist (d.h. C und die andere Klasse müssen dieselben Instanzen haben),

- `daml:equivalentTo`-Elementen (wobei jedes einen Klassenausdruck enthält) – ein solches Element hat dieselbe Bedeutung wie das `sameClassAs`-Element,

- booleschen Kombinationen von Klassenausdrücken – C muss äquivalent zu der Klasse sein, die von jeder dieser Kombinationen definiert wird,

- Aufzählungselementen – ein solches Element sichert zu, dass C genau die Instanzen enthält, die im Element aufgezählt werden.

Merkmals-Restriktionen:

Ein `daml:Restriction`-Element enthält ein `daml:onProperty`-Element, das sich auf einen Merkmals-Namen (einen URI; im folgenden sei P dieses Merkmal) bezieht, und besteht außerdem aus einem oder mehreren der folgenden Elemente:

- Elemente, die den Typ der Restriktion angeben:

- ein `daml:toClass` Element, das einen Klassenausdruck enthält – ein solches Element definiert die Klasse aller Objekte x, für die folgende Bedingung erfüllt ist: Wenn das Paar (x,y) eine Instanz von P ist, dann ist y eine Instanz des Klassenausdruckes (oder Datentypes),

- ein `daml:hasValue`-Element, das (eine Referenz auf) ein Individualobjekt oder einen Datentyp-Wert enthält – wenn y eine Instanz des Klassenausdruckes (oder Datentypes) ist, definiert ein solches Element die Klasse aller Objekte x, für die (x,y) eine Instanz von P ist,

- ein `daml:hasClass`-Element, das einen Klassenausdruck oder eine Datentyp-Referenz enthält – ein solches Element definiert die Klasse aller Objekte x, für die es (mindestens) eine Instanz y des Klassenausdruckes (oder Datentyps) gibt, so dass (x,y) eine Instanz von P ist.

Eine *toClass*-Restriktion ist analog zum universellen Allquantor der Prädikatenlogik – für jede Instanz der Klasse oder des Datentyps muss jeder Wert für P die Restriktion erfüllen. Die *hasClass*-Restriktion ist analog zum Existenzquantor – für jede Instanz der Klasse oder des Datentyps existiert ein Wert für P, der die Restriktion erfüllt.

- Elemente, die einen nichtnegativen Integer enthalten (welchen wir im folgenden als N bezeichnen), der eine unqualifizierte Kardinalitäts-Restriktion kennzeichnet:

- ein `daml:cardinality`-element – dieses definiert die Klasse aller Objekte, die genau N verschiedene Werte für das Merkmal P haben, d.h. x ist genau dann eine Instanz der definierten Klasse, wenn es genau N verschiedene Werte y gibt, so dass (x,y) eine Instanz von P ist,

- ein `daml:maxCardinality`-Element – dieses definiert die Klasse aller Objekte, die höchstens N unterschiedliche Werte für das Merkmal P haben,

- ein `daml:minCardinality`-Element – dieses definiert die Klasse aller Objekte, die mindestens N unterschiedliche Werte für das Merkmal P haben.

- Elemente, die einen nichtnegativen Integer N enthalten, der eine qualifizierte Kardinalitäts-Restriktion kennzeichnet, und die ein `daml:hasClassQ`-Element enthalten, das einen Klassenausdruck oder eine Datentyp-Referenz enthält:

- ein `daml:cardinalityQ`-Element – dieses definiert die Klasse aller Objekte, die genau N verschiedene Werte für das Merkmal P haben, wobei diese Werte Instanzen des Klassenausdruckes oder Datentyps sind. Genauer: x ist genau dann eine Instanz der definierten Klasse (x genügt der Restriktion), wenn es genau N verschiedene Werte y gibt, so dass (x,y) eine Instanz von P und y eine Instanz des Klassenausdruckes oder Datentyps ist,

- ein `daml:maxCardinalityQ`-Element – dieses definiert die Klasse aller Objekte, die höchstens N unterschiedliche Werte für das Merkmal P haben, die Instanzen des Klassenausdruckes oder Datentyps sind,

- ein `daml:minCardinalityQ`-Element – dieses definiert die Klasse aller Objekte, die mindestens N unterschiedliche Werte für das Merkmal P haben, die Instanzen des Klassenausdruckes oder Datentyps sind.

Man kann mehrere Restrikionen in einem einzigen *restriction*-Element zusammenfassen. In diesem Fall muss das Merkmal P alle diese Restriktionen erfüllen, d.h. die Zusammenfassung kann als Konjunktion verstanden werden.

Merkmals-Elemente:

Ein `rdf:Property`-Element bezieht sich auf einen Merkmalsnamen (URI, im folgenden als P bezeichnet). Merkmale, die in Restriktionen benutzt werden, sollten entweder Objekte mit anderen Objekten in Beziehung setzen (und sind dann Instanzen von *ObjectProperty*) oder Objekte mit Datentypen (und sind dann Instanzen von *DatatypeProperty*).

Ein Merkmals-Element enthält eine optionale Anzahl von

- `rdfs:subPropertyOf`-Elementen, wobei jedes einen Merkmalsnamen enthält – ein solches Element deklariert P als ein Untermerkmal des Merkmales, das im Element angegeben wird. Das bedeutet: formal: Jedes Paar (x,y), das Instanz von P ist, ist auch Instanz des benannten Merkmales,

- `rdfs:domain`-Elementen (wobei jedes einen Klassenausdruck enthält) – ein solches Element sichert folgendes zu: Ist (x,y) eine Instanz von P, dann ist x eine Instanz des Klassenausdruckes.

- `rdfs:range`-Elementen (wobei jedes einen Klassenausdruck enthält) – ein solches Element generiert folgende Bedingung: (x,y) kann nur dann eine Instanz von P sein, wenn y eine Instanz des Klassenausdruckes ist,

- `daml:samePropertyAs`-Elementen (jedes enthält einen Merkmalsnamen) – ein solches Element generiert die Bedingung, dass P äquivalent zum benannten Merkmal ist (d.h. beide müssen dieselben Instanzen haben),

- `daml:equivalentTo`-Elementen (dieselbe Semantik wie `samePropertyAs`),

- `daml:inverseOf`-Elementen (jedes enthält einen Merkmalsnamen) – ein solches Element bedeutet: Ist (x,y) eine Instanz von P, dann ist (y,x) eine Instanz des benannten Merkmales.

Eine Liste sämtlicher Sprachkonstrukte von DAML+OIL sowie die formalen Erklärungen finden sich in [7].

Literatur

[1] Die offizielle Website des Semantic Web: *http://www.semanticweb.org*

[2] Website des W3C: *http://www.w3.org*

[3] W3C-Seite zum Thema: *http://www.w3.org/2001/sw*

[4] URI-Spezifikation: *http://www.apache.org/~fielding/uri/rev-2002/draft-fielding-uri-rfc2396bis-01.html*

[5] RDF-Spezifikation: *http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/*

[6] Notation 3: *http://www.w3.org/DesignIssues/Notation3.html*

[7] DAML+OIL-Spezifikation:
http://www.w3.org/TR/2001/NOTE-daml+oil-reference-20011218

[8] OWL-Spezifikation: *http://www.daml.org/2002/06/webont/owl-ref-proposed*

[9] DAML-S Website: *http://www.daml.org/services/*

[10] Erfahrungsberichte von DAML-S-Entwicklern sowie Schwachpunkte der Ontologie: *http://www.iids.org/publications/essw03.pdf*

[11] Boeing Corporation: *http://www.boeing.com*

[12] Finanzportal *GetSee* von iSoco:
http://www.isoco.com/en/content/solutions/solution_getsee.html

[13] Website von Mozilla: *http://www.mozilla.org*
[14] *e-speak* von Hewlett Packard: *http://www.e-speak.hp.com*

[15] Microsofts Werkzeuge *.NET und Biztalk*:
www.microsoft.com/presspass/features/ 2002/sep02/09-30biztalk.asp

[16] *Application Framework for E-Business* von IBM:
http://www.research.ibm.com/journal/sj/401/flurry.html

[17] *Open Network Environment* von Sun:
http://wwws.sun.com/software/sunone/overview/

Self-Awareness

Julia Matt

Zusammenfassung. Self-Awareness im Forschungsgebiet des Autonomic Computing ist ein bis dato noch ungenau definierter Begriff. Er findet häufige Verwendung, doch eine einheitliche Definition, die für alle Systeme, die von sich behaupten self-aware zu sein, zutrifft, fehlt. Dabei bleibt auch die Frage offen, ob manche Systeme als self-aware bezeichnet werden dürfen. Dies ist ein Versuch, diesen Begriff etwas zu verdeutlichen.

1 Einleitung

Self-Awareness (Selbstbewusstsein) in der heutigen Software-Entwicklung - und vor allem im Forschungsbereich des Autonomic Computing - ist ein noch nicht genau spezifizierter Ausdruck. In der Praxis wird er im Zusammenhang mit unterschiedlichen Systemen oft verwendet. Doch sind diese Systeme tatsächlich self-aware?

Das Autonomic Computing hat diesen Begriff geprägt. Es baut darauf auf, dass autonome Systeme wie das menschliche Zentralnervensystem funktionieren sollten. Die Selbststeuerung benutzt beim Menschen motorische Nervenzellen, um indirekte Nachrichten an Organe, die auf einer Stufe sind, die weniger Bewusstsein hat, zu senden. Diese Nachrichten regulieren die Temperatur, das Atmen und den Herzschlag ohne bewusst darüber nachzudenken [1].

Ebenso besteht ein Autonomic Computing System aus vielen Ebenen und braucht daher detaillierte Kenntnisse über sich selbst – es muss self-aware sein. Das heißt aber wiederum, dass es Kenntnisse über seine Komponenten, seinen aktuellen Status und alle Verbindungen zu anderen Systemen braucht [2].

Beispiele für solche Systeme, die als self-aware bezeichnet werden sind unter anderem ABLE (Agent Building and Learning Environment) von IBM [3] und das System Definition Model, das momentan noch gemeinsam von Microsoft und HP entwickelt wird [4], [5].

Der erste Schritt in dieser Arbeit ist es Self-Awareness, wie es hier verwendet wird, zu definieren.

1.1 Definition von Self-Awareness

Ein Autonomic Computing System muss self-aware sein, d.h. es muss sich selbst kennen. Ein System wird als self-aware verstanden, wenn es seine Umwelt, seine Komponenten (Komponenten müssen eine Systemidentität haben), seinen aktuellen Status und alle seine Verbindungen zu anderen Systemen kennt. Es muss wissen, wie

groß seine Ressourcen sind, solche, die es leihen kann und solche, die es verleihen kann – welche isoliert werden sollten und welche mit anderen Systemen geteilt werden können.

Im Folgenden werden zwei Richtungen aufgezeigt, bei denen Self-Awareness verwendet wird. Zu jeder dieser Richtungen wird ein Beispiel vorgestellt, dass von sich behauptet, self-aware zu sein.

2 Self-Awareness unter Echtzeitbedingungen

Das erste Gebiet ist die Echtzeit. Dies beinhaltet, dass sich selbstanpassende Systeme zur Laufzeit selbst rekonfigurieren können müssen, d.h. sie müssen auf eine Umwelt, Ziele oder Systemressourcen, die sich verändern, reagieren können und zwar in einer bestimmten Zeitperiode.

Dieses Konzept besitzt unter anderem das Problem, dass das System fähig sein muss, seine Ziele, Ressourcen und Umgebungen zu kennen, d.h. es muss self-aware sein, um Veränderungen zu entdecken und sich selbst zu rekonfigurieren. Hinzu kommt das Problem, dass sie diese Selbstanpassung zur Laufzeit ausführen müssen und damit zeitlich begrenzt sind. Denn, wenn es versagt, könnte es, je nach Art der Echtzeitbedingung[1], zu katastrophalen Konsequenzen führen.

2.1 Beispiel: SA-CIRCA

SA-CIRCA (Self-Adaptive Cooperative Intelligent Real Time Control Architecture) wurde für intelligente, selbst-adaptive Systeme entwickelt, die Deadlines treffen müssen [6].

Es erstellt Kontrollpläne für die Ziele von Aufgaben in einem System und versucht diese einzuhalten. Ist es nicht möglich diese einzuhalten, so gibt es verschiedene Ansätze dies zu lösen.

Entwickelt wurde SA-CIRCA von der Automated Reasoning Group des Honeywell Technology Center [6]. Es basiert auf der bewährten CIRCA-Architektur, die für intelligente Echtzeitsysteme konstruiert wurde [7], [8].

2.1.1 Grober Aufbau von SA-CIRCA

Abb. 1 zeigt den groben Aufbau von SA-CIRCA. Es stellt dar, wie der Adaptive Mission Planner (AMP), das Controller Synthesis Module (CSM) und das Real Time Subsystem (RTS) interagieren, um denkfähige und adaptive Kontrolle, was die Ausführung in Echtzeit betrifft, zu erhalten. Alle Untersysteme von SA-CIRCA arbeiten dabei parallel.

[1] Harte Echtzeitbedingungen (z.B. Airbag) führen bei Nichteinhaltung zu katastrophalen Konsequenzen. Weiche (z.B. Videostream)hingegen haben keine so schwerwiegenden Konsequenzen bei Nichteinhaltung zur Folge. Ein kurzes Überschreiten der Deadline kann meist verschmerzt werden.

Ganz oben steht der AMP. Er beschäftigt sich mit Langzeitzielen, Problemstrukturen und die nächstliegenden Deadlines, um zu entscheiden, welches die nächsten Ziele sind und auf welche Probleme das nähere Denken fokussiert werden soll.

Der AMP sendet Unterziele und Problemkonfigurationen an das CSM, das Kontrollpläne zur Einhaltung der Zeitperiode, die für die Lösung einer Aufgabe zur Verfügung steht (Echtzeitkontroller), entwickelt, um die nächstliegenden Ziele zu erreichen. Diese Kontrollpläne sind wie die Menge von rückwirkenden Regeln, die spezifizieren, welche Aktionen das System in all den möglichen, verschiedenen Weltsituationen unternehmen kann.

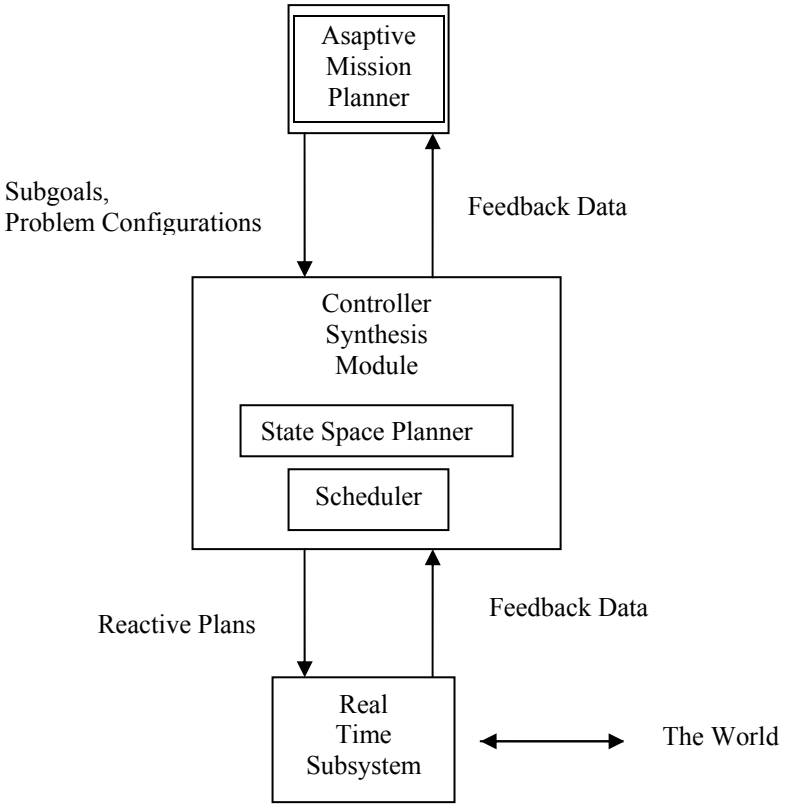


Abb. 1. grober Aufbau von SA-CIRCA

Das CSM sendet diese möglichen Kontrollpläne an das RTS, welches rückwirkend die automatisch generierten Kontrollpläne ausführt und garantierte Antwortzeiten durchsetzt.

Im Voraus gebildete Kontrollpläne werden vom RTS in einen Cache-Speicher aufgenommen. Wenn es jedoch keinen gespeicherten Kontrollplan für eine Situation gibt, dann nimmt das System seinen besten vorhandenen Kontrollplan und der AMP weist das CSM an, einen neuen, angepassten Kontrollplan so schnell wie möglich zu entwickeln. Jeder Kontrollplan, der auf dem RTS ausgeführt wird, muss das System sichern, während er auf das CSM wartet, das den nächsten Kontrollplan erzeugt.

Währenddessen beschäftigen sich der AMP und das CSM weiter mit zukünftigen unvorhergesehenen Fällen.

Das CSM beinhaltet dazu noch einen State Space Planner (SSP) und eine Komponente, die den zeitlichen Ablauf organisiert (Scheduler). Dieser SSP erzeugt Kontrollpläne, die auf einem Weltmodell und einer Menge von zuvor definierten Sicherheitsbedingungen basiert. Der Benutzer einer Domäne beschreibt seine Domäne mit Transitionsbeschreibungen, die unbedingt die Menge der erreichbaren Zustände definieren muss. Diese nutzt der SSP dann um einen nichtdeterministischen endlichen Automaten (NEA) zu erstellen. Der SSP bestimmt dazu eine Aktion zu jedem erreichbaren Zustand.

Diese Aktionen werden danach ausgesucht, wie viele Zielabsichten aus den so gewonnenen Zuständen des Systems erfüllt werden können. Auch sollen sie helfen Übergänge zu verhindern, die zu Fehlern führen.

Wird nun SA-CIRCA z.B. in einem Roboter in der Fabrikation eingesetzt, so müssen vom Menschen die Fähigkeiten des Roboters, die Prozesse in seiner Umgebung und die allgemeinen Systemziele beschrieben werden. Der Mensch sagt dem System also nicht, wann es was machen soll, sondern nur was es machen kann und dass es, wenn es dies nicht in einer bestimmten Zeit erledigt, versagt. SA-CIRCA erstellt dann eine Menge von reaktiven Richtlinien und führt formale Verifikationsprozesse durch, um zu garantieren, dass alle Deadlines durch die vom SSP erstellten Kontrollpläne eingehalten werden.

Es gibt momentan zwei verschiedene Versionen des SSP von SA-CIRCA: der originale SSP, der sich mit einer nichtdeterministischen Welt beschäftigt ohne explizit quantitative Unsicherheiten zu vertreten und der Probabilistic State Space Planner (PSSP) aus CIRCA-II[2], der sich ausschließlich mit den Wahrscheinlichkeiten der verschiedenen Übergänge und Zustände beschäftigt. In beiden Fällen kann das Zusammensetzen von Kontrollplänen durch die Abstimmung der Aspekte der SSP-Eingabe kontrolliert werden: die Problemstellung, die es lösen soll, und die Parameter, die genau die Art einer akzeptablen Lösung beschreiben. Für den nicht-wahrscheinlichkeitstheoretischen SSP besteht die Problemstellung aus folgendem:

- **Aktionsübergänge**: repräsentieren potentielle Aktionen, die der SSP eventuell aufrufen kann und dabei sicher sein kann, dass er nach einer maximalen Zeitspanne Ergebnisse bekommt
- **Ereignisübergänge**: repräsentieren unkontrollierbare, aktuelle Ereignisse
- **Zeitliche Übergänge**: repräsentieren unkontrollierbare Prozesse aus der Umwelt, die zumindest ein Minimum an Zeit benötigen

[2] Ein Vorgänger von SA-CIRCA, der nicht selbst adaptiv ist.

- **Sichere zeitliche Übergänge**: repräsentieren Prozesse, die sicher auftreten, wenn ihnen eine gewisse Zeit gegeben wird
- **Startzustände**: beschreiben mögliche Zustände, in denen das System starten oder „aufgeweckt“ werden könnte
- **Ziele**: beschreiben erstrebenswerte Zustandsmerkmale

Der PSSP benutzt zusätzliche Eingabeinformationen - unter anderem Wahrscheinlichkeitsfunktionen, die jedem Übergang beigeordnet sind oder einen einzigen Wahrscheinlichkeitsgewichtsparameter, der kontrolliert, wie zurückhaltend der PSSP beim Nachdenken über niederwahrscheinliche Zustände sein sollte.

Im obigen Beispiel eines Roboters in der Fabrikation würde der PSSP nun Wahrscheinlichkeiten aufstellen, wie wahrscheinlich es ist, dass ein Kontrollplan in einer bestimmten Situation benutzt werden kann.

Der Kontrollplan des RTS wird aus der Menge der geplanten Aktionen im NEA gewonnen und in der Form von Test Action Pairs (TAPs) gestaltet. Jedes TAP hat einen Testausdruck, der eine Untermenge von NEA (Welt-) Zuständen anerkennt und eine einzelne Aktion, die für diese Zustände geplant wurde.

TAPs, die Fehlern vorbeugen, haben zeitliche Begrenzungen. Der Scheduler organisiert sie in einem festen, zyklischen Zeitplan, damit sie durch das RTS ausgeführt werden können. TAPs, die keinem Fehler vorbeugen und nur geplant wurden um andere Ziele, die nicht sicherheitskritisch sind, zu erreichen, werden in einer bestmöglichen Art und Weise ausgeführt.

2.1.2 Sicherheit von SA-CIRCA

Dieses SA-CIRCA steht solange für Sicherheit und hochqualitative Leistung, solange eine Domäne exakt modelliert ist, d.h. solange klar ist, was von einer Aufgabe erwartet wird, und solange sie erfolgreich durch die vorhandenen Ausführungsressourcen kontrolliert werden kann. D.h. es gibt Kontrollpläne für alle Situationen, die während der Lösung der Aufgabe auftreten können. Wenn nun aber der Punkt erreicht ist, an dem das System keinen Plan bilden kann, der alle seine Ziele bewältigt und Systemsicherheit garantiert, kann SA-CIRCA dies erkennen und versucht es zu beheben.

Dies tritt z.B. ein, wenn der SSP keinen passenden Plan in der Menge der möglichen Reaktionspläne mehr findet, der Fehler verhindern könnte. Oder der SSP braucht zu lange um einen Plan zu bilden und wird von einem Timer Interrupt unterbrochen. Zuletzt bleibt noch die Möglichkeit, dass der SSP eine Menge von gewünschten Reaktionen hat, der Scheduler sie nur nicht in seinem Zeitplan unterbringen kann.

Ein Lösungsansatz dieser Probleme besteht darin, dass der SSP den Kontrollplan wegen Mangel an einem anderen Kandidaten zurückzieht, um eine andere Wahl zu treffen und einen veränderten Reaktionsplan zu erstellen.

Oder der AMP entscheidet sich stattdessen, einen Kompromiss zu erzeugen und einige Aspekte des Kontrollproblems zu vereinfachen, damit der CSM einen ausführbaren Kontrollplan erstellen kann.

Diese Kompromissmethoden sind selbst nicht heuristisch. Sie können durch einfache Prozeduren implementiert werden, indem sie eingeschränkte Veränderungsmöglichkeiten an der SA-CIRCA Datenstruktur, die das Weltmodell oder den momentanen Kontrollplan oder ähnliches beschreiben, wahrnehmen. SA-CIRCA weiß dabei genau, welche Wirkungen die verschiedenen Kompromisse haben. Welcher Kompromiss genommen werden sollte, ist eine heuristische Entscheidung, die bis dato noch nicht in Entwicklung ist.

Ein Beispiel für einen Kompromiss wäre, einfach einen oder mehrere temporäre Übergänge, die zu Fehlern im Weltmodell (siehe Abb. 1) führen, zu löschen oder zu ignorieren.

2.1.3 Fazit des Beispiels SA-CIRCA

SA-CIRCA kann automatisch individuelle Kontrollpläne für autonome Systeme zusammenfassen, da es die Fähigkeit besitzt, über Ressourcenbegrenzungen nachzudenken und die Kontrollprobleme abzuwandeln. Es kann zum Einen überbelastete Domänen, d.h. das System kann nicht auf diesem Niveau ausgeführt werden, auf dem es eigentlich ausgeführt werden sollte (einige Aspekte müssen vereinfacht werden), erkennen. Auch kann es die potentiellen Effekte von verschiedenen Veränderungen auf seine Ziele und Pläne analysieren. Und alles muss SA-CIRCA in Echtzeit bewerkstelligen.

3 Self-Awareness im Hinblick auf Funktionalität

Das zweite Gebiet ist die Funktionalität. Dies bedeutet, dass sich selbstanpassende Systeme selbst kennen müssen, um zu wissen, wann sie Änderungen durchzuführen haben und wann nicht. Sie müssen, je nachdem auf welche Änderungen sie reagieren sollen, ihre Umwelt, ihre Komponenten und auch ihre Kenntnisse über sich selbst kennen, d.h. sie müssen self-aware sein.

In diesem Abschnitt wird mehr Wert auf die korrekte Ausführung einer Aufgabe gelegt als auf die Zeit, die für deren Ausführung benötigt wird – es wird auf Funktionalität gesetzt.

Die meisten Beispiele hierfür sind im Gebiet der Software-Agenten zu finden. Zwei Beispiele in diesem Gebiet wären die Multi-Agenten Systeme und Software-Agenten und ihre Agent-Factory. Im Folgenden wird ein Beispiel zu Software-Agenten und deren Agent-Factories aufgeführt.

3.1 Beispiel: automatisierte (Weiter-) Entwicklung von Software-Agenten

Software-Agenten und ihre Agent-Factory sind unter anderem ein Forschungsgebiet von der Intelligent Interactive Distributed Systems Group, Division of Computer Science, Faculty of Sciences der Universität von Amsterdam [9].

In diesem Gebiet kann sich ein Software-Agent, der von einem Entwickler gestaltet wurde, in einer für den Entwickler ganz unerwarteten Weise weiterentwickeln. Und so wird dann aus einem entwickelten Produkt ein dynamisches Produkt.

3.1.1 Software Agenten als dynamische Produkte

Dynamische Produkte sind Produkte, die sich durch ihre Umwelt beeinflussen lassen, d.h. wenn sich ihre Umwelt verändert, verändern sie sich auch. Beispiele für dynamische Produkte sind enthalten in:

- Gebäuden, die Licht und Temperatur durch Analysieren des Raumes anpassen [10]
- Autopiloten in Flugzeugen, die die Kontrolle aus den Händen des Menschen nehmen und diese wieder zurückgeben
- Selbstgestaltung von autonomen (Raumfahrzeug-) Systemen [11]

Software-Agenten sind spezielle Typen von autonomen Systemen, die sich als ein Ergebnis der Interaktion mit ihrer Umgebung und/oder Kommunikation mit anderen Agenten verändern. Personifizierung ist ein Beispiel: Agenten, die Informationen sammeln, halten oft Profile von anderen Agenten (möglicherweise von Menschen [12]) aufrecht und adaptieren diese Profile, indem sie mit diesen Agenten interagieren.

Um dies zu können, müssen die Agenten die folgenden Eigenschaften besitzen [9]:

- **Self-Awareness Kenntnisse**: Der Agent muss über das Wissen verfügen, das die Funktionalität und das Verhalten des Agenten beschreibt.
- **Kontrollinformation**: Der Agent muss dazu in der Lage sein, seine Funktionalität und sein Verhalten in einer gegebenen Situation zu kontrollieren.
- **Kenntnisse über Selbsteinschätzung**: Der Agent muss Kenntnisse haben, die die Ausmaße festlegen, bis zu welchem Punkt ein Agent in einer gewissen Situation funktioniert.
- **Kenntnisse über den Gebrauch von Formulierungen:** Der Agent braucht Kenntnisse, mit denen er das Benötigen von Adaption bestimmen kann.
- **Integration:** Self-Awareness Kenntnisse, Kontrollinformationen, Kenntnisse über Selbsteinschätzung und Kenntnisse über den Gebrauch von Formulierungen müssen in die (interne) Funktionalität eines self-aware Agenten integriert werden.
- **Kommunikation:** Der Agent muss Kenntnisse darüber besitzen, nach welchen Kriterien er eine Agent-Factory aussucht und wie er mit ihr interagiert.

In wieweit diese Kenntnisse, Informationen und Funktionalitäten gebraucht werden, hängt von der Bauweise eines Agenten ab.

Ein sich anpassender Agent muss sich seiner Fähigkeiten bewusst sein. Dieses Wissen über seine Fähigkeiten kann einfach in seiner Form (z.B. Fakten) oder aufwendiger (z.B. ein Modell von seinem Verhalten) sein. Wichtig ist dabei, dass ein Agent erkennt, in welchen Situationen er korrekt und in welchen er nicht korrekt funktionieren wird. Auch mündet das Vorhandensein oder Nichtvorhandensein von verschiedenen Fähigkeiten in einen starken Glauben an Erfolg oder Misserfolg.

Ein Agent hat also einen gewissen Grad an Self-Awareness. Dieser reicht von fast komplettem Nichtbewusstsein bis hin zu komplettem Bewusstsein, wobei der Agent seine Fähigkeiten ganz versteht und somit mit einer hohen Sicherheit sagen kann, ob er eine Aufgabe ausführen kann oder nicht.

Wenn jedoch die Self-Awareness Kenntnisse eines Agenten sehr ungenau sind, muss die Agent-Factory die Verantwortung übernehmen – je mehr, je ungenauer die Self-Awareness Kenntnisse sind.

3.1.2 Agent-Factory

Eine Agent-Factory entwickelt und adaptiert Agenten. Sie basiert auf fünf grundlegenden Annahmen [13]:

- Agenten haben eine zusammengesetzte Struktur.
- Wieder verwendbare Teile eines Agenten können identifiziert werden.
- Es werden zwei Ebenen von Beschreibungen benutzt: eine konzeptionelle und eine detaillierte.
- Die Eigenschaften und die Kenntnisse dieser Eigenschaften sind abrufbar.
- Keine Festlegung auf bestimmte (Programmier- oder Modellierungs-) Sprachen werden gemacht.

Das Design eines Agenten in einer Agent-Factory basiert auf dem Aufbau eines Building Blocks. Building Blocks bestehen aus Gehäusen und partiellen (Agenten-) Entwicklungen.

Agent-Factories haben Prozessnummern. Manche sind mit der Kundenverwaltung, d.h. welcher Kunde verlangt welche Servicequalität, verknüpft.

Das Design-Zentrum (auch ein Prozess) ist für das Entwickeln und Weiterentwickeln von Agenten verantwortlich. Es basiert auf dem Modell für (Weiter-) Entwicklung von zusammengesetzten Systemen [14]. Dieses Modell modelliert das Nachdenken über Strategien, Anforderungen und Produktbeschreibungen.

In dem Design-Zentrum werden also die Bedürfnisse nach Adaption der Agenten als eindeutige und spezifische Anforderungen verstanden, die dann die Grundlage der Pläne für einen Agent bilden. Hier werden dann auch die Strategien dringend benötigt, um festzustellen, welche Anforderungen wann und wie manipuliert werden müssen, um dann die Pläne zu manipulieren. Weiterhin braucht man die Strategien, um den (Weiter-) Entwicklungsprozess zu leiten.

Die Agent-Factory geht bei einer Adaption eines Agenten folgendermaßen vor: Wenn der Agent nun eine Anforderung nach Adaption schickt, wird diese in mehrere Mengen von qualifizierten Anforderungen übersetzt. Auf dieser Basis wird dann der

Plan des Agenten geändert. Diese beiden Manipulationen sind jedoch nicht trivial. Es muss herausgefunden werden, wann die Manipulationsprozesse angesteuert werden können, wie parallele Aktivitäten synchronisiert werden sollen, usw.

3.1.3 Building Blocks

Building Blocks in einer Agent-Factory sind entweder Komponenten mit offenen Steckplätzen, komplett spezifizierten Komponenten oder eine Kombination von beiden. Sie werden auf einer der zwei Ebenen, entweder der konzeptionellen oder der detaillierten, definiert. Also ist eventuell eine Zuordnung zwischen den Building Blocks auf konzeptioneller Ebene und denen auf detaillierter Ebene von Nöten.

Building Blocks selbst sind konfigurierbar, können aber nicht willkürlich zusammengesetzt werden. Ihre Bibliothek besitzt eine feste Maximalgröße.

Die Zusammensetzung der konzeptionellen Building Blocks, die der detaillierten Building Blocks und die Zuordnung zwischen diesen beiden Zusammensetzungen müssen manipuliert werden. Manipulation der Building Blocks bedeutet hier, dass die Building Blocks mit spezifischen Eigenschaften abgefragt werden müssen, um richtig mit anderen Building Blocks gruppiert zu werden. Teilweise zusammenpassende Building Blocks müssen (durch andere Building Blocks) adaptiert werden, bevor sie zusammengesetzt werden können. Das „offene Steckplätze“ Konzept wird benutzt um zu regulieren, wie die Komponenten kombiniert werden können. Ein offener Steckplatz in einer Komponente besitzt Eigenschaften, die der Anwärter auf diesen Steckplatz erfüllen muss, um ihn zu bekommen.

3.1.4 Beispiel eines self-aware Information Retrieval Agenten, der mit einer Agent-Factory kommuniziert

Um die Zusammenarbeit zwischen einem Agenten und einer Agent-Factory zu verdeutlichen, soll hier ein Beispiel, dessen Szenario in Abb. 2 dargestellt wird, angeführt werden. Es besitzt einen Klienten, der eine Anfrage an einen Information Retrieval Agent schickt. Diese Anfrage soll in unserem Beispiel die Buchung eines Fluges von Frankfurt nach Mailand sein. Dazu benötigt der Agent eine Ressource, die die Informationen zu dem gewünschten Flug besitzt.

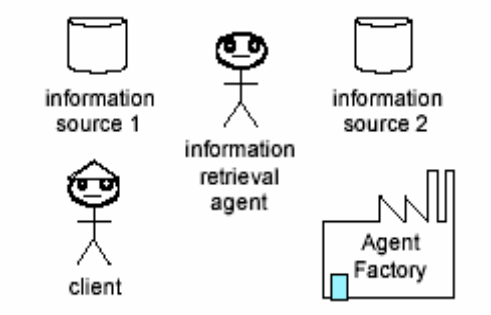


Abb. 2. Beispiel zu einem Agent – Agent-Factory - Szenario

Aus der Sicht des Agenten

Der Agent sucht nun die Ressource auf, um sich die Information zu beschaffen und sie dem Klienten weiterzugeben. Allerdings stellt er fest, dass er die Ressource nicht versteht, da sie in Deutsch geschrieben ist und er nur Englisch versteht. Der Agent hat nun einen gewissen Grad an Self-Awareness. Die Höhe dieses Grades beeinflusst die Fähigkeit des Agenten sein Bedürfnis nach Adaption auszudrücken.

In diesem Beispiel besitzt der Agent die folgenden Eigenschaften, die Interaktionen mit anderen Agenten und externen Objekten betreffen:

I_speak_protocol(http)

I_speak_protocol(fipa_acl_v1.0b)

I_understand_ontology_framework(xml)

I_understand_ontology_framework(rdf)

I_understand_ontology_framework(daml+oil)

I_understand_ontology(airplane_seat_reservation_English)

Die Informationsressource hat z.B. die folgende Meta-Beschreibung:

expressed_in_ontology_framework(daml+oil)

expressed_in_ontology(airplane_seat_reservation_German)

Aus diesen beiden Informationen kann nun der Agent herauslesen, dass er die Informationsressource nicht verstehen kann. Auch wenn er nun die Ontologie herunterladen würde, hätte er immer noch nicht das Verständnis der Ontologie. Also stellt er einen Antrag wie folgt an die Agent-Factory, damit sie ihn adaptiert:

request(possibility_for_adaption)

required_functionality(I_understand_ontology(
airplane_seat_reservation_German))

Die Agent-Factory quittiert die Anfrage und der Agent nimmt Kontakt zu einem Verzeichnisdienst auf, entdeckt dort eine Agent-Factory, die in der Nähe ist, der er vertraut und die dazu in der Lage ist, die gewünschte Veränderung durchzuführen, und wandert dorthin ab. Hier wird er anschließend adaptiert. In diesem Beispiel ist der Agent nicht „lebendig" während der Adaption. Er wird erst wieder „aufgeweckt", wenn die Adaption beendet wurde und er wieder an seinen ursprünglichen Platz zurückgekehrt ist. Der Agent versteht nun den Inhalt der Informationsressource und kann die Anfrage des Klienten bearbeiten und ihm die gewünschten Informationen liefern.

Aus der Sicht der Agent-Factory

Agenten mit einem hohen Grad an Self-Awareness sind dazu in der Lage, spezielle Bedürfnisse an Adaptionen zu stellen. So könnte z.B. der Agent aus dem obigen Beispiel seine Anfrage so gestellt haben:

request(possibility_for_adaption)

required_functionality(I_understand_ontology(
airplane_seat_reservation_German))

Ein Agent möchte aber vielleicht sein „Gedächtnis" nach der Adaption wieder haben. Dazu muss er nicht nur seinen Plan an die Agent-Factory schicken, sondern auch sein „Gedächtnis", da das eventuell auch geändert werden muss um seinem Plan wieder zu entsprechen.

