INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Diplomarbeit

Untersuchungen zum apoptotischen Verhalten verteilter Systeme

Martin Gail

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Diplomarbeit

Untersuchungen zum apoptotischen Verhalten verteilter Systeme

Martin Gail

Aufgabensteller: Prof. Dr. Dieter Kranzlmüller
Betreuer: Dr. Michael Schiffers
Christian Straube

Abgabetermin: 31. März 2012

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 31. März 2012

. . . .

(Unterschrift des Kandidaten)

Die Natur hat im Laufe von Milliarden Jahren sehr komplexe Lebensformen hervorgebracht, die aus unzähligen Einzelzellen aufgebaut sind. Um das Überleben des gesamten Organismus zu sichern, können sich Zellen kontrolliert selbst terminieren. Dieser Vorgang nennt sich Apoptose.

Auch verteilte technische Systeme haben eine Komplexität erreicht, für deren Beherrschung wir immer wieder neue Konzepte brauchen. Es bietet sich an, die Mission eines verteilten Systems formal vorzugeben, um Abweichungen davon automatisch erkennen und entsprechend reagieren zu können. Diese Mission muss nun auf die einzelnen Komponenten des Systems und deren Eigenschaften heruntergebrochen werden, damit diese sich selbst überwachen können. Sollte ein Fehler festgestellt werden, könnte ein Vorgehen ähnlich der Apoptose negative Einflüsse auf das restliche System verhindern, wodurch es seine Mission fortsetzen kann. Durch die Selbstverwaltung der einzelnen Komponenten soll Skalierungsproblemen und domainübergreifenden Zuständigkeitsfragen, wie sie bei zentralistischen Steuerungssystemen auftreten, begegnet werden. Ein generisches Konzept kann den implementatorischen Aufwand dank Wiederverwendbarkeit stark reduzieren und die Interoperabilität der Komponenten verbessern, sowie auch jenseits der IT Anwendung finden.

Diese Arbeit wendet das Konzept der Apoptose von Zellen zwecks dem Erhalt einer gemeinsamen Mission in verteilten Systemen an. Es werden mögliche Definitionen von Apoptose, Zellen eines verteilten Systems und deren Mission besprochen und Anforderungen an die technische Umsetzung gestellt. Hierfür werden Datenerfassungs- und Analysesysteme benötigt, um eine fundierte Entscheidung für eine Terminierung finden zu können. Hierfür wird ein Framework prototypisch entwickelt und auf seine praktische Umsetzbarkeit hin untersucht.

Inhaltsverzeichnis

1. Einleitung 1

2. Problemdefinition 3
2.1. Missionen von Systemen 3
2.2. Gefahren für eine Mission 5
2.3. Anwendungsfälle aus der Informatik 6
2.3.1. **1. Szenario:** Missionen im Grid Computing 6
2.3.2. **2. Szenario:** Verfalldatum bei Informationen 6
2.3.3. **3. Szenario:** Spionagesoftware 7
2.3.4. **4. Szenario:** Sensornetze 8
2.4. Anwendungsfälle bei physischen Systemen 9
2.4.1. **5. Szenario:** Heizung 9
2.4.2. **6. Szenario:** Sicherheit in der Industrie 9
2.4.3. **7. Szenario:** Smart Grid 10
2.5. Taxonomie von Fehlern 10
2.5.1. Fehlerklassen 10
2.5.2. Arten von Fehlern 13
2.6. Zusammenfassung 14

3. Anforderungsanalyse 15
3.1. Begriffsklärung 15
3.2. Missionserhalt ist das Ziel 17
3.3. Analyse der Anwendungsfälle 18
3.3.1. Definieren der Mission 19
3.3.2. Aufragserteilung und Missionsbearbeitung 21
3.3.3. Das Ende einer Mission 22
3.4. Anforderungen 24
3.4.1. **1. Anforderung:** Informationen sammeln 25
3.4.2. **2. Anforderung:** Wissen aus den Informationen ableiten 25
3.4.3. **3. Anforderung:** Reaktionen bestimmen und durchführen 25
3.4.4. **4. Anforderung:** Mit anderen Zellen kommunizieren 25
3.4.5. **5. Anforderung:** Verzeichnis- und Protokollplattform 26
3.4.6. **6. Anforderung:** Eine Zelle kennt die Mission 26
3.5. Zusammenfassung 26

4. Bestehende Konzepte zum Missionserhalt 29
4.1. Angriffsprävention in IT-Systemen 29
4.1.1. Virenscanner 29
4.1.2. Firewall 30
4.1.3. IDS und IPS 31
4.1.4. DEP und ASLR 32
4.2. Physische Gefahren erkennen und vermeiden 32
4.2.1. Schutz vor Eindringlingen 32
4.2.2. Schutz vor Elementarschäden 33
4.2.3. Ressourcenmangel 34
4.2.4. Redundanz 37
4.3. Autonomic Computing 37

4.4. Synergiediskussion . . . 38
4.4.1. Chancen in IT-Grids . . . 38
4.4.2. Power Grids . . . 39
4.4.3. Synergien nutzen . . . 39
4.5. Bewertung der Konzepte bezüglich der Anforderungen aus Abschnitt 3.4 . . . 40

5. Apoptose als Lösungskonzept 41
5.1. Sammeln von Informationen . . . 42
5.1.1. Arten von Daten . . . 42
5.1.2. Erfassen von Daten . . . 44
5.1.3. Weitergabe von Ergebnissen . . . 45
5.1.4. Normalisierung . . . 45
5.1.5. Aggregation . . . 46
5.2. Aus- und Bewerten von Daten . . . 47
5.2.1. Entscheidungsbaum . . . 48
5.2.2. Klassifizierung und Clustering . . . 48
5.2.3. Outliner Detection . . . 48
5.2.4. Neuronale Netze . . . 49
5.3. Angemessene Reaktion bestimmen und ausführen . . . 49
5.3.1. Mögliche Reaktionen . . . 49
5.3.2. Loggen und Bewerten der Entscheidungen . . . 52
5.4. Adaptive Apoptose . . . 52
5.4.1. Die Mission einer Zelle . . . 53
5.4.2. Selbstregulierende Überwachung . . . 54
5.5. Zusammenfassung . . . 54

6. Konzept eines Frameworks zum Missionserhalt durch Apoptose 55
6.1. Paradigmenwahl . . . 55
6.1.1. Prolog . . . 56
6.1.2. Skript mit funktionalen Ansätzen . . . 56
6.1.3. Objektorientierte Lösung . . . 56
6.2. Umsetzung der Anforderungen aus Abschnitt 3.4 . . . 58
6.2.1. **1. Anforderung:** Informationen sammeln . . . 58
6.2.2. **2. Anforderung:** Wissen aus den Informationen ableiten . . . 60
6.2.3. **3. Anforderung:** Reaktionen bestimmen und durchführen . . . 61
6.2.4. **4. Anforderung:** Mit anderen Zellen kommunizieren . . . 62
6.2.5. **5. Anforderung:** Verzeichnis- und Protokollplattform . . . 67
6.2.6. **6. Anforderung:** Eine Zelle kennt die Mission . . . 70
6.3. Kleines Beispiel für ein Framework . . . 73

7. Prototypische Implementierung der Apoptose 75
7.1. Ein Framework für die Steuerung von Apoptose . . . 75
7.1.1. Programmiersprache und Entwicklungsumgebung . . . 75
7.1.2. Aufbau des Projekts . . . 76
7.1.3. Aufbau einer XML-Konfigurationsdatei . . . 80
7.1.4. Datenaustauschformat zwischen den Zellen . . . 81
7.1.5. Weitere Eigenschaften des Prototyps . . . 82
7.2. Apoptose im Smart Grid . . . 82
7.2.1. Probleme im Stromnetz . . . 82
7.2.2. Beispielzenario . . . 84
7.2.3. Versuchsaufbau und -durchführung . . . 86
7.3. Apoptose beim Grid Computing . . . 88
7.3.1. Probleme im Grid . . . 88
7.3.2. Beispielszenario . . . 89
7.3.3. Auswertung des Szenarios . . . 90
7.4. Evaluierung des Prototypen . . . 91

8. Zusammenfassung und Ausblick **93**
8.1. Schwierigkeiten bei Konzeption und Umsetzung 93
8.2. Vor- und Nachteile von Apoptose 94
8.3. Weiterführende Arbeiten 94

A. Anhang **97**
A.1. XML-Listings 97
A.1.1. Kleines Beispiel für ein Framework 97
A.1.2. Aufbau einer XML-Konfigurationsdatei 98
A.1.3. XML-Dokumente des Beispielszenarios *Apoptose im Smart Grid* (7.2) 99
A.2. SQL-Listings 110
A.3. CD-ROM 111

1. Einleitung

Unsere Welt wird immer komplexer. Abhängigkeiten werden immer bedeutender und umfangreicher, wodurch die Systeme immer undurchsichtiger werden. Während ein Mensch früher einen Großteil seiner Arbeit noch weitgehend alleine erledigen konnte, muss er sich heutzutage auf die Fähigkeiten seiner Zulieferer verlassen, wie diese auch auf ihn zählen. Das ist ein Ergebnis der immer stärkeren Spezialisierung und dem damit einhergehenden Zwang zur Optimierung. In der heutigen globalisierten Welt ist jeder ein Konkurrent um Ressourcen oder ein potentieller Kunde. Deshalb legt man auf die Steigerung der Effizienz großen Wert. Oft ist es nur durch eine gute Planung und Kommunikation aller Beteiligter möglich, einen engen Zeitplan bei gleichzeitig hoher Qualität einhalten zu können. Bestimmte Hightechprodukte werden häufig in nur wenigen teuren Fabriken gefertigt, weshalb bei einem Ausfall große Versorgungslücken entstehen können. Aufgrund von Just-In-Time-Lieferungen, die eine teure Lagerhaltung ersetzen sollen, verschärfen Lieferausfälle das Risiko von Produktionsausfällen.

Konnte man mit einem Telefon früher nur telefonieren, so bieten Smartphones mit ihren vollwertigen Computerbetriebssystemen eine unendliche Erweiterbarkeit durch Zusatzprogramme. Und da alle Geräte smart, vernetzt und weltweit zugänglich sein sollen, um ein möglichst hohes Maß an Bequemlichkeit zu bieten, sind die Auswirkungen auf andere Teilnehmer kaum noch vorherzusehen.

Um den Überblick zu behalten und die Kontrolle nicht zu verlieren, werden immer mehr Daten erhoben. Diese Informationsflut kann nur noch maschinell ausgewertet werden. Da viele Abläufe bis auf Bruchteile von Sekunden genau aufeinander abgestimmt sind, müssen notwendige Reaktionen auf neue Ereignisse sehr schnell erfolgen. Hierfür ist eine Automatisierung der Steuerung mittels Computer unerlässlich. Doch auch Computer und Kommunikationsnetze erreichen irgendwann ihre maximale Kapazität. Noch mehr Daten machen eine Berechnung der Situation zwar genauer, wenn die Reaktion aber zu spät eingeleitet wird, können Ausfälle oftmals nicht mehr verhindert werden. Andererseits fehlen vielfach Daten, um richtig zu agieren. Dies ist vor allem bei verteilten Systemen der Fall, wenn die einzelnen Komponenten unterschiedlichen Domänen angehören und damit verschiedenen Herren dienen. Außerdem sind auch die Kommunikationsnetze nicht hundertprozentig ausfallsicher.

Daher ist es unerlässlich, den Fehlerfall beim Design von Anlagen mit hoher Komplexität einzuplanen. Ein bekanntes Beispiel ist der *„Blue Screen of death“*, der bei Windowssystemen einen Fehler im Kernelmode anzeigt. Ähnlich verhält es sich mit der *„Kernel Panik“* in Linux- und Unixsystemen. Hier wird durch einen klar definierten Endzustand ein nicht valider Zustand des Betriebssystems ersetzt, so dass bei einem unvorhersehbaren Ereignis Schlimmeres vermieden wird. Dies kann vom Datenerhalt bis hin zum Schutz der Hardware reichen.

Durch das Verbinden von technischen Systemen ergeben sich ganz neue Möglichkeiten und Eigenschaften, getreu dem Motto: *„Das Ganze ist mehr als die Summe seiner Teile“*. Somit erhöht sich auch die Komplexität, was das Design und den Betrieb solcher Netze deutlich erschwert. Da der Einfluss von Fehlern in derartigen großen Netzen häufig Gefahr für die Stabilität der angebundenen Einrichtungen bringt, ist ein Versagen oft schwer zu verkraften. Von der Funktionsfähigkeit wichtiger Infrastrukturen hängen oft Menschenleben ab und der wirtschaftliche Schaden bei einem Ausfall kann immens sein. Doch auch kleinere Systeme sollen ordnungsgemäß arbeiten. Schließlich stellen komplexe Systeme eine größere Investition dar, die eine klar definierte Aufgabe zu erfüllen haben. Doch immer wieder reichen die verfügbaren Kapazitäten nicht aus oder Fehler führen zum vollständigen Versagen der Technik. Es wäre demnach zu begrüßen, wenn sich die Auswirkungen von Fehlern begrenzen und deren negative Folgen minimieren ließen.

Ein schönes Beispiel ist eine Lichterkette, wie sie alle Jahre wieder zur Christbaumbeleuchtung Verwendung findet. Brennt eine Birne durch, bleibt die gesamte Kette dunkel. Sollte allerdings ein Stück des Glühfadens herunterfallen und einen Kurzschluss erzeugen, so sinkt der Widerstand in der Kerze. Sie glimmt höchstens noch, während die anderen Kerzen eine höhere Spannung erhalten und etwas heller leuchten. Deren Glühwen-

deln verschleißen durch die Überbelastung, weshalb schon bald die nächste Kerze erlischt. Da der Spannungsabfall in den verbleibenden Leuchten erneut steigt, fallen immer mehr Glühbirnen in immer kürzerem Abstand aus. Dies kommt einem kaskadierenden Blackout im Kleinen gleich.