Nun hat die Agent-Factory also (hier in diesem Falle) den Plan des Agenten angenommen und kennt die gewünschte Adaption. Es folgen einige Schritte.

Einer der ersten Schritte besteht darin, die Anfragen in konkretere Anfragen zu verfeinern:

required_functionality(I_understand_ontology(
airplane_seat_reservation_German))

in:

requirement(hard,interpret ontology(airplane_seat_reservation_German)
as ontology(airplane_seat_reservation_English))

Diese Anfrage ist nun ein Mapping, bei dem die deutsche Ontology auf die englische abgebildet wird. Diese Anfrage kann noch weiter verfeinert werden, bis aus ihr eine Menge von qualifizierten Anfragen resultiert, die dann von der Agent-Factory bearbeitet werden kann.

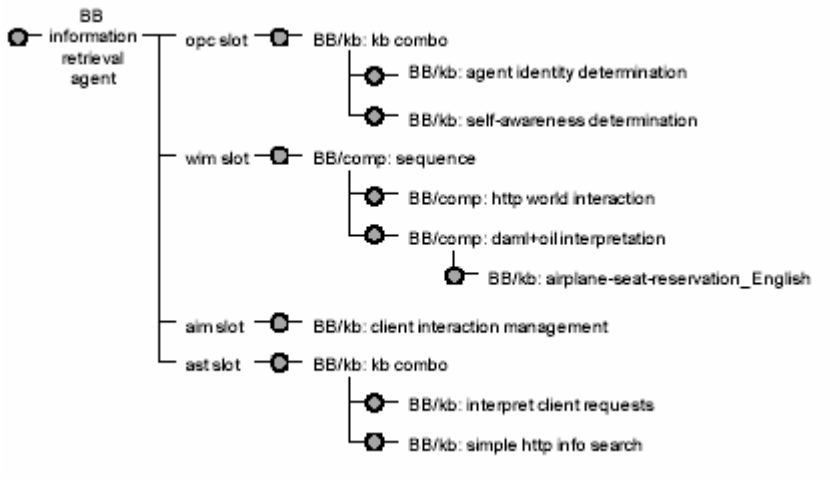


Abb. 3. Agent vor Besuch bei der Agent-Factory

Der Plan des Agenten wurde nun mit diesen spezielleren Anfragen verändert. In Abb. 3 wird der Agent vor seinem Besuch in der Agent-Factory dargestellt. Abb. 4 zeigt ihn danach.

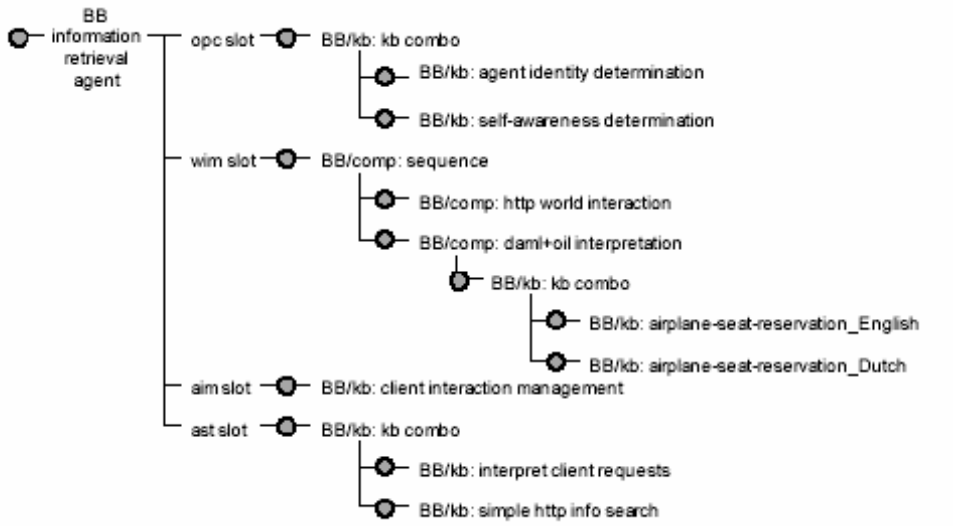


Abb. 4. Agent nach Besuch bei der Agent-Factory

Dieser Agent basiert auf dem Generic Agent Model [15], in dem separate Prozesse für die Kontrolle über die Agenten (*own process control*), Interaktionen mit Objekten aus der externen Welt (*world interaction management*), Interaktionen mit anderen Agenten (*agent interaction management*) und spezielle Tasks eines Agenten (*agent specific task*) getrennt werden.

Die Adaption eines Agenten ist ein Weiterentwicklungsprozess. Innerhalb der Agent-Factory basiert er auf dem Generic Model of Design [16]. Dieses Modell unterscheidet zwischen dem Nachdenken über die Gesamtkoordination des Weiterentwicklungsprozesses, der Manipulation von Mengen von qualifizierten Anfragen und der Manipulation der Pläne der Agenten.

3.2 Fazit des Beispiels: automatisierte (Weiter-) Entwicklung von Software-Agenten

Agenten müssen so entwickelt werden, dass sie adaptiv sind. Sie müssen erkennen können, wann ein Bedürfnis nach Veränderung vorhanden ist und sich dann an die Agent-Factory mit Anfragen wenden können. Diese müssen die Anfragen annehmen und die Agenten nach ihren Anfragen adaptieren.

4 Vergleich: Echtzeit - Funktionalität

Die beiden hier vorgestellten Gebiete, in denen Self-Awareness genutzt wird, sind sehr verschieden.

Die Echtzeit legt sehr hohen Wert auf die zeitlich korrekte Ausführung von Aktionen, während in der Funktionalität, wie schon der Name sagt, mehr Wert auf eine korrekt ausgeführte Aktion gelegt wird. Die Dauer der Ausführung spielt dabei keine Rolle.

Am Beispiel des Software-Agenten war das sehr gut zu sehen. Der Software-Agent bekommt eine Aufgabe zugewiesen. Zuerst versucht er, sie selbständig zu lösen. Wenn er nun einen, wenn auch nur recht niedrigen Grad an Self-Awareness besitzt, kann er feststellen, ob er die Aufgabe lösen kann oder nicht. Kann er es, so bekommt er eine Erfolgsmeldung. Schafft er es jedoch nicht, die Aufgabe zu lösen, so wendet er sich an die Agent-Factory und formuliert seine Anfrage. Diese nimmt seine Anfrage entgegen. Er kann nun in einem Verzeichnis nachschlagen und sich eine vertrauenserweckende Agent-Factory in der Nähe aussuchen, die seine Adaption durchführen soll. Während einer Adaption ist ein Agent „lebendig“ oder nicht. Ist er nicht „lebendig“, so wird er „aufgeweckt“, sobald die Adaption vollendet wurde und er an seinem Platz zurückgekehrt ist. Hier kann er dann endlich die Aufgabe ausführen, die er ausführen sollte.

Wenn der Agent auch sein „Gedächtnis“ behalten möchte, so muss er auch dieses zusammen mit seinem Plan an die Agent-Factory übergeben.

Es wird also deutlich, dass es keine Rolle spielt, wie lange ein Agent braucht um seine Aufgabe zu erfüllen. Es ist von Vorteil, wenn der Agent seine Aufgabe in minimaler Zeit bewältigen kann, ist hier aber nicht gefordert; es gibt hier keine zeitlichen Begrenzungen.

Im Beispiel des SA-CIRCA hingegen kann die zeitliche Begrenzung, die mit einer Aufgabe verbunden ist, sehr gut gesehen werden. Ein System, in dem SA-CIRCA eingesetzt wird, soll eine Aufgabe in einer bestimmten Zeit erledigen. Der AMP (siehe Abb. 1) unterteilt diese Aufgabe in Teilziele und erörtert, welche Ziele zuerst behandelt werden sollten, da sie die nächstliegende Deadline besitzen. Diese Teilziele schickt er weiter an den CSM, der mit Hilfe des SSP Kontrollpläne zum Erreichen gewisser Etappenzustände erstellt. Der SSP kann entweder der originale SSP sein, der sich mit einer nichtdeterministischen Welt beschäftigt ohne explizit quantitative Unsicherheiten zu vertreten, oder der PSSP, der sich ausschließlich mit den Wahrscheinlichkeiten der verschiedenen Übergänge und Zustände beschäftigt.

Die Kontrollpläne, die nun vom SSP erstellt wurden, werden dann an das RTS weitergegeben, das sie, wenn es sie nicht sofort braucht, in einem Cache speichert. Es ist aber durchaus wahrscheinlicher, dass der Kontrollplan direkt benötigt wird, da zumeist aktuelle Aufgaben erledigt werden und eigentlich keine Zeit bleibt, im Voraus schon zu wissen, welche Teilaufgaben auftreten können. Und selbst wenn diese bekannt sind, hat das CSM meist nicht genügend Zeit, die aktuell benötigten und noch weitere Kontrollpläne zu erstellen.

Probleme treten dann auf, wenn kein passender Kontrollplan im Cache vorhanden ist, der Fehler verhindern könnte, und die Zeit abläuft (evtl. harte Echtzeitbedingungen). Oder aber der SSP braucht zu lange um einen Kontrollplan zu erstellen und wird von einem Timer Interrupt unterbrochen. Oder der SSP hat eine Menge von gewünschten Reaktionen, der Scheduler kann sie jedoch nicht in seinem Zeitplan unterbringen.

Als Lösung kann ein Kompromiss eingesetzt werden, der eventuell Ziele herunterschraubt, Fehler ignorieren lässt oder ähnliches.

Das System ist also an strenge zeitliche Begrenzungen gebunden und muss bei Nichterreichen der Ziele bei der Funktionalität Abstriche machen.

Die beiden Gebiete gehen, wie hier deutlich wurde, in zwei ganz unterschiedliche Richtungen. Bei der Funktionalität wird grundsätzlich nicht darauf geachtet, welche Zeit zur Verfügung steht um eine Aufgabe zu lösen. Es ist also sinnvoll diese Richtung nur einzusetzen, wenn keine Probleme (größere oder kleinere) bei Nichteinhaltung zeitlicher Begrenzungen auftreten. Allerdings liegt diese Abwägung, ob diese Probleme tragbar sind oder nicht, beim Entwickler und/oder Administrator eines Einsatzgebietes. Wenn ein Agent eine Aufgabe beherrscht, dürfte es keine Probleme geben, ihn auch bei zeitkritischen Gebieten einzusetzen.

Die Echtzeitbedingungen hingegen setzen eher auf die Einhaltung des zeitlich vorgegebenen Rahmens und machen lieber Abstriche, was die Funktionalität anbelangt. Ob ein Entwickler und/oder Administrator das lieber in Kauf nimmt, ist die andere Frage.

In Bezug auf die Self-Awareness scheinen beide die übliche Definition richtig verstanden zu haben.

Die Systeme kennen beide ihre Umgebung, wissen was dort passiert und können auf Veränderungen reagieren. SA-CIRCA reagiert auf nichtausführbare Ziele, indem es diese Ziele herunterschraubt und somit die Aufgabe erleichtert auf oder auf neue Aufgabenstellungen. Software-Agenten reagieren auf die Anfragen, die sie bekommen und können bei nichtvorhandenem Wissen, das sie zur Bearbeitung brauchen, sich dieses bei einer Agent-Factory aneignen.

In beiden Beispielen sind die Komponenten der Systeme den Systemen bekannt – SA-CIRCA weiß z.B., welche Pläne es im Cache hat, Software-Agenten wissen z.B., über welches Wissen sie verfügen.

Auch kennen die Systeme aus beiden Beispielen ihren aktuellen Status – z.B. aktiv - nichtaktiv, nichtwissend - wissend.

Ihre maximale Kapazität ist auch beiden bekannt – SA-CIRCA kann über Ressourcenbegrenzungen nachdenken, ein Agent kann nicht alles erlernen; es kommt auf seinen Grad an Self-Awareness an.

Und zuletzt kennen beide auch ihre Verbindung zu anderen Systemen – SA-CIRCA hat die Verbindungen zwischen AMP, CSM und RTS, Agenten haben Verbindungen untereinander über die Agent-Factories oder eben zu diesen Agent-Factories. Agenten können untereinander Ressourcen ausleihen.

Es besitzen also beide eine gewisse Self-Awareness, auch wenn sie vielleicht bei den Software-Agenten deutlicher zu sehen ist. Somit haben beide ihre Behauptung, self-aware zu sein, erfüllen können.

5 Zusammenfassung und Ausblick

Self-Awareness ist zwar ein noch ungenau definierter Begriff, wird aber schon verwendet. Die beiden Beispiele, die in dieser Arbeit vorgestellt wurden, können beide von sich behaupten, self-aware zu sein. Vielleicht erfüllen sie nicht jede Eigenschaft von Self-Awareness, wie es in dieser Arbeit definiert wurde, zur vollsten Zufriedenheit, doch gehen sie bei jeder Eigenschaft in die richtige Richtung.

SA-CIRCA konzentriert sich dabei mehr auf die Erfüllung der Echtzeitbedingungen. Es versucht, die Deadlines zu treffen und wenn es dies nicht schafft, so erzeugt es neue Kontrollpläne oder schwächt eventuell auch ihre Bedingungen, die die Ausführung der Aufgabe am Anfang gestellt hat, etwas ab. Dazu benötigt es nur am Anfang die Hilfe eines Menschen. Danach arbeitet es selbständig, was Self-Awareness ausmacht.

Die Software-Agenten werden am Anfang auch durch einen Entwickler programmiert und passen sich dann nach und nach ihrer Umgebung an. Sie können sich dabei in eine für den Entwickler unvorhersagbare Richtung weiterentwickeln. Der Entwickler hat also keinen Einfluss mehr darauf, was ebenfalls Self-Awareness ausmacht.

Bei Software-Agenten gibt es die Erweiterung zu Multiagentensystemen. Sie beinhalten autonome Agenten, die gegeneinander und miteinander arbeiten. Allerdings stellen sie kein geschlossenes System dar und werden auch nicht in einem Stück entworfen. Sie stellen lediglich die Infrastruktur zur Verfügung, die für die Kommunikation und die Interaktion benötigt werden.

SA-CIRCA wird heute in Multi-Aircraft Systemen verwendet [17]. Es dient dort zur autonomen Steuerung von Flugzeugen.

Bis dato ist für solche selbst-adaptiven Systeme (sowohl SA-CIRCA wie auch Software-Agenten) noch kein Test- und/oder Beweisverfahren entwickelt worden. Weiterhin fehlt das Vertrauen, da sie nicht so leicht wie konventionelle Software zu verstehen sind. Auch sind die Mächtigkeit und die Möglichkeiten, die mit diesen Systemen einhergehen, noch nicht vollständig bekannt. Die Forschung steckt also, wie man sieht, noch in den Kinderschuhen.

Referenzen

[1] http://www.research.ibm.com/autonomic/overview/solution.html
[2] http://www.research.ibm.com/autonomic/overview/elements.html
[3] J. P. Bigus, D.A. Schlosnagle, J.R. Pilgrim, W. N. Mills III, Y. Diao, “ABLE: A tollkit for building multiagent autonomic systems”, 2002 IBM SYSTEMS JOURNAL, VOL. 41, Nr. 3
[4] http://www.gridtoday.com/03/0512/101403.html
[5] http://www.microsoft-watch.com/article2/0,4248,930310,00.asp
[6] David J. Musliner, “Imposing Real-Time Constraints on Self-Adaptive Controller Synthesis”, 2000 Proc. Int´l Workshop on Self-Adaptive Software, Honeywell Technology Center
[7] David J. Musliner, E.H. Durfee and K. G. Shin, “CIRCA: A Cooperative Intelligent Real-Time Control Architecture”, 1993 IEEE Trans. Systems, Man, and Cybernetics, vol. 23, no. 6, S. 1561-1574
[8] David J. Musliner, E.H. Durfee and K. G. Shin, “World Modeling for Dynamic Construction of Real-Time Control Plans”, 1995 Artificial Intelligence, vol. 74, no. 1, S.83-127
[9] Frances M. T. Brazier and Niek J. E. Wijngaards, “Automated (Re-) Design of Software Agenten”, 2002 Vrije Universiteit Amsterdam
[10] M. C. Mozer, “An Intelligent Environment Must Be Adaptive”, 1999 IEEE Intelligent Systems And Their Applications, 14(2), S. 11-13
[11] B. C. Williams and P. P. Nayak, “A Model-Based Approach to Reactive Self-Configuring Systems”, 1996 AAAI Press/MIT Press, 2, S. 971-978
[12] N. Wells and J. Wolfers, “Finance with a Personalized Touch”, 2000 Communications of The ACM, 43(8), S. 31-34
[13] F. M. T. Brazier and N. J. E. Wijngaards, “Automated Servicing of Agents”, 2001 Journal ofAISB, special issue on Agent Technology, S.5-20
[14] F. M. T. Brazier, C. M. Jonker, J. Treur and N. J. E. Wijngaards, „Compositional Design of a Generic Design Agent“, 2001 Design Studies Journal, 22, 439-471
[15] F. M. T. Brazier, C. M. Jonker and J. Treur, „Compositional Design and Reuse of a Generic Agent Model“, 2000 Applied Artificial Intelligence Journal, 14, 491-538
[16]F. M. T. Brazier, P. H. G. Van Langen, Zs. Ruttkay and J. Treur, “On Formal Specification of Design Tasks”, 1994Proceedings Artificial Intelligence in Design (AID´94), S. 535-552
[17] http://www.htc.honeywell.com/projects/ants/9-99-ants-review.ppt

Self-Monitoring and Self-Optimisation

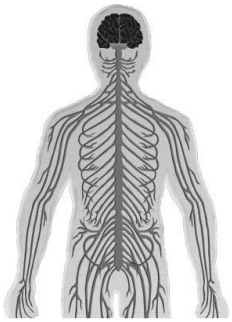
Haiko Gaißer

Zusammenfassung Immer komplexer werdende heterogene Systeme verursachen immer höhere Kosten für ihre Verwaltung. Bisher vorhandene Software zur Unterstützung ist zu sehr abhängig vom jeweiligen Hersteller.
Mit *Web Based Enterprise Management* hat die *Desktop Management Taskforce* einen herstellerunabhängigen Standard zum Management socher Systeme geschaffen und inzwischen liegen erste Implementierungen vor.

1 Einleitung

Heutige Rechnernetze werden immer komplexer und immer heterogener. Viele unterschiedliche Geräte vom Handheld-PC bis zum Großrechner interagieren. Die Verwaltung dieser Systeme ist eine sehr komplexe Aufgabe, die immer mehr Arbeitszeit in Anspruch nimmt. Diese Kosten manchen einen großen Anteil an den *Total Cost of Ownership* (TCO [1]), deren Senkung ein erklärtes Ziel der gesamten IT-Industrie ist.

Einigen Schätzungen zu Folge werden in den kommenden Jahren bis zu 200 Millionen Fachkräfte allein zur Bewältigung dieser Aufgaben benötigt [14]. Als Konsequenz hierfür investieren die großen IT-Unternehmen immer höhere Beträge in die Forschung und Entwicklung auf dem Gebiet des *Autonomic Computing* [22].



Darunter versteht man Computersysteme, die sich selbst überwachen, um sich selbstständig zu reparieren, optimieren sowie konfigurieren, ohne dabei auf externe Hilfe angewiesen zu sein. Diese Abläufe sollen völlig automatisch und vor allem transparent von statten gehen, ähnlich dem menschlichen autonomes Nervensystem.

Um eine solche Autarkie eines Systems erreichen zu können, bedarf es zuvor einer Möglichkeit, die an den verschiedenen Sensoren anfallenden Informationen zu bündeln und darauf zugreifen zu können.

Auch bisher gibt es schon recht viele Ansätze, um diesem Problem Herr zu werden, die jedoch noch einige Nachteile haben. Zum einen sind die meisten Lösungen sehr herstellerspezifisch, zum anderen werden Teilbereiche wie mobile Endgeräte überhaupt nicht abgedeckt. Zwar besteht natürlich die Möglichkeit,

[1] als TCO bezeichnet man die mit einem Computersystem verbundenen administrativen Kosten für Anschaffung, Installation, Wartung und Betrieb.

individuelle Anpassungen vorzunehmen, allerdings ist dieser Ansatz sehr teuer und es entsteht eine starke Abhängigkeit vom jeweiligen Anbieter.

Diese Ausarbeitung beginnt im zweiten Abschnitt mit einem kurzen Zusammenfassung des Managementstandards WBEM. Darauf folgt im dritten Abschnitt ein kurzer Überblick über Microsofts WBEM Implementierung *Windows Management Instrumentation* (WMI), sowie eine Übersicht über die verfügbaren WBEM Implementierungen.

2 WBEM

2.1 Einleitung

Aus den beschriebenen Gründen haben sich im Juli 1996 mehrere Hersteller, darunter BMC Software [2], Cisco Systems [3], Compaq [4], Intel [5] und Microsoft [6] zu einer Gruppe zusammengeschlossen, um den herstellerübergreifenden Standard WBEM (*Web Based Enterprise Management*)zu schaffen. Ziel war es, auf der Basis von bestehenden Web-Technologien (unter anderem HTTP und XML), einen Standard zur einheitlichen Verwaltung von verteilten Systemen, wie sie heutzutage in allen großen Unternehmen vorzufinden sind, zu schaffen und Interoperabilität zwischen den Produkten der verschiedenen Hersteller zu garantieren.

Diese Aufgabe wurde im April 1998 an die *Desktop Management Taskforce* (DMTF [1]) übergeben, der heute viele Hard- und Softwarehersteller angehören, was darauf hoffen lässt, dass die Bedeutung von WBEM wächst und damit die Chance, dass sich der Standard durchsetzen kann. Erst wenn genügend Druck auf andere Hersteller entsteht, werden diese ihre Produkte an den neuen Standard anpassen. Sollte die DMTF ihr Ziel erreichen und sich der Standard wie gehofft durchsetzen, so darf man in Zukunft auf eine verringerte Komplexität der Mangement-Anwendungen und auf eine geringere Herstellerabhängigkeit hoffen. WBEM bedeutet allerdings nicht, dass es das Ziel ist, den Browser als Management-Konsole zu verwenden, sondern dass vorhandene Techniken und Methoden verwendet werden sollen. Das bedeutet beispielsweise, dass zur Kommunikation XML via HTTP verwendet wird. Es ist folglich kein Ersatz für etablierte Enterprise-Management-Systeme wie Tivoli Enterprise von IBM/Tivoli [8] oder Unicenter TNG von Computer Associates [7] geschaffen werden. Vielmehr ist das primäre Ziel eine Vereinheitlichung der Schnittstellen und Techniken, mit denen die Managementinformationen gesammelt werden und wie der Zugriff darauf erfolgt.

2.2 Die Komponenten von WBEM

WBEM setzt sich aus den folgenden drei Komponenten zusammen (siehe auch Abbildung 1):

CIM Das *Common Information Model* (CIM) stellt allgemein und implementierungsunabhängig Format, Sprache und Methodik zur Beschreibung und zum

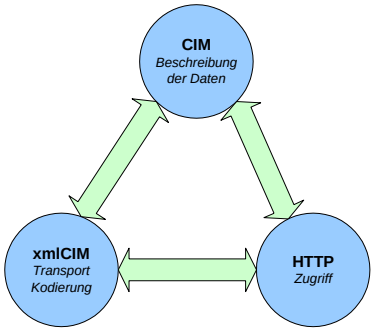


Abbildung 1. Die Komponenten von WBEM

Sammeln von Verwaltungsinformationen zur Verfügung. Aus diesen Informationen wird dann eine Datei im *Managed Object Format* (MOF) generiert. MOF ist Sprache zur textuellen Definition von Klassen und Instanzen, die dann zum *CIM Object Manager Repository* hinzugefügt werden können.

xmlCIM zur Repräsentation der CIM-Deklarationen und Operationen in Form von XML-Elementen. Mit Hilfe eines entsprechenden Übersetzers wird daraus eine MOF-Datei generiert, die die Beschreibung der Komponente enthält.

CIM/HTTP-Operationen als protokoll-unabhängige Abbildung von CIM Objekten und Operationen auf auf das weit verbreitete und bewährte Internet-Protkoll HTTP (*Hypertext Transfer Protocol*, [25]).

2.3 Aufgaben- und Problemstellung, Alternativen

Eines der Hauptprobleme ist die Verfolgung und Erkennung von Nachrichten der überwachten Geräte. Dies wird von den bisher verbreiteten Netzwerkmanagement-Protokollen relativ schlecht abgedeckt. Dazu kommt, dass das *Simple Network Management Protocol* (SNMP [20]) hauptsächlich in der Welt der IP-Netzwerke verbreitet ist, während sich das *Common Management Information Protocol* (CMIP [21]) primär auf den Telekommunikations-Sektor beschränkt. Allerdings gibt es gerade im Telekommunikationsbereich eine Tendenz hin zu IP-basierten Diensten, so dass auch dort SNMP immer beliebter wird. RMON (Remote Monitoring [19]) hingegen ist primär für den Einsatz in Geräten zur Überwachung von Netzwerken, die einen Großteil ihrer Ressourcen dieser Aufgabe widmen, gedacht. SNMP wiederum ist ein recht schlankes Protokoll, weshalb es gerade bei kleineren Netzwerk-Komponenten wie HUB's und Switches weit verbreitet ist. Solche Geräte wären mit komplexeren Protokollen relativ schnell überfordert. Also wird auch eine Möglichkeit benötigt, vorhandene Hardware, die auf bestehenden Techniken aufsetzt, zu integrieren. Das lässt sich bei CIM/WBEM recht einfach mit Hilfe von sogenannten Objekt-Providern bewerkstelligen, die als Brücke zu anderen Managementstandards dienen können.

Für die Hard- und Software Hersteller, die Mitglieder in der DMTF sind, besteht die Möglichkeit, ihre Produkte auf Standardkonformität prüfen zu lassen. Dazu muß ein Hersteller eine entsprechende Datei im MOF-Format, die eine detailierte Beschreibung der Komponente enhält, sowie einer eindeutigen Identifikation und die zugehörige CIM-Versionsnummer einreichen. Der Zertifizierungsprozeß beinhaltet ein Programm, mit dessen Hilfe die korrekte Verwendung und Definition des Schemas überprüft werden kann.

2.4 CIM-Schema

CIM ist ein abstraktes, objektorientiertes Datenmodell mit dem Ziel, alle Komponenten eines Computer-Systems vollständig beschreiben zu können. Als Datenformat dient hierzu das oben erwähnte *Managed Object Format* (MOF) - ähnlich wie die *Management Information Base* (MIB) bei den bisher verbreiteten Managementprotokollen SNMP und RMON. Die gesammelten Informationen werden in einer zentralen Datenbank im sogenannten *Management Information Format* (MIF) abgelegt. Das *Management Information Format* ist Teil des *Desktop Management Interface* (DMI), einer anderen Initiative der DMTF. Dabei handelt es sich um eine Syntax zur Beschreibung von Komponenten und deren Attributen.

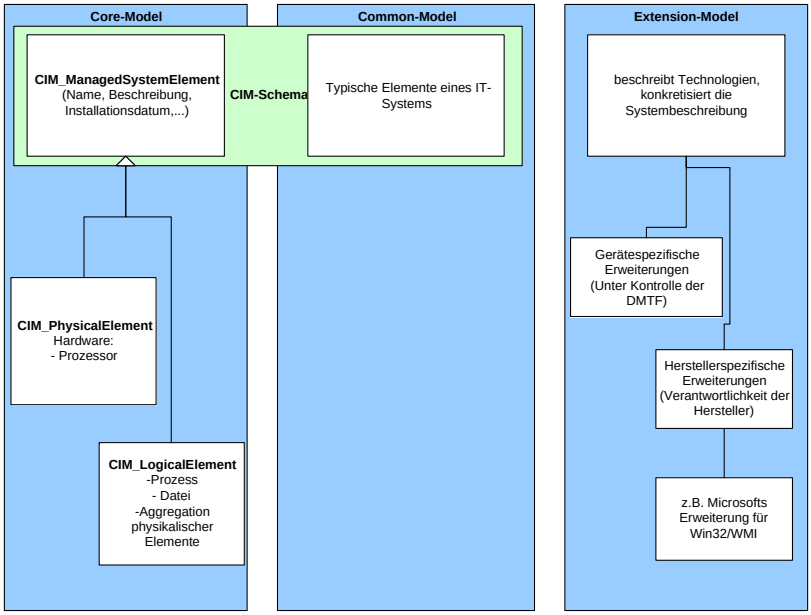


Abbildung 2. CIM Schema

Das *Core* Modell Das *Core*-Modell beschreibt die Grundelemente aus denen Rechner bestehen. Ziel ist eine grundlegende Klassifizierung der Elemente und Beziehungen der zu verwaltenden Umgebung. Die zentrale Klasse ist hier `CIM_ManagedSystemElement`. Sie repräsentiert Systeme, Systemkomponenten, jegliche Art von Diensten, Software und Netzwerke.

Von ihr sind die Unterklassen für physische (`CIM_PhysicalElement`) und logische Elemente (`CIM_LogicalElement`) abgeleitet.

Des Weiteren existieren unter anderem Klassen für Produkt-Beschreibungen, Einstellungen und Konfiguration, Sammlungen und statistische Daten. Von hier aus dehnt sich das Modell in viele Richtungen aus, mit dem Ziel, möglichst umfassend alle Problembereiche und Beziehungen zwischen den zu verwaltenden Elementen des Systems erfassen zu können.

Der Begriff *System* ist im CIM-Kontext relativ weit gefasst. Er kann für alles, von einen einzelnen Rechner bis hin zu ganzen Netzwerk-Domänen oder Anwendungsystemen, verwendet werden.

Die *Common* Modelle Die *Common*-Modelle beschreiben die typischen Elemente eines IT-Systems, die unabhängig von einer bestimmten Technologie oder Implementierung sind. Beispiele hierfür sind Rechnersysteme, Anwendungen sowie Netzwerke. Die Klassen, Beziehungen und Methoden aus dem *Common*-Modell sind möglichst detailliert gehalten, um eine Sicht auf das zu modellierende Gebiet zu bieten, die als Basis für den Entwurf oder sogar für die Implementierung von Software verwendet werden kann. Es wird davon ausgegangen, dass das *Common* Modell nach und nach durch Objekte aus dem *Extension* Modell erweitert wird. Zusammengefasst werden die *Core*- und *Common*-Modelle als *CIM-Schema* bezeichnet (Abbildung 2).

Die Common-Modelle im CIM Schema 2.7 sind folgende:

Application Das *Application Model* beschreibt Informationen, die normalerweise benötigt werden, um Software und Anwendungen zu verwalten und zu verteilen. Es bietet die Möglichkeit, Softwarestrukturen von unabhängigen Einzelplatzanwendungen bis hin zu verteilten, Internet-basierten Softwaresystemen zu beschreiben und zu modellieren. Die aktuelle Fassung beinhaltet im Wesentlichen drei Konzepte:

1. Die Struktur der Anwendung
2. Den Lebenszyklus der Anwendung
3. Die Übergänge zwischen den verschiedenen Zuständen eines Zyklus.

Event Ein *Event* beschreibt typischerweise einen Zustandswechsel (Ereignis) des Systems oder eines Teils davon, welcher meistens eine bestimmte Reaktion des Systems zur Folge hat. Ein Ereignis kann eine Zwischenfall sein, der nur gelegentlich eintritt (wie zum Beispiel ein Neustart) oder es beschreibt einen häufiger vorkommender Zwischenfall wie zum Beispiel ein eingehendes Netzwerkpaket

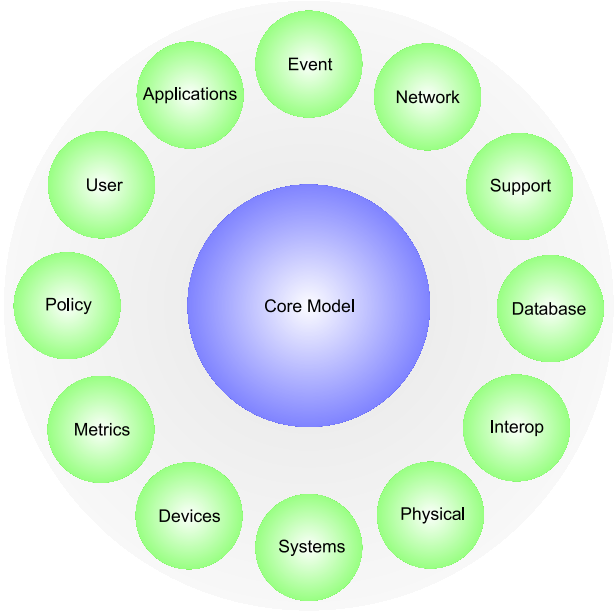


Abbildung 3. Die CIM Common Modelle

sein. Einige dieser Ereignisse erfordern eine sofortige Reaktion des Benutzers (z.B. eine ausgefallene Kühlung), andere sind erst zu einem späteren Zeitpunkt (z.B. bei der nächsten Systemwartung) von Interesse.

Network Dieses Modell dient zur Beschreibung und Verwaltung der Netzwerkanbindung und des Netzwerks selbst sowie der angebotenen Dienste und Protokolle im Netzwerk. Die verwalteten Einheiten werden grob in sieben Kategorien unterteilt:

1. Netzwerksysteme
2. Netzwerkdienste
3. logische Verbindungen und Zugriff
4. Netzwerkprotokolle
5. Netzwerktechnologien (Netzwerk-Segmente oder virtuelle Netzwerke)
6. Dienstgüte-Technologien (*Quality of Service*, QoS)
7. andere unterstützende Definitionen

Das Modell beschreibt ein Netzwerk als eine Verwaltungseinheit, die selbst wieder andere Netzwerke, Teilnetze oder Domänen enthalten kann.

Support Durch den starken Zuwachs an Komplexität und gegenseitiger Abhängigkeit der Produkte in den letzten Jahren, wurde die Zusammenarbeit der Produkte von unterschiedlichen Herstellern zunehmend bedeutender. Um dieses Ziel

erreichen zu können, erwarten die Kunden immer mehr Unterstützung durch die verschiedenen Hersteller und deren Support-Partner, was eine immer intensivere Zusammenarbeit dieser Firmen erfordert. Diese Tendenz hat einen Bedarf geschaffen, gegenseitig auf Unterstützungsinformationen zugreifen zu können. Das *Support*-Modell ermöglicht eine gemeinsame Darstellung zum Austausch von diesen Unterstützungsinformationen, mit dem Ziel, eine noch intensivere und effektivere Zusammenarbeit zu erreichen.

Database Das CIM Modell für Datenbanken orientiert sich stark an der in der RFC 1697 [26] der *Internet Engineering Taskforce* (IETF, [27]) beschriebenen Modellierung und unterteilt entsprechend eine Datenbank in drei Hauptkomponenten (Abbildung 4):

- das Datenbanksystem, das die softwaretechnischen Aspekte der Anwendung beschreibt.
- die gemeinsame Datenbasis, die eigentlichen Datenbestand repräsentiert
- den Datenbank-Dienstgeber, der den Prozeß der auf die Anfragen an die Datenbank reagiert.

Dazu kommen noch Hilfsklassen, deren Aufgabe es ist, Konfigurationsparameter, Betriebsmittel und Statistiken zu beschreiben.

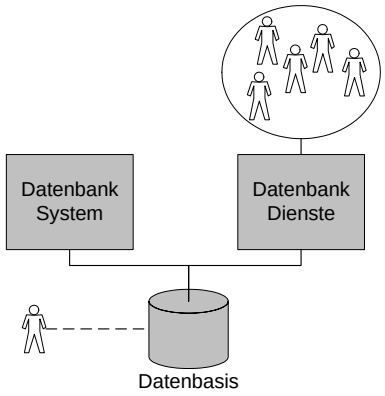


Abbildung 4. Das CIM Datenbank-Modell

Physical Das *Physical*-Modell dient der Erfassung und Verwaltung von physischen Komponenten mit dem Ziel einer Inventarisierung und der Bestandspflege. Das Zusammenspiel zwischen diesen Komponenten wird mit Hilfe von Assoziationen modelliert, meistens in Form von *beinhaltet* und *Ortsbestimmung*. Dabei ist es wichtig, zu verstehen, dass die Abstraktionen des *Physical* Modells auch typischerweise den realen physischen Komponenten eines Computersystems entsprechen und nicht der Funktionalität, die diese Komponenten bieten können.

Systems Die CIM Systems Common Modelle beschreiben einen Zusammenschluss von Teilen oder Komponenten zu einer großen, ganzen Verwaltungseinheit. Viele der Abtraktionen, die im Zusammenhang mit Computer Systemen verwendet werden, sind von der zugehörigen Klasse dieser Abtraktion abgeleitet. Neben dem Computersystem an sich befasst sich das *System*-Modell auch mit Konzepten die in den meisten Systemen zu finden sind, wie zum Beispiel Dateien und Dateisysteme oder Betriebssysteme mit ihren Prozessen.

Interop Mit Hilfe des *Interop*-Modells lassen sich die Management-Komponenten beschreiben, aus denen die WBEM Infrastruktur besteht und wie andere WBEM Komponenten mit der Infrastruktur interagieren. Die Infrastruktur besteht aus den Teilen CIM-Client, CIM Server, CIM Object Manager und Providern die, wie in Abbildung 5 gezeigt, zusammenhängen.

Das *Interop*-Modell wird in die Untermodelle *CIM Object Manager*, *Namespace* und *Provider* sowie *Protocol Adapter*-Modell unterteilt. Das *CIM Object Manager*-Modell beschreibt die WBEM-Infrastruktur und ihre Beziehungen, das *Namespace*-Modell die unterstützen Namensräume und die darin enthaltenen Informationen, das *Provider*-Modell einen Anbieters und seine Fähigkeiten und das *Protocol Adapter*-Modell zur Beschreibung von Protokoll-Adaptern.

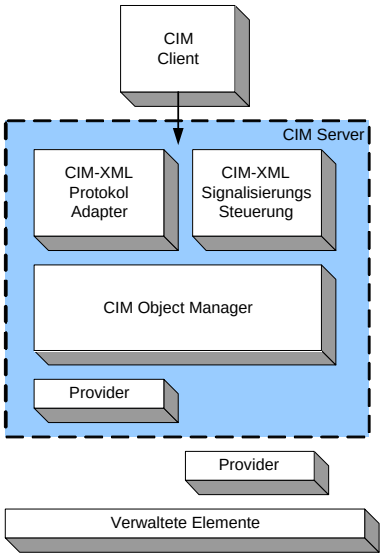


Abbildung 5. Das CIM Interop-Modell

Devices Das *Devices*-Modell beschreibt die Funktionalität, die durch die Hardware zur Verfügung gestellt wird. *Devices* werden als Komponenten eines `CIM_ System`'s dem sie angehören und mit ihren Interaktionen zu anderen Komponenten dieses Systems beschrieben. Das Modell deckt ein sehr weit gefächertes Spektrum an Geräten ab und bietet Zugriff auf Informationen über die aktuelle Konfiguration oder den aktuellen Zustand des Gerätes.

Metrics Das *Metrics*-Modell ermöglicht die dynamische Definition und Gewinnung von metrischen Daten. Es verwendet dazu ein Entwurfsmuster, das dem Muster „Dekorierer“ sehr ähnlich ist, und das auf einer CIM-Klasse zur Definition der Metrik basiert. Darin wird die Semantik sowie die Verwendung der Metrik festlegt. Zusätzlich gibt es noch eine weitere Klasse, die die eigentlichen Datenwerte enthält, die von einer bestimmten Instanz der Definitionsklasse gesammelt wurden.

Policy Das *Policy*-Modell wurde in Zusammenarbeit mit der IETF entwickelt und basiert auf den RFC's 3060 [29] und 3460 [30]. Es stellt ein objektorientiertes Rahmenwerk zur Verwaltung von Systemrichtlinien zur Verfügung. Es wurde abstrakt genug gehalten, um unabhägig von bestimmten Implementierungen zu sein und ist skalierbar genug, um große und komplexe Computersysteme effizient erfassen zu können.

User Der Schwerpunkt dieses Modells liegt zum einen auf der Sammlung von Kontaktinformationen von Personen, Abteilungen und Organisationen, zum anderen auf der Verwaltung von Benutzern, um sich für die Verwendung eines bestimmten Dienstes zu authentifizieren.

Der Begriff „Benutzer“ ist hier nicht nur auf menschliche Wesen beschränkt, sondern kann sich auch auf einen Systemdienst oder eine Gruppe von Benutzern beziehen.

***Extension* Schema** Mit Hilfe der *Extension* Modelle werden werden Technologien, beispielsweise Betriebssysteme wie Microsoft Windows oder Unix, beschrieben.

Hier wird noch weiter zwischen gerätespezifischen Erweiterungen, die von der DMTF kontrolliert werden, und herstellerspezifischen Erweiterungen, die von den Hard- und Softwareherstellern selbst eingebracht werden, unterscheiden.

3 Implementierungen

3.1 Microsoft Windows Management Instrumentation (WMI)

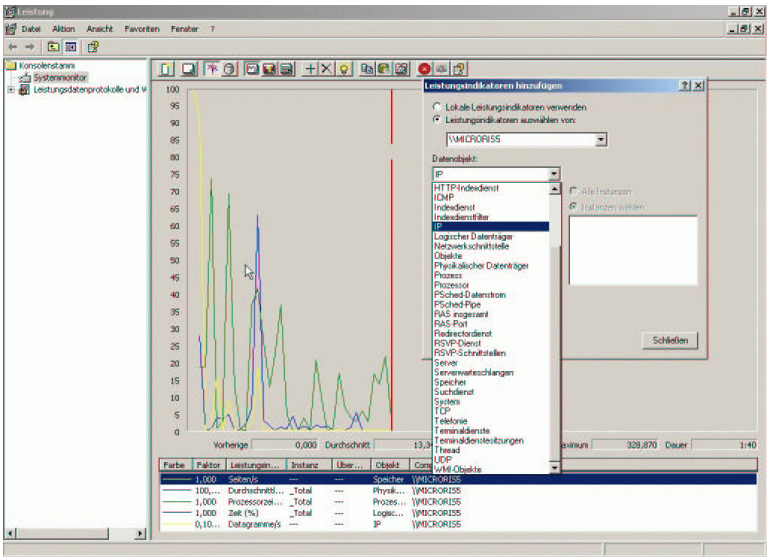


Abbildung 6. Der Windows Performance Monitor

Beschreibung Das Windows Management Instrumentation [9] ist die Implementierung des WBEM-Standards von Microsoft und Teil der Windows NT Reihe. Es ist für Windows 9x als Add-on erhältlich, womit es die gesamte 32-bit Produktpalette von Microsoft abdeckt. Der Zugriff erfolgt entweder über (D)COM oder durch eine der vielen unterstützten Skriptsprachen wie z.B.

– Microsoft Visual Basic [32]

– Microsoft Visual Basic for Applications [33]
– Microsoft VBScript [34]
– Microsoft JScript [35]
– Perl [31]

Die Schnittstellen zum *CIM Object Manager* wurden für die Skriptsprachen an deren Bedürfnisse angepasst, um eine möglichst einfache Anwendung zu ermöglichen und unterscheiden sich deshalb von denen für den Zugriff über COM. Sie verfolgen einen datenflußgesteuerten Ansatz, während für die COM-Variante ein ereignisgesteuerter Ansatz verwendet wird.

Da diese Erweiterungen relativ einfach auszubauen sind, haben bereits Andere Hersteller, darunter Intel, Compaq/Hewlett-Packard und BMC Software, inzwischen eigene Provider angekündigt um eine Anbindung an deren Produkte zu ermöglichen.

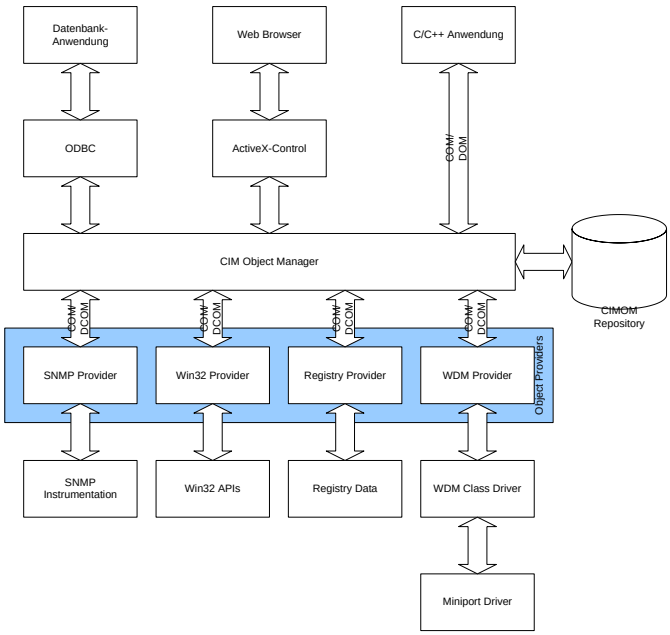


Abbildung 7. Die WBEM Implementierung von Microsoft

Das WMI-SDK und die WMI Scripting Library Von Microsoft ist unter [23] ein kostenloses WMI-Entwicklerkit zum Download erhältlich, das unter anderem die beiden nützlichen Werkzeuge *WMI Object Browser* und *WIM CIM Studio* enthält, mit denen man sich schnell einen Einblick in das objektorientierte WMI-Repository verschaffen kann. Die beiden Programme sind eine Art

Object-Browser und in Form von HTML-Seiten mit ActiveX-Steuerelementen implementiert. Sie bieten einen Überblick über die im Repository gespeicherten Klassen und Objekte, sowie deren Eigenschaften.

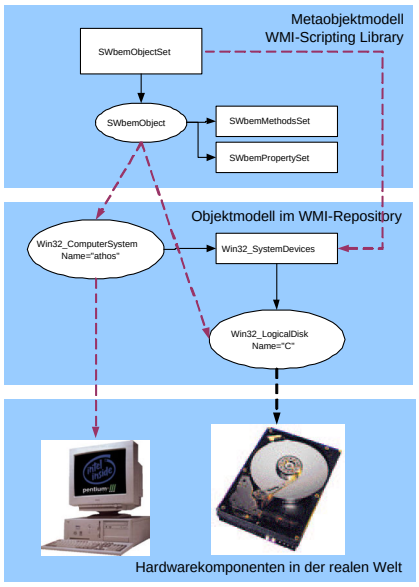


Abbildung 8. Der Objektmodell der WMI Scripting Library

Für die Programmierung von WMI mit Skriptsprachen stellt Microsoft mit der *WMI Scripting Library* [36] ein sogenanntes Meta-Objektmodell zur Verwaltung der Objekte im WMI-Repository zur Verfügung. Da dieses Meta-Objektmodell nur aus einigen wenigen COM-Klassen besteht, wird verhindert, dass sich der Programmierer direkt mit den unzähligen WMI-Klassen befassen muss.

Die zentrale Klasse in diesem Objektmodell (Abbildung 8) ist `SWbemObject`. Ihre Instanzen sind Proxy-Objekte für WMI *Managed Objects*, die durch `SWbemObject` nachgeahmt werden können. Überdies bietet die Klasse Zugriff auf Attribute mit Metainformationen zu den *Manged Objects*, die durch einen angehängten Unterstrich „_“ von den eigentlichen Attributen unterschieden werden.

Von großer Bedeutung sind außerdem die beiden Klassen `SWbemServices` und `SWbemObjectSet`. Ersteres verkörpert einen WMI-Namensraum, letzteres steht für eine Menge von *Managed Objects*.

Die WMI Query Language Die *WMI Query Language* (WQL [24]) ist eine Untermenge des, von der ANSI [28] verabschiedeten, SQL Standards, die sich auf das Anweisungen zur Abfrage von Daten beschränkt. Die *Data Definition Language* (DDL) und die *Data Manipulation Language* (DML) werden nicht

unterstützt. Unterstützt werden die Operatoren =, <, >, <=, >=, != (bzw. <>) sowie `IS NULL`, `IS NOT NULL` und `ISA`. Die zulässigen Schlüsselwörter sind in Tabelle 1 dargestellt.

Beispielsweise erhält man eine Liste der Namen und Festplattenkapazität aller Computer, deren Festplattenkapazität größer 4 GByte ist, mit dem folgenden Statement:

```
SELECT caption, size FROM Win32_DiskDrive WHERE size > 4294967296
```

Eine WQL-Abfrage kann immer nur auf einem bestimmten Namensraum ausgeführt werden.

Beispiel Um auf ein *Managed Object* zugreifen zu können, benötigt man zuerst eine Instanz des zugehörigen Namensraums. Dazu instanziiert man die Klasse `SWbemServices` und ruf deren Methode `GetObject()`-Methode auf, die eine Instanz des zu durchsuchenden Namensraums erhält.

Als Paramter erwartet `GetObject()` einen gültigen WMI-Pfad, der mit der Zeichenkette „winmgmts://“ beginnt und auf den normalerweise ein Computername folgt. Ist das nicht der Fall, so wird angenommen, dass der lokale Computer gemeint ist. Ein gültiger WMI-Pfad für den Namensraum der CIM-Objekte auf dem Rechner *Athos* wäre beispielsweise `winmgmts://mars\root\cimv2`.

Hat man das Objekt vom Typ `SWbemServices` an einen gültigen Namensraum gebunden, kann man mit der Methode `ExecQuery()` eine WQL-Abfrage absetzen und erhält als Ergebnismenge ein Objekt vom Typ `SWbemObjectSet`.

Mit den VBScript-Programmausschnitt in Abbildung 9 erhält man beispielsweise eine Liste aller Festplatten und deren Größe eines Rechners.

```
Dim query = "SELECT caption,size FROM Win32_DiskDrive"
Dim computer = "athos";
Set objServ = GetObject("winmgmts://" & computer & "\root\cimv2")
Set objOSet = objServ.ExecQuery(query)
```

Abbildung 9. Liefert die Liste aller Festplatten eines Rechners

Tabelle 1. Die Schlüsselwörter der *WMI Query Language*

Schlüsselwort	*Beschreibung*
`AND`	Verbindet zwei boolsche Ausdrücke, und liefert WAHR, falls beide Ausdrücke WAHR sind.
`ASSOCIATORS OF`	Liefert alle Instanzen, die mit einer Quellinstanz verbunden sind. Diese Anweisung wird für Schema-Abfragen und Daten-Abfragen verwendet.
`__CLASS`	Referenziert die Klasse des Objekts in einer Abfrage. Dieses Schlüsselwort ist in Windows 2000 und neuer verfügbar.
`FROM`	Beschreibt die Klasse, die die in der `SELECT`-Anweisung gewünschten Eigenschaften enthält. Das Windows Management Instrumentation (WMI) unterstützt nur Daten-Abfragen von einer Quelle gleichzeitig.
`GROUP`	Bewirkt, dass das WMI nur einen Repräsentanten für eine Gruppe von Ereignssen lierfert. Diese Klausel wird für Ereignis-Abfragen verwendet.
`HAVING`	Filtert die Ereignisse, bei einem Gruppierungsintervall, die innerhalb einer `WITHIN` Klausel empfangen wurden.
`IS`	Vergleichsoperator, der mit `NOT` und `NULL` verwendet wird. Die Syntax für diesen Ausdruck ist die folgende: `IS [NOT] NULL` (wobei `NOT` optional ist)
`ISA`	Operator, der eine Abfrage auf die Unterklassen einer bestimmten Klasse anwendet.
`KEYSONLY`	Nur ab Windows Server 2003 and Windows XP: Wird in Abfragen mit `REFERENCES OF` und `ASSOCIATORS OF` verwendet, um sicherzustellen, dass die sich ergebenden Instanzen nur mit den Schlüsseln der Instanzen belegt werden, was den Überhang des Aufrufs reduziert.
`LIKE`	Operator, der festlegt, ob eine gegebene Zeichenkette in ein bestimmtes Muster passt oder nicht.
`NOT`	Vergleichsoperator, der in einem `SELECT`-Ausdruck verwendet werden kann.
`NULL`	Beschreibt ein Objekt, dass keinen bestimmten Wert hat. und ist nicht zu verwechseln mit der Zahl 0 oder einem leeren Objekt.
`OR`	Verknüpft zwei Bedingungen. Wenn im Ausdruck mehr als ein logischer Operator verwendet wird, werden die `OR` Operatoren nach den `AND` Operatoren ausgewertet
`REFERENCES OF`	Liefert alle Instanzen einer Assoziation, die sich auf eine bestimmte Quellinstanz beziehen. Dieser Ausdruck wird in Schema- und Datenabfragen verwendet. Der `REFERENCES OF` Ausdruck ähnelt dem `ASSOCIATORS OF` Asudruck. Allerdings liefert es nicht die Instanzen der Endpunkte, sondern nur die der Assoziation.
`SELECT`	Legt die Eigenschafen fest, die in einer Abfrage verwendet werden.
`TRUE`	Boolscher Operator für WAHR der zu -1 ausgewertet wird.
`WHERE`	Verkleinert den Gültigkeitsbereicht einer Daten-, Ereignis- oder Schemaabfrage.
`WITHIN`	Legt ein ein Rundfrage- oder Gruppierungsintervall fest. Diese Klausel wird in Ereignisabfragen verwendet.
`FALSE`	Boolscher Ausdruck, der den Wert 0 hat.

3.2 Weitere verfügbare Implementierungen

Solaris WBEMServices Von Sun Microsystems gibt es gleich zwei Implementierungen des WBEM Standards. Die *Solaris WBEM Services*[10] sind eine kommerzielle Implementierung für das Sun eigene Unix-Derivat Solaris.

Sun WBEMServices Die *Sun WBEM Services* ist eine Implementierung für Java und als Open-Source Projekt unter [11] frei erhältlich.

In diesem Zusammenhang sollte noch das Projekt *Standards Based Linux Instrumentation for Manageability* [37] erwähnt werden. Dieses Open-Source Projekt definiert eine Schnittstelle, mit deren Hilfe native Provider in der Programmiersprache C entwickelt werden können, die dann in der Lage sind, mit jedem Open-Source CIMOM zusammenzuarbeiten.

The Open Group's Pegasus Pegasus [41] ist eine Open-Source Implementierung der Standards CIM und WBEM. Es wurde im Hinblick daraufhin entworfen, portierbar und sehr modular zu sein. Als Programmiersprache wurde C++ verwendet, um die objektorientierten Konzepte von CIM gewissermaßen direkt in ein Programmiermodel übersetzen zu können, ohne dabei die Ausführgeschwindigkeit und die Effizienz einer übersetzten Sprache zu verlieren.

Durch seinen portablen Entwurf kann es unter den meisten Versionen von Unix, Linux und Microsoft Windows übersetzt und betrieben werden.

SNIA Open Source CIMOM Ursprünglich wurde das *SNIA Open-Source Java CIMOM* [38] Projekt von der *Storage Networking Industry Association* ins Leben gerufen. Inzwischen wurde es an die Open Group übergeben und wird dort in deren Pegasus Projekt integriert werden, um so eine umfassende Open-Source WBEM Umgebung zu schaffen.

OpenWBEM OpenWBEM [39] ist ein Software-Paket mit einer marktreifen Implementierung der CIM und WBEM Standards der DMTF. Es beinhaltet einen *CIM Object Manager*, eine Schnittstelle für CIM-Clients, eine CIM Listener API sowie eine Unterstützung der WBEM Abfragesprache WQL.

B4wbem B4wbem [40] hat das Ziel, ein Administrationssystem für Linux zu sein. Es basiert auf der Bibliothek *libCIM*, welche als Basis für eine Implementierung des *Common Information Model* dient. Darüber hinaus sind ein *CIM Object Manager* und eine Vorlage für einen *CIM Client* enthalten.

4 Zusammenfassung und Ausblick

Mit *Windows Management Integration* für Windows, *Solaris WBEMServices* für Solaris ist ein großer Teil der sich am Markt befindlichen Systeme abgedeckt.

Die inzwischen so große Zahl an Implementierungen und Projekten rund um WBEM und CIM, zeigt deutlich, dass der Standard am Markt an Akzeptanz gewinnt. Eine herstellerunabhängige Managementlösung stellt eine wichtige Grundlage für die Entwicklung von allgemein verwendbaren Werkzeugen dar. Auf dieser Grundlage können dann Anwendungen entwickelt werden, die es zuküftigen Rechnersystemen ermöglichen, sich selbst zu überwachen und zu optimieren.

Literatur

1. Desktop Management Task Force: http://www.dmtf.org
2. BMC Software: http://www.bmc.com
3. Cisco Systems: http://www.cisco.com
4. Compaq/HP: http://www.compaq.com
5. Intel: http://www.intel.com
6. Microsoft: http://www.microsoft.com
7. Computer Associates: http://www.cai.com/
8. IBM/Tivoli Software: http://www.tivoli.com
9. Microsoft Windows Management Instrumentation: http://www.microsoft.com/management
10. Solaris WBEMServices: http://wwws.sun.com/software/solaris/wbem/
11. Sun WBEMServices: http://wbemservices.sourceforge.net/
12. Jörgens, U., Kuschke, M.: Alles neu macht das Web (iX 7/2000) S.116
13. IBM Research: http://www.ibm.com/research/autonomic/
14. IEEE Spectrum R&D Report 15.09.2002: Helping Computers Help Themselves http://www.spectrum.ieee.org/WEBONLY/publicfeature/sep02/auto.html
15. The CIM Tutorial http://www.wbemsolutions.com/tutorials/CIM/cim.html
16. Windows Management Instrumentation and the Common Information Model (MSDN, November 1998) http://msdn.microsoft.com/library/en-us/dnwmi/html/wmicim.asp
17. Windows Management Instrumentation: Background and Overview (MSDN, November 2000) http://msdn.microsoft.com/library/en-us/dnwmi/html/wmioverview.asp
18. Carey/O´Reilly: Integrating CIM/WBEM with the Java Enterprise Model: http://www.globalmanagementconference.com/alliance_contest/carey_oreilly.pdf
19. RFC 1757: Remote Network Monitoring Management Information Base http://www.ietf.org/rfc/rfc1757.txt
20. RFC 1098: A Simple Network Management Protocol (SNMP) http://www.ietf.org/rfc/rfc1098.txt
21. RC 1189: The Common Management Information Services and Protocols for the Internet (CMOT and CMIP) http://www.ietf.org/rfc/rfc1189.txt
22. IEEE Spectrum R&D Report 15.09.2002: Helping Computers Help Themselves / IT Services Keep Climbing http://www.spectrum.ieee.org/WEBONLY/publicfeature/sep02/autof1.html
23. Microsoft Windows Management Instrumentation SDK: http://download.microsoft.com/download/platformsdk/x86wmi/1.1/W9XNT4/EN-US/wmisdk.EXE
24. WMI Query Language (WQL): http://msdn.microsoft.com/library/en-us/wmisdk/wmi/sql_for_wmi.asp

25. RC 2616: Hypertext Transfer Protocol – HTTP/1.1
http://www.ietf.org/rfc/rfc2616.txt
26. RFC 1697: Relational Database Management System (RDBMS) Management
Information Base (MIB) using SMIv2 http://www.ietf.org/rfc/rfc1697.txt
27. The Internet Engineering Task Force http://www.ietf.org/
28. American National Standards Institute (ANSI): http://www.ansi.org
29. RFC 3060: Policy Core Information Model http://www.ietf.org/rfc/rfc3060.txt
30. RFC 3460: Policy Core Information Model (PCIM) Extensions
http://www.ietf.org/rfc/rfc3460.txt
31. perl - Practical Extraction and Report Language: http://www.perl.org
32. Microsoft Visual Basic: http://msdn.microsoft.com/vbasic/
33. Microsoft Visual Basic for Applications: http://msdn.microsoft.com/vba/
34. Microsoft VBScript:
http://msdn.microsoft.com/library/en-us/script56/html/vtoriVBScript.asp
35. Microsoft JScript:
http://msdn.microsoft.com/library/en-us/script56/html/js56jsoriJScript.asp
36. WMI Scripting Library:
http://www.microsoft.com/technet/scriptcenter/scrguide/sas_wmi_miat.asp
37. Standards Based Linux Instrumentation for Manageability (SBLIM):
http://www-124.ibm.com/sblim/
38. SNIA Open-Source Java CIMOM: http://www.opengroup.org/snia-cimom/
39. OpenWBEM: http://openwbem.sourceforge.net/
40. B4wbem: http://b4wbem.sourceforge.net/
41. The Open Group's Pegasus: http://www.opengroup.org/pegasus/

Selbst-Schutz

Andreas Stöckicht

Zusammenfassung Dieser Artikel beschäftigt sich mit den Sicherheitsanforderungen, die an ein autonomes Computersystem gestellt werden müssen und den Techniken um diese Anforderungen zu erfüllen. Zur Abwehr von Viren geht die Forschung in Richtung eines digitalen Immunsystems angelehnt an das biologische System von Wirbeltieren zur Abwehr von Krankheiten. Bei der Erkennung von Eindringlingen nutzt man neuronale Netze um abweichendes Verhalten erkennen und bekämpfen zu können. Desweiteren geht es um die sichere Architektur autonomer Computersysteme und die effektive, automatische Nutzung von Sicherheitsrichtlinien auf verschiedenen Ebenen.

1 Einleitung

Heutige Computersysteme werden immer schneller, leistungsfähiger und komplexer, stellen zugleich aber auch entsprechend höhere Anforderungen an die Administratoren. Ob in einer großen Firma mit einer entsprechenden EDV-Anlage oder beim Privatnutzer zu Hause, die Konfiguration und Fehlersuche nimmt immer mehr Zeit in Anspruch und führt oft zu Produktionsausfällen, Datenverlust und Frustration. Deshalb befasst man sich in der Forschung mit sich selbst überwachenden Systemen, so genannten *autonomen Computersystemen* [1] [2]. Sie sollen einen Großteil der administrativen Tätigkeiten übernehmen und geben den IT-Fachkräften somit wieder mehr Zeit für ihre eigentlichen Aufgaben.
Heutige Sicherheitstechniken würden ohne den Menschen nicht funktionieren, weil sie ausgeführt, überwacht, trainiert und durch Updates auf dem neuesten Stand gehalten werden müssen. Ein autonomes System benötigt also eine Art Selbst-Schutz mit dem es in der Lage ist sich vor Viren und Angriffen zu schützen.
Als Vorbild für autonome Computersysteme galten von Anfang an Organismen aus der Natur, die sich selbst steuern. Bestes Beispiel dafür ist der Mensch mit seinem Nervensystem, welches alle lebenswichtigen Funktionen automatisch reguliert. Daher ist es nur konsequent bei der Suche nach einem Selbst-Schutz für ein autonomes Computersystem den Schutz des menschlichen Körpers, das *Immunsystem* und eventuelle Analogien näher zu betrachten um Techniken die in Millionen Jahren Evolution entstanden sind zu nutzen.
In den folgenden Abschnitten wird auf diese Analogie zwischen Mensch und Computer sowie den daraus folgenden Konsequenzen für den Schutz moderner Computersysteme eingegangen. Im letzten Kapitel geht es dann um die generelle Architektur autonomer Systeme und den daraus folgenden Möglichkeiten des Selbst-Schutzes.

2 Digitales Immunsystem

2.1 Entstehung

Die Idee das Immunsystem von Wirbeltieren als Vorbild zum Schutz von Computersystemen zu verwenden kam nicht erst durch den Trend zu autonomen Systemen auf. Schon 1994 bzw. 1995 machten sich Kephart und White in [3] Gedanken über diese Analogie. Das Problem war und ist es heute immer noch, auf neue Computerviren möglichst schnell reagieren zu können. Zur Bekämpfung bekannter Viren bestehen geeignete Methoden [4] [5] wie Scanner, Integritätstester und Aktivitätsmonitore, die vom Benutzer ausgeführt und überwacht werden. Aktivitätsmonitore melden unübliche Verhaltensweisen von Programmen - etwa den Versuch, bestimmte Interruptadressen zu verändern, Integritätstester stöbern auffällige Änderungen an Dateien auf und Scanner durchsuchen den Haupt und Sekundärspeicher eines Computers nach Viren. Zur Abwehr von Eindringlingen benutzt man unter anderem neuronale Netze, die von Administratoren trainiert werden um normales Verhalten von Angriffen zu unterscheiden. Diese Techniken versagen jedoch bei bisher unbekannten Viren.
Taucht nun ein neuer Virus auf müssen möglichst schnell folgende Funktionen in Form eines Updates für Anti-Viren Software bereitgestellt werden:

1. Erkennung des Virus in einem infizierten Programm
2. Wiederherstellung des Programms in seinen Originalzustand

Üblicherweise besorgen hunderte Experten diese Information indem sie den Virus zerlegen und analysieren. Dadurch erhalten sie Informationen über sein Verhalten und wie er sich mit Programmem verknüpft. Danach wird eine für den Virus charakteristische Signatur festgelegt (16-32 Bytes), die in all seinen Instanzen existiert, nicht aber in normaler Software. Diese Signatur wird dem Scanner durch ein Update zur Verfügung gestellt.
Die Schwachstelle bei diesem Verfahren ist eindeutig der Mensch [6]. Durch die weltweite Vernetzung können sich Viren und Würmer immer schneller ausbreiten und innerhalb weniger Tage Epidemien auslösen. Desweiteren ist der Trend neue Computerviren zu entwickeln ungebrochen und wird durch Automation sogar noch vereinfacht. Selbst Experten können es nicht mit dieser Entwicklung aufnehmen und schnell genug Gegenmittel bereitstellen, die auch noch garantiert keine Autoimmunreaktion auslösen, also reguläre Programme für einen Virus halten. Da der biologische- und der Computervirus nicht nur den Namen gemeinsam haben sondern auch die Art, wie sie durch ihren Wirt überleben und sich vervielfältigen lassen, forschte man an einem digitalen Immunsystem. Diese Ideen werden heute bei autonomen Computersystemen wieder aktuell, da sie dasselbe Ziel verfolgen: Der Computer soll sich selbstständig vor bekannten und unbekannten Viren schützen können.

2.2 Anforderungen an ein Immunsystem

Wenn der Mensch bei jedem Schnupfen, der sich zur Winterzeit ausbreitet auf ein Gegenmittel vom Gesundheitsministerium angewiesen wäre um wieder gesund zu

werden, dann hätte unsere Spezies nicht lange überlebt. Genau so schützen wir aber zurzeit unsere Computer vor Viren. Wie im vorigen Abschnitt beschrieben warten wir auf ein Update für unsere Anti-Viren Software, aktualisieren damit unser System und bekämpfen den Virus. Menschen brauchen sich aber nicht auf eine zentrale Einrichtung verlassen, denn sie tragen ihren Schutz gegen Krankheiten immer bei sich. Das Immunsystem von Wirbeltieren besitzt einige bemerkenswerte Eigenschaften [7]:

1. Wiedererkennen bekannter Viren
2. Eliminierung/Neutralisation von Viren
3. Erkennen von bisher unbekannten Viren
4. Selektive Nutzung von Nachbildung und Vermehrung zur schnellen Erkennung und Bekämpfung von Eindringlingen

Diese Eigenschaften sind für das Sicherheitssystem eines Computers ebenfalls wünschenswert. In den folgenden Abschnitten wird erläutert, wie diese Funktionen nach [6] im Modell des digitalen Immunsystems realisiert werden und wie die entsprechenden Funktionen in der Natur gelöst sind.

2.3 Wiedererkennung bekannter Viren

Das Immunsystem von Wirbeltieren bzw. die speziell dafür vorgesehenen Antikörper erkennen Antigene (Fremdkörper) anhand einer kurzen Folge von Aminosäuren an deren Oberfläche. Exakte Übereinstimmung mit dem gesamten Muster ist also nicht erforderlich.
Der Computer erkennt Viren wie das menschliche Immunsystem an einer kurzen charakteristischen Sequenz (Kapitel 2.1). Die Vorteile dieser Methode sind zum einen der geringe Speicherplatzbedarf und zum anderen können so auch Varianten eines Virus entdeckt werden. Für das biologische wie das digitale Immunsystem ist diese Eigenschaft lebenswichtig, da Viren sehr oft mutieren.

2.4 Eliminierung von Viren

Wenn im menschlichen Organismus ein Antikörper auf ein Antigen trifft, verbinden sich die beiden und das Antigen ist neutralisiert. Ist aber schon eine Zelle durch das Antigen infiziert und produziert Kopien des Virus, dann wird diese Zelle durch eine T-Zelle zerstört. Für den Körper ist dies eine vernünftige Vorgehensweise, da Zellen leicht zu ersetzen sind. Der Computer hingegen würde sich mit dieser Strategie selbst zerstören, weil viele Daten und Funktionen einzigartig und unersetzbar sind. Darin liegt jedoch auch der Vorteil. Wenn ein Virus ein bestimmtes Programm zerstört wird er sofort bemerkt und entfernt. Deshalb zerstören Viren Programme im allgemeinen nicht, sondern ändern nur die Struktur zu ihren Gunsten. Durch passende Algorithmen können diese Dateien dann wieder in ihren Originalzustand versetzt werden.

2.5 Erkennung von unbekannten Viren

Wird ein Fremdkörper im menschlichen Organismus entdeckt, so weiß das Immunsystem sofort, dass es diesen Eindringling bekämpfen muss. Es hat nämlich eine Vorstellung davon aus welchen Zellen es seit der Geburt besteht und handelt nach dem Grundsatz: Alles was nicht zum Körper gehört ist fremd und muss bekämpft werden. Es kann lernen, wie es bisher unbekannte Eindringlinge bekämpfen muss und sich Jahre später noch durch aufbewahrte Rezeptoren daran erinnern. Diese Rezeptoren werden bevor sie zum Einsatz kommen durch eine Art Trainingslager geschleust um Autoimmunreaktionen zu verhindern.
Die Frage, was zu einem Computersystem gehört und was nicht, ist schwerer zu beantworten als beim Menschen, da ständig neue Software installiert wird, die vom Computer natürlich nicht als feindlich angesehen werden darf. Mehr über das Problem dem Computer eine gewisse Form von Bewußtsein für sich und seine Komponenten zu geben kann man in der Ausarbeitung *Selbst-Bewusstsein autonomer Computersysteme* nachlesen.
Um unbekannte Viren aufzuspüren benutzt das digitale Immunsystem verschiedene Überwachungsprogramme, wie z.B. die Integritätskontrolle. Dabei werden laufend Prüfsummen von Dateien gebildet und diese mit früheren Ergebnissen verglichen. Die Aktivitätskontrolle überwacht zur Zeit laufende Programme um unübliche Tätigkeiten zu bemerken. Schlagen mehrere dieser Programme Alarm wird der Virenscanner aktiviert, der bei Erfolg nach gewohnter Art den bekannten Virus bekämpft. Findet der Scanner jedoch nichts, gab es entweder falschen Alarm oder es ist ein bisher unbekannter Virus im System. Nun werden in einem geschützten Bereich so genannte Köder mit allerhand Zugriffsrechten, Schreiberlaubnis und Interaktion mit dem Betriebssystem ausgelegt um den Virus zu locken. Programme dieser Art sind bei Viren sehr beliebt und werden nach jeder Ausführung nach Veränderungen durchsucht. Werden solche Unregelmäßigkeiten an einem Köder festgestellt, befindet sich ein Virus im System und eine Probe davon zur weiteren Analyse direkt im geschützten Bereich. Nun wird analog zum biologischen Immunsystem automatisch eine in allen Variationen des Virus invariante Signatur extrahiert, auf Autoimmunreaktionen getestet und dem Scanner zur Verfügung gestellt. Der automatischen Bestimmung der Signatur liegen komplexe Algorithmen zugrunde, die von den Firmen geheimgehalten werden. Das digitale Immunsystem muß zusätzlich Informationen bereitstellen, wie der Virus sich mit Dateien verknüpft, da im Gegensatz zum menschlichen Organismus befallene Einheiten repariert werden müssen.

2.6 Reproduzierung und selektive Vermehrung

Im biologischen Immunsystem reproduzieren sich gezielt Immunzellen, deren Rezeptoren zu gerade eingedrungenen Antigenen passen. So werden selektiv die aktuell benötigten Antikörper vermehrt um eine Ausbreitung der Krankheit zu verhindern.
Die Ausbreitung von Viren in Computernetzwerken wird auf ähnliche Weise eingedämmt. Ist ein Computer infiziert schickt er an alle Nachbarn ein sogenanntes

kill-Signal (siehe Abbildung 1) welches eine Signatur und Wiederherstellungsinformationen für infizierte Dateien beinhaltet. Wenn der Empfänger dieses Signals ebenfalls befallen ist, schickt er es weiter an seinen Nachbarn. Ist er aber frei von Viren, sendet er das Signal nicht weiter sondern aktualisiert nur seine Anti-Viren Datenbank. Theoretische Studien haben erwiesen, dass diese Form der Epidemiebekämpfung von Computerviren sehr effektiv ist [8] [9].