Die Probleme, die in dieser Arbeit behandelt werden, entstehen durch die Komplexität verteilter Systeme. Zum einen skaliert eine zentrale Steuerung der Prozesse oft nicht. Dann können die einzelnen Komponenten auch unterschiedlichen Systemen angehören, die jeweils einer eigenständigen Administration unterliegen. Gerade die Interdomänkommunikation und die Verbindung von Komponenten verschiedenen Typs stellen immer wieder zu überwindende Schwierigkeiten dar. Das Ziel ist es dabei, die Auswirkungen von Fehlern in einem verteilten System zu minimieren. Da eine Komponente sich selbst und ihre Aufgabe am besten kennen sollte, erscheint ein dezentraler Ansatz der Überwachung sinnvoll. Wenn ein Fehler entdeckt wird, sollte eine Komponente sich selbst abschalten können, sofern sie für das Problem ursächlich ist. Dies entscheidet sie entweder auf Basis von Messpunkten selbst oder sie wird von außen durch andere Komponenten dazu angeregt.

In dieser Arbeit wird ein Konzept besprochen, das der Natur entlehnt ist: Apoptose. Dabei handelt es sich um den geplanten Zelltod, der von einer Zelle selbstständig durchgeführt wird, wenn sie dem Körper nicht mehr nutzt oder gar zu einer Gefahr für diesen zu werden droht.

Interessant ist nun, in wieweit sich ein zur Apoptose ähnliches Prinzip in der technischen Welt umsetzen lässt und ob es die geschilderten Probleme lösen kann. Daher soll in dieser Arbeit ein Framework konzeptioniert und implementiert werden, mit dem eine künstliche Apoptose untersuchen werden kann. Doch wie lässt sich dieses Konzept auf die einzelnen Komponenten eines verteilten Systems übertragen, um dessen Überleben, beziehungsweise die korrekte Ausführung der zugeteilten Arbeit sicher zu stellen? Ist ein zum biologischen Verfahren der Apoptose kongruentes Vorgehen überhaupt sinnvoll?

Im folgenden Kapitel *Problemdefinition* (2) wird auf die *Missionen von Systemen* (2.1) und die *Gefahren für eine Mission* (2.2) eingegangen, zu denen *Anwendungsfälle aus der Informatik* (2.3) und *Anwendungsfälle bei physischen Systemen* (2.4) beschrieben sind. Weiterhin wird auf die *Taxonomie von Fehlern* (2.5) eingegangen und deren Bedeutung für diese Arbeit erläutert. Es folgt eine *Anforderungsanalyse* (3) an eine dezentrale kontrollierende Instanz, um eine Mission besser erhalten zu können. Zunächst findet eine *Begriffsklärung* (3.1) statt, es werden formale Anwendungsfälle (3.3) vorgestellt, aus denen *Anforderungen* (3.4) abgeleitet werden. Die zuvor in Abschnitt 2.2 beschriebenen Szenarien werden bezüglich der Anforderungen analysiert.

In Kapitel 4 sind diverse Verfahren beschrieben, die eine Fortsetzung einer Mission auch bei Fehlern ermöglichen sollen. In Abschnitt 4.5 wird untersucht, in wie weit diese Konzepte die Anforderungen aus Abschnitt 3.4 erfüllen. Dabei wird sich zeigen, dass keines allen Anforderungen gerecht werden kann. Weiterhin findet eine *Synergiediskussion* (4.4) statt, die die Vorteile von der kombinierten Auswertung von Werten aus der digitalen und der physischen Welt aufzeigt.

Danach wird die *Apoptose als Lösungskonzept* (5) vorgestellt, um dezentral auf negative Einflüsse reagieren zu können, die die korrekte Ausführung einer Mission gefährden könnten. Apoptose ist ein Mechanismus in der Natur, der dem selbst herbeigeführten Tod einer Körperzelle entspricht. So sollen Gefahren für den Körper, wie zum Beispiel eine Krankheit, abgewehrt werden, indem sich infizierte Zellen terminieren, um das Ausbreiten der Krankheitserreger zu verhindern. Der Tod der Zelle kann auch vorprogrammiert sein, um ein natürliches Lebensende zu definieren und so Platz für Neues zu schaffen. Dieses Konzept der Natur wird nun adaptiert, um den Missionserhalt in verteilten technischen Systemen zu verbessern. Hierzu müssen die Anforderungen aus Abschnitt 3.4 umgesetzt werden: das *Sammeln von Informationen* (5.1) in all ihren Arten, das *Aus- und Bewerten von Daten* (5.2), auch mittels Techniken der *Knowledge Discovery in Databases*, sowie das Finden einer angemessenen Reaktion 5.3.

Nun folgt ein *Konzept eines Frameworks zum Missionserhalt durch Apoptose* (6), dass sich an den zuvor definierten Anforderungen (3.4) orientiert. Dessen Implementierung ist in Kapitel 7 beschrieben. Der dabei entstandene Prototyp wird für zwei Szenarien angepasst, *Apoptose beim Grid Computing* (7.3) und *Apoptose im Smart Grid* (7.2), an denen die Apoptose und deren Wirkung verdeutlicht werden kann.

Im Schlusskapitel 8 wird auf die *Schwierigkeiten bei Konzeption und Umsetzung* (8.1) des Frameworks eingegangen. Auch die *Vor- und Nachteile von Apoptose* (8.2) in einem verteilten System werden beleuchtet, bevor noch einige mögliche *Weiterführende Arbeiten* (8.3) vorgestellt werden.

2. Problemdefinition

Während die Menschheit über den Sinn des Lebens philosophiert, ist dieser bei technischen Systemen in der Regel vorgegeben. Maschinen werden gebaut, um Probleme zu lösen, die für den Menschen nur unbequem, schwer, gefährlich oder gar nicht zu bewältigen wären. Technische Geräte haben demnach meistens eine klar definierte Mission zu erfüllen. Alles was dieser zuwider handelt, stellt ein Versagen und damit eine potentielle Gefährdung des ganzen Systems dar.

Sobald die Aufgabe, die ein System zu erfüllen hat, zu komplex wird, empfiehlt es sich, das Problem und damit das System aufzuteilen. Durch *teile und herrsche* lässt sich vieles, das vorher überwältigend erscheint, auf mehrere, einfachere und oftmals wiederverwendbare Teile reduzieren. Das liegt an dem modularen Ansatz, der eine Spezialisierung beim Bau der einzelnen Komponenten ermöglicht. Unsere moderne Welt wäre ohne Spezialisierung in der Ausbildung und dem Verteilen der Verantwortung nicht denkbar. Bei dem Fachwissen, das schon für die Herstellung der kleinsten Bauteilen nötig ist, kann kein Mensch mehr eine komplette Produktion beherrschen. Das fängt beim Bäcker an, der auf den Müller als Mehllieferanten angewiesen ist, der seinerseits Getreide von einem Bauern bezieht, und hört beim Konstruieren, Fertigen, Betreiben und Nutzen von Computersystemen noch lange nicht auf. Auch bei Software versucht man, mittels Schichten und Middlewares zu abstrahieren, um die Verwendung und Weiterentwicklung zu vereinfachen. Hierdurch wird auch eine hohe Dynamik bei der Nutzung der Ressourcen – und damit eine Effizienzsteigerung durch eine optimale Auslastung – realisierbar. Betriebsmittel lassen sich flexibel belegen, so dass das resultierende Konglomerat geeignet ist, die ihm zugedachte Aufgabe schnell und gut lösen zu können.

Ein System, das die hierfür benötigten Komponenten anbietet, vernetzt und verwaltet, wird auch als Grid bezeichnet. Grids findet man nicht nur in der Informationstechnologie und dem *High Performance Computing*, sondern häufig auch im Zusammenhang mit Infrastruktur. Beispielsweise sollen die im Englischen als *Power Grids* bezeichneten Stromnetze mittels IT-Technik intelligenter werden. Die *Smart Grids* genannten Netze können sich besser an Veränderungen der Auslastung anpassen, indem Methoden des *Grid Computings* bei der Ressourcenverwaltung Verwendung finden. Denn Stromnetze haben immer mehr ein Problem, das auch bei verteilten Computersystemen erkannt wurde. Ian Foster benannte 2001 dieses *Grid-Problem*:

> *„The real and specific problem that underlies the Grid concept is coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations“* [FKT 01].

Die Schwierigkeit bei der Nutzung eines Grids liegt demnach bei dem Teilen von Ressourcen zwischen den verschiedenen Aufträgen. Es müssen nicht nur die benötigten Komponenten und ausreichend Kapazitäten bereitgestellt werden, sondern auch Verträge sowie Sicherheitsrichtlinien eingehalten und eine entsprechende Vergütung berücksichtigt werden. Oft besteht ein Grid aus Teilnetzen, die unterschiedlichen Domänen zugeordnet sind. Gerade die Zuständigkeit bei grenzüberschreitenden Anwendungen ist alles andere als trivial. Da ein Grid sehr groß werden kann, also eine Vielzahl, teils unterschiedlicher Komponenten enthält, skalieren zentralistische Ansätze nicht immer. Hierarchien können helfen, ziehen aber wieder erneut Zuständigkeitsgrenzen, deren Überschreitung wiederum problematisch ist. Auch muss eine Zentrale die Fähigkeiten und Zustände aller Komponenten kennen. Ein dezentraler Ansatz kann daher den Verwaltungsoverhead verringern, die Interdomänadministration vereinfachen und einen *single Point of Failure* vermeiden. Allerdings müssen die einzelnen Komponenten intelligenter werden, miteinander sprechen und über die allgemeine Lage informiert sein. So ist die Durchführung einer gemeinsamen Mission im Grid möglich.

2.1. Missionen von Systemen

Eine Maschine kann eine Aufgabe auf zwei Arten zugeteilt bekommen: entweder ist das Gerät explizit für die Erfüllung dieser Aufgabe entwickelt worden, oder die Maschine kann allgemeine Arbeitet verrichten.

Die Peripherie eines Personal Computers gehört in der Regel in die erste Kategorie. So ist die Aufgabe die Tastatur, Maus, Webcam, Drucker oder Scanner haben, bereits bei deren Design bekannt. Daher ist es einfach, diese Geräte auf ihren Verwendungszweck hin zu optimieren. Der PC selbst ist eine generische Maschine, deren Aufgabe durch die Software, die zur Ausführung gebracht wird, definiert ist. Somit ist die Mission bei der Fertigung des Computers nicht bekannt, eine Optimierung kann nicht erfolgen. Viele der bereitgestellten Funktionen werden für eine anstehende Aufgabe nicht gebraucht, können aber unter ungünstigen Umständen das korrekte Abarbeiten des Programms verhindern. Die Berechnung erfolgt dann sehr langsam, das Ergebnis ist fehlerhaft oder die Anwendung stürzt mit einer Fehlermeldung ab. Im schlimmsten Fall friert das ganze Betriebssystem ein, wie das bei defekter Hardware oder Treibern häufig passiert.

Eine weitere Gefahrenquelle besteht in gezielt eingeschleustem Schadcode, der das System dazu veranlasst, entgegen den Bestimmungen des ursprünglichen Auftrags zu handeln. Durch das Hacken fremder Systeme entstehen eventuell negative Nebeneffekte, die die Stabilität des gesamten Apparats beeinträchtigen können. Vor allem wenn Technik jenseits ihrer Spezifikation betrieben wird, sind Risiken für die eigene Stabilität, andere Hardware in der Umgebung oder das Leben von Menschen nicht auszuschließen. Wenn der Computer zuhause abstürzt, ist dies meistens nur ärgerlich, wird jedoch ein Computer zur Steuerung von Infrastruktur oder Industrieanlagen durch falsche Bedienung, fehlerhafte Software oder einen böswilligen Angreifer zerstört, kann dies neben teuren Sachschäden auch den Schaden an oder sogar den Tod von vielen Menschen verursachen.

Solange ein System isoliert ist, sind die Auswirkungen von Fehlern noch einigermaßen überschaubar. Doch wenn ein System aus vielen Einzelkomponenten besteht, steigt die Komplexität gewaltig an. Eine Mission wird nicht mehr nur für ein Gerät spezifiziert, sie wird auf verschiedene Maschinen verteilt. Dies kann gleichmäßig erfolgen, wenn die Hardware die gleiche Funktionalität bereitstellen kann. Auf Geräten unterschiedlicher Art muss die Mission abhängig von den Fähigkeiten der Maschine aufgeteilt werden. So kann ein Drucker nur drucken, ein Scanner nur scannen, ein Austausch der Aufgaben ist wenig sinnvoll. Wenn nun eine Mission *„Kopieren“* heißt, hängt diese neben dem kontrollierenden Computer auch von der Peripherie ab. Ist dabei ein Gerät fehlerhaft, kann die ganze Mission scheitern. Besteht Ersatz, zum Beispiel ein weiterer Drucker im Netz, so könnte der Nichtfunktionierende seine Aufgaben an diesen weiterleiten. Obwohl einzelne Teile des Systems ausfallen, ist eine Erfüllung der Mission eventuell möglich. Behauptet der kaputte Drucker allerdings weiterhin funktionsfähig zu sein, so kann die Mission *„Kopieren“* nicht erfolgreich abgeschlossen werden.

Es ist demnach erstrebenswert, jedes Gerät in die Lage zu versetzen, sich selbst dahingehend zu überwachen, ob es fähig ist, die ihm im Rahmen einer Mission gestellte Aufgabe korrekt zu erfüllen. Ist dies nicht der Fall, so sollte das Gerät sich selbst abschalten und damit aus der Gleichung nehmen, wodurch die Anzahl potentieller Fehlerquellen reduziert wird. Hierdurch kann es sein, dass das übergeordnete System in der Lage bleibt, seine Mission zu erfüllen, womit die Lebendigkeit dieser gewahrt ist. Das Schicksal des Einzelsystems spielt dabei keine Rolle, so lange es für die Mission nicht unverzichtbar ist.

Man kann auf zwei Arten überprüfen, ob ein System seine Mission erfüllt: man testet es auf Korrektheit oder auf Fehler. Wenn es seine Aufgaben ordnungsgemäß verrichtet, dürfte alles in Ordnung sein, wenn es etwas macht, wofür es nicht vorgesehen ist, liegt ein Defekt vor.