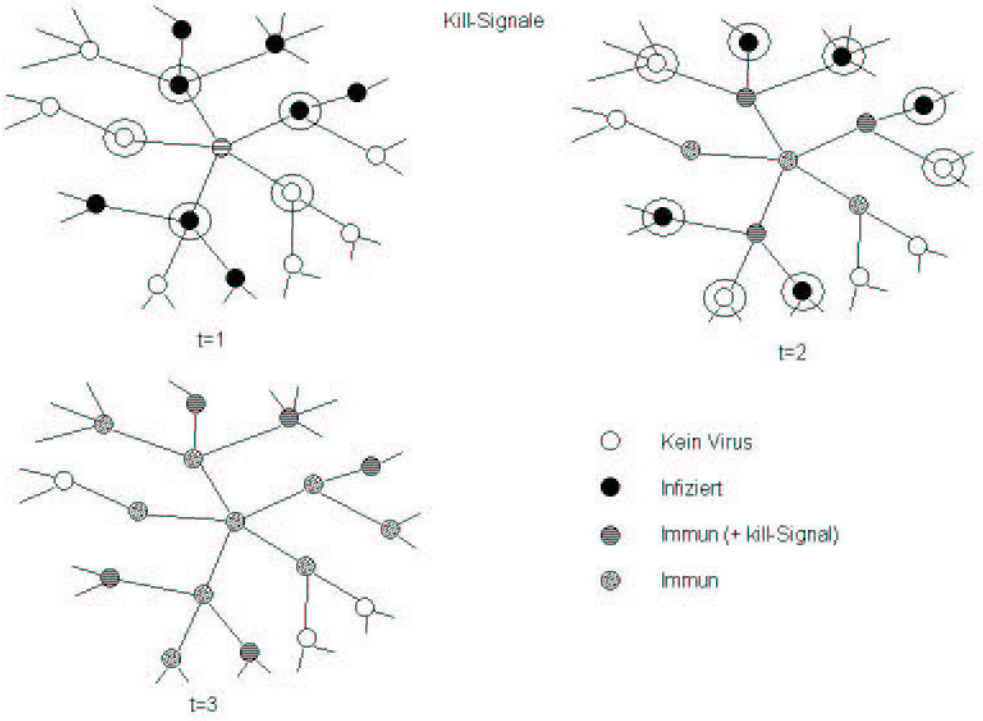


Abbildung 1. Kill-Signale

3 Prototyp eines digitalen Immunsystems

In Kapitel 2 wurde ein theoretisches digitales Immunsystem entwickelt, bei dem viele wichtige Funktionen aus der Natur übernommen und nachgeahmt werden. Mit Hilfe dieses Vorwissens und aufbauend auf [10] wurde am IBM Research Center ein *kommerziell taugliches Immunsystem* [11] entwickelt mit dem Anspruch bisher unbekannte Viren schneller erkennen, analysieren und heilen zu können, als sie sich ausbreiten. Desweiteren soll das System seine Aufgaben autonom ausführen können. In den folgenden Abschnitten wird der Aufbau und die

Funktionsweise dieses Immunsystems erläutert und auf verschiedene Probleme eingegangen, die beim Entwurf berücksichtigt werden mussten.

3.1 Aufbau und Funktionsweise

Das Immunsystem besteht aus einem zentralen Analysesystem, einem aktiven Netzwerk und den derzeit angeschlossenen Konzernen (siehe Abbildung 2). Die Funktionsweise der einzelnen Komponenten wird anhand eines Beispiels erläutert, bei dem ein Client auf seinem Computer einen potentiellen Virus entdeckt, ihn zum Administrator weiterleitet, und der von dort über das aktive Netzwerk zum Analysezentrum gelangt. Danach wird ein Update verbreitet, welches diesen Virus erkennt und infizierte Dateien wiederherstellt. Alle Schritte können automatisiert werden, so dass das System unabhängig vom Menschen arbeiten kann.

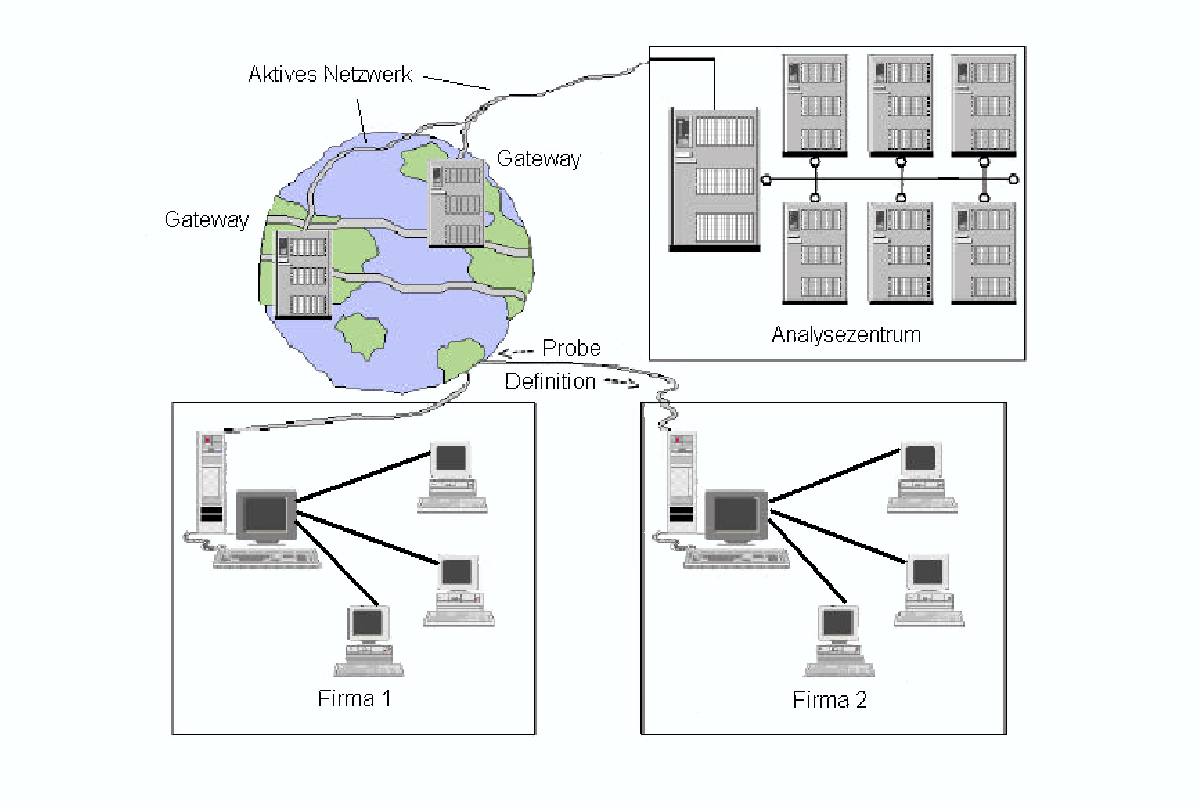


Abbildung 2. Aufbau des Immunsystems

Potentieller Virus Wird ein unbekannter Virus von einem Client in einer Firma entdeckt (siehe Abschnitt 2.5 und [12] [13]) so wird eine Kopie in einer harmlosen Form an das Administratorsystem gesendet.

Administratorsystem Dieses System entscheidet, welche Daten das interne Netzwerk der Firma verlassen und an das Analysezentrum gesendet werden. Wenn es aktuellere Virendefinintionen als der Client besitzt kann das System den Virus eventuell erkennen und somit das Problem beheben. Findet es keinen bekannten Virus wird die potentiell infizierte Datei von vertraulichen Daten bereinigt und an die nächst höhere Stufe des Immunsystems zur Analyse gesendet. Werden in einer Firma tausende Proben in kurzer Zeit an das Administratorsystem gesendet, so besteht die Möglichkeit, dass alle vom selben Virus befallen sind. Nun werden ein paar repräsentative Muster an das Analysezentrum gesendet um das aktive Netzwerk nicht zu überlasten. Erhält es die Signatur und Wiederherstellungsinformationen zurück, werden die wartenden Exemplare damit verglichen und wenn möglich repariert.

Aktives Netzwerk Proben werden vom aktiven Netzwerk über das Internet zum Analysezentrum unter Verwendung von Standard Internet Transport- und Sicherheitsprotokollen geleitet. Seine Konstruktion erlaubt den Umgang mit Epidemien, wobei sehr viele mit dem selben Virus infizierte Dateien das Netzwerk überfluten. In so einem Fall verbleibt ein Großteil der Proben im aktiven Netzwerk und das Analysezentrum kann sich auf wenige Exemplare konzentrieren und wird nicht von Datenmengen arbeitsunfähig gemacht.
In unserem Beispiel handelt es sich um einen neuen Virus, der nicht innerhalb des aktiven Netzwerks behandelt werden kann und somit zum Analysezentrum gelangt.

Analysezentrum Automatische Analyse von Viren ist eine der Schlüsseltechniken des Immunsystems. Aus unserer Probe wird eine Sequenz extrahiert, anhand derer der Virus von Scannern zu erkennen ist. Desweiteren wird herausgefunden, wie der Virus sich mit seinem Wirt verbindet und wie man befallene Daten in ihren Ausgangszustand zurückversetzen kann. Dieser Bereich ist Bestandteil vieler aktueller Forschungsprojekte worüber die Unternehmen ungern Informationen freigeben.

Update verbreiten Hat das Analysezentrum erfolgreich ein Update erstellt, so wird dieses über denselben Weg zurück an das Administratorsystem übermittelt. Dieses untersucht eventuelle Proben in Warteschlangen und schickt bei Erfolg das Update zu den jeweiligen Clients, damit diese ihr System von dem Virus säubern können.

3.2 Aktives Netzwerk zur Bewältigung von Epidemien

Das aktive Netzwerk (siehe Abbildung 3) darf, wie in Abschnitt 3.1 angedeuted, selbst in Ausnahmesituationen nicht zusammenbrechen und muß das Analysezentrum vor einer Datenüberflutung bewahren. Um diese Aufgabe zu bewältigen ist es aus Knoten, so genannten *Gateways* zusammengesetzt, wobei das ganze System einen Baum darstellt. Die Blätter des Baumes sind verschiedene Administratorsysteme und die Wurzel stellt das Analysezentrum dar. Diese Struktur stellt sicher, dass für die einzelnen Administratorsysteme immer genügend Kapazität vorhanden ist, selbst in Ausnahmefällen.

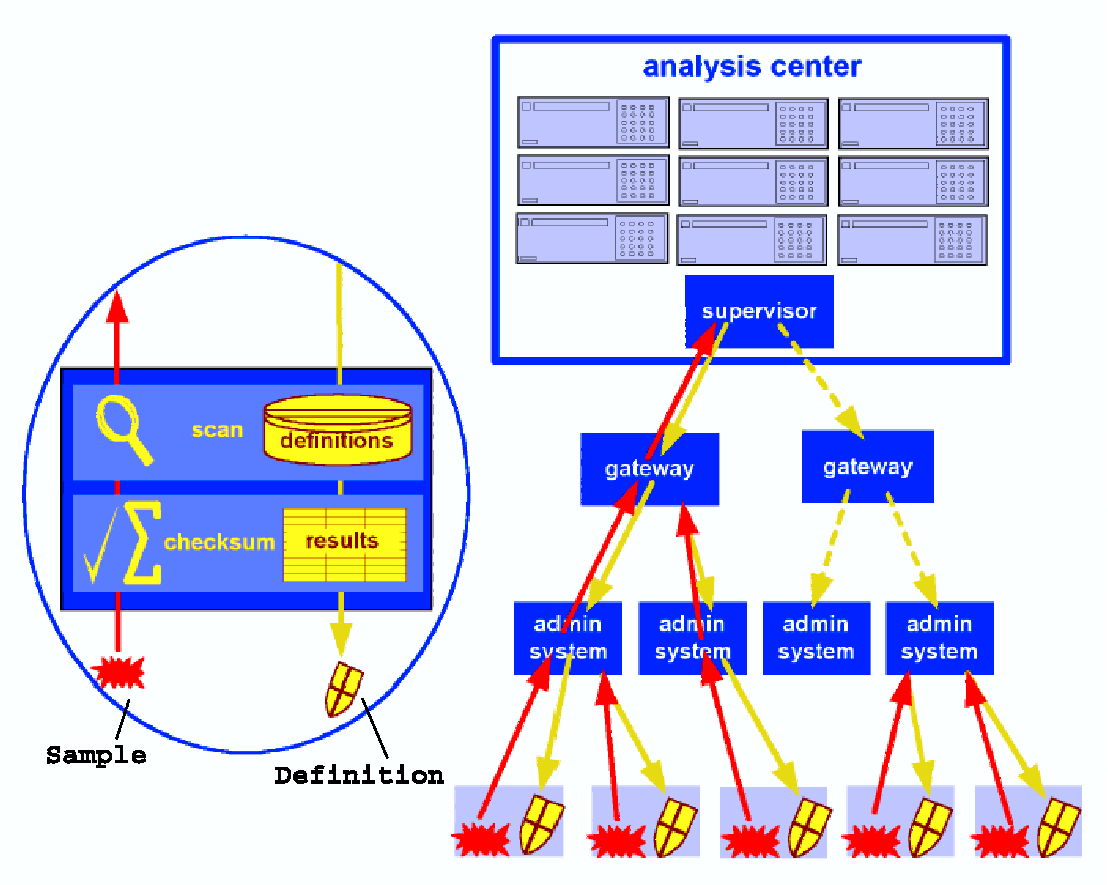


Abbildung 3. Aktives Netzwerk

Jedes Gateway hat zwei primäre Funktionen. Erstens, es versucht das Problem selber zu lösen, indem die Prüfsumme des ankommenden Virus gebildet und diese mit seiner Datenbank schon analysierter Proben verglichen wird. So

können unter anderem schon früher analysierte, virenfreie Proben herausgefiltert werden. Bei Erfolg werden die nötigen Informationen zurückgesendet. Wenn dieselbe Datei schon in höheren Ebenen des aktiven Netzwerks existiert, so nimmt das Gateway die Probe nicht an. Stattdessen wartet es auf ein Ergebnis von oben und schickt dieses an alle Gateways oder Administratorsysteme der Ebene darunter, die ein Exemplar gesendet haben.
Als zweite Funktion überprüft das Gateway die Probe mit den neuesten Anti-Virus Definitionen. Es kann sein, dass Updates, die gerade erst vom Analysezentrum bereitgestellt wurden, noch nicht bei den Administratorsystemen eingetroffen sind. Führt dies auch nicht zum Erfolg, so wird die Probe an die nächst höhere Ebene also ein Gateway oder das Analysezentrum gesendet.
Wird nun vom Analysezentrum ein Update bereitgestellt, aktualisiert das Gateway seine Datenbank und überprüft die Warteschlange. Paßt die neue Definition zu einzelnen Proben, so werden sie gelöscht und das Update für diese Dateien den Baum hinunter zu den Administratorsystemen gesendet, die die Proben in Auftrag gegeben haben.

3.3 Analysezentrum

Das Analysezentrum (siehe Abbildung 4) besteht aus einem Supervisor- und mehreren Rechensystemen. Das Supervisorsystem kontrolliert den Ablauf der Analyse einer Probe, richtet wo nötig Warteschlangen ein und verteilt Aufgaben an die Rechensysteme, bis es schließlich ein fertiges Update an das aktive Netzwerk übergeben kann. Falls bei der automatischen Analyse Probleme auftreten entscheidet es, ob weitere Maßnahmen ergriffen werden oder der Mensch eingeschaltet werden soll.

Im ersten Schritt der Analyse wird versucht den Virus zu klassifizieren, damit spezielle Programme die weitere Analyse übernehmen können. Je nach Typ des Virus können sich die Proben nun in speziellen Umgebungen vermehren. Dies dient zum einen der Gewissheit, dass es sich um einen Virus handelt, und zum anderen hat man danach genug Proben zur weiteren Analyse. Ein weiterer Vorteil der kontrollierten Replikation ist, dass durch die Beobachtung der Ausbreitung des Virus erste Erkenntnisse über ein Gegenmittel gesammelt werden können.
Ziel der nun folgenden Analyse ist es, eine möglichst gute Signatur zu extrahieren sowie Wiederherstellungsinformationen bereitzustellen. Hat sich ein Virus in der Replikationsphase unterschiedlich vermehrt, so werden alle ungleichen Proben einzeln analysiert und am Schluß eine allen gemeinsame, invariante Signatur [14] [15] bestimmt.
Anhand der Ergebnisse der Analyse wird nun eine neue Virendefinition generiert und im Archiv gespeichert. Bevor diese jedoch als Update für den eingereichten Virus an das aktive Netzwerk übergeben wird, muß sie einen Test durchlaufen, indem alle infizierten Dateien des Analysezentrum ohne Ausnahme erkannt und repariert werden müssen. Danach wird das Update verbreitet.

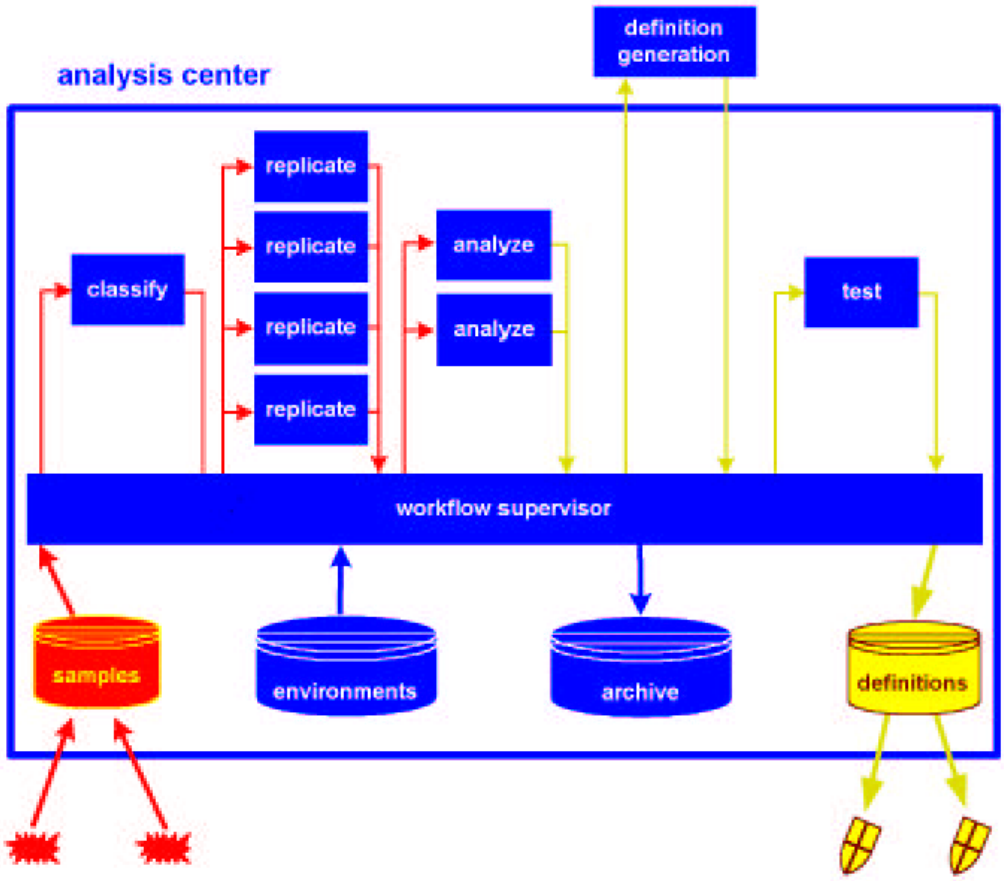


Abbildung 4. Analysezentrum

4 Entdeckung von Eindringlingen

Bisher ging es um die Frage, welche Techniken benötigt werden damit autonom arbeitende Computersysteme sich selbst vor Viren schützen können. Ein weiterer Sicherheitsaspekt ist die Entdeckung von Eindringlingen, sogenannten *Hackern*, die sich unbefugt im System aufhalten und immensen Schaden anrichten können.
Gegenwärtig gibt es zwei gängige Methoden unberechtigte Zugriffe zu entdecken. Zum einen werden Verhaltensmuster von allen Benutzern oder mehreren Gruppen in Datenbanken gespeichert, um abweichendes Verhalten erkennen und Alarm schlagen zu können. Zum anderen wird jede Aktivität der Benutzer mit typischen Verhaltensweisen, üblicherweise in Form eines regelbasierten Systems gespeichert, von Eindringlingen verglichen.

Diese Ansätze benötigen jedoch regelmäßige Updates der Administratoren um zuverlässig funktionieren zu können. Sobald ein neuer Mitarbeiter das Computersystem nutzt oder neue Projekte begonnen werden, müssen die Verhaltensmuster manuell angepasst werden. Wenn Eindringlinge neue Wege gefunden haben das System zu infiltrieren, so muss dieses Verhaltensmuster manuell in Form einer Regel kodiert und den typischen Verhaltensweisen von Hackern hinzugefügt werden. Regelbasierte Systeme sind desweiteren sehr unflexibel, was kleine Abweichungen vom typischen Verhalten angeht. Nur kleine Änderungen in den von den Regeln vorgesehenen Mustern für Hacker reichen einem Eindringling aus um unerkannt zu bleiben.

4.1 Neuronale Netze

Ein künstliches neuronales Netz besteht aus einer Sammlung von verbundenen Verarbeitungseinheiten, die eine Menge von Eingaben in eine gewünschte Menge von Ausgaben umsetzen. Das Ergebnis dieser Transformation wird durch die Charakteristik der einzelnen Einheiten und die Gewichte der Verbindungen festgelegt. Durch Modifizierung der Verbindungen zwischen den Knoten ist das Netzwerk in der Lage die gewünschte Ausgabe zu erzielen [16].
Mit neuronalen Netzen Eindringlinge zu erkennen ist ein weiterer Ansatz, der aber einen großen Vorteil gegenüber den im letzen Abschnitt genannten Techniken bietet. Anders als bei den regelbasierten Systemen wird hier keine absolute Übereinstimmung zwischen den vorgegebenen und den zu untersuchenden Daten benötigt um Alarm schlagen zu können. Das neuronale Netz analysiert die Eingabe und schätz die Wahrscheinlichkeit mit der die Daten zu den Charakteristiken passen, die es trainiert wurde zu erkennen. Wenn nun die Wahrscheinlichkeit eines Angriffs höher liegt als die vom Administrator gewählte, kann Alarm ausgelöst werden obwohl das Verhalten nicht absolut mit dem Muster eines Angriffs übereinstimmt.
Trotz der Vorteile neuronaler Netze kann sich ein Computer mit ihnen noch nicht selbst beschützen. Um einen Angriff erkennen zu können muss das Netz darauf trainiert werden, d.h. zu Beispieleingaben müssen durch Modifizierung der Verbindungen zwischen den Knoten die gewünschten Ausgaben produziert werden. Daher werden neuronale Netze vorzugsweise bei speziellen, statischen Problemen eingesetzt für die es gut trainiert werden kann.
Um die Vorzüge dieser Technik jedoch auch zur Abwehr von Angriffen auf autonome Computersysteme nutzen zu können, forscht man an selbstlernenden neuronalen Netzen [17]. Diese bekommen Rückmeldungen auf ihre Ergebnisse vom Sicherheitssystem, ob sie erfolgreich waren oder immer noch Lücken bestehen. Mit diesen Informationen können sie ihre Konfiguration selbstständig anpassen.

5 Sicherheit autonomer Computersysteme

Die in dieser Ausarbeitung bisher vorgestellten Methoden zum Selbst-Schutz von Computersystemen dienen alle der Behandlung von Viren oder dem Entdecken

von Eindringlingen, die bereits das System korrumpieren. Autonome Computersysteme benötigen zusätzlich eine von Grund auf neue, sichere Architektur um ihre vielfältigen Möglichkeiten schützen zu können. Sie werden in einer sich ständig ändernden Welt zurechtkommen und mit anderen vielleicht unsicheren Systemen und Menschen zusammenarbeiten, sowie persönliche Daten nach unterschiedlichen Gesetzen schützen müssen. Um auf diese Veränderungen reagieren zu können, haben sie die Möglichkeit zur Selbst-Konfiguration und Selbst-Optimierung. Jeder neue Zustand in den das System übergeht muss geprüft und mit den Sicherheitsbestimmungen konform sein. Es bedarf einem strikten und einsehbaren Sicherheitskonzept, damit Menschen dem Computer vollständig ihre Daten und wichtigen Prozesse anvertrauen können.

5.1 Architektur autonomer Systeme

Ein Vorschlag von IBM [18] geht in die Richtung große autonome Systeme aus kleineren zusammenzusetzen. Die kleinste nicht mehr unterteilbare Einheit ist dabei ein autonomes Element, welches sehr einfach aufgebaut ist und festgelegte Funktionen ausführen kann. Es arbeitet über längere Zeiträume mit denselben autonomen Elementen zusammen. In höheren Ebenen arbeiten autonome Systeme in dynamischen Umgebungen und es bleiben nur die Hauptaufgabe sowie zentrale Richtlinien gleich. Wie Aufgaben erledigt werden und welche Mittel zur Verfügung stehen kann sich jedoch sekündlich ändern.
Ein autonomes Element wird aus einer *funktionalen Einheit* und einer *Management Einheit* bestehen. Die funktionale Einheit beinhaltet alle Funktionen, wie speichern, Webdienste usw., die dieses Element ausführen kann. Die Management Einheit hingegen überwacht die Operationen der funktionalen Einheit, sorgt für ausreichende Ressourcen, rekonfiguriert sie bei sich ändernden Bedingungen und führt Verhandlungen mit anderen autonomen Elementen, die Dienste in Anspruch nehmen wollen oder die vom autonomen Element Aufgaben zugeteilt bekommen.

Abbildung 5 zeigt eine einfache Darstellung eines autonomen Elements, bei dem die dünnen senkrechten Pfeile der Management Einheit deren Kommunikation mit anderen Elementen oder externen Ressourcen andeuten. Die dicken senkrechten Pfeile der funktionalen Einheit deuten Eingaben oder Ressourcen an, die sie zur Ausführung ihrer Funktionen benötigt und Ausgaben mit denen sie ihre Ergebnisse anderen Elementen weitergibt. Die Striche zwischen den Einheiten repräsentieren die Sensoren und Steuerungsmöglichkeiten mit denen die Management Einheit die funktionale Einheit überwacht und steuert. Speziell die Pfeile zwischen der Management Einheit und den Kreisen um die Funktionspfeile deuten die Zugriffskontrolle an mit der die Management Einheit die Ein- und Ausgänge der funktionalen Einheit überwacht.
Dieser Aufbau eines autonomen Elements trägt maßgeblich zur Sicherheit eines großen autonomen Systems bei, denn kein anderes Element kann diesem autonomen Element Ressourcen anbieten oder einen Dienst verlangen, ohne die Erlaubnis der Management Einheit. Der Vorteil dieser Aufteilung ist ebenfalls in Abbildung 5 zu erkennen. Wenn ein Angreifer es geschafft hat eine Hintertür

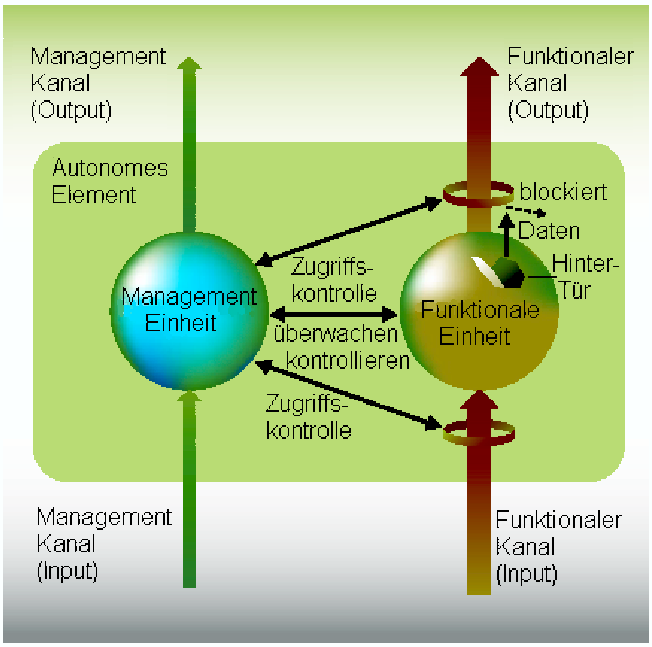


Abbildung 5. Autonomes Element

im System zu erzeugen, so kann er heutzutage durch diese undichte Stelle Daten löschen oder sich senden lassen. Wenn er diese Hintertür jedoch bei funktionalen Elementen ansetzt und diese dann fremden Code ausführen lässt, so wird dieser Vorgang sofort von der Management Einheit bemerkt und gestoppt, da er nicht mit den Sicherheitsbestimmungen für dieses autonome Element übereinstimmt. Damit Manangement Einheiten ihre überwachenden und leitenden Aufgaben korrekt erfüllen können benötigen sie Zugriff auf Richtlinien, nach denen sie ihr Verhalten ausrichten müssen. Anders als bisherige Computer, deren Verhalten explizit programmiert war, hat die Management Einheit eines autonomen Elements eine große Auswahl an möglichen Strategien um die aktuelle Aufgabe unter Berücksichtigung der Richtlinien zu erfüllen. Einige davon sind Sicherheitsrichtlinien, die der Management Einheit z.B. mitteilen, welche Sicherheitsstufe auf die von der funktionalen Einheit benutzten Daten anzuwenden ist oder Regeln mit denen das Vertrauen in andere Elemente bestimmt werden kann,

mit denen das Element kommuniziert. Außerdem muss die Management Einheit wissen, welche kryptografischen Protokolle in verschiedenen Situationen benutzt werden sollen und unter welchen Umständen das autonome Element Sicherheits- oder andere Updates annehmen darf usw. Diese Richtlinien sind entweder vom Menschen vorgegeben oder werden aus Rahmenrichtlinien höherer Ebenen abgeleitet.

5.2 Datenschutz und Vertrauen

Einer der Gründe warum Computersysteme immer komplexer werden ist die zunehmende Menge an Informationen die verarbeitet werden kann. Dabei sind viele dieser Informationen privater Natur, die in unterschiedlichen Ländern nach verschiedenen Sicherheitsstandards geschützt werden müssen. Es muss autonomen Systemen also möglich sein, zuverlässig und automatisch die Sicherheitsstufe der Daten mit denen sie arbeiten festzustellen und passende Richtlinien zu verwenden.
Sicherheitsrichtlinien sowie Verhandlungen unter autonomen Elementen müssen auch komplexe politische und geographische Situationen berücksichtigen. Wenn bestimmte Daten in einigen Ländern verboten sind, so muss das autonome Element, welches diese Daten zur Bearbeitung verschicken möchte, diese Information besitzen und mit den Daten verschicken. Andere Elemente wissen nun sofort auf welchen Routen die Daten weitergesendet werden dürfen.

6 Autonome Software

Bei dem Thema autonome Computersysteme stößt man immer wieder auf die Firma IBM, die in diesem Bereich ihre Forschungen stark vorantreibt. Außer Theorien und Blaupausen gibt es auch schon Software mit autonomen Fähigkeiten, die Benutzern und Administratoren das Leben erleichtern soll. IBM hat für diese Art Software im Bereich Selbst-Schutz fünf Implementierungsstufen festgelegt [19]:

Stufe 1: Elementar Lokale Sicherheitskonfigurationen benötigen Administratoren, die Sicherheitseinstellungen und Veränderungen an jeder Komponente unabhängig vornehmen. Zum Schutz der Daten werden lokale Datensicherungsprogramme benutzt und Ergebnisse von Systemprüfungen kann der Administrator abrufen. In dieser Stufe wird sehr viel Mitarbeit des Menschen verlangt um laufende Programme und dauerhaft Daten zu schützen.

Stufe 2: Geführt Durch Managementprogramme wird die Sicherheitsadministration zentralisiert und es können Benutzeridentifikatoren vergeben sowie Zugriffe auf priviligierte Ressourcen gesteuert werden. Prüfprogramme sammeln Daten über unbefugtes Eindringen in das System, die daraufhin manuell durchgesehen und geeignete Maßnahmen zur Abwehr zukünftiger Attacken ergriffen werden. Zentralisierte Backupsysteme und Wiederherstellungswerkzeuge bieten eine schrittweise Sicherungsfähigkeit über viele Ressourcen.

Stufe 3: Voraussehend Sicherheitsrichtlinien einer Firma können mit Hilfe von Programmen unternehmensweit konsistent verbreitet werden. Identifikatoren und Zugriffsrechte werden quer durch viele Anwendungen koordiniert und wenn nötig zurückgenommen. Sensoren können Sicherheitsverletzungen registrieren und eventuelle Eindringlinge erkennen, die dann von anderen Programmen bearbeitet werden.

Stufe 4: Lernfähig Sicherheitsmanagement konzentriert sich auf fortgeschrittene Automation, wobei ohne Mithilfe des Administrators neue Benutzer angelegt und Identifikatoren gelöscht werden von denen die ein Unternehmen verlassen. Zugänge zu Systemen und Anwendungen, die für ein neues Projekt nötig sind werden ebenso automatisch gewährt, wie die für abgeschlossene Aufgaben gelöscht werden. Wenn Zugriffsverletzungen oder Eindringlinge auftreten werden durch Selbst-Konfiguration quarantäne Bereiche eingerichtet und verantwortliche Identifikatoren gesperrt.

Stufe 5: Autonom In dieser Stufe geht es um lernende Systeme, die Sicherheitsrichtlinien niedriger Ebenen oder einzelnen Komponenten zu unternehmensweiten Vorgaben anpassen können. Die enge Zusammenarbeit der einzelnen Systeme ermöglicht schnelles rekonfigurieren, automatisches installieren von Sicherheitsupdates und modifizieren der Überwachungsprogramme. In diese Stufe ist auch das in Kapitel 3 beschriebene Immunsystem sowie das aus Kapitel 4 bekannte lernende neuronale Netz einzuordnen.

6.1 Produkte

IBM hat in seiner Tivoli Produktfamilie einige Programme, die einen gewissen Selbst-Schutz bieten. Zum einen gibt es den *Tivoli Storage Manager*, der automatische Backups durchführt und diese in verschiedenen Umgebungen speichert. Der *Tivoli Access Manager* hilft unautorisierten Zugriff auf das System zu verhindern. Er benötigt einen eigenen Sicherheitsserver mit dem er eine große Auswahl an Authentifizierungsmethoden anbieten kann und das komplette System überwacht. Mit dem *Tivoli Risk Manager* werden systemweit potentielle Bedrohungen wie Viren und Eindringlinge erkannt und automatisch durch Selbst-Konfiguration darauf reagiert. Er sammelt Informationen von Firewalls, Virenscannern, Aktivitätskontrollen und anderen Überwachungsprogrammen und wertet sie aus.

7 Zusammenfassung und Ausblick

Anhand von Kapitel 2 und 3 läßt sich erkennen, dass zwischen Theorie und Praxis noch eine große Lücke klafft. Die Forderung nach einem Immunsystem für jeden PC wird noch nicht erfüllt. Stattdessen gibt es nun ein großes Immunsystem, worin alle Rechner integriert und eine zentrale Rechenanlage die Funktion

des Immunsystems übernimmt. Ob diese Architektur nicht nur in Testläufen sondern auch im Alltag funktioniert wird sich noch zeigen.
Man ist jedoch mit dieser interdisziplinären Zusammenarbeit zwischen Medizin und Computertechnik auf dem richtigen Weg, da es definitiv eine Vielzahl von Parallelen zwischen elektronischen und menschlichen Viren gibt. Bei Projekten an denen Mediziner und Anti-Viren Spezialisten gemeinsam gearbeitet haben hat sich gezeigt, dass beide Seiten voneinander lernen können.
Für den Selbst-Schutz zukünftiger autonomer Computersysteme müssen die Konzepte und Theorien noch in praxistaugliche Anwendungen umgesetzt werden. Zur Zeit ist man noch nicht so weit den Schutz wichtiger Systeme ganz dem Computer zu überlassen. Zwar ist vieles automatisiert (Kapitel 6.1), doch sitzen am Ende immer noch Experten an den wichtigen Stellen um auf Meldungen und Warnungen reagieren zu können.
Die Forschung im Bereich autonomer Computersysteme wird jedoch enorm vorangetrieben, sodass mittelfristig wohl das Ziel eines selbstständig arbeitenden Computers jedenfalls ansatzweise erreicht wird.