Der erste Fall ist schwierig zu prüfen, da Korrektheit kaum zu beweisen ist. Zunächst muss der kontrollierenden Instanz verständlich gemacht werden, was die Aufgabe eines Geräts ist. Da ein Computer keine Intelligenz besitzt, ist eine formale Beschreibung der Mission notwendig, mit der die Arbeit der Maschine verglichen wird. Da dies in einem Rechner nur mit mathematischen Vergleichen gemacht werden kann, müssen Mission und Qualität der Arbeit in Zahlen ausgedrückt werden. Dies ist nicht immer einfach, da sich Parameter in der Praxis meistens von theoretischen Vorgaben unterscheiden. Es sind also Toleranzen bei den Grenzwerten einzuplanen. Weiterhin schließt ein innerhalb seiner vorgegebenen Grenzen arbeitendes Gerät ein Fehlverhalten nicht grundsätzlich aus. Spionageprogramme zum Beispiel verstecken sich meist tief im System und tarnen ihre Kommunikation, um nicht erkannt zu werden. Doch auch wenn daraus ein Schaden für den Betreiber einer Infrastruktur entstehen kann, so gefährdet derartige Software die Ausführung der Mission nicht direkt. Zuletzt muss natürlich auch das Prüfsystem selbst fehlerfrei sein, damit dessen Aussage als korrekt angesehen werden kann.

Die zweite Art, einen Fehler zu finden, ist, gezielt danach zu suchen. Wenn sich ein Programm abnormal verhält, kann dies leicht festgestellt werden. Doch ein unbekannter Fehler kann durch das Raster fallen und nicht gefunden werden. Auch sagt ein sich normal verhaltendes System nichts darüber aus, ob es auch seine tatsächliche Mission erfüllt. Ganz im Gegenteil, wird das Raster zu eng gewählt, sprich die Toleranzen zu ge-

ring gesetzt, so können sich auch ordnungsgemäß ablaufende Prozesse in diesem verfangen. Das Ergebnis sind falsch positiv gemeldete Fehler, die ihrerseits zu einem Abbrechen des Prozesses führen können. Dadurch fällt eine Ressource aus, was durch gleichartige Ressourcen kompensiert werden muss. Wenn die erhöhte Last bei jenen ihrerseits die zulässigen Parameter sprengt, führt dies zu weiteren Abschaltungen. Eine derartige Kettenreaktion führt häufig zum Totalausfall kritischer Teile eines Netzes und damit zum Scheitern der Mission. Kaskadierende Effekte können auch durch berechtigte Terminierungen oder Einschränkungen von Betriebsmitteln entstehen und selbstverstärkend wirken. Einer Lawine gleich kann ein geringfügiger Auslöser zu einer gewaltigen Katastrophe führen, wenn die Kettenreaktion nicht rechtzeitig unterbrochen wird.

2.2. Gefahren für eine Mission

Da technische Anlagen dazu geschaffen werden, bestimmte Aufgaben zu erledigen, ist ein System, das seiner Mission nicht gerecht wird, nutzlos. Die Mission ist entweder designentscheident beim Aufbau von Anlagen, oder wird in Form von Software einem generischen System obtruiert. Software lässt sich allerdings nur mit großem Aufwand und in sehr geringem Umfang auf Korrektheit prüfen. Fehler bei der Entwicklung sind daher immer anzunehmen, normale Software kommt auf bis zu 25 Fehler je 1000 Zeilen Code, gute auf immer noch bis zu 2 Fehler [Huck 11]. Hinzu kommen Defekte der Hardware oder bei der Bedienung, die beim Entwurf der Programme nur teilweise beachtet werden können. Daher werden Abstraktionsebenen zwischen ein Programm und die Hardware gezogen, die diese Fehler abfangen und das laufende Programm auf Lebendigkeit und Unschädlichkeit hin überwachen sollen. Dies sind bei Computern unter anderem das Betriebssystem, Laufzeitumgebungen, Middlewares und virtuelle Maschinen. Diese sollen die für die Ausführung benötigten Ressourcen bereitstellen und deren Fehlen möglichst kompensieren.

Gerade in einem verteilten System mit vielen nebenläufigen Zugriffen muss das Ressourcenmanagement diverse Probleme lösen [FKT 01]. So kann eine Ressource nicht oder nicht mehr verfügbar sein. Falls es nun äquivalente Ressourcen gibt, die statt dessen genutzt werden können, so ist ein Ersatz möglich, mit dem die Mission weiter fortgesetzt werden kann. Dabei können Betriebsmittel temporär blockiert sein. Dann muss der Prozess so lange angehalten werden, bis die gefragte Ressource wieder verfügbar ist. Eine Schwierigkeit stellen dabei Deadlocks dar, die einen Ablauf für alle Zeit blockieren. Sie treten auf, wenn ein Prozess auf das Freiwerden eins Betriebsmittels wartet und gleichzeitig ein weiteres Mittel hält, welches von einem anderen Prozess zum Fortsetzen seiner Mission benötigt wird. Jener wiederum blockiert die Ressource, auf die der erste Prozess wartet. Dieser wechselseitige Ausschluss kann auch bei mehr als nur zwei Prozessen auftreten, womit ein Wartezyklus entsteht [CoEl 71]. Nun wartet jeder auf den anderen, keiner kann weitermachen und die Mission ist so gut wie gescheitert. Diese Zwickmühle kann gelöst werden, indem die Prozesse den Deadlock erkennen und einer von sich aus terminiert, um die von ihm gehaltenen Betriebsmittel für die wartenden Prozesse frei zu geben. Somit soll die Lebendigkeit der Mission wiederhergestellt werden.

Schlimmer als eine nicht nutzbare Ressource ist eine störende. Ähnlich einer natürlichen Zelle, die mit einem Virus infiziert ist und andere Zellen in einem Körper ansteckt, kann eine technische Zelle ihrer Nachbarschaft schaden. Auch für Computerprogramme gibt es Viren, die genau wie ihre natürlichen Vorbilder den Quellcode, dem künstlichen Äquivalent der DNA, in einer Zelle umschreiben. So eine Malware zwingt einen Computer dazu, nicht mehr seine Aufgaben wahrzunehmen. Hierdurch kann die Leistungsfähigkeit des gesamten Systems beeinträchtigt werden, im schlimmsten Fall wird die Mission abgebrochen.

Aber auch wenn kein technischer Defekt vorliegt, kann eine Komponente dem erfolgreichen Abschluss einer Mission im Wege stehen. Gerade wenn es um die Einsparung von Rohstoffen geht, konkurrieren die Zellen miteinander. Warum also sollte eine Zelle, die ihre Aufgabe erfüllt hat und nicht mehr benötigt wird, Ressourcen verbrauchen, die andere und wichtigere Zellen dringend benötigen? Auch stellt eine abgeschaltete Zelle ein geringeres Risiko für die Mission da als eine, die sich im Leerlauf befindet. Sie kann nicht gehackt werden, keine Malware ins Netz streuen oder Konflikte durch eine fehlerhafte Konfiguration der Netzwerkschnittstelle produzieren. In jedem Fall ist eine deaktivierte Zelle eine Variable weniger in einem komplexen System. Besonders beim Ende einer Mission oder bei deren Abbruch muss auch eine Zelle die Arbeit an der Mission einstellen und die von ihr gebundenen Mittel freigeben. Damit hat eine Zelle ihr natürliches Ende erreicht, wenn sie je nach Implementierung auch weiterhin anderen Aufgaben zur Verfügung steht.

2.3. Anwendungsfälle aus der Informatik

2.3.1. 1. Szenario: Missionen im Grid Computing

In einem modernen Rechenzentrum werden Server meistens virtualisiert, um die vorhandene Hardware besser auszunutzen [BDF+ 03]. Somit könnte jede dieser virtuellen Maschinen eine eigene Mission haben oder als eigenständige Komponente im verteilten System betrachtet werden. Demnach liefe für jeden virtuellen Server ein eigenes Framework, das dessen Arbeit überwacht und notfalls auch suspendieren oder beenden kann. Dieses Framework kann dabei im Code des Servers fest integriert sein oder als eigenständiger Prozess ausgeführt werden. Trifft letzteres zu, so muss das Framework sich auf die Aufgabe des jeweiligen Servers einstellen lassen. Dies setzt eine möglichst generische Implementierung voraus. Diese hat den Vorteil, dass die Überwachungssoftware nur in einer Version getestet und gewartet werden muss, denn ein generisches Framework lässt sich leichter wiederverwenden.

So könnte es auch auf dem Hypervisor laufen, um die Mission, die Hardware zu schützen, zu überwachen. Damit kann die Hardware mit dem Hypervisor wiederum als Komponente des Verbundes der auf dem Rechner installierten virtuellen Maschinen gesehen werden. Mitteilungen von den Frameworks, die in jenen laufen, dienen dabei als Eingänge für das im Hypervisor integrierte Framework und umgekehrt.

Gesetzt den Fall, ein Rechner in einem Rechenzentrum vieler gleichartiger Computer würde ein Hardwareproblem feststellen, so ist er also eine Komponente in einem größeren Gesamtsystem, dessen Mission es ist, virtuelle Maschinen zu beherbergen. Diese ist nun unter Stress, weil sie ihre Aufgabe nicht mehr wahrnehmen kann. Sie kann nun den anderen Computern im Rechenzentrum mitteilen, dass sie sich bald beenden wird und daher jemand anderer ihre Aufgaben übernehmen muss. Damit bricht die Ressource *Hypervisor* für die auf ihr laufenden virtuellen Maschinen weg. Diese Zellen beenden sich daraufhin selbst, da sie ihre Aufgabe nicht mehr erledigen können, wodurch sie für die entsprechende Mission nicht mehr verfügbar sind. Der Hypervisor verschiebt nun die virtuellen Maschinen zu seinen Nachbarn, wo diese neu starten. Für die höheren Missionen sehen diese neuen Server wie neue Ressourcen aus, die die Aufgaben der ausgefallenen Komponente übernehmen können. Somit ist die Ausführung aller Missionen gesichert, obwohl ein Computer im Rechenzentrum nicht mehr mitarbeitet.

> **Die Mission** eines Computergrids, wie die eben beschriebene Serverlandschaft auch gesehen werden kann, wird durch den auszuführenden Job bestimmt. Dieser wird auf die einzelnen Komponenten im Grid verteilt. Dadurch, dass sich jene selbst abschalten, sofern sie ihre Arbeit nicht mehr wahrnehmen können, wird mit wichtigen Rohstoffen sparsamer umgegangen. So belegt eine nicht gebrauchte virtuelle Maschine nicht unnötig IP-Adressen und Ports, Speicher und CPU-Zeit des Wirtsrechners, der wiederum, wenn er sich selbst beendet, Rohstoffe wie elektrischen Strom, Bandbreite im Netz oder Kapazitäten des Kühlsystems einspart. Somit wird nicht nur reaktiv auf den Ausfall von Hardware reagiert, sondern auch proaktiv gehandelt, um Engpässe und weitere Ursachen für mögliche Fehler zu vermeiden.

2.3.2. 2. Szenario: Verfalldatum bei Informationen

Aufgrund der Entwicklung von Computern ist es möglich, gewaltige Mengen an Daten verarbeiten und speichern zu können. Daher werden große Anstrengungen unternommen, um diese Daten sicher zu verwahren. Neben dem Schutz der Informationen vor digitalen Angreifern, spielt auch die Beständigkeit der Speichermedien eine wesentliche Rolle. Als besonders wichtig eingestufte Informationen werden auf beständigem Mikrofilm in versiegelten Fässern in gut klimatisierten Bergwerkstollen gelagert [fBuK 11]. Dienstleister vermieten spezialisierte Tresore an Firmen, die ihre Backups auf Bändern sicher verwahrt sehen wollen. Und auch Privatleute haben immer öfter ein RAID-System in ihrem Network Attached Storage oder erstellen Sicherheitskopien ihrer Fotos auf externe Datenträger. Der neueste Trend sind Storagelösungen in der Cloud, um immer und überall seine Daten verfügbar zu haben.

Doch viele Daten sind bei weitem nicht so wichtig, wie wir das zunächst annehmen. Bei wahrscheinlich jedem länger laufenden System sammelt sich im Laufe der Jahre ein ordentlicher Datenmüllberg an. Alte Software,

die von dem neuen Betriebssystem nicht mehr unterstützt wird, dazu die nicht mehr aktuellen Anleitungen und Zusatzprogramme, Links zu nicht mehr existierenden Ressourcen, Videos in nicht mehr unterstützten Formaten mit miserabler Qualität - die Liste ließe sich endlos fortsetzen. Doch im Gegensatz zu einem Umzug in der analogen Welt, der immer eine prima Gelegenheit zum Ausmisten bietet, sind Umzüge von einem Computer zu einem neuen Gerät einfacher und schneller erledigt, wenn man die gesamten Daten ungefiltert kopiert. Da die Datenspeicher auch immer größer und günstiger werden, fallen die ehemals immensen Datenberge auf der neuen Platte kaum ins Gewicht. Auch Unternehmen haben längst gelernt, dass es oft günstiger ist, immer neue Festplatten zu kaufen, als Menschen zu beschäftigen, die die alten Datenbestände neu ordnen.

Möchte man nun seine Daten im Internet aufräumen, hat man eine wahre Sisyphosarbeit vor sich. In Zeiten sozialer Netzwerke ist ein wichtiges und leider kaum funktionierend umgesetztes Prinzip des Datenschutzes, Informationen wieder aus dem Internet nehmen zu können. Das Internet wurde von Anfang an entwickelt, Informationen und den Zugang zu diesen zu erhalten. Der Schutz von Privatsphäre, ein unrechtmäßiges Vervielfachen von Software oder der Wunsch nach der Hoheitsgewalt über seine Daten, wurde damals nicht eingeplant. Die bisher angedachten Lösungen für ein Verfalldatum von Informationen und Software können noch nicht wirklich überzeugen. Zum einen können Daten nur gestreamt werden, aber Streams lassen sich mitschneiden, ein Kopieren kann daher nicht ausgeschlossen werden. Die andere Lösung arbeitet mit den Methoden des *Digital Rights Management* (DRM), die durch Verschlüsselung der Daten einen unkontrollierten Zugriff verhindern sollen. Bei diesem als *„digitalen Radiergummi“* [MB] beworbenen Verfahren benötigt man, um eine Information sehen zu können, einen Schlüssel, der nur begrenzte Zeit von einem Server abrufbar ist.

Doch wie alle anderen DRM-Systeme hat auch diese Technik Schwachstellen und dürfte daher über kurz oder lang geknackt werden. Zum einen ist es möglich, einmal abgerufene Schlüssel zu speichern und wieder zu verwenden. Zum anderen können einmal entschlüsselte Datensätze kopiert und ohne Schlüssel erneut ins Netz gestellt werden. Das größte Problem ist aber die Integration dieser Technik in bestehende Systeme. Dies erfolgt meistens über Plugins, die zumindests von bestimmten Browser- oder Betriebssystemversionen abhängig sind und extra installiert werden müssen. Eine Durchsetzung dieser Technik kann nur über einen langen Zeitraum und auf Basis einer offenen Standardisierung erfolgen. Eine physische Selbstzerstörung, aus Filmen wie „Mission Impossible“ bekannt, ist in den meisten Fällen nicht möglich und auch eine vielleicht mögliche digitale Löschung der Daten durch einen, einem Virus ähnlichen, Anhang ist eher kritisch zu sehen.

> **Die Mission** eines Datums ist die Verfügbarkeit, sofern der Anfragende die nötigen Zugriffsrechte hat. Das Entziehen dieser Rechte gestaltet sich schwierig bis unmöglich. Praktikable Lösungen, mit denen die Verfügbarkeit für nicht mehr benötigte oder zurückgezogenen Daten zuverlässig beendet werden kann, sind leider kaum realisierbar. Dennoch zeigen Konzepte wie der digitale Radiergummi ein berechtigtes Interesse, Informationen auch wieder löschen, also die Mission der Verfügbarkeit abbrechen zu können. Das Ziel ist die Schaffung sich selbst zerstörende Informationen, was aber kaum umsetzbar ist.

2.3.3. 3. Szenario: Spionagesoftware

Die wichtigste Eigenschaft von Spionagesoftware ist es, unentdeckt zu bleiben. Droht eine Enttarnung, ist es demnach wichtig, dass sich Root-Kits und trojanische Pferde selbst löschen können. Denn wenn so ein Programm erst einmal gefangen und analysiert worden ist, ist es nur eine Frage der Zeit, bis die ausgenutzten Sicherheitslücken geschlossen und die Signaturen der Virenscanner entsprechend angepasst werden. Viel schlimmer jedoch ist die Möglichkeit, dass der Spionierende enttarnt wird. Daher hat der Chaos Computer Club auch im Oktober 2011 [Chao 11] das Bundesministerium des Innern informiert, dass ihnen ein staatlich genutztes trojanisches Pferd zugespielt worden sei. Somit sollte den Kriminalämtern die Gelegenheit gegeben werden, im Einsatz befindliche Spionagesoftware aus der Ferne zu löschen, um laufende Ermittlungen nicht zu gefährden.

Gerade hier muss die Selbstzerstörung zuverlässig arbeiten, um bei einer forensischen Analyse des infizierten Computers unauffindbar zu bleiben. Neben anderen Fehlern wurde auch die Umsetzung der Löschfunktion des als Bundes- oder Staatstrojaners bekannten Spionageprogramms zur Quellentelekommunikationsüberwachung nur ungenügend implementiert. Statt den Code auf der Festplatte zu überschreiben, wurde er lediglich als

entfernt markiert. Somit war es dem Chaos Computer Club möglich, diesen Code wiederherzustellen und zu disassemblieren.

Die Mission einer heimlichen Überwachung kann als gescheitert angesehen werden, wenn sie entdeckt wird. Somit ist deren Schutz, sowie die Aufrechterhaltung einer späteren Verwertbarkeit der gesammelten Daten vor Gericht wichtiger, als die weitere Durchführung der Überwachung. Da der für den Einsatz benötigte Gerichtsbeschluss auch eine Beschränkung der Einsatzdauer vorgibt, ist es sinnvoll, neben der ferngesteuerten Abbruchfunktion auch ein vorgegebenes Ende der Mission einzubauen.

2.3.4. 4. Szenario: Sensornetze

Dank Massenproduktion und einer hohen Integrationsdichte bei Halbleitern können immer kleinere, energiesparendere, vielseitigere und preiswertere Sensoren hergestellt werden. Moderne Smartphones nehmen mit nur einer Handvoll Chips ihre Umwelt auf viele verschiedene Weisen wahr. Sie sind in der Lage, diese Informationsflut in Echtzeit zu verarbeiten, einen passenden Kontext zu bestimmen und dem Nutzer als Datensatz oder sogar als aufbereitetes Wissen verfügbar zu machen. Weiterhin zeichnen sich Smartphones durch eine Internetanbindung und diverse Nahbereichskommunikationsschnittstellen wie Bluetooth, WLAN oder Near Field Communication aus.

Reduziert man nun ein Smartphone auf einige wenige Funktionen, die ausreichen, um einen vorher klar definierten Auftrag ausführen zu können, erhalten wir kostengünstigere Knoten eines Sensornetzes. In großen Mengen ausgebracht, lassen sich so ganze Landstriche feingranular überwachen. Die Sensoren erfassen ihre unmittelbare Umgebung, versehen die gewonnenen Daten möglichst mit einem Zeitstempel und dem Lageort des Sensors und senden diese Datenpakete über das Netz an eine Auswertungsstation oder eine Fernübertragungseinrichtung. Von anderen Sensoren empfangene Sendungen werden ebenfalls weitergeleitet. Den Rest der Zeit schläft der Sensor, um Energie zu sparen. Da die Bandbreite und die Dauer des Funks stark limitiert sind, ist es notwendig, dass die Sensoren sich möglichst selbst verwalten.

Um genügend Daten zu gewinnen, aus denen sich realistische Schlüsse ziehen lassen und um das Kommunikationsnetz aufrecht zu erhalten, sind eine gewisse Menge an Sensoren nötig. Das heißt aber noch lange nicht, dass alle Sensorknoten überleben müssen, um die Mission erfolgreich durchzuführen. Es kann also sehr wohl sinnvoll sein, dass sich einzelne Sensoren selbst abschalten. Schon um Strom zu sparen, geschieht dies periodisch. Doch auch ein endgültiges Aus für einen Sensor kann gewollt sein. Wenn ein Sensor nur noch falsche oder unnötige Daten schickt, oder die Übertragung anderer Sensoren stört, sollte dieser dauerhaft deaktiviert werden. Da derartige Sensoren oft mit einer hohen Redundanz ausgebracht werden, können doppelt besetzte Positionen entstehen. Hier kann entweder der Sensor mit den unwichtigeren Daten oder mit der schlechteren Erreichbarkeit im Netzgraphen ausgeschaltet werden, oder ein Knoten wird so lange schlafen gelegt, bis die Batterie des Nachbarknoten fast leer ist. Dann kann jener den schlafenden Sensor aufwecken, um für ihn zu übernehmen, und sich daraufhin selbst endgültig zu beenden.

Sensoren, die nur noch ein oder zwei Millimeter groß sind, werden auch als Smart Dust bezeichnet [KKaFP 99], [ASSC 02]. Sie werden in großen Mengen produziert und weiträumig ausgebracht. Sie liegen oder bewegen sich in nur einigen Zentimetern Abstand zueinander, so dass sie mit nur sehr geringen Energien kommunizieren und ihre Umgebung äußerst feingranular aufgelöst überwachen können. Sind sie zu weit auseinander, können sie ihre Mission nicht mehr erfüllen, liegen sie zu nahe, kann eine Ermittlung der relativen Position eines Partikels zu seinen Nachbarn gestört werden [VVKH 00]. Es ist also sinnvoll, störende Sensoren auszuschalten. Auch bei der Produktion oder beim Transport beschädigte Smart Dust Partikel können durch eine Selbstabschaltung daran gehindert werden, fehlerhafte Daten zu verbreiten.

Gerade bei Sensornetzen für die militärische Nutzung ist die Fähigkeit der Selbstabschaltung der Knoten oft besonders wichtig. So sollten die Sensoren möglichst Funkstille wahren, um nicht vom Feind entdeckt zu werden. In Gebieten, die an die Gegenseite gefallen sind, müssen sich die Knoten ruhig verhalten oder gar selbst zerstören, damit die Technik nicht dem Feind in die Hände fällt. Aber auch bei ziviler Nutzung sollten sich Sensoren selbst zerstören, wenn sie ihr Einsatzgebiet verlassen. Wird ein Acker beispielsweise mit Smart Dust

überwacht, so will man diesen nicht in der späteren Ernte finden. Auch Sensoren, die helfen sollen, das Wetter besser zu erforschen, sollen keinen Schaden anrichten, wenn sie versehentlich vom menschlichen Körper aufgenommen worden sind. Und zwecks Überwachung von Körperfunktionen in die Blutbahn eingebrachte Sensoren sollen sich auch lieber auflösen, als ein Blutgefäß zu verstopfen.

Die Mission eines Sensors ist weitgehend durch dessen Fähigkeiten festgelegt. Ein Sensor, der seine Aufgabe nicht mehr wahrnehmen kann ist demnach nur Ballast. In Netzen von Sensoren, bis hin zum Smart Dust, ist eine hohe Redundanz meistens die Regel, mit Ausfällen wird gerechnet. Somit ist gerade in einem so massiv verteilten System eine gewisse Selbstverwaltung unerlässlich. Und besonders bei Smart Dust sollen unerwünschte Seiteneffekte für den Menschen und die Umwelt durch eine gezielte physische Selbstzerstörung vermieden werden.

2.4. Anwendungsfälle bei physischen Systemen

2.4.1. 5. Szenario: Heizung

Für Heizkörper gibt es Motor betriebene Steuerungen, die eine vorgegebene Temperatur im Raum automatisch halten. Wenn nun ein Fenster geöffnet wird, kühlt der Raum schnell ab. Um nicht nach draußen zu heizen, bleibt das Heizungsventil dicht, bis das Fenster wieder geschlossen wird. Die *Fenster-offen*-Erkennung findet bei günstigen Geräten nur über eine schnelle Änderung bei der Temperaturmessung statt. Genauer wäre sie beim Einbinden von Sensoren einer Alarmanlage, die zuverlässig geöffnete Fenster und Türen melden können.

Die Mission , die Temperatur in einem Raum konstant zu halten, ist bei einem geöffneten Fenster nicht mehr zu erfüllen. Daher ist es sinnvoll, die Mission so lange auszusetzen, bis die Bedingungen wieder besser sind.

2.4.2. 6. Szenario: Sicherheit in der Industrie

Besonders in Industrieanlagen und Rechenzentren spielt Sicherheit eine große Rolle. Einerseits ist auf den Erhalt der Systeme zu achten, andererseits müssen diese vor unberechtigtem Zugriff geschützt werden. Computerviren können wertvolle Daten und teure Hardware zerstören, Hacker Datenbanken mit sensiblen Informationen kopieren oder Kriminelle die Rechnerinfrastruktur für illegale Aktivitäten missbrauchen. So wurde der im Juni 2010 bekannt gewordene Stuxnet-Wurm wahrscheinlich geschrieben, um gezielt Zentrifugen zu beschädigen, die im iranische Atomprogramm Verwendung finden [Rieg 10]. Es wird vermutet, dass Stuxnet zuerst Programmiergeräte infizierte, um im Huckepackverfahren bei der nächsten Wartung in die nicht ans Internet angeschlossene Steuerungsanlage eingeschleppt zu werden. Hier wurde die Prozessvisualisierung soweit verändert, dass die Manipulation des Steuerungscomputers nicht mehr auffiel. Zuletzt wurden fehlerhafte Werte an die Frequenzumrichter gesendet, so dass die Zentrifugen mit für sie schädlichen Geschwindigkeiten betrieben wurden, wodurch sie sehr viel schneller als normal verschlissen.

Im Falle des Stuxnet-Wurms hätten sich die Systeme idealerweise schützen können, indem sie sich aufgrund des abnormalen Verhaltens selbst abgeschaltet hätten. Das gilt schon für das infizierte Programmiergerät, das als trojanisches Pferd missbraucht wurde, erst recht aber für die Zentrifugen, die mit für sie ungünstigen Frequenzen betrieben wurden.

Die Mission im industriellen Umfeld ist immer das Erbringen von vertraglich zugesicherten Leistungen, die meist klar definiert sind. Nur der Schutz der Arbeiter und der Erhalt der Einsatzfähigkeit haben höhere Priorität. Dabei ist auch immer auf die Wahrung der wertvollen Geschäftsgeheimnisse sowie des Datenschutzes zu achten. Industriespionage und Sabotage sind oft unterschätzte Gefahren, denen zu begegnen oft nicht einfach ist. Daher könnte eine formale Definition der Mission den Komponenten, die jene ausführen, helfen, eine Manipulation an sich zu bemerken und zu verhindern.

2.4.3. 7. Szenario: Smart Grid

Im Rahmen der Diskussionen um die Stromproduktion durch erneuerbare Energien und dem damit verbundenen notwendigen Ausbau der Stromnetze wird auch die Schaffung intelligenter Stromnetze, auch als Smart Grids bezeichnet, gefordert. Dabei sollen die Verbrauchszähler und letztendlich alle Elektrogeräte im Haushalt den momentanen Verbrauch in Echtzeit an ein zentrales Element melden. So sollen die Stromproduzenten besser auf Lastspitzen reagieren und das Stromnetz besser ausnutzen können.