Literatur

1. D.M. Chess, J.O. Kephart ,,The Vision of Autonomic Computing" IEEE Society 2003.
2. Http://www.research.ibm.com/autonomic/
3. J.O. Kephart, Steve White ,,Biologically Inspired Defenses Against Computer Viruses" Proceedings of IJAI, Montreal, 1995.
4. E.H. Spafford ,,Computer Viruses: A Form of artificial life?" Artificial Life 2. Studies in the Sciences of Complexity, 727-747. Addison-Wesley, 1991.
5. S.R. White, and D.M. Chess ,,Computers and epidemiology" IEEE Spectrum 20-26, May 1993.
6. J.O. Kephart ,,A Biologically Inspired Immune System for Computers" MIT Press, 1994.
7. W.E. Paul ,,Immunology: Recognition and Response" Readings from Scientific American, New York: W.H. Freeman and Company, 1991 .
8. S.R. White, J.O. Kephart ,,Measuring and modeling computer virus prevalence" Proceedings of the 1993 IEEE Computer Society Symposium on Research in Security and Privacy. Oakland, California, 1993.
9. J.O. Kephart ,,How topology affects population dynamics" Artificial Life 3. Studies in the Sciences of Complexity, Addison-Wesley, 1994.
10. G.B. Sorkin, M. Swimmer and S.R. White ,,Blueprint for a Computer Immune System" Proceedings of the International Virus Bulletin Conference, San Francisco, California, 1997.
11. S.R. White, M. Swimmer, E.J. Pring, W.C. Arnold, D.M. Chess, J.F. Morar ,,Anatomy of a Commercial-Grade Immuns System"
12. Symantec Corporation ,,Understanding Heuristics: Symantecs Bloodhound Technology" Symantec White Paper Series, Volume XXXIV, http://www.symantec.com/avcenter/reference/heuristic.pdf.
13. G. Tesauro, G. Sorkin ,,Neural Networks for Computer Virus Recongnition" IEEE Expert, Vol. 11, 1996

14. W.C. Arnold, J.O.Kephart ,,Automatic Extraction of Computer Virus Signatures"Proceedings of the 4th International Virus Bulletin Conference, Jersey, UK, 1994.
15. D.M. Chess, G.B. Sorkin ,,Automatic Analysis of a Computer Virus Structure and Means of Attachment to its Hosts" U.S. Patent 5,485,575.
16. D. Hammerstrom ,,Neural Networks At Work" IEEE Spectrum, pp. 26-53, 1993.
17. J. Cannady ,,Next Generation Intrusion Detection: Autonomous Reinforcement Learning of Network Attacks" Proceedings of the 23rd National Information Systems Security Conference, 2000.
18. D.M. Chess, C.C. Palmer, S.R. White ,,Security in an autonomic computing environment" IBM Systems Journal, Vol 42, No 1, 2003.
19. IBM ,,How Tivoli software products support the IBM Autonomic Computing Initiative" A technical view of autonomic computing, October 2002.

Self-Healing

Oliver Hessel

Zusammenfassung. In Zeiten wachsender Komplexität der Hard- und Software, Wachstum der Einsatzgebiete von Computern, wird es schwieriger, als Mensch den Fehler zu finden, wie oft ist die „Lösung“, Neustarten und sehn ob es hilft?
Als Ausweg aus diesem Dilemma erscheint ein neuer Ansatz in der Informatik: das Autonomic Computing (AC). Diese Seminararbeit setzt sich mit einem Teilbereich des AC besonders auseinander, dem Self-Healing (SH).

Einleitung

Ein SH System [16] muss fähig sein, Probleme, oder potentielle Probleme zu entdecken und angemessen auf diese zu reagieren. Anfangs werden die „Heilungen“ durch Autonome Systeme noch den vorgegebenen Regeln der Menschlichen Experten folgen, aber mit der Steigerung der Intelligenz der Systeme werden sie eigene Regeln entwickeln, die spezifizierten Ziele zu erreichen.

Zu SH gibt es verschiedene Ansätze, bei denen manche im Kern auf ähnlichen Strategien basieren. Einige von Ihnen werden in dieser Arbeit vorgestellt, bewertet und verglichen, ist es vielleicht sogar möglich verschiedene Konzepte miteinander zu verbinden?

1. SH durch Zellregeneration basiert auf naturähnlichen Strukturen, die einzelnen „Zellen“ beobachten ihre direkte Umgebung und verhalten sich wie Endliche Automaten.

2. ARMORs sind gekapselte Umgebungen für kommerzielle Software, die es erlauben, ihren Zustand zu überwachen und gegebenenfalls neu zu starten. Eine ARMOR Architektur besteht aus mehreren Ebenen, die in hierarchischer Struktur für System-Verfügbarkeit sorgen.

3. Die Zeitliche Auswirkung von Neustarts ist mit ausschlaggebend für die Verfügbarkeit eines Systems, da einige SH Ansätze auf Neustarts basieren, ist es sinnvoll die Zusammenhänge zwischen Verfügbarkeit, Ausfall und Wiederherstellungszeit zu kennen.

4. Rekursive Neustartbarkeit und Wiederherstellungsorientiertes Rechnen basiert auf der Idee, einzelne Komponenten fehlerbehafteter Software (die Software die ohne Fehler ist werfe den ersten Stein...) durch Neustart aus einem festgefahrenen, oder abgestürzten Status zurückzuholen.

5. JAGR, ein selbst wiederherstellender Webserver, der durch mehrere Kombinierte SH Techniken hohe Verfügbarkeit garantiert, nutzt Statistiken früherer Ausfälle um Teile seiner Anwendungen neu zu starten.

6. Durch SH auf Architekturebene ist es möglich, während der Laufzeit Komponenten des Software-Systems auszutauschen, Reparaturen bzw. Änderungen zu planen und vor Anwendung anhand von Modellen zu testen.

7. Checkpointing ist eine Strategie ein System zu sichern, damit nach einem Neustart der letzte als funktionierend bekannte Status wiederhergestellt werden kann, als wäre nie neu gestartet worden.

8. All diese Ansätze gehen davon aus, zu wissen, wann ein „kranker Zustand" erreicht ist, um dann zu Heilen. Kann man aber nicht auch schon präventiv Heilen, oder bei den ersten Anzeichen einer schleichenden Verschlechterung der Leistung? – Wie aber stellt man einen solchen Zustand fest?

„Es ist Zeit, dass Computer lernen sich selbst zu heilen"

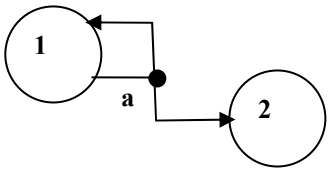
1 Selbst-Heilung durch Zellregeneration

Es herrscht steigender Bedarf an Softwaresystemen, die sich veränderten Ressourcen, Anforderungen und Störungen anpassen, oder auf Störung von Komponenten und Angriffe von außen reagieren [1]. Solche SH Systeme können sich in Situationen, in denen unterbrechungsfreier Betrieb entscheidend, oder manuelle Reparatur nicht möglich ist, als sehr nützlich erweisen. Diese Ziele zu erreichen, lässt die Forschung nach neuen Ansätzen suchen. Hier zeigt uns die Natur ihr annähernd perfektes Konzept, Zell-Basiertes Programmieren. Die Programmierung der Natur basiert auf einigen außergewöhnlichen Eigenschaften:

1. *Umgebungs-Bewusstsein*: Obwohl Zellen nur eingeschränkte Kommunikations-Fähigkeiten haben, sind sie doch in der Lage auf ihre direkte Umgebung zu reagieren.
2. *Anpassung*: Viele Zellen haben erstaunliche Anpassungsfähigkeiten, so kann z.B. eine Keimzelle bei den ersten Zellteilungen den Verlust einer zweiten ausgleichen, das ist ein Hinweis darauf, dass alle Zellen in ihrer Entwicklung das gleiche „Programm" ablaufen lassen.
3. *Redundanz*: Biologische Systeme sind in hohem Masse redundant, so wird ein Zell-Ausfall während der Entwicklung meist von anderen mit übernommen, sodass am Ende ein funktionierender Organismus steht.
4. *Dezentralisierung*: Es gibt keine globale Koordination im Entwicklungsprozess und eingeschränkte Kommunikation. Die Zellen spüren die Eigenschaften ihrer Umgebung [2] und werden von nahe liegenden Zellen beeinflusst (Neueste Forschungen haben gezeigt, dass die Genetische Programmierung für bestimmte Körperteile nicht spezialisiert ist, so hat man einem Fliegen-Ei ein Augen-Gen

einer Maus eingesetzt und trotzdem ist ein Facettenauge entstanden). Zellen können andere nahe liegende zu bestimmten Aktionen veranlassen, aber es gibt keine zentrale Kontrolle darüber.

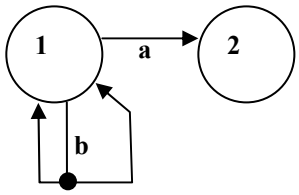
In dem hier beschriebenen Ansatz entspricht das Zell-Programm einem Automaten. Die lokalen Umgebungsbedingungen dienen als Zustandseingaben, die Ausgaben sind Übergänge, oder Verzweigungen in mehrere Zustände. Das Zell-Programm beginnt mit einer Grundkonfiguration und folgt den Übergangsregeln wie ein Endlicher Automat. Während der Laufzeit bestimmt ein Umgebungs-Simulator externe Reize, die beispielsweise durch Operationen, Eingaben oder Ausfälle des Systems hervorgerufen werden. Da die einzelnen Zellen Änderungen in ihrer Umgebung spüren, können sie auf diese reagieren, beispielsweise können sie eine Fehlerbehandlung (Heilung) oder Neuanordnung (Regeneration) vornehmen.
Das Prinzip hinter dieser Art von Selbst-Heilung ist, dass die einzelnen Zellen Ständig ein Lebenssignal aussenden, wodurch eine nahe liegende Zelle den Ausfall bemerken und kompensieren (Zellteilung) kann.
Ein Beispiel für ein einfaches Zellprogramm (Abb. 1) produziert eine Reihe von Zellen, solange die Eingabe *a* besteht, *a* kann z.B. eine Wachstumsvorrausetzung darstellen.
Ein zweites (Abb. 2) ist für die Zellteilung bei einem Angriff zuständig, welcher durch die Eingabe *b* repräsentiert wird, es werden zusätzliche Zellen Produziert, so dass einige den Angriff „überleben“.



a – Positive Vorraussetzung

Kreise: Zustände,
Pfeil: Übergang,
Punkt: Zellteilung

Abb. 1: Erstellung einer Reihe von Zellen



b – Negative Vorraussetzung

Kreise: Zustände,
Pfeil: Übergang,
Punkt: Zellteilung

Abb. 2: Neubildung von Zellen gegen Angriff

Durch diese Art von Programmierung soll es zukünftig möglich sein, robuste SH-Systeme zu erstellen die komplexe Aufgaben erledigen können. Am Ende dieser Forschung soll eine Hochsprache stehen, deren Compiler, Programme auf Basis von Zellularautomaten erstellt.

2 Selbst-Heilung durch ARMOR

Traditionell wird Fehlertoleranz durch dedizierte Hard bzw. Software erreicht [3, 4], die erreichte Fehlertoleranz durch Hardware ist meist statisch während der gesamten Laufzeit, wobei man durch verteilte Software die gewünschte Stufe der Redundanz anpassen kann. Diese Anwendungen müssen jeweils für die Spezielle Umgebung angepasst werden.
In heutigen Rechnersystemen müssen sowohl Kommerzielle wie auch Wissenschaftliche Anwendungen koexistieren, die alle verschiedene Level der Verfügbarkeit und Verlässlichkeit benötigen. Da es nicht gerade Kosten sparend ist, für jede Anwendung spezielle Hardware bzw. angepasste Software zu haben ist die Bestrebung, hohe Verfügbarkeit mit Soft- und Hardware von der Stange zu erreichen. Hier ist einer der Ansätze die Softwarearchitektur Chameleon, die in einer vernetzten Umgebung und sogar auf verschiedenen Betriebssystemen arbeitet. Der Kern dieser Architektur sind ARMORs (Anpassungsfähige, Rekonfigurierbare, Mobile Objekte für Zuverlässigkeit), sie sind die Komponenten, die in der Chameleon Umgebung alle Operationen kontrollieren.
Unter diesen gibt es 3 verschiedene Typen:
Manager, die andere ARMOR überwachen und Ausfallbehandlung vornehmern, Deamons, welche zur Fehlerfindung und Kommunikation dienen und Standard ARMOR, die die anwendungsabhängige Zuverlässigkeit bereitstellen. Dadurch, dass ein ARMOR das „Betriebssystem“ für eine bestimmte Anwendung darstellt, kann es schlank gehalten werden, indem der Anwendung nur die von ihr benötigten Funktionen, wie z.B. Checkpointing (siehe Abschnitt 7) bereitgestellt werden. Da die untersten ARMOR nicht voneinander abhängen und jeweils gekapselte Rahmenbedingungen anbieten, können sich abstürzende Programme nicht gegenseitig beeinflussen. Eine abgestürzte Anwendung, bzw. ARMOR kann neu gestartet werden, sobald sie kein Lebenssignal mehr von sich gibt.

3 Welche Zeitliche Auswirkung haben Neustarts auf die Verfügbarkeitsstatistik?

„*Wenn ein Problem keine Lösung hat, so muss es keines sein,*
aber eine Tatsache, die nicht gelöst werden, sondern mit der man mit der Zeit zurechtkommen muss“ - Shimon Peres

Dieser Leitspruch bestimmt auch das Wiederherstellungsorientierte Rechnen (ROC), Fehler durch Menschen, Hard- und Software sehen wir als Tatsachen, nicht als zu lösende Probleme und schnelle Wiederherstellung ist, wie wir damit Fertigwerden.
Wenn man an ROC bzw. RR (Rekursive Neustartbarkeit) herangeht wird man auf einige Fragestellungen stoßen:

- Warum Besser öfter kurze Störungen als wenige lange?
- Warum Neustarten, wenn gerade gar kein Fehler passiert?

– Warum ist ein Menschlicher Fehler nicht auszuschließen und was hat das mit MTTR zu tun?

Verfügbarkeit berechnet sich aus der Durchschnittlichen Zeit zu Versagen (MTTF) und der Durchschnittlichen Zeit zur Wiederherstellung (MTTR) also: Verfügbarkeit = MTTF/(MTTF+MTTR) [5, 6]. Durch diese Gleichung ist zu erkennen, dass eine Verringerung der Wiederherstellungszeit ebenso gut ist, wie eine Verringerung Versagensrate. Einige Ansätze zeigen sogar, dass es besser ist die Wiederherstellungszeit zu verringern:

1. Die MTTF heutiger Hardware ist so hoch, dass sie sehr schwer zu messen ist.
   Ein Versagen tritt immer unerwartet ein, und ist nicht planbar, wohingegen die Wiederherstellungszeit ein sehr gut bekannter Gegner ist.
2. Bei Webseiten kann eine verringerte MTTR direkt die Erfahrung eines Benutzers beim Ausfall beeinflussen. Ein kurzer Ausfall wird eher toleriert, als ein langwieriger, deshalb genügt es nicht, die Wahrscheinlichkeit eines Versagens zu verringern.
3. Kurze Ausfälle sind weniger Medienwirksam als lange, sofern sie überhaupt bemerkt werden. Ebay hatte April 2002 einen 4,5 Stunden-Ausfall der meisten seiner Systeme, laut der Verfügbarkeitsformel, selbst bei 2 solcher Ausfälle pro Jahr (abgesehen von Routinewartungen) hätte Ebay eine Verfügbarkeit von: (182x24std)/(182x24std+4std)=99,9%. (Durch einen 22 Std. Ausfall 1999 fiel der Aktienkurs um 26% und Ebay selbst musste an seine Kunden Zugeständnisse im Wert von 3-5 Millionen US$ machen).
4. Am ehesten zu akzeptieren ist wohl noch eine kurzzeitige Verlangsamung der Webseite, was für die Wiederherstellung bedeutet, dass möglichst kleine Komponenten einer Anwendung neustartbar sein müssen, die dann nur verlangsamend wirken.
5. Neustarten um einen zukünftigen Fehler zu verhindern ist speziell wirksam wenn es sich um Speicherüberladung durch fehlerhafte Software handelt.
6. Es gibt zwei verschiedene Arten menschlicher Fehler, Ausrutscher, wenn man nicht das tut, was man beabsichtigt und Planungsfehler, wenn man tut was man beabsichtigt, aber die Falsche Richtung gewählt hat. Ein wichtiges Kriterium ist, dass der Mensch 75% seiner Fehler direkt erkennt, falls es dann schon zu spät ist, ist Wiederherstellung gefragt. Menschliche Fehler sind unvermeidlich, bei über 50% der Ausfälle handelte es sich um einen Bedienungsfehler- da kann die MTTF noch so hoch sein, hier ist nur die Wiederherstellungszeit von Bedeutung.

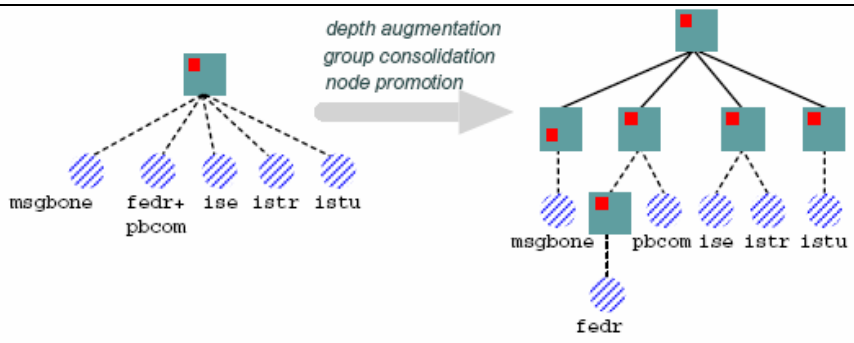
Die MTTF zu erhöhen kann nie ein ausfallfreies Zeitintervall garantieren, wobei die Verringerung der MTTR den Ausfall kalkulierbarer macht.

4 Rekursive Neustartbarkeit (RR)/ Wiederherstellungs-orientiertes Rechnen (ROC)

Selbst nach Jahrzehnten der Softwaretechnikforschung gibt es immer noch Softwareausfall [9, 10], vorrangig durch nichtdederministische Fehler, der sich durch Neustarten lösen lässt. Wahrscheinlich muss man sich mit dieser Art Fehlern, die als Heisenbugs bekannt sind abfinden. Softwarefehler führen zu Abstürzen, Lifelocks, Deadlocks (In einem Deadlock sind die Prozesse hoffungslos blockiert, während in einem Lifelock noch die Möglichkeit des Fortschritts besteht, aber keine zeitliche Garantie dafür gegeben werden kann.), Endlosschleifen oder zu Speicherfehlern. Die meisten dieser Fehlerzustände sind nur zufrieden stellend durch Anwendungs- oder Systemneustart zu beheben. Es ist Anzunehmen, dass diese Art von Fehlern auch zukünftig, besonders durch immer komplexer werdende Systeme erhalten bleibt. Obwohl RR (Rekursive Neustartbarkeit) kein Freibrief ist, minderwertige Software zu erstellen, ist sie doch ein Instrument mit günstiger Software von der Stange zurechtzukommen, besonders wenn sie bereits auf dem System implementiert ist. Neustarts sind ein effektives Workaround für vorübergehende Ausfälle, bringen sie doch die Anwendung in den am besten getesteten Zustand: den Startzustand, außerdem werden Speicherfragmentierungen durch den Neustart wieder freigegeben. Leider zeichnen sich Neustarts bei den meisten Systemen durch Zeitintensivität und Statusverlust aus, alleinige Neustarts der betroffenen Subsysteme sind jedoch meist nicht möglich.
RR-Systeme tolerieren Neustarts sehr gut, da die einzelnen Komponenten sehr feingranular auf ihre gegenseitigen (Neustart-)Abhängigkeiten hin in Neustart-Bäumen organisiert sind. Die Blätter, welche die eigentliche Softwarekomponente darstellen sind gegenüber anderen isoliert, so hat beispielsweise jedes Blatt seine eigene Java virtual machine, sodass ein Absturz nicht andere Komponenten mit in den „Tod“ reißt.
Ein Beispiel für RR ist eine Bodenstation [10], die nur kurze Zeit Kontakt zu einem Satelliten hat, während dieser Zeit ist ein Ausfall besonders schwerwiegend, da wichtige Messdaten nicht empfangen werden können. Eigens dafür wurde die Architektur Mercury eingeführt, die nach 2 Jahren menschlicher Administration die Verfügbarkeit der Empfangsstation sichert.

Die Neustartbefehle selbst werden, inklusive der Spezifikation wo neu zu starten ist, von dem Orakel, das die Neustart-Strategie darstellt, an den Restarter weitergeleitet. Führt der Neustart nicht zum Erfolg, wird der Neustart einen Knoten höher im Baum ausgeführt. Ob eine Komponente noch läuft, erkennt das Orakel durch ein Lebenssignal, das von ihr ausgesandt wird, falls dieses ausbleibt wird der Restarter angesteuert. Der einzige aktive Eingriff, den das Orakel machen kann, ist den Restarter neu zu starten, falls dieser ausfällt, im Gegenzug überwacht der Restarter das Orakel auf Ausfall. Ein Ausfall beider gleichzeitig, wäre für das System fatal.
Durch die Einführung von RR wurde die Verfügbarkeit der Bodenstation erheblich vergrößert, was in folgendem Schaubild (Abb. 3) verdeutlicht ist [7]. Die einzelnen Komponenten wurden auf Abhängigkeit untersucht und soweit möglich in einzelne Neustart-Blätter aufgeteilt. Wie man in der Tabelle ablesen kann hat man hier mit RR

große Erfolge in der Wiederherstellungszeit zu verbuchen, besonders wichtig bei den oft ausfallenden Komponenten (MTTF Messung nach 2 Jahren Betrieb durch Personal).



Failed component	msgbone	fedr	pbcom	ise/istr	istu
MTTF	1 month	10 min	1 month	5 hours	5 hours
MTTR before	28.9 sec	28.9 sec	28.9 sec	28.9 sec	28.9 sec
MTTR after	4.7 sec	5.0 sec	21.9 sec	6.1 sec	5.8 sec

Abb. 3: Die Mercury Architektur bevor und nach Einsatz von RR

Komponenten:
msgbone: interne XML Kommunikationsplattform; fedr: interner XML Kommando Übersetzer; pbcom: Komponenten Datenbus; ise: Satteliten Positionsschätzer; istr: Satteliten Kontakthaltung; istu: Funkgerät

Die Neustart-Fähigkeit hat eines der bekanntesten Raumfahrzeuge [17] davor bewahrt, allzu früh Weltraumschrott zu werden, bei der Pathfinder Mission hat sich im Echtzeit-Betriebssystem VxWorks des Pathfinder ein Deadlock von verschiedenen Steuerungen, durch Prozessprioritätsfehler ergeben. Nur durch den Watchdog-Timer, der wenn über bestimmte Zeit kein Lebenssignal mehr von den Systemprozessen kommt, das System neu startet, konnte Pathfinder gerettet werden. Der nur schwer reproduzierbare Fehler wurde letztendlich durch ein Update des Systems, mit dem Prozessprioritätsvererbung eingeführt wurde, gelöst (d.h. Ein Prozess mit niedriger Priorität läuft, währenddessen startet eine Prozess mit hoher Priorität, der den Niedrigprioren verdrängen sollte, Gibt es aber Prozessabhängigkeiten, so erbt der Niedrige kurz bis zu seiner Fertigstellung die Priorität des hohen.).

5 JAGR: Selbst wiederherstellender Webserver

Für die JAGR [19] Architektur wurden einige SH Techniken in den Open Source J2EE (Java 2 Enterprise Environment) Server JBoss integriert, das daraus resultierende System nennt sich JAGR-JBoss. Aus der Annahme heraus, dass Fehler nicht zu verhindern sind, setzt dieses System auf ROC [7]. Dieser Ansatz zielt im

Besonderen auf Internetdienste die traditionell mehrstufig aufgebaut sind. Ausgehend davon kann man Annahmen über ihre Struktur machen, um anwendungsunabhängig nur durch Veränderung der Middleware (*Hardware und Software zum Betreiben eines Programms auf anderem Computermodell, z.B.: JAVA)* zu erreichen.
JAGR benutzt Anwendungsunabhängige Fehlerpfad Spekulation (AFPI), AFPI nutzt kontrollierte Fehlererzeugung und Statistik, Ausfallabhängigkeiten zu erkennen. Als zweite Strategie wird zur Wiederherstellung RR eingesetzt.
Im Falle eines Ausfalls wird ein, zwischen dem HTTP Server und dem Endbenutzer liegender, Proxy, der die Verbindung verlangsamen oder stoppen kann, aktiviert um dem Benutzer nach der Wiederherstellung wieder zu bedienen.
JAGR stützt sich in der Ausnahmebehandlung auf verschiedene Fehlerüberwachungen:

1. ExcMon, er überwacht Java Ausnahmen. Dieser Fehlermonitor instruiert die JAGR Komponenten, die Ausnahme ohne Weiterleitung an den Benutzer zu unterbrechen. Sobald dies geschehen ist, werden der Fehler und Information über betroffene Komponenten an den Wiederherstellungsmanager weitergeleitet. Die Fehler, weitergeleitet durch Java Ausnahmen, die so erkannt werden können beinhalten viele Anwendungsfehler (z.B.: Nullzeiger), Ressourcenausfälle (z.B.: zu wenig Speicher) und Netzwerkfehler.

2. E2Emon, ist ein Client Simulator, der Anwendungs spezifische HTTP Anfragen durch ein Webfrontend schickt um Fehler, wie vom Benutzer erlebt, zu erkennen. Diese werden an den ExcMon weitergeleitet um dessen Fehlererkennungen zu validieren.
   Level, auf denen E2Emon Fehler erkennt:
   Netzwerk-Level: Verbindung zwischen Klient und JAGR geschlossen, verweigert oder Zeitüberschreitung
   HTTP-Level: alle HTTP Fehler- und Rückmeldungen die auf Ausfälle hinweisen werden untersucht (z.B.: Zugriff verboten, Serverfehler)
   HTML-Level: leere HTML Seiten, oder Schlüsselwörter, die in Fehlerseiten zu finden sind
   Per Definition kann E2Emon nicht komplett Anwendungsabhängig sein, die Schlüsselwörter müssen Anwendungsspezifisch angepasst werden, um nicht Hilfeseiten, die *Error* als FAQ Punkt enthalten als Fehler gewertet werden.

3. PPMon basiert auf Pinpoint [20], dieser benutzt grobkörnige Nachverfolgung der Client-Anfragen, folgt ihrem Weg durch das System und bewertet ihre Komponenten- und Ressourcennutzung. Daraufhin vergleicht PPMon diese mit früheren Statistiken von „gutem“ Verhalten des Systems. Falls das Verhalten signifikant von der Statistik abweicht, wird dem Wiederherstellungs-Manager (RM) die möglicherweise versagende Komponente mitgeteilt.

Wiederherstellungsmanager (RM) und Widerherstellungsagent (RA) stehen außerhalb von JBoss, sie führen Automatische Wiederherstellung durch und verlangen nur, falls diese fehlschlägt nach dem Administrator.
Der RM lauscht an einem UDP Port auf Meldungen der Fehlermonitore, aus der so erhaltenen Information baut er eine Repräsentation der Ausfallvererbung in Form eines Graphen auf. Nachdem er den Graph auf den neusten Stand gebracht hat, prüft

er, welche Komponente beeinflusst wurde, aktiviert den Proxy und startet alle Knotenpunkte in dieser Komponente neu. Diese Neustarts führt der RA aus, der verantwortlich ist, Micro-Reboots auszuführen, indem er EJBs (Enterprise Java Beans) aus dem System entfernt und neu hinzufügt. Dieses Entfernen ist natürlich nur mit einem kleinen Teil der EJBs möglich, die keine Sitzungsdaten beinhalten.
Falls diese Neustart-Basierte Wiederherstellung fehlschlägt, kommt RR zu tragen. Der RA startet einzelne EJBs oder die ganze Anwendung neu. Der RM speichert vorherige Neustarts, im Falle eines wiederholten Fehlers wählt er die vorherig erfolgreiche Strategie. Da RM Komponenten aufgrund des Ausfall-Pfades neu startet, muss das System auf eine gespeicherte Repräsentation dessen zurückgreifen. Den Fehlerpfad von Menschen erstellen zu lassen ist unzuverlässig, daher wird dieser autonom erstellt, AFPI (Autonome Fehlerpfaderkennung) nutzt systematische Injektion von Java-Ausnahmen und stellt die festgestellten Abhängigkeiten im AFPI-Graph dar.

6 Selbst-Heilung auf Architekturebene

Softwarearchitekturen bieten ein hohes Abstraktionslevel um die Struktur, das Verhalten und die Haupteigenschaften einer Software zu repräsentieren [13].
Eventbasierte Softwarearchitekturen, ohne gemeinsame Speichernutzung (ohne dass verschiedene Anwendungen bzw. Prozesse auf den gleichen Speicherbereich zugreifen) eignen sich, um eine sehr wichtige Fähigkeit erweitert zu werden: die Selbst-Heilung. Aufgrund dessen, dass die einzelnen Komponenten nicht direkt aufeinander zugreifen dürfen, sind sie unabhängig von ihrer Lage in der Softwarearchitektur. Dadurch ist es möglich, Komponenten hinzuzufügen, zu entfernen oder zu ersetzen ohne Änderungen am ihrem Code vorzunehmen. Diese Selbstständigkeit ist nötig für Softwarereparatur ohne Voraussicht, die nicht extra in die Komponenten programmiert werden muss. Auch „Software von der Stange“ kann durch speziell programmierte Hüllen (Diese Hüllen verhalten sich wie ein Emulator, der der Anwendung ihre nötige Umgebung schafft und nach außen wie das lokale System arbeitet), in diese eventbasierten Systeme eingebunden werden.
Dynamische Reparaturen zur Laufzeit haben einige Voraussetzungen [14]:

- Die Fähigkeit die aktuelle Architektur zu beschreiben
- Die Fähigkeit eine beliebige Änderung der Architektur als Reparaturplan zu erstellen
- Die Fähigkeit das Ergebnis der Reparatur zu analysieren und zu validieren
- Die Fähigkeit den Reparaturplan auf einem laufenden System ohne Neustart auszuführen

Bevor eine Reparatur ausgeführt wird, wird eine Kopie der aktuellen Architektur gemacht, auf der die in xADL 2.0 beschriebenen Änderungen validiert werden.
Wird die Reparatur letztendlich ausgeführt, werden die Komponenten auf die Änderung vorbereitet:

1. Zu entfernenden Komponenten und Vermittlern wird die Möglichkeit gegeben Code auszuführen um Variablen zu entladen, Benachrichtigungen über den Austausch an andere Komponenten weiterzugeben, oder weitere Mitteilungen zu versenden.
2. Angrenzende Komponenten und Vermittler werden vorübergehend angehalten, um Benachrichtigungen an gerade hinzugefügte oder zu entfernende Komponenten zu verhindern.
3. Angrenzende Komponenten und Vermittler werden wieder gestartet, und neuen Komponenten wird die Möglichkeit gegeben erste Meldungen zu versenden

Trotz, dass diese Taktik nicht für jede Reparatur ausreichend ist, ist sie es für die meisten, derart können Reparaturen an der Architektur viel sicherer bewerkstelligt werden, als direkte Ausführung der Änderungen.

7 Checkpointing

Checkpointing [15] erlaubt es lang laufenden Programmen den aktuellen Status zu sichern um bei Neustarts oder Systemfehlern ohne Verzögerung und Datenverlust wieder zu einem funktionierenden Status zurückzukehren. Zu Checkpointing gibt es 4 Hauptansätze, wem die Implementierung zu überlassen ist:

1. Programmierer
2. Support im Compiler
3. Betriebssystem, der Programmierer bestimmt lediglich die Intervalle (System-Level CP)
4. Laufzeit Bibliothek (Benutzerdefiniertes CP)

Wenn dem Programmierer das Behandeln von Ausfällen überlassen wird, kann es sehr komplex und aufwendig werden. Einige Untersuchungen haben gezeigt, in einer Parallelen Anwendung Fehlertoleranz ohne Betriebssystemsupport einzubauen, kann bis zu 50% des Programmcodes umfassen, ganz abgesehen von dem wesentlich aufwendigeren Debugging. Das führt zu dem Schluss, dass entweder das Betriebssystem, oder die Laufzeit-Bibliothek Fehlertoleranz unterstützen sollten.

Im Compilerbasierten Ansatz ist die Platzierung der Checkpoints (CP) transparent für den Programmierer, der Compiler sorgt dafür, dass der CP an der bestmöglichen Stelle gesetzt wird und unnötige Teile des Speichers ausgelassen werden. Obwohl dies eine gute Möglichkeit des Checkpointing ist, mangelt es an Portierbarkeit, da nicht alle Compiler dieses Feature anbieten. Außerdem könnte der Einsatz bei verteilten oder parallelen Anwendungen problematisch sein, da der Compiler beim setzen des CP nicht den Status der aktuellen Prozesskommunikation bestimmen kann.

Der Checkpoint-Ansatz auf Betriebssystemebene hat einige Vorteile, so nimmt er dem Anwendungs-Programmierer die Arbeit ab, sich um CP zu kümmern, ein

weiterer Vorteil ist Transparenz. Aber dieser Ansatz hat auch Nachteile, die Anwendung wird vom Betriebssystem als Blackbox angesehen, es hat keinerlei Anhaltspunkte, welche Bereiche bei einer bestimmten Anwendung wichtig sind, es wird einfach der ganze Status gesichert, was zu großem Overhead führt.

Die Alternative zu den vorhergehenden Ansätzen ist, CP als Laufzeit-Bibliothek bereitzustellen, die Inhalte und Platzierung der Checkpoints werden vom Anwendungs-Programmierer festgelegt, für den Endbenutzer ist das Checkpointing nicht sichtbar.

Die beiden letzten Ansätze erscheinen am besten ausgereift, doch auch sie haben ihre Vor- und Nachteile.
Transparentes System-Level CP ist unkompliziert einzusetzen, da kein zusätzlicher Programmieraufwand nötig ist, außerdem können auch Anwendungen, die kein eingebautes CP haben widerherstellbar genutzt werden.
Benutzerdefiniertes CP vereinfacht die Speicherung des aktuellen Status, da der Anwendungs Programmierer genau den Zeitpunkt und die zu sichernden Daten festlegen kann, sodass eine sichere Widerherstellung auf den gespeicherten Status möglich ist. Außerdem wird hierdurch der CP Speicherbedarf stark verringert, da z.B. viele Daten einfacher beim Wiederherstellen neu berechnet werden können.

8 Selbst-Heilung: Wann ist ein Softwaresystem überhaupt krank?

Moderne Computersysteme [11] sind viel komplexer als die einfachen Programme, auf denen unsere Abhängigkeitsmodelle aufbauen, diese Modelle sind begründet auf präzisen Spezifikationen. Im Gegensatz zu diesen Annahmen variieren die Kriterien für annehmbares Verhalten des Systems von Zeit zu Zeit und Benutzer zu Benutzer. Aufbauend auf diesen Modellen sind Softwaresysteme oft anfällig und verwundbar für unerwartete Situationen außerdem sind sie schwer an veränderte Anforderungen anzupassen.
Traditionelle SH Modelle machen genaue Unterscheidungen zwischen Normalen, Defekten, und Zuständen mit geringer Leistung. In heutigen Systemen ist der Übergang zwischen diesen aber fließend, somit stellt sich natürlich die Frage, wann denn eine Heilung einzuleiten ist. Ein Ausweg daraus ist, durch Hintergrund-Aktivitäten ständig nach hoher Gesundheit des Systems zu streben, ob es nun defekt ist, oder nicht. Gleichgewichtssysteme reagieren im Gegensatz zu anderen nicht auf den direkten Fehlerzustand, sondern schon viel früher, bei einer Änderung. (z.B.: Die Iris des Auges, sie verkleinert sich bei veränderndem Lichteinfall, nicht erst wenn man schon nichts mehr sehen kann.)
Allgemeine Beispiele für Gleichgewichtssysteme beinhalten Garbage Collection im Hintergrund, die den unbenutzten Speicher wieder freigibt unabhängig davon ob noch genug verfügbar ist, oder nicht, sodass es erst gar nicht zu Engpässen kommt. Ein anderes Beispiel dafür ist das Paketrouting im Internet, das immer den besten Weg für die Pakete sucht, egal ob ein Knoten ausgefallen ist oder nicht.