Um gegen die Klimaerwärmung vorzugehen und langfristig auf den Einsatz von fossilen oder radioaktiven Brennstoffen verzichten zu können, wird immer stärker auf umweltfreundliche und regenerative Stromproduktion gesetzt. Dies stellt ganz neue Anforderungen an ein modernes Stromnetz, da statt einiger Großkraftwerke viele kleine eingebunden werden müssen, die teilweise vom Wetter und damit tages-, und jahreszeitsowie standortabhängig sind. Auch werden klassische Konsumenten zu Produzenten, wenn sie mit Hilfe staatlicher Subventionen Fotovoltaikanlagen installieren. Um Lastspitzen und Versorgungslücken ausgleichen zu können, werden auch viele, verteilte und leistungsfähige Energiespeicher benötigt. All dies erschwert eine zentrale Überwachung und Steuerung.

Einige Probleme bezüglich Datensicherheit und Schutz der Privatsphäre sind allerdings noch ungeklärt. Eine mögliche Lösung könnte eine dezentrale Auswertung der Daten sein. Wenn der Stromverbrauch übermäßig stark steigt oder die Produktion einbricht, könnten sich unkritische Geräte selbst deaktivieren, um Ressourcen für wichtigere Systeme frei zu machen.

So kommt es in den USA immer wieder zu Stromausfällen und geplanten Teilabschaltungen, wenn aufgrund einer Hitzewelle alle Klimaanlagen auf Volllast betrieben werden und die Atomkraftwerke wegen zu warmen Kühlwassers weniger Strom produzieren. Es wäre daher schön, nicht nur einzelne Gebiete, sondern einzelne Geräteklassen ausschalten zu können. So hat Unterhaltungselektronik – wie Fernsehgeräte und Spielekonsolen – eine geringere Priorität als Kühlschränke, Pumpen und Aufzüge. Auch sind Krankenhäuser und Telekommunikation wichtiger als Kinos, Diskotheken oder Wohnhäuser. Wenn die Versorgungslage über das Stromnetz mitgeteilt würde, könnten die angeschlossenen Geräte sich selbst abschalten und so vor den Folgen eines plötzlichen Stromverlustes schützen.

> **Die Mission** eines Stromnetzes ist immer gleich: alle angeschlossenen Systeme mit ausreichend elektrischer Energie zu versorgen. Doch ist dies nicht immer möglich. Es ist schon schwierig genug, die normalen Schwankungen bei Produktion und Verbrauch im Griff zu behalten. Doch extreme Wetterlagen und Naturkatastrophen können diese Mission ernsthaft gefährden. Dann müssen Verbraucher mit einer geringen Priorität sich selbst abschalten, um die stark limitierten Ressourcen für wichtigere Systeme frei zu halten.

2.5. Taxonomie von Fehlern

2.5.1. Fehlerklassen

Es gibt viele Gründe, die für das Scheitern von Missionen verantwortlich sein können. Abbildung 2.1 zeigt sechs verschiedene Klassen Fehler produzierender Faktoren. Da dies mit Hilfe dieser Arbeit nach Möglichkeiten verhindert werden soll, empfiehlt es sich, die Klassen dieser Fehler genauer zu betrachten.

Herkunft und Absicht von Fehlern

Bei Unfällen stellt sich immer die Frage, ob es sich um menschliches Versagen oder höhere Gewalt handelt. Während Ersteres vermeidbar ist, aber auch beabsichtigt sein kann, so ist ein zufälliges oder natürliches Ereignis nie zu verhindern. Um dennoch ein Maximum an Ausfallsicherheit bei Unfällen, Unwettern und Naturkatastrophen zu haben, sind Sicherheitsvorkehrungen vorzunehmen, kritische Elemente gut zu warten, und

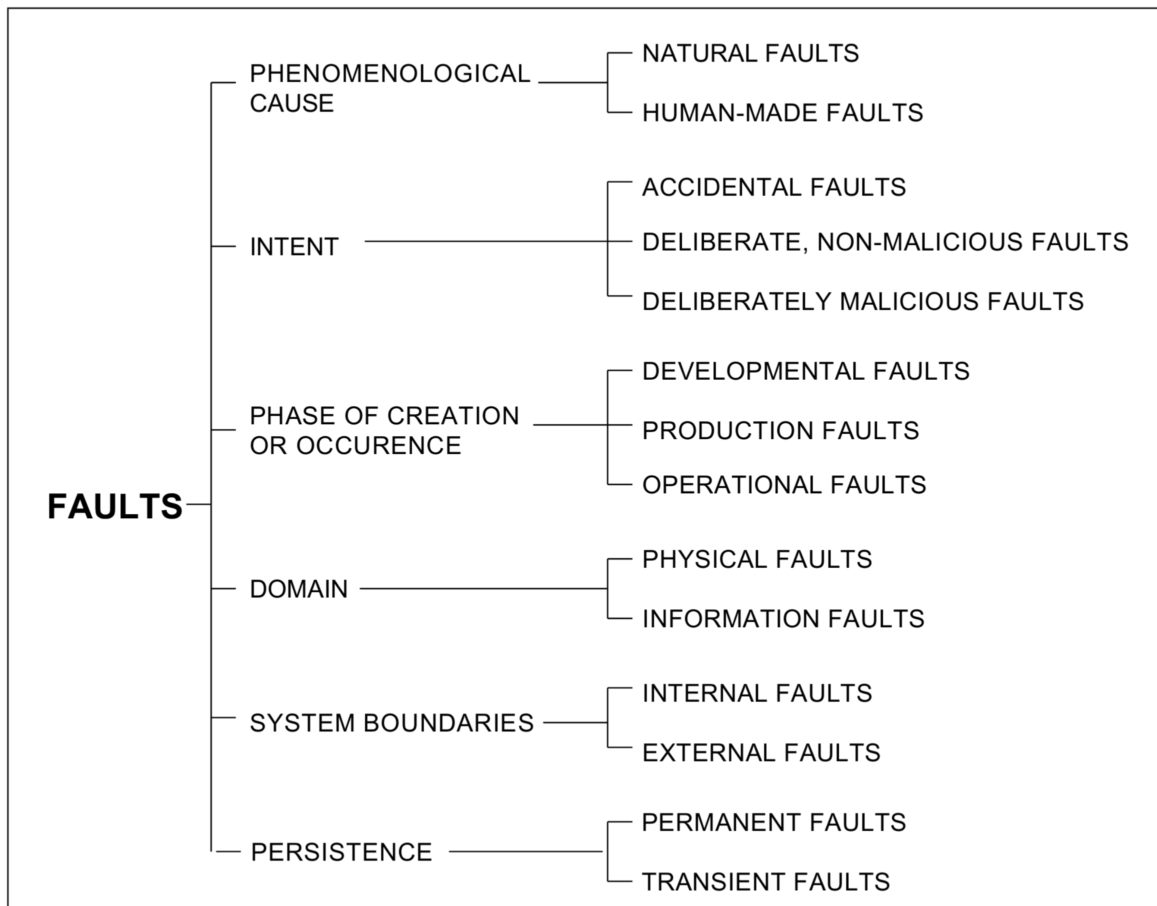

Abbildung 2.1.: Grundsätzliche Fehlerklassen aus [AcLR 01]

wenn möglich, redundant auszulegen und Notfallpläne zu entwickeln, die den Verlust von Menschenleben minimieren. So finden bei Naturkatastrophen in vielen Fällen Notabschaltungen statt, deren Ziel in erster Linie der Schutz von Menschen ist.

Fehler können aber auch Absicht sein. Einbrecher, Saboteure und Randalierer haben das Ziel, technische Anlagen zu schwächen oder zu vernichten. Dies kann mit roher physischer Gewalt, geschickter Manipulation oder auch auf digitalem Wege geschehen. So dienen Alarmanlagen und Wachschutz der Sicherheit gegen Eindringlinge in Bürogebäude und Fabrikhallen Aber auch die IT-Infrastruktur muss gegen Angriffe von außen und innen gesichert werden (***6. Szenario:*** *Sicherheit in der Industrie* (2.4.2)).

Letztendlich ist die Ursache eines Fehlers für diese Arbeit eher zweitrangig, da ein allgemeines Konzept zum Umgang mit Fehlern erstellt werden soll. Auch ob ein Fehler mit Absicht hervorgerufen worden ist, spielt abstrakt betrachtet keine Rolle, da es letztendlich nur die Art der Reaktion beeinflusst, nicht aber den Entscheidungsweg.

Vorkommensphase und Dauer

Menschliches Versagen ist eine sehr häufige Ursache für Un- und Ausfälle. Das beginnt schon beim Entwurf neuer Komponenten oder deren Vernetzung. Fehler, die in dieser Phase gemacht werden, lassen sich später kaum ausbessern. Wenn der Bau nicht ordnungsgemäß erfolgt, ist in der Regel eine aufwendige und teure Sanierung die Folge. Derartige Fehler sind meistens permanent und nur selten durch Automatismen während des Betriebs zu kompensieren.

Auch trotz einer sorgfältigen Planung und Konstruktion entstehen im Betrieb Fehler, die zum Komplettver-

sagen des gesamten Systems führen können. Oft verschwinden diese mit der Zeit wieder von alleine, was zwar positiv ist, aber wenn man die Ursache nicht findet, jederzeit erneut Probleme machen kann. Sporadisch auftretende Defekte bereiten einem mehr Kopfzerbrechen als dauerhafte, vor allem, wenn sie nicht reproduzierbar sind. Daher ist eine genaue Protokollierung der Ereignisse und ein gewissenhaftes Nachgehen Pflicht. Schon kleine Diskrepanzen können sich zu einer gewaltigen Sache auswachsen. So wurde Ende der achtziger Jahre eine Hackergruppe, die von Hannover aus für den Ostblock amerikanische Rechner ausspionierte, nur entdeckt, weil der Astronom Clifford Stoll in der Computerabteilung des Lawrence Berkeley National Laboratory einem Abrechnungsfehler von lediglich 75 US-Cent nachging [Stol 89].

Für diese Arbeit sind hauptsächlich die operativen Fehler interessant, die der Ausführung einer Mission, also deren Lebendigkeit, gefährlich werden können.

Physische und informationstechnische Fehler

Seit es Computer gibt, kann nicht nur ein Gerät physisch versagen, sondern auch auf digitaler Ebene nicht seinen Anforderungen genügen. Fehler in der virtuellen Welt sind häufig die Ursache für Sicherheitslücken, die von Schadsoftware und Hackern ausgenutzt werden, um ein System zu kompromittieren. Neben aufmerksamen Systemadministratoren und Taskforces werden auch spezielle Schutzprogramme wie Firewalls und Virenscanner eingesetzt. Sie helfen bei simplen und ungerichteten Angriffen, sowie bei bekannten Gefahren. Ist der Angreifer aber übermächtig, zum Beispiel bei einer DDOS-Attacke, helfen auch die besten Vorkehrungen und Verfahren nur wenig. Bei leisen Angriffen, die einen möglichst unbemerkten Missbrauch der Ressourcen zum Ziel haben, kann eine vorsorgliche Isolation der gefährdeten beziehungsweise gefährdenden Komponente vom System eine Option sein. So verschieben Virenscanner Software, die positiv auf Schadcode getestet worden ist, in Quarantäne. Die Schutzmaßnahmen werden in Abschnitt 4.1 näher erläutert. Wird ein Number Cruncher zur Spamschleuder oder lauscht ein Datenbankserver auf einem eigentlich geschlossenen Port, so widerspricht dies den Missionsparametern. Beim Grid Computing oder in Serversystemen im Allgemeinen spielen IT-Sicherheit und die Einhaltung von Service Level Agreements eine wichtige Rolle (***1. Szenario:*** *Missionen im Grid Computing* (2.3.1)). Gerade letzteres ist nicht immer einfach, da oft mehrere Missionen gleichzeitig erfüllt werden sollen, so dass ein geeignetes Ressourcenmanagement darüber entscheiden kann, ob ein Vertrag eingehalten wird oder nicht [SMS$^+$ 02]. Dies setzt eine ausreichend große und genaue Datenbasis voraus, um die richtigen Aufgaben zu priorisieren oder zu terminieren. Ein Beenden eines sich irregulär verhaltenden Prozesses kann Schlimmeres verhindern. Um ungewöhnliches Verhalten zu ermitteln, eignet sich das *Active Probing* [Stra 11], dessen Messwerte für die Entscheidungsfindung bezüglich eines Abschaltens der betroffenen Komponente herangezogen werden kann.

Des weiteren können Fehler zu falschen Ergebnissen führen, einen Nichtdeterminismus hervorrufen oder den Absturz des Systems zur Folge haben. Um die Mission trotz des Ausfalls eines Systems fortsetzen zu können, wird kritische Infrastruktur redundant ausgelegt. Dies findet vor allem bei günstigen Sensoren wie dem Smart Dust Anwendung (***4. Szenario:*** *Sensornetze* (2.3.4)), damit bei einem Defekt sofort Ersatz verfügbar ist (4.2.4).

Redundanz hilft auch bei Hardwarefehlern, die diverse Ursachen haben können. Neben Eindringlingen (4.2.1) sind die Naturelemente (4.2.2) ein wichtiger Punkt, wenn es um die Absicherung von Infrastruktur geht. Auch der Mangel an Betriebsmitteln (4.2.3), wie elektrischer Strom (4.2.3), Kühlung (4.2.3) oder Kommunikationsleitungen (4.2.3), kann eine Mission ernsthaft gefährden. Letztendlich kann alles zu technischem Versagen führen. Verschleiß und Überlastungen bedingen immer irgendwann den Verlust von Ressourcen, was wiederum oft den Ausfall weiterer Komponenten zur Folge hat. Gerade Stromnetze sind eine kritische Infrastruktur, deren Versagen katastrophale Zustände erzeugen kann (***7. Szenario:*** *Smart Grid* (2.4.3)). Darum ist eine gute Wartung entscheidend für die Zuverlässigkeit eines Systems.

Systemgrenzen

Ein Problem bei Fehlern ist, dass diese nicht nur innerhalb von Systemgrenzen, sondern auch außerhalb auftreten können. Die meisten Maßnahmen zur Beseitigung von Defekten können aber nur innerhalb des zugehörigen Systems angewendet werden, da sie für andere Domänen nicht zuständig sind. Das ist teilweise auch so gewollt, gerade wenn es sich um politische Systeme wie Staaten handelt. So soll nur die berechtigte Stelle Zugriff auf Überwachungstechnik haben, um Missbrauch vorzubeugen (***3. Szenario:*** *Spionagesoftware*

(2.3.3)). Auch Unternehmen sehen es nur ungern, wenn sich die Konkurrenz in ihre Angelegenheiten einmischt (***6. Szenario:*** *Sicherheit in der Industrie* (2.4.2)). Doch bei vernetzten technischen Systemen, wie zum Beispiel dem Internet, ist es unerlässlich, dass alle Teilnehmer zusammenarbeiten. Einige wenige Protokolle können über Systemgrenzen hinweg agieren, allerdings nur recht eingeschränkt. Bei Störungen hilft dann nur ein Unterstützungsgesuch bei der Administration des betroffenen Teilnetzes (***1. Szenario:*** *Missionen im Grid Computing* (2.3.1)). Gerade das Garantieren von Leistungen ist besonders schwer, wenn zusätzliche Parteien involviert sind, auf deren Technik man selbst keinen Einfluss hat [BSC 99]. Angenehmer wäre da eine Möglichkeit, automatisiert in einem fremden Teilnetz eine Fehlerbehandlung anregen zu können.

Dies ist ein interessanter Punkt, da klassische zentralistische Kontrollsysteme nicht in fremde Zuständigkeitsbereiche eindringen können, selbst wenn der Fehler dort ausgemacht worden ist. Würden sich die Komponenten des Systems miteinander über eine standardisierte Schnittstelle absprechen und autonom reagieren, ließen sich Störungen viel frühzeitiger und gezielter beheben. Dies gilt besonders, wenn auch externe Sensoren zur Steuerung der Mission herangezogen werden können, wodurch sich die Genauigkeit bei der Erkennung von Fehlern verbessern lassen würde.

2.5.2. Arten von Fehlern

Auch die möglichen Auswirkungen von Fehlern sind vielfältig. Abbildung 2.2 unterteilt sie grob in drei Arten.Manchmal können kleinere Fehler ignoriert werden, da keine wichtigen Systeme betroffen sind und andere Teile die defekten Stellen kompensieren können (***4. Szenario:*** *Sensornetze* (2.3.4)). Sind die Ausfälle jedoch schwerwiegender, kann ein schnelles Eingreifen Schlimmeres verhindern.

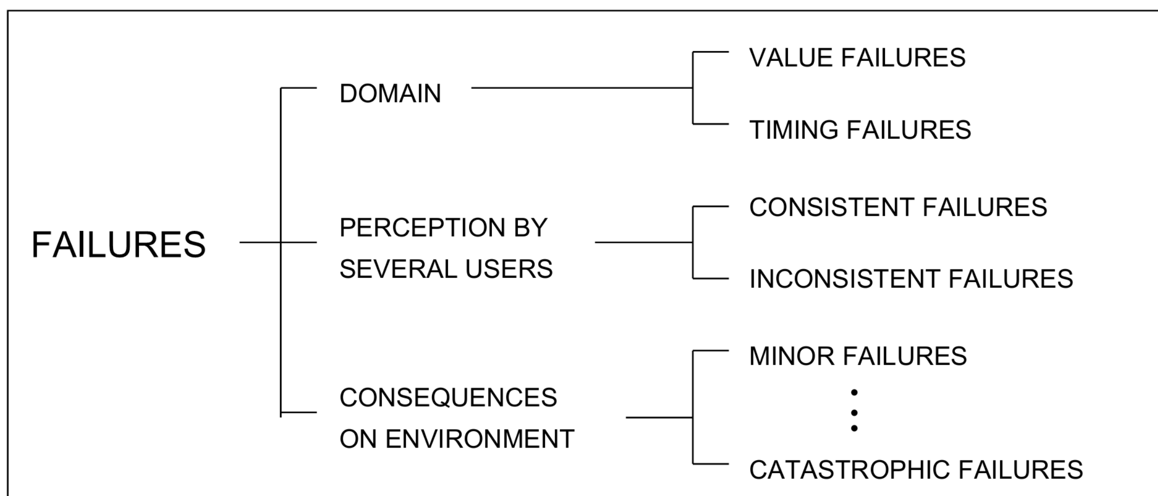

Abbildung 2.2.: Arten des Versagens aus [AcLR 01]

Neben einem zu überwachenden Wert ist auch der Zeitpunkt wichtig, an dem er nicht mehr stimmt oder verfügbar ist. Es ist also notwendig, auch den Kontext bei Entscheidungen, wie mit einem Fehler verfahren werden soll, heranzuziehen. So können bestimmte Informationen nur in bestimmten Bereichen, nur zu gewissen Zeiten oder nur gegenüber einigen vertrauenswürdigen Personen freigegeben sein (***2. Szenario:*** *Verfalldatum bei Informationen* (2.3.2)). Der Kontext muss sich demnach messen und automatisiert verarbeiten lassen. Hierbei spielt auch der Nutzer, beziehungsweise die von ihm gestartete Mission, eine Rolle. So kann ein Fehler nur eine Mission betreffen oder alle. Vielleicht ist auch ein für eine Mission gültiges Datum für eine andere Mission ein Fehler.

Immer zu bedenken sind auch die Auswirkungen eines Fehlers auf eine Mission. Wenn die Gegenmaßnahmen schlechter für deren Lebendigkeit sind, als der Fehler selbst, ist von Reaktionen abzusehen oder sie sind entsprechend zu ändern.

2.6. Zusammenfassung

Systeme werden geschaffen, um Aufgaben zu erfüllen (2.1). Verteilt sich das System auf viele Komponenten, die alle ihren Teil zur Bewältigung der Mission beisteuern, erhöht sich die Komplexität und damit die Gefahr, dass die Mission scheitert, deutlich (Abschnitt 2.2). Ein Fehler in einer Komponente beeinflusst nicht nur diese, sondern auch alle anderen, die mit der Komponente verbunden und von ihr abhängig sind.

Die vorgestellten Szenarien (2.3 und 2.4) zeigen Missionen verteilter Systeme auf, bei denen die einzelnen Komponenten in der Regel nicht wissen, warum sie etwas tun oder lassen sollen, da ihnen der Überblick über das Gesamtsystem fehlt. Sie können nicht bemessen, inwieweit ihre Aktionen der Mission aller nutzen oder schaden. Doch selbst wenn sie abschätzen könnten, welche Auswirkungen ihr Handeln auf andere Komponenten hat, so ist es ihnen meistens nicht gestattet, dieses zu ändern.

Eine Lösung könnte ein überwachendes und im Gefahrenfall eingreifendes Framework bringen, welches eine Komponente beeinflussen und sogar abschalten, beziehungsweise aus dem Verbund, der eine Mission erfüllt, abziehen kann. So ließen sich durch Fehler hervorgerufene negative Einflüsse auf die Lebendigkeit der Mission vermeiden oder zumindest die Auswirkungen minimieren. Derartige Fehler wurden ebenfalls in diesem Kapitel in Abschnitt 2.5 vorgestellt.

Doch wie könnte so ein Framework aussehen und welchen Anforderungen muss es gewachsen sein? Wie lässt sich damit eine Mission sichern? In dieser Arbeit soll ein Konzept für ein Framework erarbeitet werden, mit dem sich diese Fragestellungen untersuchen lassen. Hierfür werden zunächst in Kapitel 3 einige Anwendungsfälle formal besprochen (3.3), aus denen dann Anforderungen an das Framework abgeleitet werden (3.4).

3. Anforderungsanalyse

Um den Gefahren für eine Mission begegnen zu können, scheint eine Kontrollinstanz sinnvoll, die die verwendeten Ressourcen überwacht und im Fehlerfall korrigierend eingreift. Eine mögliche Reaktion kann das kontrollierte Abschalten eines Gerätes sein, um von diesem, sowie der Mission, Schaden fern zu halten. Als überwachendes Element könnte ein Framework dienen, das Informationen sammelt und darauf basierend Entscheidungen bezüglich einer vorgegebenen Mission trifft, um diese zu schützen. Zunächst werden in Abschnitt 3.1 die verwendeten Begrifflichkeiten geklärt, bevor in Abschnitt 3.2 das zu erreichende Ziel bestimmt wird. Hierzu werden in Abschnitt 3.3 formale Anwendungsfälle beschrieben und analysiert, um in Abschnitt 3.4 unter Verwendung der Szenarien aus Kapitel 2 die Anforderungen an das Framework zu stellen, dessen Konzept in Kapitel 6 vorgestellt wird.

3.1. Begriffsklärung

Ein *System* besteht aus vielen verteilten *Komponenten*, die untereinander über ein Netz kommunizieren können. Wird dem System nun eine Aufgabe gestellt, wird diese an die Teilnehmer, je nach ihren Fähigkeiten, verteilt. Jene versuchen nun den ihnen zugewiesenen Teil der *Mission* des Gesamtsystems zu erfüllen.

Große Netze mit einer hohen Dynamik werden schnell zu komplex, um sie von einem zentralen Element zu steuern. So ist ein Mobilfunknetz in kleinere Teilnetze, die Location Update Areas, unterteilt, die sich weitgehend selbst verwalten können [RaSt 96]. Sie bestehen ihrerseits wieder aus einzelnen Funkzellen, die die eigentliche Mission, die Kommunikation der Teilnehmer, erfüllen. Dies ist eine Analogie zu den biologischen Zellen, die die Mission haben, ihren Wirtskörper am Leben zu erhalten. Dabei sind sie meist spezialisiert, um ihren Teil der Aufgabe wahrnehmen zu können.

Der Begriff der *Zelle* stellt eine Abstraktionsebene dar, die zwischen das System und die Komponenten, die das System bilden, gezogen wird. Eine Zelle repräsentiert eine Komponente mit ihren Fähigkeiten und Eigenschaften gegenüber der Mission, die ein System erfüllt. Somit kann man im Rahmen dieser Arbeit eine Zelle als eine Komponente im Kontext einer Mission begreifen.

Dies soll auch helfen, die Analogie zur Natur anschaulicher zu machen. Wenn man einen menschlichen Körper als ein System sieht, das Befehle vom Gehirn umzusetzen hat, dann sind die Zellen die einzelnen Komponenten, aus denen der Körper besteht und die letztendlich die Mission erfüllen. Dabei kennt eine Zelle das große Programm, welches in Form von DNA im Zellkern vorliegt, und allen anderen Missionen zugrunde liegt: Überleben. Allerdings sind die Befehle, denen eine Zelle folgt, speziell für diese ausgelegt, die Befehle an andere Zellen und deren Teilaufgaben im Körper tangieren sie nicht.

Die technischen Zellen in dieser Arbeit sind Komponenten, die um ein Framework erweitert werden, welches ständig überprüft, ob die Mission noch erfüllt oder ihr zuwider gehandelt wird. Jede Zelle hat dabei genau ein solches Framework, dessen Entwicklung ein zentraler Aspekt dieser Arbeit ist.

Eine *Zelle* ist demnach ein Teil eines Systems. Sie kann sich weitgehend selbst verwalten, aber auch auf Einflüsse von außen reagieren sowie selbst Nachrichten an ihre Umgebung senden. Eine Zelle kann dabei einer Komponente entsprechen, aber auch eine andere Granularität aufweisen. Um ihre Mission erfüllen zu können, benötigt sie *Ressourcen*. Das kann ein unintelligentes Betriebsmittel wie elektrischer Strom sein, aber auch Fähigkeiten oder Informationen von anderen Zellen. Im einfachsten Fall entspricht eine Zelle einer Komponente, die ihren Teil einer Mission erfüllt und somit eine Ressource für das System darstellt. Man könnte eine Zelle ihrerseits als ein System aus vielen Zellen betrachten. Dies lässt sich beliebig lang rekursiv fortsetzen.

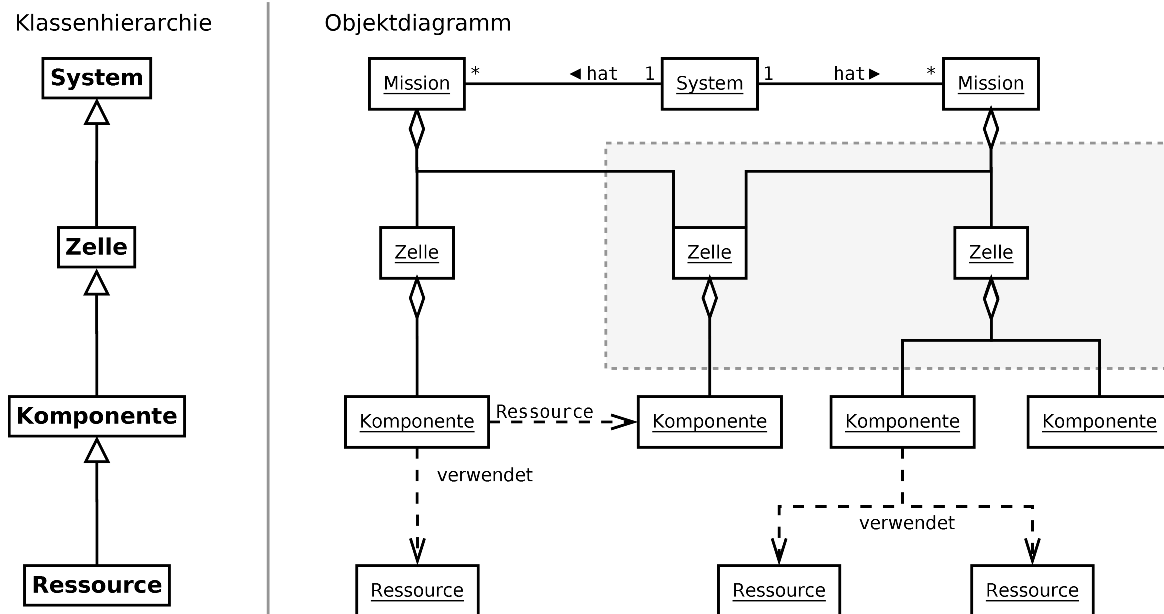

Abbildung 3.1.: Jede **Komponente** ist genau einer **Zelle** zugeordnet

Eine Mission ist also die Aufgabe, die ein System zu erledigen hat. Dabei kann diese auf die Komponenten aufgeteilt werden, die ihren Fähigkeiten entsprechend Teile der Aufgabe erledigen. Ein System kann durchaus auch mehrere Missionen gleichzeitig erfüllen, wobei je Mission nur ein Teil des Systems beansprucht wird.