9 Zusammenfassung und Ausblick

Selbst-Heilende Computersysteme - ein interessantes Feld. Wenn man die Entwicklung über die letzten Jahre beobachtet wird eines sofort klar, wir können es uns überhaupt nicht mehr leisten, dass Rechner das nicht können. ...„Noch 1990 hat eine EDV-Abteilung 80% seines Budgets für Technologie ausgegeben“...„Im Jahr 2000 waren es 50%, Heutzutage sind es nur noch 30-40%, der heute größte Teil geht in die Personalkosten“...[18]. Wenn es so weiterginge wie bisher, könnten sich Firmen keine neuen Rechnersysteme leisten, weil mit der Anschaffung ein Vielfaches an Personalkosten nötig wäre um die Systeme am laufen zu halten. Ganz abgesehen von der Tatsache, dass dies in vielen Firmen eine bewährte Praxis ist, unter dem Leitspruch „Never change a running system“ sind noch viele Proprietäre Systeme im Einsatz. Meiner Meinung nach ist dies ein weiterer Grund, Autonomic Computing und Self-Healing voranzutreiben, vielleicht wird es ja so möglich wieder in Computersysteme zu vertrauen, ohne große Angst, bei einer Umstellung gingen viele Daten verloren, auch da die Benutzer sich noch nicht mit dem neuen System auskennen und Fehler machen. Man muss sich jedoch die enormen Vorteile Selbstheilender Computersysteme klarmachen, statt sich um Fehler und Ausfälle zu kümmern könnten Mitarbeiter besser ihrem Job nachgehen oder andere sinnvolle Dinge tun.
Weitere Vorteile sieht man an mit Rechnern ausgestatteten Systemen, die fern von Menschen arbeiten, auf dem Mars, im Weltall, im Meer, „Intelligente“ Roboter eben, bei denen man natürlich, wie auch in Abschnitt 3 gezeigt, nie von absoluter Systemkorrektheit ausgehen kann. Bei diesen Systemen kann man nicht vorbeischauen und reparieren, im Fehlerfall bleibt ohne SH bestenfalls teurer Weltraumschrott. Dazu gibt es schon einige Implementierungen, wie den Watchdog in Echtzeitbetriebssystemen, die auch bei industrieller Automatisierung zum Einsatz kommen, die sich, selbstverständlich auch keine Ausfälle leisten kann.
Aber auch die neueren Ansätze haben großes Potential wie z.B. die Chameleon Architektur, die recht ausgereift und komplex erscheint. Sie arbeitet auf heterogenen Systemen im Netzwerk und unterstützt bereits viele der Anforderungen an ein SH-System. Sie kann im Fehlerfall freie Ressourcen über das Netzwerk nutzen, bringt eine Statussicherung mit und ist im Fehlerfall fähig einzelne Komponenten neu zu starten, ohne andere zu beeinflussen. Wie in vielen anderen Lösungen wird hier der Fehlerfall durch ein Lebenssignal der einzelnen Komponenten überwacht, kommt irgendwann kein Signal mehr, wird von einem Ausfall ausgegangen und neu gestartet. Was aber passiert wenn das Lebenssignal fälschlicherweise noch gesendet wird, obwohl die Komponente nicht mehr richtig funktioniert? Hierzu gibt Abschnitt 8 einen Lösungsvorschlag, der solche Fälle zwar nicht verhindern, wohl aber verringern kann. Der Wiederherstellungs-ARMOR muss dafür sorgen, dass die Komponente oder auch das gesamte System sich immer in einem „Gesunden“ Zustand befindet, um Ausfälle zu vermeiden.
JAGR geht ganz anders an das Erkennen von Systemfehlern heran, statt ein Lebenssignal zu erwarten wird hier die Prozess- und Komponentenkommunikation auf Anomalien überwacht. Die Autonome Fehlerpfaderkennung ist eine weitere Neuerung, die in JAGR implementiert ist, möglicherweise könnte sie als Vorbild für andere SH-Systeme dienen. JAGR ist speziell auf den Webserver JBoss

zugeschnitten, womit natürlich nur ein kleiner Teil der SH-Anwendungsmöglichkeiten abgedeckt ist. J2EE bietet für JAGR gute Vorraussetzungen seine Aufgabe zu erfüllen, in seinem Bereich tut JAGR sein Bestes, fraglich ist nur, ob diese Strategie auch auf allgemeinere Anwendungsfelder übertragbar ist.
Architekturbasiertes SH ist besonders für Hochverfügbarkeitssysteme mit Administratorzugriff geeignet, da der Administrator jeweils Veränderungen, wie Softwareupdates oder Architekturanpassungen vorher begutachten und noch ändern kann, obwohl die Veränderungen auch autonom übernommen werden könnten. Wie bei Chameleon besteht auch hier die Möglichkeit bisher nicht SH-Fähige Software zu integrieren, am besten passt natürlich die in Abschnitt 6 spezifizierte Software, aber durch mehr oder weniger Programmieraufwand lässt sich fast jede Software integrieren.
Besonders hervorzuheben bei diesem Ansatz ist natürlich, dass Updates während der Laufzeit eingespielt werden können und die Architektur sich darum kümmert, dass keine Fehler durch Zugriff auf nicht existierende Komponenten passieren.
Rekursive Neustartbarkeit oder Wiederherstellungsorientiertes Rechnen kann in geeigneten Systemen implementiert werden um mit Kommerzieller Hard- und Software zurechtzukommen, ohne auf das letzte Bugfix warten zu müssen mit dem ja doch nicht jeder Fehler behandelt werden kann (siehe Abschnitt 4). Dieser Ansatz ist teilweise vergleichbar mit Chameleon und Architekturbasiertem SH (ASH), auch hier sind die einzelnen Komponenten gegeneinander isoliert, Damit bei einem Absturz möglichst wenig „kaputt geht“. Im Gegensatz hierzu ist aber bei ASH nur eventbasierte Kommunikation möglich und dort dürfen Komponenten nicht wie bei RR voneinander abhängen, was auch bei RR kein gern gesehenes, aber mögliches Feature ist.
Checkpointing, ist eine der schon etwas älteren Ideen, worauf man aber achten muss, ist, dass man die Bedeutung nicht falsch versteht, mit Checkpointing ist die Sicherung des gesamten Systemstatus gemeint, d.h. wenn eine Anwendung abstürzt, oder neu gestartet wird, kann man durch Laden des CP wieder an der gespeicherten Stelle weiterarbeiten, da sowohl der Speicher, als auch die Prozesse gesichert werden. Im Gegensatz zum klassischen Checkpoint, wie ihn auch Windows XP benutzt, wenn man den Systemstatus sichert, hiermit ist nämlich nur die Festplatte gemeint, wichtige Daten die im Hauptspeicher liegen sind damit natürlich weg. Aber dafür ist dieser CP auch nicht gedacht, er ermöglicht es, Windows auf den letzten funktionierenden Status zurückzusetzen, so wird automatisch vor jeder Treiberinstallation ein solcher CP gemacht.
Zuletzt bleibt noch Zellbasiertes Programmieren, es hört sich viel versprechend an, nach dem Vorbild der Natur zu programmieren bzw. zu kompilieren, allerdings stellt sich die Frage ob es mit diesem Modell je so weit kommen kann.

9.1 Könnte man aus diesen Verschiedenen SH-Ansätzen denn nicht einen machen?

Bei Chameleon stellt sich diese Frage gar nicht mehr, es ist schon alles Integriert, abgesehen von der neuen Ansicht wann ein System überhaupt krank ist, doch auch das sollte darin zu integrieren sein.
JAGR ist ein Konzept mit Potential, jedoch gibt es kaum Ansätze um noch andere der hier vorgestellten SH Strategien zu integrieren. Bei JAGR sind noch Verbesserungen beim Neustart der EJBs nötig, um allgemein EJBs Neuzustarten ist Checkpointing nötig, um Sitzungsbezogene Daten nicht zu verlieren .
Architekturbasiertes SH (ASH) muss nicht mit RR kombiniert werden, da hier ja keine Komponenten voneinander abhängen und ASH seine eigene Wiederherstellungsstrategie hat. Aber System-Level Checkpointing und Ständige Heilung lassen sich sicher Integrieren um diese Strategie noch leistungsfähiger zu machen. Allerdings müsste die Architektur daraufhin angepasst werden um in den CPs die Prozesskommunikation zu speichern, was bei diesem Modell aber viel versprechend ist, da ASH seine Komponenten nicht als Black Box ansieht, sondern sich über die Kommunikation zwischen ihnen bewusst ist. Dafür kann sicher eine Möglichkeit gefunden werden, beispielsweise kann vor setzen eines CPs „Funkstille“ vereinbart werden um die Kommunikation danach erfolgreich wieder aufzunehmen, wie auch nach einem Neustart auf diesen CP. Die Ständige Heilung kann zu der Wiederherstellungsstrategie parallel betrieben werden, im Notfall hilft dann doch der Neustart, sofern es soweit kommt.
Auch RR könnte mit Ständiger Heilung kombiniert werden, System Level CP kann hier möglicherweise durch voneinander Abhängige Komponenten nicht eingesetzt werden, alle anderen CP Strategien müssten wieder gesondert implementiert werden und würden somit den Vorteil, Kommerzielle Software von der Stange einzusetzen zu Nichte machen. In RR könnte man zusätzlich die in Abschnitt 5 (JAGR) vorgestellte Fehlerpfaderkennung implementieren, um dem Menschen, der in komplexen Systemen sowieso nicht dazu geeignet ist, die Erstellung der Wiederherstellungsreihenfolge und der Abhängigkeiten abzunehmen.

Referenzen

[1] S. George, D. Evans, L. Davidson, „A Biologically Inspired Programming Model for Self-Healing Systems“ November 2002 Proceedings of the first workshop on self-healing Systems

[2] G. Homsy, T. F. Knight, Jr. Radhika Nagpal, „Amorphous Computing“, May 2000/Vol. 43, No. 5 Communications of the ACM

[3] R.K. Iyer, Z. Kalbarczyk, K. Whisnant, S. Bagchi, „A Flexible Software Architecture for High Availability Computing“ 1998, Proceedings of the 3rd IEEE Conference on High Assurance Systems Engineering

[4] S. Bagchi, K.Whisnant, Z. Kalarczyk, R.K. Iyer, „The Cameleon Infrastructure for Adaptive, Software Implemented Fault Tolerance“, 1999, IEEE Transactions on Parallel and Distributed Systems
[5] A. Fox, D. Patterson, „When Does Fast Recovery Trump High Reliability?
„, Stanford/Berkeley Recovery-Oriented Computing Project (ROC), Oktober 2002, „In Proceedings of the EASY 2002, San Jose, CA“
[6] A. Fox, „Toward Recovery-Oriented Computing“, August 2002, „28th International Conference on Very Large Data Bases“
[7] „Recovery Oriented Computing: Motivation, Definition, Techniques, and Case Studies“, 15.3.2002, „Computer Science Technical Report UCB/CSD-02-1175, U.C. Berkeley“
[8] G. Candea, A.Fox, „ Reboot-Based High Availability“, Oktober 2000, „In Symposium for Operating System Design and Implementation (OSDI), San Diego, CA“
[9] G. Candea, A.Fox, „Recursive Restartability: Turning the Reboot Sledgehammer into a Scalpel“, Mai 2001, „Proc. 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII), Schloss Elmau, Germany“
[10] G. Candea, J. Cutler, A. Fox, R. Doshi, P. Garg, R. Gowda, „Reducing Recovery Time in a Small Recursively Restartable System“, Juni 2002, „International Conference on Dependable Systems and Networks (DSN-2002), Washington, D.C.“
[11] M. Shaw, „Self-Healing: Softening Precision to Avoid Brittleness“, November 2002, „Proceedings of the First ACM SIGSOFT Workshop on Self-Healing Systems (WOSS '02) Charleston, South Carolina“
[12] M.Schmerl, D.Garlan, „Exploiting Architectural Design Knowledge to Support Self-Repairing Systems“, Juli 2002, „The 14th International Conference on Software Engineering and Knowledge Engineering, Ischia, Italy“
[13] M. Mikic-Rakic, N. Metha, N. Medvidovic, „Architectural Style Requirements for Self-Healing Systems“, 2002, „ACM SIGSOFT Workshop on Self-Healing Systems (WOSS'02)“
[14] E.M. Dashofy, A. Van der Hoek, R.N. Taylor, „Towards Architecture-Based Self-Healing Systems“, 2002, „ACM SIGSOFT Workshop on Self-Healing Systems (WOSS'02)“
[15] L.M. Silva, J.G. Silva, „System-Level versus User-Defined Checkpointing“, Oktober 1998, „In Proceedings of the Seventeenth Symposium on Reliable Distributed Systems“
[16] Autonomic Computing: „IBM’s Perspective on the State of Information Technology“
[17] H. Wörn, Brinkschulte: „Vorlesung Echtzeitsysteme SS2003 Universität Karlsruhe TH“
[18] Alan Ganek, IBM’s pice president of autonomic computing, 2003
[19]G. Candea, P. Keyani, E. Kiciman, S. Zhang, and A. Fox, „Jagr: An autonomous self-recovering application server“, Juni 03, „In Proc. 5th International Workshop on Active Middleware Services, Seattle, WA“
[20] M. Chen, E. Kiciman, E. Fratkin, E. Brewer, A. Fox, „ Pinpoint: Problem determination in large, dynamic Internet Services“, Juni 2002, „In Proc. International Conference on Dependable Systems and Networks, Washington D.C.“

ROC – Recovery Oriented Computing

Michael Kohlbecker

Zusammenfassung. Wegen der schlechten Erreichbarkeit heutiger Computersysteme wird in diesem Paper ein Versuch aufgezeigt, Erreichbarkeit durch das schnelle Beheben von Fehlern und nicht wie traditionell durch Fehlervermeidung, zu garantieren. Diese Idee wird allgemein als ROC – Recovery Oriented Computing - bezeichnet. Nach einer Einführung von ROC wird auf das Testen von Recovery-Mechanismen und auf automatische Diagnoseverfahren zur Ursachenforschung von auftretenden Fehlern eingegangen.

1 Einleitung

Recovery Oriented Computing (ROC) baut auf der Grundlage auf, dass Hardware-, Software- und Anwenderfehler Fakten sind, die behandelt werden müssen, und keine Probleme, die gelöst werden können. Dadurch, dass es Fehler akzeptiert, werden diese schneller behoben und damit die Erreichbarkeit verbessert. Da eine Hauptaufgabe heutiger Systemadministratoren darin besteht, durch Fehler ausgefallene Systeme zu reparieren, senkt ROC auch die Betriebskosten.

1.1 Vergangenheit

In den letzten 20 Jahren hat man sich beim Erforschen neuer Techniken hauptsächlich auf das Verbessern der Performance konzentriert. Ein Grund dafür war, dass die Performance anhand von Benchmarks gemessen werden kann und man dadurch den Forschritt bzw. die Verbesserungen bzgl. älterer Systeme aufzeigen kann.

Durch diese Einschränkung bei der Entwicklung wurden andere wichtige Aspekte außer Acht gelassen. Einer davon waren die Betriebskosten. Betriebskosten sind heute 5 bis 10mal höher als die Anschaffungskosten von Hardware und Software [2]. Diese Tatsache lässt sich durch Moore's Law, Open Source Software und den steigenden Gehältern der Administratoren erklären.

Ein anderer Aspekt war die Erreichbarkeit von Systemen. Zwar hat man gedacht, dieses Problem wäre gelöst, doch das war nur ein Trugschluss. Die Idee erschien einfach: man kauft einen Fehler-toleranten Server, benutzt ein ständig erreichbares Datenbanksystem und schließt das ganze in einen Raum ein, in den nur hoch qualifizierte Administratoren Zutritt haben. Doch die heutige Computerwelt hat sich mehr zu einer Verbindung aus verschiedenen Hardware- und Softwarekomponenten, in der die Daten und Anfragen für den Endbenutzer transparent werden, entwickelt. In

einem solchem System, indem das schwächste Glied der Kette über die Zuverlässigkeit entscheidet, genügt es nicht einen ausfallsicheren Server zu haben.

Doch wie wichtig Erreichbarkeit auch bei solchen Internetsystemen ist, wird deutlich, wenn man sich den Schaden bei einem Ausfall anschaut. So liegt der Schaden pro Stunde bei Hunderttausenden von Euro bei Einzelhändlern (z.B. Amazon, Ebay) oder sogar Millionen von Euro (z.B.Maklern) [2].

1.2 Heute

Obwohl einige Firmen bei ihren Systemen 99,999% Erreichbarkeit versprechen, berichteten - laut einem Artikel von InterWeek aus dem Jahr 2000 - 65% der IT-Manager, dass ihre Internetseiten mindestens einen Ausfall in den letzten sechs Monaten zu beklagen hatten, 25% beklagten sogar mehr als drei im gleichen Zeitraum [1].

Diese Fehleinschätzung der Firmen resultiert aus einigen Annahmen, die man als Grundlage bei der Entwicklung der Systeme genommen hat [3]:

1. Fehlerraten von Software und Hardware sind niedrig und werden ständig verbessert
2. Menschen machen keine Fehler während der Instandsetzung
3. Systeme können zuverlässig gemacht werden und auftretende Fehler können vorhergesagt werden

Nimmt man diese drei Annahmen zusammen, sieht man, dass diese in die Richtung gehen, Fehler zu vermeiden und dadurch die Erreichbarkeit zu verbessern. Da diese Annahmen, wie später noch gezeigt wird, nicht der Realität entsprechen, hat man sich einen neuen Weg überlegt, eine hohe Erreichbarkeit zu garantieren.

Die Überlegungen basieren auf folgenden, geänderten Annahmen [3]:

1. Hardware und Software werden immer die Ursache für Fehler sein
2. Fehler können nicht im Voraus erkannt werden
3. Menschen machen Fehler

Diese Annahmen resultieren in einer Denkweise, die die Wichtigkeit des Aufspürens von Fehlern und deren Behebung, sowie dem Bereitstellen menschgerechter Reparaturprozeduren, berücksichtigt.

1.3 Hardware- und Softwarefehler sind unvermeidlich

In der hier betrachteten Internetservicewelt ist Funktionalität wichtig. Funktionalität ändert sich allerdings wöchentlich; und mit ihr die Software. Damit ist klar, dass fehlerfreie Software mit den traditionellen Programmiertechniken, die eine hohe Erreichbarkeit und wenige Fehler garantieren, mit annehmbarem Aufwand nicht möglich ist. Außerdem haben viele Angestellte der Internetportale nicht das entsprechende Know-How, um qualitativ hochwertige Software zu entwickeln [1].

Auch auf Seiten der Hardware sieht es nicht besser aus. Da viele Anbieter oft an der Gewinngrenze operieren, können sich die meisten keine teure, fehlerresistente Hardware leisten. Sogar die einfachsten Fehler-Erkennungsmechanismen werden aus Kostengründen weggelassen. So haben die meisten Systeme nicht einmal ECC-Memory. Auch werden billigere Desktop-Komponenten (IDE-Disk anstatt SCSI) eingesetzt. Diese Sparmaßnehmen resultieren in höheren Hardwarefehlerraten [1]. Google z.B. hat eine Ausfallquote der Server von 2-3% pro Jahr. Davon resultiert ein Drittel aus Speicherfehlern, die durch den Einsatz von ECC-Memory verhindert werden könnten [1]. Auch der Mehrpreis von ECC-Memory im Vergleich zu normalem ist niedriger als die Kosten eines Ausfalls (ECC-Memory zur Zeit (Sommer 2003) etwa 50 Euro teurer bei einem 512 MB Modul).

Wenn man sich nun noch überlegt, dass Google aus über 8000 Servern besteht, wird klar, dass schon eine Fehlerrate von 1% zu mehr als einem Serverausfall pro Woche führt. Jetzt ist noch zu bedenken, dass diese Serveranzahl ständig steigt, und mit ihr die Ausfälle pro Woche.

1.4 Menschliche Fehler sind unvermeidlich

Menschen machen Fehler. Das ist eigentlich eine Tatsache, derer sich jeder bewusst ist. Und diese Fehler nehmen in Stresssituationen deutlich zu. Psychologen haben herausgefunden, dass die Fehlerraten von Menschen in Stresssituationen auf 10% bis 100% ansteigen können [1].

In den letzten Jahrzehnten hat sich die Quote der vom Menschen verursachten Fehler nicht signifikant verbessert. So war in den 1970er der Anwender für 50-70% der Fehler in elektronischen Systemen, 20-53% in Raketensystemen und 60-70% von Fugzeugunglücken verantwortlich. In der Mitte der 1980er waren es 42% der Fehler und 1993 über 50% in einem VAX System [1]. Abbildung 1 zeigt das Verhältnis bei der U.S Public Switched Telephone Network und drei Internetseiten aus dem Jahr 2000/2001.

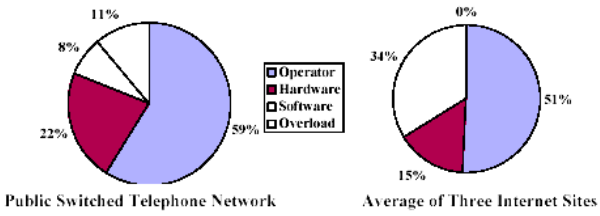


Abb. 1. Fehleranteile

Wie man sieht ist auch hier der menschliche Faktor der Größte und auf keinen Fall vernachlässigbar gering.

Auch der Versuch, menschliche Einflüsse durch Automation zu verringern, ist fehlgeschlagen. Das liegt vor allem daran, dass nur die leichten Benutzeranfragen vom System erledigt werden können. Die komplexen Aufgaben muss weiterhin der

Benutzer abwickeln. Menschen, die bei den einfachen Aufgaben schon Fehler machen, sind bei solchen komplexen überfordert. Die Ironie bei der Automation ist es, dass es dem Anwender das Training entzieht, wodurch er fehleranfälliger wird [2].

1.5 Unvorhersehbare Fehler sind unvermeidlich

Auch die Tatsache, dass unvorhersehbare Fehler nicht zu vermeiden sind, leuchtet ein. Allein wenn man sich eine Situation überlegt, in der der Mensch mit dem System interagiert. Menschen ignorieren oft Warnungen und Fehlermeldungen, wenn diese nicht ihrem Verständnis, wie das System reagieren sollte, entsprechen [1].

Weiter könnte man argumentieren, dass in der Praxis Fehler auftreten, die in der Theorie als unmöglich bezeichnet wurden [1].

2 ROC

„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." - Shimon Peres (Peres's Law)

Die obigen Annahmen und Peres's Law bilden die Grundlagen für eine neue Philosophie, die Erreichbarkeit heutiger Systeme zu verbessern. Diese wird „Recovery Oriented Computing" (ROC) genannt.

Das Beheben von auftretenden Fehlern wird dabei durch verschiedene Mechanismen durchgeführt – von einfachen wie ein Neustart, Redundanz und Isolation bis zu komplexen wie dem Wiederherstellen von einem Systemzustand via Backup, Checkpoints oder Logs.

Diese Themen werden hier nicht weiter behandelt. Allerdings findet sich dazu eine große Anzahl von Papers [9].

Zusätzlich muss ein recovery-oriented Systemdesign weiter gehen, als nur diese Mechanismen zur Verfügung zu stellen. Schließlich haben heute fast alle Systeme schon Recovery-Mechanismen in der Form von Backups und dem Loggen von Systemzuständen; doch dies allein hat das Problem der Erreichbarkeit noch nicht gelöst. Um ein echtes recovery-oriented Design zu haben, müssen die Mechanismen als Hauptbestandteile implementiert werden und in ein System eingebunden werden, dass ihre Steuerung übernimmt und ihre Effektivität garantiert.

Dieses System muss verschiedene Dinge garantieren. Zuerst muss es sicherstellen, dass Probleme und Fehler schnell gefunden werden, damit diese behoben werden können, bevor sie sich fortpflanzen. Als Teil dieser Erkennung, sollte ein recovery-oriented System versuchen, versteckte Fehler zu finden bevor diese aufgerufen werden.

Später wird noch gezeigt, dass diese versteckten Fehler für viele schwerwiegendere Fehler verantwortlich sind. Das passiert dann, wenn einige dieser versteckten Fehler in einer bestimmten Reihenfolge auftreten bzw. aktiviert werden.

Weiterhin sollte ein recovery-oriented System Mechanismen bereitstellen um die Ursache der Fehler aufzuspüren. Dadurch wird eine schnelle Wiederherstellung des Systems garantiert. Mehr dazu wird in den folgenden Abschnitten erläutert.

Zusätzlich zu der schnellen Fehleraufdeckung muss ein ROC System auch sicherstellen, dass seine Mechanismen vertrauenswürdig und fehlerfrei sind, egal ob sie komplett automatisch ablaufen, oder mit Hilfe des Menschen. Das ist heute noch ein großes Problem, da der Code von Recovery-Mechanismen schwer zu testen ist. Die Lösung dieses Problems ist das regelmäßige Testen dieser Mechanismen während dem normalen Betrieb. Das heißt ein ROC-System das für die Produktion verantwortlich ist, wird während dem laufenden Produktionsbetrieb getestet. Dabei spielt keine Rolle, ob die Mechanismen menschliche Interaktionen benötigen oder nicht.

Ein weiterer Schlüssel für ein funktionierendes ROC-System ist, dass das System Fehler, die während dem Beheben von Fehlern auftreten, ebenfalls behandelt. In großen Systemen, wie sie heute existieren, kann dieser „doppelte Fehler“ häufig auftreten. Außerdem sind auch Administratoren in das Beheben von Fehlern mit eingebunden.

Zum Schluss bleibt noch festzuhalten, dass ROC keineswegs traditionelle Fehlerbehebung ausschließt. Die traditionellen Mechanismen garantieren schließlich, dass ein Fehler seltener auftritt. Das bedeutet dann in Verbindung mit ROC, dass die Recovery-Mechanismen seltener benötigt werden.

3 Online Verification of Recovery Mechanisms

Ein ROC-System hängt von der Qualität seiner Recovery-Mechanismen ab. Nur wenn diese schnell, effizient und zuverlässig arbeiten, ist auch das Recovery-System schnell, effizient und zuverlässig.

Die Wichtigkeit einer korrekten Funktion dieser Mechanismen, wird an dem folgenden Beispiel erläutern.

3.1 TMI – Three Mile Island

Charles Perrow hat Unglücke, wie auch das von Three Mile Island, einem Atomkraftwerk in Pennsylvania, analysiert [8]. Um Unglücke zu verhindern sind Reaktoren durch mehrere Schichten von redundanten Systemen abgesichert.

Reaktoren sind große, komplexe, eng verschlungene Systeme mit vielen Interaktionen. Deshalb ist es für Benutzer schwer, den Zustand des Systems zu erkennen oder die Auswirkungen ihrer Aktionen vorauszusehen. Außerdem gibt es Fehler in der Implementierung und in Notfallsystemen die die Situation erschweren. Zusätzlich können vermehrt Fehler auftreten, die von Computerspezialisten als statistisch unmöglich bezeichnet wurden. Aber es gibt auch versteckte Fehler die sich im System sammeln und auf ein Ereignis warten um aktiviert zu werden.

Zusätzlich sind Notfallsysteme oft mangelhaft implementiert bzw. gewartet. Da sie im normalen Betrieb nicht benötigt werden, testet sie nur ein Notfall; und versteckte Fehler machen sie unbrauchbar.

Im Fall des TMI hatten zwei Notfall-Bewässerungssysteme ihre Ventile fälschlicherweise geschlossen. Dadurch konnte kein Kühlwasser in den Reaktor gelangen. Als der Notfall eintrat, versagten diese beiden redundanten Notfallsysteme.

Letztendlich wurde der Reaktor doch noch gekühlt und die Zementeinfassung war als letzter Schutz auch noch da, aber durch den Ausfall dieser Notfallsysteme wurde der Kern zerstört.

3.2 Folgerung

Wie man an diesem Beispiel gesehen hat, ist es wichtig, Notfallsysteme bzw. Recovery-Mechanismen regelmäßig zu testen. Auch ist das Entfernen von versteckten Fehlern essenziell.

Beim Testen der Mechanismen sollten zufällige oder kontrollierte Testdaten benutzt werden. Dabei werden die Fehler direkt in das System als Störung eingepflanzt. Diese Art der Fehlereinpflanzung (Fault Injection) dient der Simulation eigentlicher Fehler. Alle diese Tests müssen natürlich auch „Online“, d.h. in der eigentlichen Arbeitsumgebung des Systems ausgeführt werden.

3.3 Beispiel: FIG

3.3.1 Einführung

Momentan läuft FIG (Fault Injection in glibc) auf Unix Systemen. Dabei setzt es eine Bibliothek zwischen die Applikation und anderen Bibliotheken ein. Diese eingesetzte Bibliothek fängt Aufrufe zwischen der Applikation und dem System ab. Wenn ein Aufruf abgefangen wurde entscheidet FIG aufgrund seiner Testdirektiven, ob der Aufruf normal durchgelassen wird oder ob ein Fehler zurückgeliefert wird, welcher ein Fehler im Betriebssystem simuliert. Die Testdirektiven können vom Benutzer verändert oder neue hinzugefügt werden. Manche Aufrufe die von FIG abgefangen werden, benötigen auch eine Behandlung durch den Benutzer. Mehr dazu aber später.

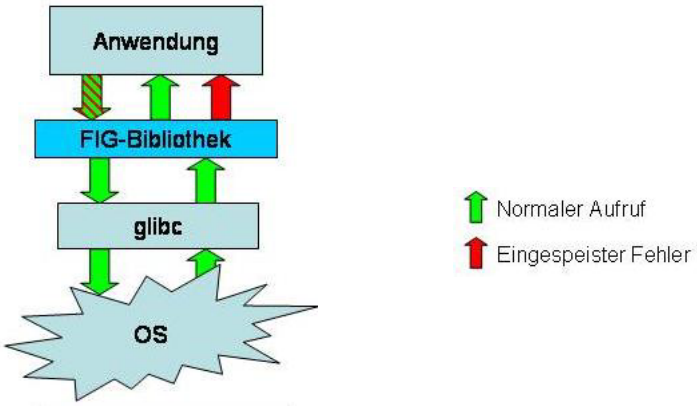


Abb. 2. Implementierung von FIG

FIG ist einfach an- bzw. auszuschalten. Es verursacht einen minimalen Overhead während dem Betrieb und die Installation lässt sich auch einfach bewältigen. Es kann ein einzelnes Programm oder eine Gruppe von Programmen testen. Außerdem kann es Fehler in einzelnen Komponenten oder einen systemweiten Ausfall simulieren.

Wenn FIG zum Testen von Software eingesetzt wird, kann diese durch das Beseitigen der von FIG gefundenen Fehlern und Schwachstellen robuster und widerstandsfähiger gemacht werden. Zusätzlich werden durch FIG Fehler aufgedeckt, die bei herkömmlichen Testprozeduren nicht gefunden werden. FIG setzt voraus, dass das System auf dem es eingesetzt wird, schon Recovery-Mechanismen besitzt, welche dann die auftretenden Fehler beheben.

3.3.2 Implementierung

Architektur

Da FIG die Systemaufrufe zwischen einer Applikation und dem Betriebssystem abfangen muss, wird eine Methode gesucht, die die FIG-Bibliothek zwischen der Applikation und dem Betriebssystem implementiert. Im Fall von glibc Aufrufen ist diese das Interface zwischen der Applikation und der glibc Laufzeitbibliothek (shared libraby).

UNIX stellt eine Methode zur Verfügung, mit der man Code in dieses Interface einfügen kann. Indem die LD_PRELOAD Umgebungsvariable auf die libfig.so Bibliothek gesetzt wird - das ist die Bibliothek von FIG - setzt man diese an den Anfang der Bibliothekslisten. Da Symbole, die am Anfang der Bibliotheksliste stehen gleichnamige Einträge weiter hinten überschreiben, hat FIG die glibc Routinen überschreiben.

Diese Art der Anbindung ist einfach umzusetzen. Deshalb kann FIG an jede andere Laufzeitbibliothek, nicht nur glibc, angehängt werden.

Da FIG die glibc Symbole überschreibt, können keine direkten Aufrufe von libfig.so nach glibc gemacht werden, um die ursprünglichen Bibliotheksfunktionen auszuführen. Es wird eine der folgenden Methoden benutzt, um diese Einschränkung zu umgehen. Viele glibc Symbole haben mehrere Definitionen: zwei Symbole, die auf den gleichen Code zeigen. In diesem Fall nimmt man die zweite, nicht überschriebene, Funktion.

Für Bibliotheksfunktionen ohne eine solche zweite Definition nimmt man die „dlsym()“ Routine um den Code der Funktion zu lokalisieren und ruft ihn indirekt, durch den zurückgegeben Funktionspointer, auf.

Automatische Erzeugung von Testdirektiven in der FIG-Bibliothek

Um den Umgang mit FIG zu erleichtern wurde ein Mechanismus integriert, der automatisch Aufrufe in der FIG-Bibliothek erzeugt. Dies erleichtert es neue Funktionen zu FIG hinzuzufügen. Der Entwickler fügt ein paar Dinge, wie z.B. den Namen der Funktion, das Symbol sowie die Namen und Typen der Parameter, zu einer Liste von Routinen hinzu. Als nächstes generiert ein AWK Script den Quellcode der Aufrufe, welche dann in eine angepasste FIG-Bibliothek aufgenommen werden können.

Nur ein paar Aufrufe können diesen automatischen Mechanismus nicht benutzen. „malloc()“ z.B. interagiert mit der FIG-Kontrolllogik und muss manuell behandelt werden.

Das FIG-Interface

FIG besteht aus zwei wesentlichen Bestandteilen: der libfig.so Bibliothek und der „fig“ Kommandozeile. Die libfig.so Bibliothek implementiert das Abfangen von Systemaufrufen und das Einpflanzen von Fehlern, wie es oben beschrieben wurde. „fig“ setzt die Umgebung so, wie libfig.so es haben will und führt das Programm aus, das getestet werden soll. Nach Beenden des Programms gibt „fig“ Timing Informationen aus.

Auf „fig“ wird an dieser Stelle nicht näher eingegangen, da diese Kommandozeile darüber hinaus lediglich der einfacheren Konfiguration des FIG-Tools dient.

Overhead

Da es eine Aufgabe von FIG ist, Fehlereinpflanzungen in laufenden Systemen zu machen, ist es wichtig, seine Auswirkungen auf die Performance zu minimieren. Sogar während des Entwicklungsprozesses werden schnellere Tools bevorzugt.

	User Time	System Time	Real Time	Overhead
ohne FIG	12.11	21.15	33.46	
FIG: kein Loggen	13.20	20.87	34.28	2.5%
FIG: Loggen ohne Zeitstempel	24.29	23. Feb	47.83	42.9%
FIG: Loggen mit Zeitstempel	36.66	24.25	61.74	84.5%
strace	65.06	47.14	112.85	237.3%

Abb. 3. Overhead

Wie aus Abbildung 3 ersichtlich wird, stellt FIG eine Reihe von Optionen zur Verfügung, mit denen man einstellen kann, wie viel geloggt werden soll – und dies hat direkten Einfluss auf den Overhead. Mit dem niedrigsten Loglevel beträgt der Overhead lediglich 2,5%. Da es wenig Sinn macht, jeden Bibliotheksaufruf mitzuloggen, ist dieser Overhead repräsentativ für den Einsatz für FIG im laufenden Betrieb.

Entwickler dagegen werden wahrscheinlich mehr Informationen benötigen. FIG kann sowohl alle glibc Aufrufe als auch nur solche mit einem Fehler mitloggen (egal ob der Fehler natürlich aufgetreten ist oder eingepflanzt wurde). Wenn alles mitgeloggt wird liegt der Overhead bei 43%. Wenn der Benutzer auch noch den Zeitstempel von jedem Log haben will, verdoppelt sich der Overhead auf 85%.

Aber selbst wenn alles geloggt wird und die Zeitstempeloption eingeschaltet ist, verursacht FIG immer noch deutlich weniger Overhead als z.B. das „strace“ Tool. Obwohl „strace“ nicht die gleichen Funktionen wie FIG hat, speichert es doch eine ähnliche Vielfalt von Aufrufen, aber verpasst den Logs keinen Zeitstempel. Es verursacht also fünfmal soviel Overhead als FIG.

3.3.3 Test Methode

Der erste Test von FIG bestand darin, Fehler in UNIX Programme einzupflanzen. Das Ziel war es viele verschiedene Programme zu nehmen, darunter einfache Befehlszeilenprogramme, Desktopapplikationen und zuverlässige Server. Alle Applikationen wurden auf einer Workstation mit Red Hat Linux (kernel 2.4) getestet. Es wurde überprüft wie die einzelnen Programme auf Fehler von malloc(), read() und write() reagierten. Bei manchen Programmen wurden auch die open(), close() und select() Aufrufe, soweit vorhanden, getestet. Diese Aufrufe wurden deshalb ausgewählt, weil sie von den getesteten Programmen am häufigsten benutzt werden.

Für jeden Systemaufruf wurde die Art und die Häufigkeit der Fehler erzeugt, welche als realistisch für ein stark beanspruchtes System mit Hardware- und Softwarefehlern angenommen wurden. Die Fehler bestanden aus Ressourcenknappheit (ENOMEM – nicht genug Speicher, ENOSPC – nicht genug Festplattenspeicher), Laufwerksfehlern (EIO – I/O Fehler) und solchen, die während dem normalen Betrieb auftreten können, aber häufiger sind, wenn das System ausgelastet ist (EINTR – Systemaufruf wurde unterbrochen).

Die getesteten Programme waren die folgenden:

1. ls
2. GNU Emacs 20.7.1
3. Apache 1.3.22
4. Berkeley DB 4.0.14
5. Netscape Communicator 4.76
6. MySQL Server 3.23.36

3.3.4 Ergebnisse

Die Ergebnisse der Tests findet man in Abbildung 4 wieder.

	read()		**write()**		**select()**	**malloc()**
	EINTR	EIO	ENOSPC	EIO	ENOMEM	ENOMEM
Emacs - ohne X	o.k.	exit	warn	warn	o.k.	crash
Emacs - mit X	o.k.	crash	o.k.	crash	crash/exit	crash
Netscape	warn	exit	exit	exit	n/a	exit
Berkeley DB - mit Xact	retry	detected	Xact abort	Xact abort	n/a	Xact abort
Berkeley DB - ohne Xact	retry	detected	data loss	data loss	n/a	detected, or data loss
MySQL	Xact abort	retry, warn	Xact abort	Xact abort	retry	restart process
Apache	o.k.	req dropped	req dropped	req dropped	o.k.	n/a

Abb. 4. Testergebnisse

Wie man sehen kann hat Emacs deutlich besser ohne X abgeschnitten als mit: EIO und ENOMEM Fehler verursachten Abstürze mit X. Netscape dagegen hat sich bei EIO oder ENOSPC normal beendet, aber hat keine Warnung ausgegeben und hat bei EINTR den Seitenaufbau unterbrochen. Allgemein gesehen zeigte Netscape das fragwürdigste Verhalten. Berkeley DB ist im Geschäftsmodus nur an Speicherfehlern gescheitert. Im anderen Modus konnte es bei Schreibfehlern zu Datenverlust führen. MySQL und Apache waren die Besten in diesem Test.