Es gibt zwei mögliche Betrachtungsweisen für den Begriff der Zelle. So kann sie eine Komponente sein, die bereit ist, eine Aufgabe zu übernehmen. In Abbildung 3.1 sieht man im grau hinterlegten Teil, dass eine Zelle eine Komponente zwei unterschiedlichen Missionen gegenüber repräsentiert, gleichzeitig aber auch mehrere Komponenten zugleich aggregieren kann. Somit kann sie gleichzeitig mehrere Missionen unabhängig voneinander ausführen, solange es ihre Kapazitäten erlauben.

Weiterhin kann eine Komponente gegenüber einer anderen als Ressource auftreten. Oft benötigt eine Komponente dabei mehrere Betriebsmittel gleichzeitig. Der Fall, dass eine Zelle ihrerseits ein System von Subzellen ist und ihre Mission an diese weiterverteilt, wurde wegen der Übersichtlichkeit nicht dargestellt, lässt sich aber aus dem vereinfachten Klassendiagramm ableiten.

Das Leben einer Zelle endet erst, wenn sie keine Komponenten mehr repräsentiert. Allerdings ist eine Zelle ohne eine einzige Mission denkbar. Sollte eine Komponente die Bearbeitung einer Mission einstellen, so zieht sie sich aus dieser zurück, was zur Folge hat, dass die Zelle aus Sicht der Mission zu existieren aufhört.

Andererseits lässt sich eine Zelle auch als eine Komponente mit genau einer Mission sehen, wie in Abbildung 3.2 gezeigt. Gegenüber der vorherigen Grafik hat sich nur der grau hinterlegte Ausschnitt verändert. Eine Komponente wird gegenüber jeder Mission, der sie zugeteilt ist, durch eine eigenständige Zelle vertreten. Eine Zelle kann also nicht mehreren Mission gleichzeitig angehören. Des weiteren entspricht eine Zelle genau einer Komponente.

Da eine Zelle gleich einer Komponente im Rahmen einer Mission ist, existiert sie auch nur für die Dauer einer Mission. Wenn eine Komponente die Arbeit an einer Mission aufgibt, endet automatisch das Leben der Zelle. Sollte die Komponente der Mission erneut zugeteilt werden, so schafft sie damit eine neue Zelle, die den Platz der alten einnimmt.

Beide Betrachtungsweisen unterscheiden sich hauptsächlich in der Implementierung der Verwaltung. Aus Sicht der Mission spielt es hingegen keine Rolle, ob eine Zelle eine Komponente ist, die im Rahmen ihrer Fähigkeiten beliebig viele Mission bearbeitet, oder eine Komponente im Kontext einer bestimmten Mission darstellt. Sie sieht die Zelle immer nur solange diese ihr zugeteilt ist, alles andere existiert für sie nicht. Solange sich die Missionen nicht gegenseitig blockieren, da sie um die selben Ressourcen konkurrieren, können diese

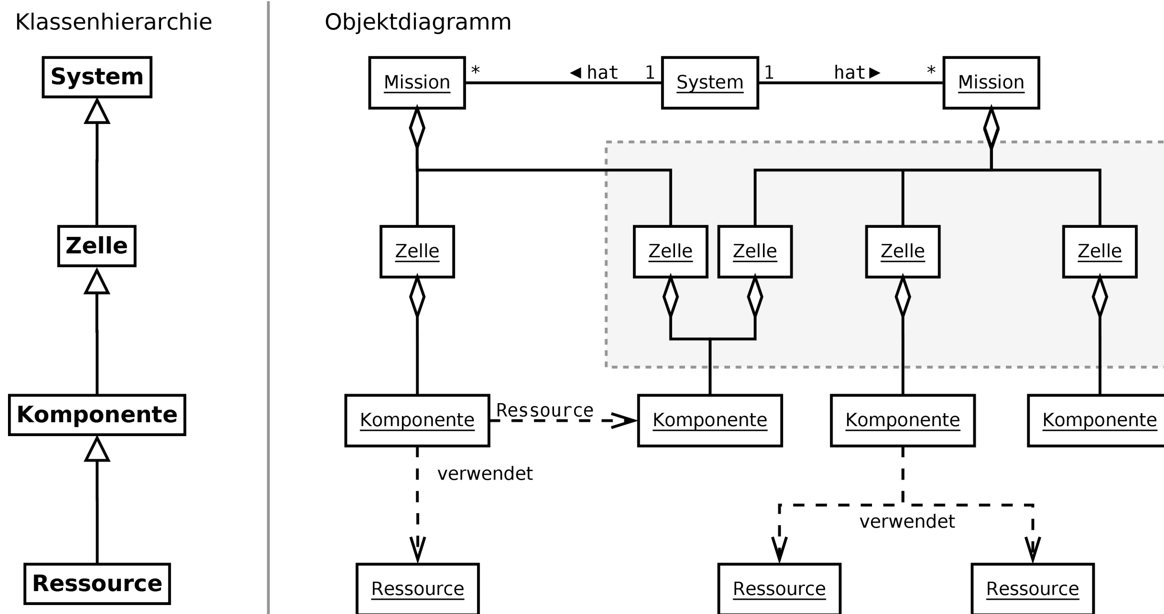

Abbildung 3.2.: Jede **Zelle** ist genau einer **Mission** zugeordnet

auch isoliert betrachtet werden. Dies soll zur Vereinfachung im Folgenden als gegeben angenommen werden, weshalb immer nur ein System mit nur einer Mission betrachtet werden wird. Somit repräsentiert eine Zelle immer eine Komponente gegenüber einer Mission. Zelle, Komponente, Ressource, Teilnehmer und Teilsystem werden im Folgenden weitgehend synonym verwendet, ausgenommen, wenn von Ressourcen im Sinne von Rohstoffen die Rede ist.

3.2. Missionserhalt ist das Ziel

Wie bereits im vorherigen Kapitel 2 festgestellt worden ist, hat der Erhalt der Mission eines Systems oberste Priorität. Dies bedeutet allerdings nicht zwangsläufig den Erhalt der einzelnen Komponenten. Im Gegenteil, deren Opfer kann durchaus positiv für den Fortbestand der Mission sein.

Um die Fälle, in denen die Anwesenheit einer Komponente dem Missionserhalt mehr schadet als nutzt genauer untersuchen zu können, ist ein Framework notwendig, das alle relevanten Daten sammelt, bewertet und eine Entscheidung bezüglich der Abschaltung einer Zelle trifft. Eine Zelle ist dabei eine Komponente, die in einem System mit einer Mission eingebunden ist, also eine logische Einheit eines eine Mission erfüllenden Verbundes. Jede Zelle kennt ihren Platz in diesem Zellverbund und damit ihre Aufgabe im Sinne der Mission. Sie überwacht sich selbst und kann sich selbst terminieren. Dies bedeutet, dass die Zelle nicht mehr länger dem Verbund zur Verfügung steht, aber nicht unbedingt, dass die Ressource, die durch die Zelle repräsentiert wurde, ebenfalls beendet ist. Falls es technisch möglich und sinnvoll ist, kann sie neu gestartet oder fortgesetzt werden und somit als neue Zelle in den Verbund reintegriert werden.

Wenn eine Zelle, beziehungsweise die mit ihr assoziierte Ressource, mehrere Aufgaben gleichzeitig erfüllen soll, so bedeutet ein Abschalten nicht zwangsläufig, dass die anderen Missionen davon betroffen sind. Ähnlich dem ACID-Prinzip, das bei der Verwaltung von Datenbanken Anwendung findet, soll eine logische Unabhängigkeit der einzelnen Missionen gewahrt bleiben. Diese können analog zu Transaktionen in einem Datenbankmanagementsystem gesehen werden, so dass auch für eine Mission die Bedingung der *Isolation* gilt.

Da es sich bei dem überwachenden Framework nicht um eine Datenbank handelt, die Informationen halten soll, spielen die anderen Punkte von ACID – Atomarität der Transaktionen, Konsistenz und Dauerhaftigkeit der Daten – nur eine untergeordnete Rolle. Tatsächlich ist es fraglich, ob Konzepte wie *Commit* und *Rollback* bei einem überwachenden Framework überhaupt gebraucht werden. Ein Rückgängigmachen des Abziehens

einer Zelle von einer Mission ist oft nur nach einem regulierenden Eingreifen möglich, dann auch gerne mit veränderten Parametern. Dies entspricht aber eher dem Hinzufügen einer neuen Zelle zur Mission. Lediglich die Atomarität von Ereignissen, wie dem Belegen oder Freigeben von Ressourcen, ist interessant, wenn es auch auf einer anderen logischen Ebene als der des Frameworks abläuft. Jenes könnte aber beim Scheitern einer Anforderung von einer unbedingt benötigten Ressource Konsequenzen für den Fortbestand der Zelle haben.

Wenn eine Zelle sich selbst abschaltet, so ist sie für die anderen Komponenten, die der gleichen Mission folgen, verloren. Dennoch kann die Zelle im Rahmen anderer Missionen weiter bestehen. Es stellt sich die Frage, ob eine Zelle immer genau eine Komponente im Rahmen von genau einer Mission entspricht, oder ob eine Zelle immer gleich einer bestimmten Komponente ist, die beliebig viele Missionen erfüllen kann. Der größte Unterschied liegt dabei in der Implementierung der Zellenverwaltung. Zur Vereinfachung soll das Abschalten einer Zelle immer im Kontext einer Mission betrachtet werden. Ob sie weiterhin anderen Missionen zur Verfügung steht, oder komplett ausfällt ist aus Sicht der Mission unerheblich. Nur wenn Schaden von der Hardware abgehalten werden soll, beziehungsweise diese nicht mehr arbeitsfähig ist, bedeutet eine Terminierung den tatsächlichen Ausfall der Ressource für alle auf ihr laufenden Missionen, ohne die Chance auf eine baldige, vollautomatische Wiederkehr in den Zellverbund.

3.3. Analyse der Anwendungsfälle

Die in Kapitel 2 aufgezeigten Szenarien zeigen, dass sich die Zellen dank *„teile und herrsche“* oft selbst regulieren können. Die dazu erforderlichen Vorgänge sind abstrakt gesehen relativ einfach erfassbar. Nach einer Initialisierung einer Zelle führt diese ihre Aufgabe aus bis sie wieder beendet wird. Dabei wirken auf eine Komponente im Kontext einer Mission neben der Zelle selbst drei weitere Aktoren ein, die sich aus den beschriebenen Szenarien ableiten lassen.

Der Auftragsteller ist eine Entität, die die Mission definiert, an die Komponente weiterreicht und die Ergebnisse, sofern welche vorliegen, einsammelt. Dabei kann es sich um einen Menschen beziehungsweise eine *virtuelle Organisation* (VO) oder um ein übergeordnetes System handeln. Letzteres vertritt dann gegenüber der Komponente die VO und passt die globale Missionsbeschreibung an die Eigenschaften der Komponente an.

Ohne den Auftragsteller gäbe es keine Mission und damit keinen gültigen Kontext in dem eine Komponente existieren könnte.

Eine andere Komponente mit der selben Mission entspricht einer benachbarten Zelle, die am selben Auftrag wie die betrachtete Komponente arbeitet. Sie steht für alle anderen Zellen im Verbund, mit denen die Komponente Nachrichten austauscht. Wenn sie in der Lage ist, kann sie die Arbeit einer ausfallenden Komponente übernehmen, aber auch selbst bei einem Fehler um Hilfe bitten.

Ohne wenigstens eine weitere Komponente gäbe es kein verteiltes System, was die betrachteten Szenarien trivial oder gar unmöglich machen würde.

Ein negativer Einfluss auf die Mission ist meistens ein Fehler oder der Ausfall einer wichtigen Ressource. Mögliche Gründe für das Versagen von Technik wurden in Abschnitt 2.5 erläutert. Eine Komponente soll nun in die Lage versetzt werden, derartige Gefahren für das gesamte System zu verhindern, indem sie sich selbst aus der Gleichung nimmt. Eine Bedrohung für die Mission kann entweder von außen kommen, wie das bei Angreifern oder Naturkatastrophen der Fall ist, oder in und durch die Komponente selbst erzeugt werden, zum Beispiel durch einen Virus, technisches Versagen oder aufgrund einer Fehlkonfiguration. Auch eine Aufforderung zur Selbstabschaltung durch eine oder mehrere andere Zellen kann als ein negativer Einfluss auf die Mission gewertet werden.

Ohne ein negatives Ereignis, das die Fortsetzung der Mission eines Systems bedroht, gäbe es keinen Grund, Maßnahmen für einen Fall, wie er auch in den Szenarien beschrieben ist, zu entwickeln und zu testen.

3.3.1. Definieren der Mission

Einem komplexen, verteilten System werden Aufgaben gestellt. Dies kann bereits während der Planungsphase geschehen, aber auch erst später in Form von Software einem *General-Purpose*-System mitgeteilt werden. Während sich bei einem für einen bestimmten Zweck geschaffenen Apparat die Definition der Aufgabe nur wenig ändert, so kann dies bei einem von Programmen gesteuerten Computer ständig passieren.

In der Abbildung 3.3 wird eine solche Mission formal definiert. Sie soll der Maschinerie beschreiben, was ihre Aufgabe ist und an welche Regeln sie sich zu halten hat. Dazu werden Messpunkte wie Sensoren, Prozessinformationen und Kommunikationsschnittstellen anderer Komponenten angegeben, die für die Aus- und Bewertung des eigenen Zustands einer Zelle und des Systems herangezogen werden. Mit Hilfe einer ebenfalls vorgegebenen Logik erfolgt die Evaluierung und ein Vergleich mit den Parametern, die die Grenzwerte der Mission darstellen. Werden jene über- oder unterschritten, kann die festgelegte Reaktion zur Ausführung kommen. Dabei ist es notwendig, die Missionsdefinition an die jeweilige Komponente anzupassen. Diese kann manuell erstellt, oder über eine Konvertierung aus einer generischen und folglich abstrakteren Missionsbeschreibung extrahiert werden.