3.3.5 Fazit

Die Ergebnisse von FIG waren, dass sogar zuverlässige Programme schlechte Recovery-Mechanismen haben. Außerdem kann die zukünftige Programmentwicklung von FIG profitieren.

3.4 Folgerung

Wie man am Beispiel von FIG gesehen hat, ist es nicht schwierig, Recovery-Mechanismen durch Fehlereinpflanzung zu testen bzw. das Verhalten des System oder einzelner Applikationen im Falle eines Fehlers heraus zu finden. Auch bietet die Fehlereinpflanzung die Möglichkeit, Administratoren oder Benutzer an den Umgang mit Fehlern zu gewöhnen. Dadurch wird Ihnen die Möglichkeit gegeben, Modelle für die Fehlerbehebung zu erstellen, welche Ihnen helfen, weniger Fehler bei der Beseitigung der selbigen zu begehen [2].

4 Integrated Diagnostic Support

Damit ROC erfolgreich sein kann, muss die Behebung von Fehlern schnell und effizient erfolgen. Das setzt voraus, dass das ROC System das Vorhandensein von Fehlern schnell entdeckt und die Ursache dieser findet, damit diese schnell beseitigt werden kann. Außerdem müssen versteckte Fehler aufgedeckt werden, bevor diese aktiv werden und einen Crash verursachen können.

Diese Ziele können durch Integrated Diagnostic Supprt in der Form von Selbsttests und automatischer Ursachenforschung realisiert werden. Alle Module in einem ROC System sollten Selbsttests durchführen und sollten das Verhalten aller Komponenten, von denen sie abhängen bzw. auf die sie aufbauen, kennen. Diese Tests müssen wiederum „Online“, d.h. in der normalen Umgebung bzw. bei der normalen Arbeitsweise des Systems erfolgen. Beim Testen werden, ähnlich der Fehlereinpflanzung, Eingaben, auch falsche, in das System eingespeist, um die korrekte Funktionsweise nachzuprüfen. Neben den Selbsttests sollten die Komponenten zusammenarbeiten um Abhängigkeiten zwischen Modulen, Ressourcen und Benutzeranfragen herauszufinden, da diese Abhängigkeiten wichtige Informationen für die automatische Fehlerursachenanalyse und die menschliche Diagnose enthalten.

4.1 Beispiel: Pinpoint

4.1.1 Hintergrund

Momentan aktuelle Analysetechniken von Fehlerursachen benutzen Annäherungen, die die Komplexität von großen Systemen nicht ausreichend erfassen. Außerdem benötigen sie umfassende Informationen über das System. Die meisten

Analysetechniken von Fehlerursachen, auch Ereigniszuordnungssysteme, basieren auf statischen Abhängigkeitsmodellen die die Abhängigkeiten der Hardware- und Softwarekomponenten im System beschreiben. Diese Abhängigkeiten werden benutzt um festzustellen, welche Komponenten die Ursache für ein Problem sein könnten.

Ein großes Problem dieser traditionellen Abhängigkeitsmodelle ist es, ein genaues Modell von sich ständig verändernden Internetservices zu erstellen. Ein weiteres Problem ist, dass sie nur ein logisches System modellieren und nicht zwischen redundanten Komponenten unterscheiden können. Da aber Internetservices viele redundante Komponenten haben, ist es wichtig zu wissen, welche Instanz einer Komponente den Fehler verursacht.

4.1.2 Das Folgen des Pfades

Einem richtigen Befehl auf seinem Weg durch das System zu folgen ermöglicht die Problembehandlung auch in dynamischen Systemen, in denen traditionelle Abhängigkeitsmodelle nicht funktionieren. Dieses Verfolgen eines Befehls ermöglicht es zwischen den einzelnen Instanzen einer Komponente zu unterscheiden.

Wenn eine Anfrage an das System fehlschlägt, sieht man anhand des Pfades, welche Komponenten an diesem Fehler beteiligt sind. Dabei wird auch angenommen, dass diese Komponenten den Fehler verursachen. Wenn man jetzt analysiert, welche Komponenten bei den fehlgeschlagenen, aber nicht bei den erfolgreichen, Anfragen beteiligt waren, findet man heraus welche Komponenten Fehler verursachen. Dabei findet man einzelne fehlerhafte Komponenten oder zusammengehörige Gruppen.

Für Pinpoint werden nun zwei Annahmen gemacht. Erstens muss das System Komponenten in verschiedenen Kombinationen zusammen benutzen. Wenn zwei Komponenten immer zusammen benutzt werden würden, könnte man bei einem Fehler nicht feststellen, welche der Beiden dafür verantwortlich wäre. Zweitens muss man davon ausgehen, dass Anfragen nicht wegen der Aktivität einer anderen Anfrage fehlschlagen. Diese beiden Annahmen sind zutreffend für heutige Internetservices, da Serviceanfragen, wegen der Beschaffenheit von HTTP, dazu tendieren, unanhängig zu sein [5].

4.1.3 Pinpoint - Umgebung

Um die oben geäußerte Theorie in die Tat umzusetzen, wurde Pinpoint, eine Umgebung um Probleme in Internetservices zu finden, implementiert. Abbildung 5 zeigt diese Umgebung.

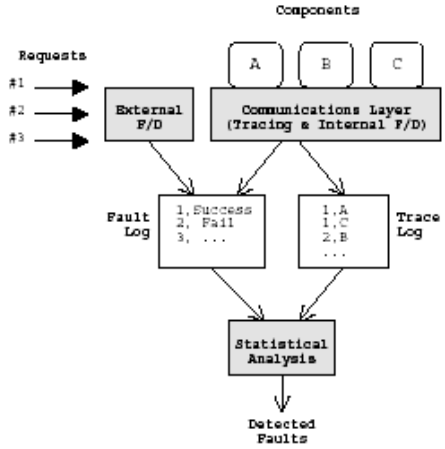


Abb. 5. Pinpoint Umgebung

Diese Umgebung sieht drei verschiedene Funktionalitäten vor, um das Finden der Fehlerursache zu erleichtern:

Anfragepfad des Klienten

Wenn eine Anfrage eines Klienten durch das System läuft, sind alle Komponenten interessant, die benutzt werden. Dabei interessiert man sich sowohl für die Maschinen und Cluster die benutzt werden, als auch im spezielleren für die einzelnen Softwarekomponenten, Versionen der Komponenten und sogar vereinzelt für Dateien. Das Ziel ist es, so viele Faktoren wie möglich zwischen einer erfolgreichen und einer fehlgeschlagenen Anfrage zu sammeln.

Bei einer Anfrage eines Klienten ist das pfadfolgende Untersystem verantwortlich, der Anfrage eine ID zuzuteilen und ihren Weg zu verfolgen. Um eine unnötige Komplexität zu verhindern, werden einfache Logs in der Form <Anfrage_ID, Komponenten_ID> erstellt. Diese Logs werden in eine Liste, welche speziell für diesen einen Aufruf da ist, aufgenommen.

Indem die Middleware verändert wird, kann die ID von jeder Anfrage, die an einer speziellen Komponente ankommt, aufgezeichnet werden, ohne dass die Applikation bekannt ist oder die Komponenten verändert werden. Wenn eine Komponente eine Andere aufruft, gibt die Middleware nur die ID und die Aufrufdaten weiter.

Aufspüren von Fehlern

Während das pfadfolgende Untersystem die Anfragen verfolgt, versucht dieses Untersystem herauszufinden, ob eine Anfrage erfolgreich war oder nicht. Obwohl es nicht möglich ist alle Fehler aufzuspüren, sind einige leichter und andere schwerer zu finden. Dafür erlaubt die Umgebung beides, internes und externes Aufspüren von Fehlern.

Internes Aufspüren von Fehlern wird dazu benutzt Fehler zu finden, die, bevor sie der Benutzer zu sehen bekommt, versteckt werden könnten. Interne Fehlerdetektoren

können auch die Middleware verändern, um Annahmen und Besonderheiten, die von den Applikationskomponenten geschaffen werden, zu erfüllen.

Externes Aufspüren von Fehlern spürt Fehler auf, die für den Benutzer sichtbar sind. Das beinhaltet Servicefehler wie Netzwerkausfälle und Systemcrashs.

Sobald eine Anfrage beendet wird, egal ob erfolgreich oder mit einem Fehler, loggt das das Subsystem zusammen mit der Anfragen-ID. Damit die IDs zwischen diesem und dem pfadfolgenden Subsystem gleich sind, müssen die beiden entweder die IDs austauschen oder müssen deterministische Algorithmen verwenden, die die IDs basierend auf der Anfrage des Klienten erstellen.

Datenanalyse

Sobald die Logliste einer Anfrage verfügbar ist, werden sie an das Analyse Untersystem weitergegeben. Die Datenanalyse verwendet einen Anhäufungsalgorithmus um festzustellen, welche Komponenten in Verbindung mit Fehlern gebracht werden können. Dieser ordnet die Logs dann gemäß Tabelle 6 an.

Klient Anfrage-ID	Fehler	Komponente A	Komponente B	Komponente C
1	0	1	0	0
2	1	1	1	0
3	1	0	1	0
4	0	0	0	1

Abb. 6. Geordnete Logs

4.1.4 Implementierung

Ein Prototyp von Pinpoint wurde in die J2EE Middleware implementiert. Dieser besteht aus einem Netzwerk-Sniffer und einem Analysierer. Pinpoint benötigt keine Veränderung der J2EE Programme. Nur das externe Aufspüren von Fehlern benötigt applikationsbezogene Überprüfungen – doch diese benötigen keine Veränderung der Applikationskomponenten. Daher kann Pinpoint mit fast jeder J2EE Applikation verwendet werden.

4.1.5 Ergebnisse

Um den Overhead von Pinpoint zu testen, wurde die Performance einer Applikation auf einem normalen J2EE Server und auf einem, der mit Pinpoint modifiziert wurde, getestet. Das Ergebnis war ein Overhead von 8%, was getrost als Peanuts bezeichnet werden kann. Die Frage nach der Effektivität von Pinpoint sieht man am besten in Abbildung 7.

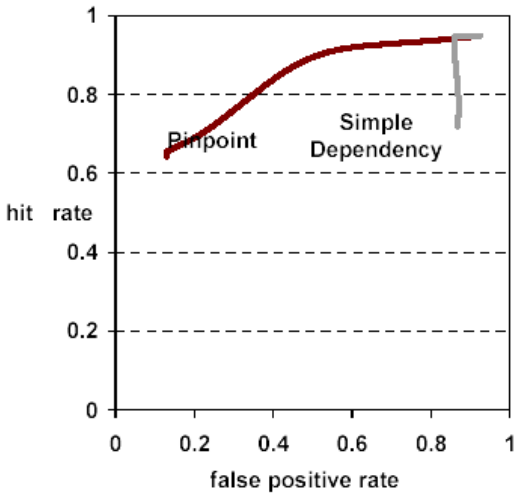


Abb. 7. Pinpoint Trefferquoten

Wie man erkennen kann, variiert die Anzahl der korrekt als defekt angezeigten Komponenten. Wenn man die Wahrscheinlichkeit, falsche Komponenten zu entdecken erhöht, erhöht sich auch automatisch die Anzahl der fälschlicherweise als defekt bezeichneten Komponenten.

4.1.6 Fazit

Pinpoint liefert sowohl was Overhead als auch was Genauigkeit angeht zufrieden stellende Ergebnisse. Ein Problem von Pinpoint ist allerdings, dass es nicht zwischen Komponenten unterscheiden kann, die ständig zusammen benutzt werden. Es werden allerdings Tests vorbereitet, die auch solche Komponenten trennen. Eine weitere Schwachstelle ist, dass Pinpoint nicht mit Fehlern funktioniert, die den Zustand verändern und andere Anfragen beeinflussen.

4.2 Folgerung

Das Beispiel von Pinpoint zeigt, wie man die Benutzer in der Fehlersuche bzw. in den Bemühungen die Fehlerursache zu finden, unterstützen kann. Da durch diese Hilfe das Beheben von Fehlern schneller von Statten geht, wird die Erreichbarkeit des Systems deutlich verbessert. Außerdem werden die Betriebskosten deutlich gesenkt, da der Administrator nicht mehr so lange braucht, um den Fehler zu finden. Im Idealfall muss er den Fehler nämlich nur noch beheben.

5 Hardwarebasierende Lösungen

5.1 ROC-1 Internetserver

Neben den oben vorgestellten Softwareumsetzungen von ROC gibt es auch Hardwarebasierende. Im Folgenden wird der ROC-1 Internetserver kurz vorgestellt.

Der ROC-1 Prototyp besteht aus 64 speziellen Knoten (werden hier als Module bezeichnet). Abbildung 8 zeigt den Aufbau eines solchen Moduls.

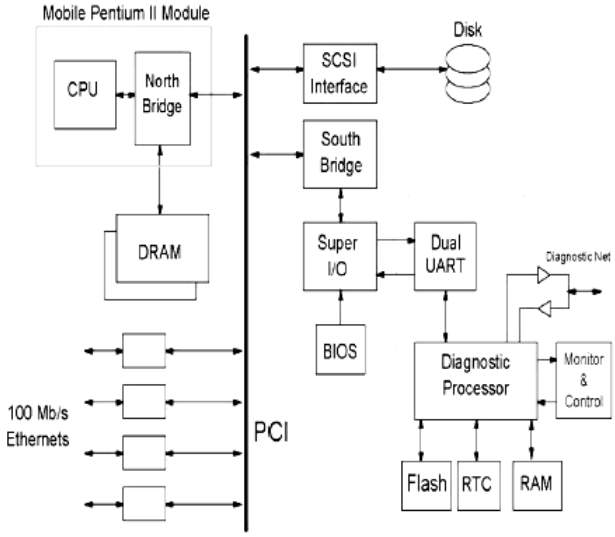


Abb. 8. Modul-Layout

Jedes Modul besteht aus einem mobilen Pentium II Prozessor mit 266Mhz, einer 18GB SCSI-Festplatte, 256 MB ECC-Speicher, vier redundanten 100 Mb/s Netzwerkinterfaces und einem 18Mhz Motorola MC68376 Diagnoseprozessor, welcher mit einem eigenen Diagnosenetzwerk verbunden ist. Wegen des Platzbedarfs und der Energieeffizienz sind die Module in einzelnen halbhohen 5 ¼ Zoll Rahmen untergebracht. Abbildung 9 zeigt ein solches Modul.

Abb. 9. ROC-1 Modul

Die ROC-1 Module sind komplett über redundante Ethernet Netzwerke verbunden. Jedes der vier Netzwerkinterfaces eines Moduls ist mit einem von 16 first-level Netzwerkswitches verbunden. Jedes dieser Switches ist über einen Gigabit Uplink mit einem von zwei Gigabit Switches verbunden. Als Netzwerk wurde ein 100 Mb/s Netzwerk eingesetzt.

Obwohl ROC-1 als Cluster strukturiert ist, hat es doch eine höhere Prozessor-Disk Leistung als die meisten, zu dieser Zeit vergleichbaren, Cluster. Damit wurde sichergestellt, dass genug Rechenleistung zur Verfügung steht – nicht nur für die Applikationen sondern auch für die Überwachungsfunktionen.

Der Hauptunterschied zwischen ROC-1 und anderen vergleichbaren Systemen ist allerdings der Diagnoseprozessor. Das Diagnoseuntersystem wird von einer Reihe von Sensoren, welche auf dem Modul und auf der Hauptplatine sitzen, versorgt. Jedes Modul enthält drei Thermometer, vier Spannungsmesser, einen Sensor der erkennt, ob das Modul entfernt wird, eine Batteriewarnung für das SRAM des Diagnoseprozessors und einen Beschleunigungsmesser auf der Festplatte um ungewöhnliche Vibrationen festzustellen.

Der Diagnoseprozessor erlaubt es, den Strom der Festplatte, der CPU und des Netzwerkinterfaces getrennt ein- und auszuschalten. Den Strom auszuschalten ist eine gute Möglichkeit, einen Fehler zu simulieren. Außerdem kontrolliert der Diagnoseprozessor zusätzliche Hardware, die es ihm ermöglicht, Fehler in den SCSI- und den Speicherbus sowie in das Netzwerk einzupflanzen. Der Diagnoseprozessor kann auch noch die CPU zurücksetzen. Aufgezeichnete Daten werden zwischen den Diagnoseprozessoren über das Diagnosenetzwerk ausgetauscht.

Wie gesehen ist der ROC-1 teilweise redundant aufgebaut (siehe Netzwerkinterface). Die Isolationen einzelner Komponenten wird auch unterstützt (wenn der Strom des Netzwerkinterfaces abgestellt wird, ist dieses Modul isoliert). Durch das Einpflanzen von Fehlern im laufenden Betrieb werden Tests simuliert, wie das System auf Fehler reagiert und damit die Recovery-Mechanismen getestet. Außerdem verfügt das System über eine Reihe von Sensoren, welche der Fehlerdiagnose dienen.

Wie man also sieht, verfügt der ROC-1 über eine Reihe von ROC Prinzipien. Allerdings besitzt der ROC-1 auch ein gutes Preis-Leistungsverhältnis, welches für Internetserver wichtig ist.

5.2 IBM pSerie

Als „Nachfolger“ des ROC-1 könnte man die pSerie von IBM sehen. Diese setzt auf RAS – Reliability (Zuverlässigkeit), Availability (Verfügbarkeit), Serviceablilty (Betriebsfähigkeit) um die Erreichbarkeit zu steigern. Eigentlich stehen RAS und ROC für die gleiche Sache. Allerdings setzt RAS verstärkt auf die traditionellen Fehlervermeidungsstrategien wie z.B. dem Vermeiden von Fehlern, dem Vorhersagen von Fehlern, einfache Redundanz und hat diese als Hauptbestandteil ihrer Philosophie angenommen.

Jedoch sind auch einige Gemeinsamkeiten mit ROC und speziell mit dem ROC-1 Prototyp erkennbar. So besitzt die pSerie ebenfalls einen Diagnoseprozessor – hier Serviceprozessor genannt. Dieser führt während dem Betrieb Tests durch und schreibt fehlerhafte Komponenten in das AIX Register – ein Register, in dem alle Hardware- und Softwarefehler des Systems gespeichert werden. Zusätzlich werden die Fehler analysiert und falls erforderlich das Servicecenter von IBM benachrichtigt (home call).

Neben dem Serviceprozessor gibt es auch noch einen Serviceagenten. Dieser hat eine ähnliche Funktion wie der Serviceprozessor, ist allerdings als Software umgesetzt.

Zusätzlich zu diesen beiden Features führt die pSerie auch so genannte BIST (Built-In Self-Test) und POST (Power-On Self-Test) durch. Diese überprüfen den Prozessor, den Cache und verbundene Hardware, die für ein korrektes Booten des Betriebssystems verantwortlich sind. Zusätzliches Testen kann während dem Starten ausgewählt werden, um den Systemspeicher und die Chipverdrahtung vollständig zu überprüfen.

Parity-Fehler im PCI Bus haben bei ältern Systemen eine systemweite Überprüfungsunterbrechung, die u. U. zu einem Neustart geführt hat, hervorgerufen. Im p630 System der pSerie erlauben das I/O Untersystem, die Firmware und das AIX sowohl eine transparente Behebung dieser Fehler, solange diese periodisch auftreten als auch einen sauberen Übergang in die nicht-Erreichbarkeit des I/O Gerätes im Falle eines permanenten Parity-Fehlers. Dieser Mechanismus wird Extended Error Handling (EEH) genannt.

Durch die RIO (Remote I/O) Verbindung ist der Prozessor mit den I/O Geräten verbunden. Dieses RIO benutzt eine Schleifenverbindungstechnologie um für redundante Wege zu den I/O Geräten zu sorgen. RIO ermöglicht CRC (cyclic redundancy checking) Überprüfungen auf dem RIO-Bus, wobei bei einem Bus-Timeout das Packet nochmals gesendet wird. Wenn eine Verbindung ausfällt, wird der RIO-Bus neu eingeteilt um die Daten über einen alternativen Pfad zu dem gewünschten Ziel zu leiten.

Außerdem verfügt die pSerie auch noch über ECC Mechanismen. Der normale Speicher besitzt Einzelbitfehlerkorrektur- und Zweierbitfehlererkennungs-Mechanismen. Der Zweierbitfehlererkennungsmechanismus hilft die Datenintegrität

beizubehalten. Die Speicherchips sind so aufgebaut, dass der Fehler eines Speichermoduls nur ein Bit in einem ECC-Wort verändern kann. Deshalb kann das System weiterarbeiten, wenn ein ganzer Speicherchip ausfällt (Chipkill recovery). Diese Mechanismen sind bei der pSerie allerdings nicht nur auf den Speicher beschränkt. Sie besitzt einen identischen Mechanismus, welcher den Prozessor durch einen Interrupt unterbricht. Dabei wird durch den Hardwarezustand die Adresse der verantwortlichen Instruktion ermittelt. Dadurch kann die fehlerhafte Software oder Firmware ermittelt werden. Der Vorteil dieses Mechanismus ist es, dass die fehlerhafte Software beendet werden kann anstatt das ganze System neu zu starten.

Auf weitere Aspekte, die über Online Verification of Recovery Mechanism und Integrated Diagnostic Support hinausgehen, wird hier nicht näher eingegangen [10][11].

6 Zusammenfassung und Ausblick

In diesem Paper wurde das Prinzip von ROC vorgestellt. Dabei wurde aufgezeigt, dass Erreichbarkeit und niedrige Betriebkosten essenziell für die heutige Computerwelt sind [1]. Und genau diesem Problem nimmt sich ROC an. Dabei benutzt es Mechanismen, um auftretende Fehler schnell zu erkennen bzw. Fehler zu beseitigen, bevor diese auftreten. Auf die Fehlerbeseitigung wurde hier allerdings nicht eingegangen [9].

Wie gezeigt wurde sind die bisherigen Bemühungen nur ein erster Schritt in die richtige Richtung. Allerdings müssen Methoden gefunden werden, um Recovery Soft- und Hardware zu testen. Nur dadurch kann garantiert werden, dass diese auch effektiv, zuverlässig und schnell arbeiten.

Referenzen

[1] Brown, A. and D. A. Patterson, “Embracing Failure: A Case for Recovery-Oriented Computing (ROC)”, *2001 High Performance Transaction Processing Symposium*, Asilomar, CA, October 2001

[2] Patterson, D. A., A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, N. Treuhaft, “Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies”, *UC Berkeley Computer Science Technical Report UCB//CSD-02-1175*, March 15, 2002

[3] Brown, A, “Accepting Failure: Availability through Repair-Centric System Design”, *UC Berkeley Qualifying Examination Proposal*, Berkeley, CA, April 2001

[4] Broadwell, P., N. Sastry and J. Traupman, “FIG: A Prototype Tool for Online Verification of Recovery Mechanisms”, To appear in *Workshop on Self-Healing, Adaptive and self-MANaged Systems (SHAMAN),* New York, NY, June 2002

[5] Chen, M., E. Kiciman, E. Fratkin, E. Brewer and A. Fox, “Pinpoint: Problem Determination in Large, Dynamic, Internet Services”, *Proceedings of the International Conference on Dependable Systems and Networks (IPDS Track)*, Washington D.C., 2002

[6] Oppenheimer, D., A. Brown, J. Beck, D. Hettena, J. Kuroda, N. Treuhaft, D.A. Patterson, and K. Yelick, “ROC-1: Hardware Support for Recovery-Oriented Computing”, *IEEE Transactions on Computers*, vol. 51, no. 2, February 2002
[7] Brown, A. and D. A. Patterson, “To Err is Human”, *Proceedings of the First Workshop on Evaluating and Architecting System dependabilitY (EASY '01)*, Göteborg, Sweden, July 2001
[8] Perrow C., “Normal Accidents: Living with High Risk Technologies”, Perseus Books, September 1999
[9] Mattes, O. “Oriented Computing: Modularisierung und Redundanz”, *Autonomic Computing Seminar*, Universität Karlsruhe, Juli 2003
[10] www.ibm.com
[11] http://www1.ibm.com/servers/eserver/pseries/hardware/whitepapers/p630_ras.pdf

Recovery Oriented Computing: Modularisierung und Redundanz

Oliver Mattes

Zusammenfassung. Diese Ausarbeitung gibt einen Überblick über Recovery Oriented Computing und die dort eingesetzten Techniken. Es wird gezeigt, nach welchen Regeln ein System entworfen werden muss, um eine hohe Verfügbarkeit sicherzustellen. Auf Modularisierung und Redundanz wird verstärkt eingegangen. Die Möglichkeiten von Recursive Restartability, einer Technik um hohe Verfügbarkeit zu erreichen, werden an Hand des Mercury Ground Station Networks vertiefend behandelt. Die daraus ersichtlichen Probleme im Einsatz von ROC werden ebenso angesprochen, wie auch die Vorteile, die durch ROC erlangt werden.

1 Einleitung

Der rasante technische Fortschritt ermöglicht immer größere und leistungsfähigere Computer. Doch während die Hardware durch den technologischen Fortschritt immer zuverlässiger wird, wird der menschliche Einfluss auf die Stabilität immer offensichtlicher. Die komplexen Systeme werden immer schwerer zu betreuen. So übertreffen die Ausgaben für die Wartung der Systeme die Hardwarekosten um das 10- bis 20-fache.

Dazu kommt, dass der Ausfall einer angebotenen Leistung viele Firmen sehr teuer zu stehen kommt. So belaufen sich die direkten Kosten für eine Stunde Nichterreichbarkeit von 25.000 $ für Netzwerkgebühren eines Providers über 180.000 $ im Fall eines großen Webshops wie Amazon.com bis zu 6.000.000 $ für den Ausfall eines Systems für Börsen-Transaktionen [1], [2], [3].

Hochverfügbare Systeme werden deshalb immer wichtiger. Doch wie kann sichergestellt werden, dass die benötigte Leistung möglichst immer zur Verfügung steht? Was ist die Idee hinter Recovery Oriented Computing (ROC), was für Techniken werden dabei eingesetzt und wie können damit die beschriebenen Probleme umgangen werden?

2 Idee von Recovery Oriented Computing

„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.” Simon Peres [1]

Recovery Oriented Computing geht davon aus, dass Hardware-, Software- und Wartungs-Fehler Tatsachen sind, die nicht gänzlich verhindert werden können und die man deshalb so gut wie möglich in den Griff bekommen muss. Trotz großer Anstrengungen Software zu verifizieren, sind verborgene, nicht eindeutig identifizierbare und reproduzierbare Fehler eine der Hauptursachen für die Nichtverfügbarkeit heutiger Systeme. Allgemein als Heisenbugs bekannt, sind sie nur sehr schwer nachzuvollziehen oder hängen von externen Ereignissen ab [4].

Dazu kommt, dass in Zeiten immer schnellerer Produktionszyklen Software vermehrt von teilweise unerfahrenen Programmierern geschrieben wird oder die Zeit für eine langwierige Testphase und Fehlersuche nicht ausreicht. Die Software wird deshalb oft fehlerbehaftet ausgeliefert und somit das Testen im laufenden Betrieb den Benutzern überlassen. Ebenso werden Systeme immer vielfältiger und unterschiedlicher und deshalb für Administratoren immer schwerer zu warten. Studien haben ergeben, dass die Ursachen für Systemausfälle bei mehr als 40% auf eine fehlerhafte Bedienung zurückzuführend sind [1]. Das liegt aber auch daran, dass in vielen Fällen kein richtiges Training und Einlernen möglich ist, da es aus Kostengründen kein Testsystem gibt, weswegen die Administratoren ihre Erfahrungen am laufenden System erlangen müssen. Für jedes System wäre aber eigentlich ein eigener Spezialist nötig. Aber auch diese Spezialisten sind Menschen und machen Fehler.

Ziel von ROC ist es, die Auswirkungen eines Ausfalls zu minimieren, indem spezielle Methoden eingesetzt werden, um ein System möglichst schnell wieder in einen lauffähigen Zustand zu versetzen, die Ausfallzeit so gering wie möglich zu halten und dadurch die Verfügbarkeit zu steigern und sicherzustellen.

Im ersten Golfkrieg gab es einen schwerwiegenden Zwischenfall, der die Auswirkungen dieser Probleme aufzeigt: Damals wurden 28 Soldaten getötet und mehr als 98 verletzt, weil das Raketenabwehrsystem nicht bereit war. Das eingesetzte Patriot-Raketenabwehrsystem musste auf Grund eines Fehlers in der Steuerungssoftware alle 8 Stunden neu gestartet werden und die Zeit zum Neustarten war zu lange, um noch rechtzeitig reagieren zu können [4].

2.1 Verfügbarkeit der Systeme

Einige große Computer- und Softwarefirmen wie Sun Microsystems, Motorola oder Cisco Systems preisen ihre Produkte mit einer 99,999%igen Verfügbarkeit an [5], [6], [7]. Dies ist eine Wunschvorstellung der Werbewirtschaft. Die Wahrheit sieht vielfach nicht so schön aus. Bis Mitte Juli 2003 wurden zum Beispiel bei eBay jeden Freitag zwei Stunden lang Wartungsarbeiten vorgenommen und dazu jedes Mal das System heruntergefahren. Umgerechnet bedeutete das nur noch eine Verfügbarkeit von 98,8%. Neu ist nun, dass nur noch diejenigen Funktionen vorübergehend nicht zur Verfügung stehen, die gerade gewartet werden und dadurch das Kaufen und Verkaufen auch fast immer während der Wartungsphase möglich ist [8].

Die Verfügbarkeit berechnet sich aus dem Quotienten aus der MTTF (mean time to failure), also der mittleren Zeitspanne bis ein Fehler auftritt, und der MTBF (mean time between failures), der mittleren Zeitspanne zwischen zwei Ausfällen. Die MTBF

ist wiederum die Summe aus der MTTF und der MTTR (mean time to recover), der Zeit für die Wiederherstellung des Systems (vgl. Abb. 1). Daraus ergibt sich folgende Formel für die Verfügbarkeit: V = MTTF / MTBF = MTTF / (MTTF + MTTR) [2], [3].

Abbildung 1. Zeitstrahl mit MTBF, MTTF und MTTR

Die MTTF steigt durch die technische Weiterentwicklung immer weiter an. Sie ist jedoch nur sehr schwer zu messen, da sie bei bestimmten Komponenten wie zum Beispiel Festplatten schon bei mehr als 100 Jahren liegt. Wie schon der Name andeutet, ist die mittlere Zeitspanne, bis ein Fehler auftritt, nur ein mathematischer Mittelwert. Es ist also in keiner Weise sichergestellt, dass ein Fehler nicht auch schon viel früher auftreten kann. Die MTTR hingegen ist relativ kurz und kann deshalb auch gut gemessen werden.

Verkürzt man nun die MTTR, erreicht man rein rechnerisch dasselbe Ergebnis, wie wenn man die MTTF um denselben Faktor erhöht. Eine höhere MTTF kann auch niemals eine fehlerfreie Ausführung garantieren, eine kurze MTTR kann jedoch die Auswirkungen eines Ausfalls lindern. Beispielsweise hat ein Ausfall von 10 Stunden im Jahr die gleiche Auswirkung auf die Verfügbarkeit wie jede Woche ein Ausfall von 12 Minuten Länge. Obwohl die Verfügbarkeit gleich hoch ist, hat der kurzzeitige Ausfall einige Vorteile: [9]

- Er ist weniger nachrichtenwirksam. Fällt zum Beispiel ein großer Emailanbieter für einen ganzen Tag aus, wird darüber in vielen Nachrichtendiensten berichtet. Fällt er jedoch nur kurzzeitig aus, ist es den Nachrichtendiensten oft keine Meldung wert oder sie erfahren selbst nicht davon.
- Die Ausfallkosten sind bei kurzzeitigen Ausfällen oft geringer. Der Online-Auktionsanbieter eBay gewährt laut seinen AGB bei unvorhergesehenen Systemausfällen, die länger als 2 Stunden dauern, Gutschriften von Gebühren für Angebote und verlängert die betroffenen Angebote um 24 Stunden [10].
- Die Benutzer merken weniger davon und es sind weniger Benutzer direkt davon betroffen. Der Ausfall eines Mailservers für einen Tag wird sehr viele Leute am Emailabholen hindern. Bei einem Ausfall von 10 Minuten hat ein Großteil der Benutzer die Möglichkeit, es kurze Zeit später noch mal zu versuchen.

Bei der Softwareentwicklung wird einerseits durch bessere Verifikation die Zahl der Fehler pro Zeilen Quellcode verringert, andererseits wächst jedoch die Größe von Programmen um einiges schneller, so dass insgesamt wieder mehr Fehler vorhanden sind und die MTTF nicht gesteigert werden kann [11].

Ein in den Nachrichten sehr publik gewordenes Beispiel für einen Systemausfall war im Jahre 2001 der 24-stündige Ausfall bei Microsoft, wodurch deren Webseiten nicht erreichbar waren. Sollte Microsoft nun versuchen wollen wieder eine

99,999%ige Verfügbarkeit ihrer Systeme zu erreichen, müssten die Rechner die nächsten 150 Jahre fehlerfrei durchlaufen [12], [13].

2.2 Wo wird ROC eingesetzt?

ROC wird überall dort eingesetzt, wo eine hohe Verfügbarkeit und eine hohe Datensicherheit verlangt wird. Dazu gehören Datenbanken, die sicherstellen müssen, dass bei einem Systemfehler keine Daten verloren gehen, Webserver von Internet-Kaufhäusern, die jederzeit erreichbar sein sollen, oder Email-Server, die ein Undo-Mechanismus integriert haben. Auch ein Update von Softwareteilen oder ein Umbau der Hardware während des laufenden Betriebs durch nur teilweisen Neustart ist ebenfalls möglich.

Eine weitere wichtige Einsatzmöglichkeit ergibt sich bei zeitkritischen Systemen, die zu bestimmten Zeitpunkten einsatzbereit sein müssen. Was passiert, wenn gerade in diesem Moment ein Fehler eintritt? Ein oft zitiertes Beispiel in diesem Zusammenhang ist das Mercury Ground Station Network (MGSN) [14], [11]. Dabei wurde ein Empfangssystem für Satelliten nach ROC-Kriterien entworfen. Eine Bodenstation bei MGSN hat mehrmals am Tag für einen kurzen Zeitraum von 25 Minuten Kontakt mit einem Satelliten. Wird dieser Kontakt verpasst, weil ein Fehler aufgetreten ist, gehen die gesammelten Daten unwiederbringlich verloren. Kann der Fehler jedoch in kürzester Zeit behoben werden, reicht ein Teil der Kontaktzeit für die Datenübertragung aus. Es wurde demnach versucht, ein System zu entwickeln, das möglichst schnell wieder lauffähig ist [15].

Ganz allgemein wird ROC überall dort eingesetzt, wo eine hohe Verfügbarkeit wichtiger ist als ein hoher Durchsatz. Dies wird dadurch erreicht, dass mit der Geduld des Benutzers gehandelt wird. Bemerkt er überhaupt etwas von einem Fehler und wie lange lässt er sich vertrösten und wartet? Untersuchungen haben ergeben, dass Ergebnisse innerhalb von einer Sekunde als schnell empfunden werden, ab acht Sekunden als etwas langsam, und nach zehn Sekunden verliert ein Benutzer die Geduld und wechselt zu einer anderen Aufgabe [16], [12]. Gelingt es nun, das System während dieser Zeit wieder lauffähig zu machen oder wenigstens das teilweise Funktionieren mit eingeschränktem Umfang sicherzustellen, war der Einsatz von ROC erfolgreich.

3 Techniken hinter ROC

Die Ideen von ROC können mit Hilfe bestimmter Techniken in schon existierenden Systemen eingesetzt werden. Dazu gibt es verschiedene Möglichkeiten, die sich unterschiedlich auf den Arbeitsaufwand auswirken.

3.1 Redundanz

Eine einfache Möglichkeit der Redundanz kann durch die mehrfache Auslegung einzelner Systemkomponenten erreicht werden. Beispielsweise werden in Systeme

mehrfache Netzwerkkarten, Netzteile oder Festplatten eingebaut. Bei der Datenspeicherung ist vor allem RAID (Redundant Arrays of Inexpensive / Independent Disks) ein Begriff. Dabei existieren verschiedene Varianten: Von RAID 0, bei dem nur auf verschiedenen Platten abgespeichert wird, über RAID 1, dem sogenannten Mirroring, also mehrfachem Speichern auf verschiedenen Platten, bis zu RAID 4, 5 und 6, bei denen die Daten auf mehrere Platten verteilt und über Prüfverfahren abgesichert sind. Alle diese Verfahren sind jedoch nur mit einem erheblichen Kostenmehraufwand zu erreichen [17], [18], [19].

Außerdem kann es Probleme geben, wenn die Sicherungssysteme den Hauptsystemen genau gleichen. Tritt nun ein kritischer Fehler in der Software auf, ist dieser Fehler auch in den anderen Systemen vorhanden. Beispielsweise könnte dann ein gezielter Angriff auf das System nicht abgefangen werden. Aus diesem Grund setzt zum Beispiel der Internet-Provider 1&1 Internet AG in seinem Karlsruher Rechenzentrum bei Netzanbindung zwei unterschiedliche Router von Cisco und Juniper ein [20].

Cluster-Systeme bestehen schon von der Idee her aus gleichen Komponenten. Hier bekommt im Normalfall jeder Knoten eine eigene Aufgabe zugeordnet. Beim Ausfall eines Knotens kann das System flexibel reagieren und die Aufgaben können von anderen Knoten übernommen werden. Dies kann sich jedoch auch nachteilig auswirken, wenn die übrig bleibenden Knoten mit der anstehenden Arbeit überfordert werden. Beispielsweise war der Webauftritt von CNN.com nach den Ereignissen vom 11. September 2001 zeitweise nicht mehr erreichbar. Der eingesetzte Cluster war der Flut der ankommenden Anfragen nicht gewachsen. Diese war sogar so hoch, dass die Administratoren sich nicht mehr in das System einloggen konnten. Als dann einzelne Knoten aus Überlastung den Betrieb einstellten und sich abschalteten, wurden die Anfragen auf die übrigen Knoten verteilt, die nach und nach ebenfalls den Dienst quittierten. Schließlich war der ganze Cluster lahmgelegt. Erst nach einem Neustart und der Bereitstellung einer kleinen Nachrichtenseite ging die Seite wieder online [16].

Ein wichtiger Punkt bei der Redundanz ist aber auch die gegenseitige Überwachung mehrfach ausgelegter Systeme. Hierbei kommt die Selbst-Diagnose ins Spiel. Paradebeispiel hierfür ist das Space Shuttle der NASA. Hier werden zur Fly-by-Wire-Steuerung fünf Computer eingesetzt, von denen sich vier gegenseitig kontrollieren und Fehler einzudämmen versuchen. Dazu können die Rechner auch fehlerhafte Rechner außer Betrieb stellen. Wird das Problem trotz der mehrstufigen Absicherung nicht behoben, kann bei Bedarf noch der fünfte Rechner, der mit einer anderen Software ausgestattet ist, von Hand gestartet werden [21], [22].

3.2 Modularisierung

Mit Hilfe der Modularisierung wird versucht, ein System in viele eigenständige und möglichst unabhängige Abschnitte aufzuteilen. Ziel sind Einzelteile, die auch nach einem Ausfall eines Partners weiterarbeiten und eigenständig darauf reagieren.

Hierbei ist auch die Wahl geeigneter Protokolle bei der Kommunikation ein entscheidendes Kriterium. Beispielsweise ist HTTP ein Protokoll, das auf zustandlose Kommunikation setzt, also keine lange Verbindung zwischen Client und Server

benötigt. Nach einer Antwort auf eine Anfrage bricht die Verbindung wieder ab. Kommt also das Ergebnis einer Anfrage von einem Rechner, kann die Antwort auf eine weitere Anfrage ohne weiteres von einem anderen Rechner kommen, ohne dass der Benutzer davon etwas merkt.

Durch sogenannte Hot-Swap-Techniken können solche eigenständige Systeme während des laufenden Betriebs herausgenommen werden, um nach erfolgter Reparatur oder Upgrade wieder integriert zu werden. Damit kann sehr flexibel auf unterschiedliche Gegebenheiten reagiert werden.

Auch bei der Wahl geeigneter Datenstrukturen kann Unabhängigkeit erreicht werden. So können Daten mit Hilfe von Hashing einfach verteilt werden. Bei einer Änderung der Komponenten müssen nur die Hash-Tabellen angepasst werden.

Virtuelle Maschinen (VM) wie Java sind ein Extremfall eines abgeschlossenen Systems. Hierbei wird ein komplettes System simuliert, was zur Folge hat, dass auch mehrere davon ohne gegenseitiges Wissen nebeneinander betrieben werden können. Webhoster setzen dies gerne ein, um den Benutzern für sich abgeschlossene Hosts auf ein und demselben Rechner zu ermöglichen, ohne dass die Arbeit anderer Nutzer beeinflusst werden kann. Bei Bedarf muss nur die betroffene VM neu gestartet werden, um den Betrieb wieder sicherzustellen [1].

Dieser Ansatz bildet die Grundlage für die sogenannten Recursive Restartability, also dem rekursiven Neustarten einzelner unabhängiger Komponenten. Diese Methode wird im folgenden Abschnitt ausführlicher behandelt, weil sie ein wichtiger Teil von ROC ist und dabei fast alle anderen angesprochenen Methoden Verwendung finden. Außerdem zeigt sich hier besonders klar, was heutzutage schon möglich ist und woran in Zukunft noch gearbeitet werden muss.

4 Recursive Restartability

Bei Recursive Restartability (RR) wird besonderen Wert auf eine Verringerung der MTTR durch gezieltes Neustarten einzelner unabhängiger Komponenten gelegt und darauf, dass das System währenddessen zumindest teilweise weiter funktioniert.

4.1 Reboot

Um die Startzeit zu verringern, wird als Mittel ein harter Reboot, also ein Neustart ohne vorheriges Herunterfahren verwendet. Dies geht sogar so weit, dass einzelne Komponenten schon neu gestartet werden, bevor ein Fehler auftritt, er jedoch immer wahrscheinlicher wird. Dadurch wird verhindert, dass ein Fehler das ganze System beeinflusst. Ziel ist es, das Neustarten so schnell und unauffällig wie möglich ablaufen zu lassen.

Ein Reboot hat auch einen entscheidenden Vorteil: So ist die Startphase eines Systems eine der am besten verstandenen und getesteten Abschnitte überhaupt. Es ist also genau bekannt, was dabei abläuft und es kann dadurch sichergestellt werden, dass nach einem Neustart das System in einen klar definierten Zustand zurückgesetzt wird.

Wird nun ein Reboot des ganzen Systems in einzelne Micro-Reboots aufgeteilt, werden deren Auswirkungen verringert und sie können gezielter eingesetzt werden, ohne gleich das ganze System lahm zu legen [23], [14], [4]. Durch diese Micro-Reboots wird auch erreicht, dass nur die fehlerbehafteten Teile neugestartet werden, was oft eine beachtliche Verkürzung der Startzeit ermöglicht.

Die Idee vom Reboot eines Systems, kann sogar so weit fortgeführt werden, dass man sich, wenn die Software und Hardware keine Probleme bei einem jederzeitigen Neustart machen würden, die Zeit und Funktionen zum Herunterfahren der Systeme sparen könnte.

Durch gezieltes Neustarten können auch blockierte Systeme wieder zum Leben erweckt werden. Ein Beispiel hierfür ist der Mars Pathfinder. Dieses mobile Gefährt kollidierte bei seinen Erkundungstouren auf der Marsoberfläche mit einem Stein und durch ein fehlerhaftes Scheduling-Verfahren blockierte das Steuerungssystem. Erst nach einem Neustart konnte eine neue Software eingespielt werden und die teure Mission konnte fortgesetzt werden [14], [4].

4.2 Probleme beim Reboot

Wie beschrieben ist die Zeitdauer bei einem Reboot entscheidend für dessen Einsatz. Benötigt nämlich die anschließende Kontrolle des Systems beim Neustart mehr Zeit als ein richtiges Herunterfahren und Neustarten, so werden die Vorteile eines Reboots wirkungslos. Heutige Systeme sind jedoch oft nicht für Reboots konzipiert. Das liegt vor allem daran, dass Programme nicht mit Rücksicht auf diese Neustarts programmiert wurden. Vielfach wird davon ausgegangen, dass ein spontaner Neustart nie auftritt und deshalb alle Anstrengungen auf die Steigerung der Geschwindigkeit des Programmablaufs gerichtet werden, auch wenn dadurch ein Datenverlust wahrscheinlicher wird.

Beim sicheren Neustarten von Programmen, dem sogenannten Clean Reboot, besteht die Nichtverfügbarkeit aus der Zeit, um das Programm herunterzufahren, und der Zeit, um es neu zu starten. Nach einem Absturz und anschließendem Neustart, dem Crash Reboot, ist nur die Zeit zur Fehlerbehebung entscheidend. Vergleicht man die Startzeiten der wichtigsten Betriebssysteme mit und ohne vorheriges Herunterfahren, wird besonders deutlich, wie sich gut gemeinte Sicherungssysteme auch negativ auswirken können [24].

Tabelle 1. Startzeit von clean und crash reboots [24]

System	Clean Reboot	Crash reboot
RedHat 8 (with ext3fs)	104 sec	75 sec
JBoss 3.0 application server	47 sec	39 sec
Windows XP	61 sec	48 sec

Wie in Tabelle 1 ersichtlich, ist die Startzeit bei einem Neustarten nach einem Crash um einiges schneller als beim Herunterfahren und anschließenden sicheren Neustart. Beim dem hier getesteten Linux-System ist vor allem das Dateisystem für die lange Startzeit verantwortlich. Um langsame Plattenzugriffe zu vermeiden, werden Datenänderungen, vor allem von Metadaten wie Dateinamen, Datum etc. im

Arbeitsspeicher zwischengespeichert. Das hat zur Folge, dass bei einem Absturz das Dateisystem einen inkonsistenten Zustand erreicht und mehr Zeit zum Testen und Reparieren benötigt wird. In diesem Beispiel werden für einen höheren Datendurchsatz im Betrieb Einbußen bei der Sicherheit und Reparaturzeit und damit indirekt eine langsamere Startgeschwindigkeit in Kauf genommen.

Die Auswirkungen dieser Zwischenspeicherung können jedoch durch Verkürzung der Zeitabstände zwischen dem Abspeichern, dem sogenannten Checkpointing, verringert werden. Dies wird seit langem vor allem beim Einsatz von Datenbanken angewendet. Um die Datenintegrität bei einem Absturz sicherzustellen, verwenden viele Datenbank-Systeme weitere Mechanismen, die dem ACID-Prinzip entsprechen [25]. Durch ACID (Atomicity Consistency Isolation Durability) werden wichtige Grundregeln von Transaktionen beschrieben. Bei einer Postgres Datenbank wird beispielsweise eine Logdatei und ein nichtüberschreibender Speicher verwendet. Nach einem Absturz kann die letzte Transaktion einfach abgelesen, rückgängig gemacht oder wiederholt werden [24], [11].

Ein weiterer Punkt beim sicheren Reboot besteht in der Unabhängigkeit der Teilsysteme. Es muss sichergestellt werden, dass ein System richtig reagiert, wenn einzelne Teile ohne Vorwarnung neu gestartet werden und deshalb zwischenzeitlich nicht erreichbar sind. Die übrigen Teilsysteme müssen gegebenenfalls ihren Funktionsumfang reduzieren, sollten jedoch nicht selbst ihren Betrieb einstellen oder auf einen fehlenden Teil warten. Etwas genauer beschrieben wird dies im Abschnitt 4.4 zum Thema Crash-Only Design.

4.3 Restart Trees

Die Startzeit für ein Gesamtsystem ist meist recht groß, mindestens jedoch so lange wie die längste Startzeit eines Teilsystems. Eine Verkürzung kann durch geschicktes Aufteilen und speziell abgestimmte Kontrollmechanismen erreicht werden.

Rekursiv startbare Systeme können durch einen Restart Tree beschrieben werden [15], [11]. Dieser besteht aus einer Hierarchie neustartbarer Komponenten, deren Knoten streng abgegrenzt sind und bei der ein Neustart eines Knotens gleichzeitig den ganzen dazugehörigen Teilbaum neu startet. Teilbäume in einem Restart Tree werden Restart Groups genannt. In einer solcher Gruppe G aus n Elementen $g_0,g_1,...,g_n$ beträgt die MTTF kleiner oder gleich dem Minimum aller Teil-Ausfallzeiten $MTTFg_i$. Die entsprechende MTTR beträgt $MTTR_G \geq \max(MTTRg_i)$.

Durch gezielte Umstrukturierung kann nun das schnelle Starten beeinflusst werden. Die verschiedenen Kriterien können sehr gut am bereits angesprochenen Beispiel der Mercury Ground Station gezeigt werden. Die Bodenstation besteht größtenteils aus Standarthardware und mit Java geschriebener Software. Nachrichten zwischen den Einzelteilen werden über einen auf TCP/IP basierenden Nachrichtenbus ausgetauscht. Das System ist in fünf Komponenten unterteilt, die jeweils in einer eigenen Java Virtual Machine (JVM) ausgeführt werden, womit eine strenge Abgrenzung sichergestellt wird. Dazu kommen noch Kontrollsysteme, mit deren Hilfe durch Selbst-Diagnose und Selbst-Test Fehler bemerkt werden und das Neustarten einzelner Komponenten eingeleitet werden kann. Alle Abbildungen und Zahlen in diesem Abschnitt sind der Arbeit [15] entnommen.

Ausgangspunkt für die Optimierung ist die Struktur wie sie in Baum I aus Abbildung 2 dargestellt ist. Dies ist der einfachste Baum, denn dort gibt es nur eine Ebene, wodurch jedoch nur das ganze System zusammen gestartet werden kann. Das hat zur Folge, dass alle Teilsysteme die selbe MTTR haben (siehe $MTTR^{I}$ in Tabelle 2).

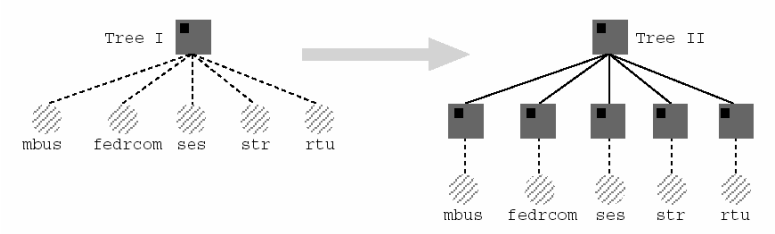


Abbildung 2. Einfache Erhöhung der Ebenen führt von Baum I zu Baum II

Der erste Schritt zur Optimierung wird durch Einfügen einer zusätzlichen Ebene und eines eigenen Teilbaums für jede eigenständige Komponente erreicht (siehe Baum II in Abbildung 2). Nun können die einzelnen Teilbäume unabhängig voneinander gestartet werden, wodurch oft deutlich Zeit eingespart werden kann (siehe $MTTR^{II}$ in Tabelle 2).

Tabelle 2. Zeit zur Entdeckung fehlerhafter Komponenten und um die einzelnen Teilsysteme neu zu starten (in Sekunden)

Failed node	mbus	ses	str	rtu	fedrcom
$MTTR^{I}$	24,75	24,75	24,75	24,75	24,75
$MTTR^{II}$	5,73	9,50	9,76	5,59	20,93

Von Fall zu Fall können einige Komponenten in Unterkomponenten aufgespaltet werden, die eine sehr unterschiedliche MTTR und MTTF haben können. In unserem Beispiel stellt „fedrcom“, ein bidirektionaler Proxy zwischen XML-Nachrichten und Funksignalen, so eine Komponente dar. Durch Ansprechen von externer Hardware und Warten auf deren Reaktion benötigt fedrcom lange, um zu starten, und durch Instabilitäten im Kommandoübersetzer stürzt es oft ab. Fedrcom hat also eine hohe MTTR und eine niedrige MTTF. Nun wird jedoch fedrcom in „pbcom“, ein TCP-Socket, und das dazu passende Frontend „fedr“ aufgespaltet. Pbcom ist stabil, aber langsam startend, fedr ist weiterhin instabil und fehlerbehaftet, benötigt jedoch nur wenig Zeit zum Neustarten. Daraus ergibt sich Baum II', der durch Einfügen einer zusätzlichen Ebene in Baum III umgewandelt wird (siehe Abbildung 3).

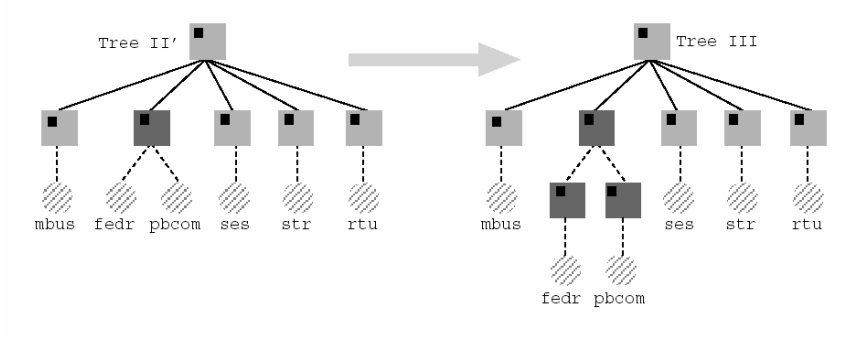


Abbildung 3. Aufspaltung von fedrcom in fedr und pbcom und Einfügen einer zusätzlichen Ebene ergibt Baum III.

Weitere Möglichkeiten zur Optimierung sind durch die Vereinigung verschiedener Gruppen und durch das Umhängen einzelner Komponenten gegeben. Die Vereinigung von Gruppen wird bei (funktionalen) Abhängigkeiten zweier Komponenten eingesetzt. In unserem Beispiel müssen sich die Module „ses“ und „str“ beim Neustart synchronisieren. Dazu müssen jedoch beide Teile neu gestartet werden. Durch die Vereinigung werden bei einem Fehler nun beide Teile automatisch neu gestartet. Dadurch wird oft Zeit gewonnen, da das Diagnosesystem schneller den Fehler feststellen kann. (siehe Baum IV in Abbildung 4).

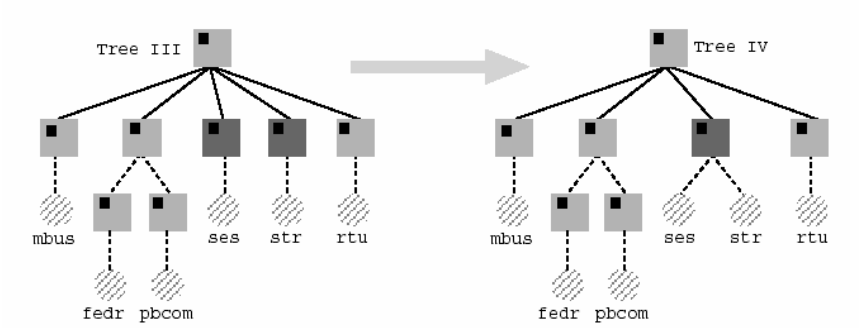


Abbildung 4. Vereinigung von ses und str zu einer Gruppe in Baum IV

Das Umhängen kann in ähnlicher Weise dazu genutzt werden, voneinander abhängige Komponenten nach bestimmten Kriterien umzuordnen. Hier wird pbcom eine Ebene höher im Baum eingehängt, da für einen sicheren Betrieb bei dessen Neustart auch fedr neugestartet werden sollte. Andererseits benötigt fedr keinen Neustart von pbcom. Ansonsten könnte eine Vereinigung beider Komponenten wie bei „ses“ und „str“ durchgeführt werden, was dann jedoch die eben erreichten Vorteile wieder zunichte machen würde. Als Ergebnis erhalten wir Baum V aus Abbildung 5.

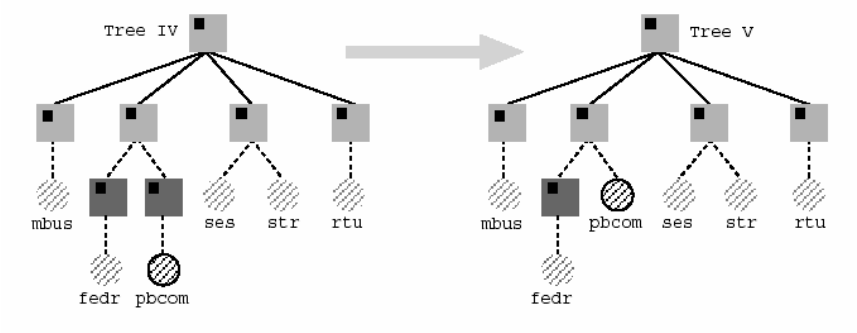


Abbildung 5. pbcom wird in eine höhere Ebene umgehängt.

Wie man in Tabelle 3 sieht, führt jedoch nicht jeder Schritt auch wirklich zu der gewünschten Verringerung der Startzeit. Im vorgestellten Beispiel ist die Architektur wie in Baum V geringfügig langsamer als die von Baum IV. Hierfür ist u.a. das Diagnosesystem, das für die Fehlererkennung und die Einleitung des anschließenden Neustarts zuständig ist, verantwortlich, da es etwas länger braucht um die defekte Komponente festzustellen.

Tabelle 3. MTTRs aller Komponenten in den verschiedenen Bäumen (in Sekunden)

Baum	mbus	ses	str	rtu	fedr	pbcom	Fedrcom
I	24,75	24,75	24,75	24,75	-	-	24,75
II	5,73	9,50	9,76	5,59	-	-	20,93
III	5,73	9,50	9,76	5,59	5,76	21,24	-
IV	5,73	6,25	6,11	5,59	5,76	21,24	-
V	5,73	6,25	6,11	5,59	5,76	21,63	-

Außerdem tritt oft eine geringfügige Verlangsamung bei der Aufspaltung einer Komponente in zwei Teile auf. Dies ist dann vor allem auf die erschwerte Kommunikation zurückzuführen, da nun nicht mehr Daten im gemeinsamen Adressraum ausgetauscht werden können, sondern nun der Umweg über IPCs genommen werden muss. Der dadurch entstehende Overhead wird jedoch im Normalfall durch den Zeitgewinn der Aufspaltung mehr als gutgemacht.

4.4 Crash-Only Design - Architektur schnellstartender Systeme

Um den Einsatz von Recursive Restartability zu ermöglichen, müssen die Teilsysteme möglichst gut mit jederzeitigen Neustarts umgehen können. Idealerweise sollten nach einem Crash überhaupt keine Reparatur- und Kontrollmaßnahmen nötig sein. Solche Systeme werden auch als crash-only bezeichnet, da bei ihnen keine speziellen

Funktionen nötig sind, um ein sicheres Beenden zu gewährleisten. Bei ihnen geschieht das Ausschalten durch einen Crash und das Einschalten einfach nur durch Neustarten. Um solche Systeme zu entwerfen, gibt es verschiedene Möglichkeiten. Bei vielen heutigen Internet-Systemen kommt davon jedoch immer nur eine Teilmenge zum Einsatz [24], [1].

Zustandsspeicher – Session State Stores
Bei einem normalen Programmablauf wird eine Vielzahl von Programmzuständen zwischengespeichert. Das geht zum Beispiel beim Surfen im Internet von mehreren Browserfenster bis hin zum Vor- und Zurückblättern in den zuletzt besuchten Seiten. Ideal ist es nun, wenn alle wichtigen, nichtflüchtigen Zustände von speziellen Zustandsspeichern verwaltet werden. Die Programme enthalten dann nur noch die eigentliche Programmlogik. Die Programme werden damit zustandslose Klienten von auf schnelle und sichere Datenspeicherung optimierten Zustandsspeichern wie objektorientierten Datenbanken, verteilten Datenstrukturen oder Hash-Tabellen. Diese Zustandsspeicher müssen natürlich selbst auch crash-only sein, denn sonst wird das Problem nur verschoben. Viele heutige Datenspeicher sind absturzsicher, jedoch nicht crash-only. In den meisten Fällen kann durch Variieren bestimmter Einstellungen, wie zum Beispiel durch häufigeres Abspeichern der Zustände (Checkpointing), die Reparaturzeit günstig beeinflusst werden [26].

Strenge Grenzen zwischen den Komponenten
Die einzelnen Komponenten müssen streng nach außen hin abgesicherte Grenzen haben. Eine solche strenge Isolierung kann zum Beispiel durch virtuelle Maschinen erreicht werden. Die darin ablaufenden Programme sind logisch voneinander isoliert. Die Grenzen zwischen Komponenten ermöglichen es auch, dass beim Bearbeiten von Anfragen eindeutige und individuell wiederherstellbare Zustände verwendet werden.

Wartezeiten
Jegliche Interaktion zwischen einzelnen Komponenten unterliegt einer festgelegten maximalen Wartezeit. Wenn während dieser Zeit keine Antwort erhalten wurde, wird vom anfragenden Prozess angenommen, dass die andere Komponente ausgefallen ist. Daraufhin wird das Diagnosesystem benachrichtigt, das dann einen Neustart einleiten kann. Bei crash-only-Systemen ist es sogar akzeptabel, Komponenten schon beim reinen Verdacht auf einen Fehler neu zu starten, da dadurch das Risiko eines Ausfalls verringert werden kann und die Komponenten in einen garantiert fehlerfreien Zustand versetzt werden. Die fehlerfreien Komponenten müssen aber auch auf die zeitweise Unerreichbarkeit anderer Komponenten reagieren und die eigene Ausführung gegebenenfalls durch Einschränkungen im Funktionsumfang sicherstellen.

Anbindung von Ressourcen
Durch das Leasen von Ressourcen an Stelle einer durchgehenden Anbindung, wird die Trennung der Komponenten von den Ressourcen sichergestellt. Wird die Leasing- oder Wartezeit überschritten, kann einfach darüber entschieden werden, ob die Ressource überhaupt weiterhin benötigt wird.

Selbstbeschreibende Anfragen
Die Benutzung von vollständig selbstbeschreibenden Anfragen durch die Angabe der Zustände und des Kontextes ermöglicht es, einer neuen Instanz einer Komponente dort weiter zu machen, wo vorher mit der Bearbeitung aufgehört wurde. Dies wird häufig im Internet bei Anfragen an Webserver über das eigentlich zustandslose HTTP-Protokoll verwendet. Die Kontextdaten können hier zum Beispiel mit der URL übermittelt werden. Diese Anfragen beinhalten immer auch eine bestimmte Lebensdauer (time-to-live), nach deren Überschritt die Anfrage neu gesendet oder eine Fehlermeldung ausgegeben wird.

Diagnosesysteme
Mit Hilfe geeigneter Diagnosesysteme wird der Zustand der Komponenten ständig überwacht und getestet. Dazu werden oft eigene unabhängige Prozessoren und Nachrichtenleitungen eingesetzt. Diese Infrastruktur ist ebenfalls für das Neustarten der einzelnen Komponenten verantwortlich [27].

Testen durch absichtliches Einschleusen von Fehlern
Außerdem gibt es die Möglichkeit zum absichtlichen Einschleusen von Fehlern in ein laufendes System. Damit kann das Verhalten gezielt getestet werden und können kritische Situationen simuliert werden. Das hat den Vorteil, dass Systembetreuer dadurch besser in ein System eingearbeitet werden und sie das richtige Verhalten bei Fehlern in einem ungefährlichen Rahmen erlernen können [27].

Verzicht auf Funktionen
Schlussendlich kann noch versucht werden, auf alle nicht unbedingt benötigten Funktionen zu verzichten. Was nicht vorhanden ist, kann auch nicht ausfallen, benötigt unter Umständen nur zusätzlichen Wartungsaufwand und verlangsamt mit großer Wahrscheinlichkeit auch das Starten des Systems. Ganz aktuell gibt es Bestrebungen, bisherige BIOS-Routinen durch schnellere und individuell zugeschnittene Programme zu ersetzen, da heutige Betriebsysteme nicht mehr auf die Unterstützung durch das BIOS angewiesen sind. Stattdessen wird zum Beispiel beim Linux-BIOS nach den initialisierenden Setup-Befehlen sofort ein Kernel geladen, mit dessen Hilfe dann die anderen Komponenten über Treiber angesprochen werden. Dadurch ist es möglich, dass ein System schon nach circa 2 Sekunden einsatzbereit ist [28].

5 Probleme von ROC

Oftmals wird der Sinn und Zweck von ROC nicht richtig verstanden und keinen Wert auf ein entsprechendes Design gelegt. Die Methoden von ROC müssten noch in viel größerem Ausmaß gelehrt werden, um ein Umdenken der Systementwickler zu erreichen. Bisher wird viel Wert auf die Steigerung der MTTF gelegt und dabei vernachlässigt, wie viel Erfolg durch eine kürzere MTTR erreicht werden kann. Die

Zeit, die mit der Suche nach unwahrscheinlichen Fehlern verbraucht wird, könnte zu einem großen Teil besser in einen flexibleren Umgang mit diesen gesteckt werden.

Zudem werden bisher die Ideen der System-Architektur und der Softwaretechnik wie Mikrokernarchitekturen und Modularisierung nur sehr eingeschränkt eingesetzt. Erst wenn von großen Programmpaketen Abstand genommen wird und stattdessen auf einen Verbund von vielen Einzelteilen zurückgegriffen wird, kann ROC richtig zum Zuge kommen.

Die Erfolge von ROC lassen sich nur schwer messen und vermitteln, solange die Hersteller lieber um möglichst schnelle Systeme wetteifern und als Benchmark nur der Idealfall getestet wird. Die wirkliche Leistung eines Systems zeigt sich aber nicht nur in hohen SPEC-Werten im idealen Umfeld, sondern während des harten Einsatzes im Alltagsleben. Dort ist es oftmals wichtiger, eine höhere Sicherheit auf Kosten eines möglichst großen Durchsatzes zu erreichen [17], [11], [29].

Doch es sollten deswegen nicht alle bisherigen Errungenschaften über Bord geworfen werden, denn ROC steht nicht für die Entwicklung fehlerhafter Software, die Produktion von unausgereifter Hardware, das Vernachlässigen von sorgfältigem Testen und Administrieren oder das Wegwerfen von existierenden nützlichen Techniken und Werkzeugen. ROC befürwortet vielmehr das Verkürzen der Startzeit, das Entwerfen von schnelleren Reparaturmechanismen in den verschiedensten Arten von Software und das Abschauen von Ideen aus der Natur, von Systemen, Internetprotokollen und der Psychologie [3].

6 Zukunft, Ausblick

Für die Entwickler und Software-Architekten müssen geeignete Werkzeuge entwickelt werden, mit deren Hilfe das Erstellen von rekursiv neu startbaren Systemen vereinfacht wird. Dazu müssen neue, einfache Modelle beschrieben und unterstützt werden. Es muss ein Weg gefunden werden, um die Eignung schon existierender Software für Neustarts zu beurteilen und neue Messverfahren und Benchmarks zu erstellen. Zudem müssen vor allem Hilfsmittel entwickelt werden, wie die Methoden von ROC in schon vorhandene Systeme eingebaut werden kann und welche Dinge geändert werden müssen.

7 Schlussbetrachtung

ROC ist eine Methode, um die Verfügbarkeit von Systemen zu erhöhen und deren Wartung zu vereinfachen. ROC widerspricht nicht den schon etablierten Methoden, vielmehr ist es eine sinnvolle Ergänzung zu redundanter Hardware und guter Software. Die Sicherheit der schon durch etablierte Methoden abgesicherten Systeme wird dadurch aber noch weiter gesteigert.

Im Vergleich zu Autonomic Computing [3] liegt das Ziel von ROC nicht darin, den Administratoren die Arbeit abzunehmen und sie durch Kontrollsysteme zu ersetzten, sondern darin, ihnen vielmehr die richtigen Werkzeuge in die Hand zu geben, um schneller und sicherer auf auftretende Fehler reagieren zu können. Menschen sind

sehr gut im Erkennen der eigenen Fehler und sie können aus ihnen lernen. Die Auswirkungen solcher Fehler können verringert und das danach folgende richtige Verhalten in automatische Prozesse integriert werden. ROC widerspricht den Errungenschaften von Autonomic Computing nicht. Beides kann sich sinnvoll ergänzen.

Referenzen

1. Dave Patterson, Aaron Brown, Armando Fox: Recovery Oriented Computing. University of California at Berkeley (2001).
2. James Cutler: Recovery-Oriented Ground Systems. Space Systems Development Laboratory, Software Infrastructures Group, Stanford University, GSAW (2003).
3. Armando Fox, David Patterson: Toward Recovery-Oriented Computing. Stanford University, UC Berkeley (2002).
4. George Candea, Armando Fox: Recursive Restartability: Turning the Reboot Sledgehammer into a Scalpel. Stanford University, 8th Workshop on Hot Topics in Operating Systems – HotOS-VIII (2001).
5. IS-SecCity - Solid und Sun: Fokus auf 99,999 % Verfügbarkeit, 29. Dezember 2002. http://www.itseccity.de/?url=/content/produkte/hochverfuegbarkeit/021229_pro_hoc_solid.html
6. Sun Microsystems GmbH: Sun in der Öffentlichkeit: Pressemitteilung / Sun Microsystems setzt High-Availability Strategie für Netzwerke der nächsten Generation konsequent fort, 4. Dezember 2001. http://de.sun.com/SunPR/Pressemitteilungen/2001/PM01_154.html
7. Motorola: Produktinformation Motorola HXP2000 High-Availability 16-Slot CompactPCI System. http://www.powerbridge.de/daten/p_picmg2.16-chassis/hxp/hxp2000.htm
8. eBay Deutschland: System-Mitteilungen: Wichtige Änderung bei den Wartungsarbeiten. http://www2.ebay.com/aw/marketing-de.shtml
9. Armando Fox: Toward Recovery-Oriented Computing. Stanford University (2002).
10. eBay Deutschland: Grundsätze zu Systemausfällen. http://pages.ebay.de/help/index_popup.html?policies=everyone-outage.html
11. George Candea, James Cutler, Armando Fox: Improving Availability with Recursive Micro-Reboots: A Soft-State System Case Study. Stanford University, Performance Evaluation Journal (2003).
12. Armando Fox, David Patterson: When Does Fast Recovery Trump High Reliability? Stanfod University, UC Berkeley (2002).
13. Robert Lemons, Melanie Austria Farmer: Microsoft blames technicians for massive outage, CNET news.com, 24. Januar 2001. http://news.com.com/2100-1001-251427.html?legacy=cnet
14. George Candea, James Cutler, Armando Fox: Applying Recursive Restartability to Real Systems. Stanford University (2002).
15. George Candea, James Cutler, Armando Fox, Rushabh Doshi, Priyank Garg, Rakesh Gowda: Reducing Recovery Time in a Small Recursive Restartable System. Stanford University, International Conference on Dependable Systems and Networks – DNS-2002 (2002).
16. George Candea und viele andere: On the Way to ROC-2 (JAGR: JBoss + App-Generic Recovery). Stanford & Berkeley (2003).
17. David Patterson, Aaron Brown, Pete Broadwell, George Candea und viele andere: Recovery Oriented Computing (ROC): Motivation, Definition, Techniques and Case

Studies. Computer Science Technical Report UCB//CSD-02-1175, U.C. Berkeley, March 15, 2002.
18. Jan Siekermann: Fault Tolerant Computer Systems, RAID. RWTH Aachen (2002).
19. Peter Chen, Edward Lee, Garth Gibson, Randy Katz, David Patterson: RAID: High-Performance, Reliable Secondary Storage. University of Michigan, DEC Systems Research Center, Carnegie Mellon University, UC Berkeley.
20. 1&1 Internet AG: Das 1&1 Hochleistungs-Rechenzentrum. http://hosting.1und1.com/?__page=about.tech
21. National Aeronautics and Space Administration (NASA): Shuttle Frequently Asked Questions, Second Generation Computers FAQ. NASA Spacelink, Education Programs, http://www.education.nasa.gov/, http://www.spacelink.msfc.nasa.gov/, (1994).
22. Max Breitling: Fehler und Fehlertoleranz, Begriffe und Techniken. Technische Universität München (2000).
23. George Candea, Armando Fox: Designing for High Availability and Measurability. Stanford University, Workshop on Evaluating and Architecting System Dependability – EASY (2001).
24. George Candea, Armando Fox: Crash-Only Software. Stanfod University, 9th Workshop on Hot Topics in Operating Systems – HotOS-IX (2003).
25. Joseph M. Hellerstein: Object-Oriented & Object-Relational DBMS. UC Berkeley (2000).
26. Benjamin Ling, Armando Fox: A Recovery-Friendly, Self-Managing Session State Store. Stanford University (2000).
27. Michael Kohlbecker: ROC - Recovery Oriented Computing. Autonomic Computing Seminar, Universität Karlsruhe (2003).
28. Axel Urbanski: Linux-BIOS: Von der Idee zur Technik. IX Magazin für professionelle Informationstechnik (7/2003), Heise Zeitschriften Verlag, Hannover
29. TOP500 Supercomputer sites, http://www.top500.org/