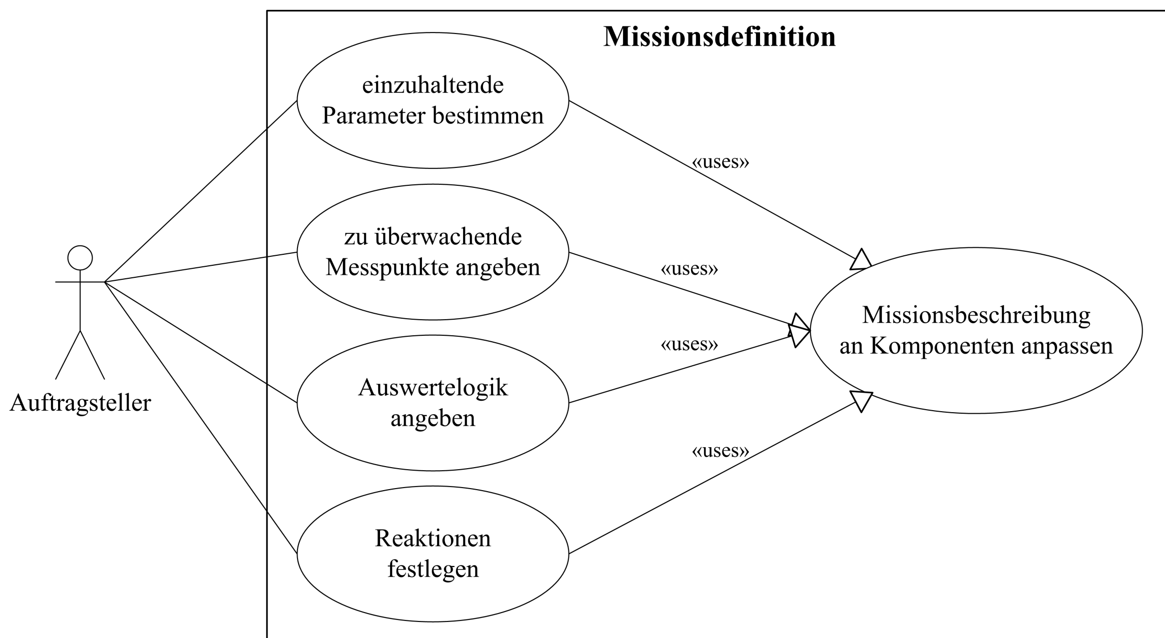

Abbildung 3.3.: Definieren einer Mission

Anwendungsfall: Einzuhaltende Parameter bestimmen

Kurzbeschreibung	Es werden Parameter angegeben, die die Grenzen einer korrekten Mission festlegen. Inkludiert *„Anwendungsfall: Missionsbeschreibung an Komponenten anpassen“*.
Vorbedingung	Valide Parameter mit Einheit.
Nachbedingung	Valide Parameter mit Einheit, die die Grenzen der Mission für eine Komponente festlegen.
Primärszenario	Es werden Parameter einer Mission in eine für die jeweilige Zelle angepasste Datenstruktur geschrieben und an die entsprechende Kontrollinstanz beim Start der Arbeit übertragen.
Sekundärszenario	Eine Komponente kann die verlangten Bedingungen und damit die Mission nicht erfüllen. Nun kann die Mission angepasst, ein Ersatz für die Zelle oder deren Modifikation veranlasst werden. Dies wiederholt sich bis die Mission durchgeführt werden kann.

Anwendungsfall: Auswertelogik angeben

Kurzbeschreibung	Es wird eine Logik angegeben, die missionsrelevante Informationen auswertet, um Grenzüberschreitungen festzustellen. Inkludiert *„Anwendungsfall: Missionsbeschreibung an Komponenten anpassen"*.
Vorbedingung	Valide Auswertelogik mit Angabe der Datenquellen.
Nachbedingung	Valide Auswertelogik mit Datenquellen, die konkret zu der jeweiligen Komponente gehören.
Primärszenario	Es wird eine Beschreibung einer Auswertelogik in eine für die jeweilige Zelle angepasste Datenstruktur geschrieben und an die entsprechende Kontrollinstanz beim Start der Arbeit übertragen.
Sekundärszenario	Eine Komponente kann die verlangten Bedingungen und damit die Mission nicht erfüllen. Nun kann die Mission angepasst, ein Ersatz für die Zelle oder deren Modifikation veranlasst werden. Dies wiederholt sich bis die Mission durchgeführt werden kann.

Anwendungsfall: Reaktionen festlegen

Kurzbeschreibung	Es werden Reaktionen bestimmt, die ausgelöst werden, wenn die Mission ihre Grenzen überschreitet. Inkludiert *„Anwendungsfall: Missionsbeschreibung an Komponenten anpassen"*.
Vorbedingung	Valide Reaktionen mit Auslösebedingungen.
Nachbedingung	Valide Reaktionen für eine Komponente mit passenden Auslösebedingungen.
Primärszenario	Es wird eine Beschreibung einer Reaktion in ein für die jeweilige Zelle angepasstes Format geschrieben und an die entsprechende Kontrollinstanz beim Start der Arbeit übertragen.
Sekundärszenario	Eine Komponente kann die verlangten Bedingungen und damit die Mission nicht erfüllen. Nun kann die Mission angepasst, ein Ersatz für die Zelle oder deren Modifikation veranlasst werden. Dies wiederholt sich, bis die Mission durchgeführt werden kann.

Anwendungsfall: Zu überwachende Messpunkte angeben

Kurzbeschreibung	Es werden Messpunkte bestimmt, die relevante Daten über den aktuellen Zustand einer Mission in einer Komponente liefern. Inkludiert *„Anwendungsfall: Missionsbeschreibung an Komponenten anpassen"*.
Vorbedingung	Valide Messpunkte mit allen für eine Abfrage notwendigen Daten (URI, Datenformat, Nomalisierungsfunktion, etc.).
Nachbedingung	Valide Messpunkte für eine Komponente mit allen notwendigen Daten.
Primärszenario	Es wird eine Reihe von Messpunkten relativ zur jeweiligen Komponente bestimmt und an die entsprechende Kontrollinstanz beim Start der Arbeit übertragen.
Sekundärszenario	Eine Komponente kann die verlangten Bedingungen und damit die Mission nicht erfüllen. Nun kann die Mission angepasst, ein Ersatz für die Zelle oder deren Modifikation veranlasst werden. Dies wiederholt sich bis die Mission durchgeführt werden kann.

Anwendungsfall: Missionsbeschreibung an Komponenten anpassen

Kurzbeschreibung	Konvertiert die Missionsbeschreibung in ein für die betreffende Komponente brauchbares Format. Dabei werden abstrakte Referenzen und Daten konkretisiert. Nicht benötigte Definitionen werden entfernt.
Vorbedingung	Alle für die Mission bezüglich einer Komponente wichtigen Daten sind auswertbar.
Nachbedingung	Nur die relevanten Daten sind in einer auf die konkrete Komponente angepassten Form.
Primärszenario	Referenzen werden auf reale Messpunkte umgesetzt und die Daten in eine für die Komponente verständliche und valide Form gebracht.
Sekundärszenario	Eine Referenz kann nicht aufgelöst werden oder ein Datum ist nicht valide. Die Missionsbeschreibung muss angepasst oder eine andere Komponente gesucht werden.

3.3.2. Aufragserteilung und Missionsbearbeitung

Abbildung 3.4 zeigt, wie eine Mission an ein verteiltes System übergeben wird, das diese dann ausführt. Zunächst wird der Auftrag an die einzelnen Komponenten weiter gereicht. Dabei kann es sein, das die Missionsdefinition [3.3.1] an die Fähigkeiten und die Teilaufgabe der jeweiligen Komponente angepasst werden muss.

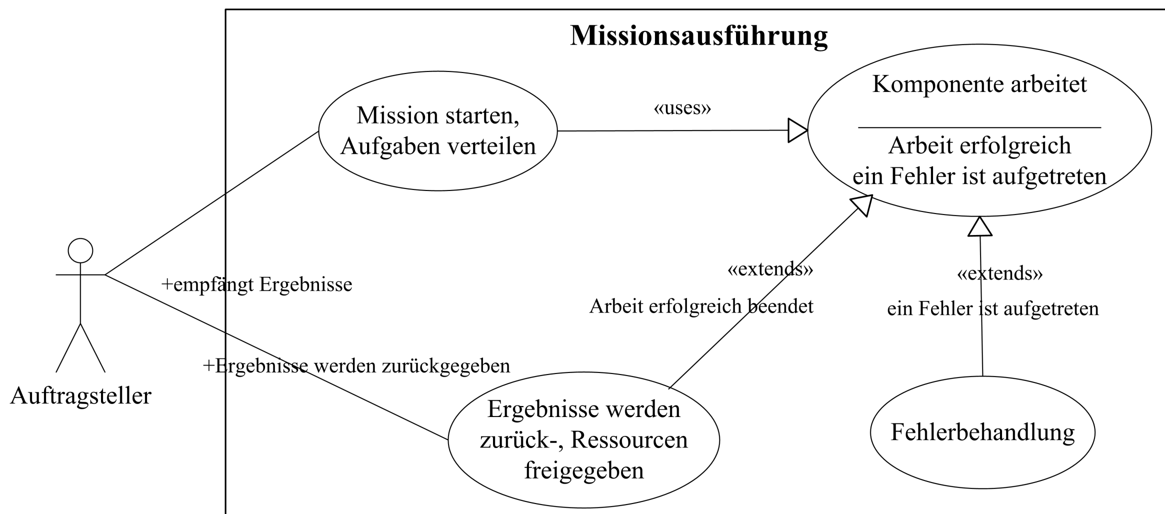

Abbildung 3.4.: Eine Mission wird gestartet und ausgeführt

Anwendungsfall: Mission starten, Aufgaben verteilen

Kurzbeschreibung	Startet die Mission und verteilt die Mission an die einzelnen Komponenten des Systems. Unter Umständen *„Anwendungsfall: Missionsbeschreibung an Komponenten anpassen“*. Inkludiert *„Anwendungsfall: Komponente arbeitet“*
Vorbedingung	Eine Entität, wie ein Mensch oder ein übergeordnetes System, hat die Zugriffsberechtigung für und Kenntnisse über die Schnittstellen, um eine Mission, die zuvor definiert worden ist, zu starten. Alle benötigten Komponenten sind verfügbar und einsatzbereit.
Nachbedingung	Die Mission wird von allen Komponenten bearbeitet.
Primärszenario	Die Mission startet und führt ihre Aufgabe aus.
Sekundärszenario	Die Mission kann nicht gestartet werden. Eine Fehlerbehandlung wird ausgeführt.

Anwendungsfall: Komponente arbeitet

Kurzbeschreibung	Die Komponente arbeitet an der Mission. Erweitert *„Anwendungsfall: Ergebnisse werden zurück-, Ressourcen freigegeben“* und *„Anwendungsfall: Fehlerbehandlung“*.
Vorbedingung	Komponente wurde korrekt initialisiert.
Nachbedingung	Ergebnisse liegen vor.
Primärszenario	Die Komponente führt die Mission aus und erweitert *„Anwendungsfall: Ergebnisse werden zurück-, Ressourcen freigegeben“*, falls sie normal beendet wird.
Sekundärszenario	Ein Fehler tritt auf, die Mission wird abgebrochen. Erweitert *„Anwendungsfall: Fehlerbehandlung“*.

Anwendungsfall: Ergebnisse werden zurück-, Ressourcen freigegeben

Kurzbeschreibung	Die Arbeit an der Mission wird normal beendet. Siehe auch *„Anwendungsfall: Beendet Auftrag“*.
Vorbedingung	Komponente bekommt ordentlichen Terminierungsbefehl.
Nachbedingung	Die Komponente ist wieder in Bereitschaft.
Primärszenario	Die Komponente gibt die Ergebnisse an das aufrufende System zurück und von ihr gebundene Ressourcen frei. Sie meldet sich als verfügbar für weitere Missionen.

Anwendungsfall: Fehlerbehandlung

Kurzbeschreibung	Behandelt Fehler, die während der Abarbeitung der Mission in der Komponente aufgetreten sind. Siehe auch *„Anwendungsfall: Abschaltung der Komponente zum Schutz der Mission“*.
Vorbedingung	Die Mission ist mit einem Fehler gescheitert.
Nachbedingung	Falls die Komponente noch arbeitsfähig ist, geht sie wieder in Bereitschaft.
Primärszenario	Eine für den gegebenen Fehlerfall festgelegte Reaktion wird ausgeführt und ins Protokoll geschrieben. Benachbarte Komponenten werden über den Fehler informiert. Wenn dies die Mission nicht wieder erfüllbar macht, werden die gebundenen Ressourcen freigegeben und falls möglich, Teilergebnisse zurückgeliefert. Die Komponente stellt sich neuen Aufgaben zur Verfügung.
Sekundärszenario	Die Komponente ist dauerhaft geschädigt, so dass sie nicht mehr für weitere Missionen verfügbar ist bis sie gewartet worden ist.

3.3.3. Das Ende einer Mission

Eine Zelle kann die Arbeit aus drei Gründen einstellen, wie es in Abbildung 3.5 dargestellt ist. Im Normalfall hat sie die Mission erfolgreich abgeschlossen, beziehungsweise wurde ein vorher festgelegtes Ende erreicht. Allerdings können auch Einflüsse von außen eine Komponente zum vorzeitigen Abbruch der Mission veranlassen. Dies kann durch den Auftragsteller, eine andere Zelle oder einen Messwert außerhalb der Missionsparameter ausgelöst werden. Solche Messwerte können entweder innerhalb der Zelle erhoben oder von externen Sensoren, Nachbarzellen oder dem System selbst geliefert werden. Andere Missionen, die von der selben Komponente verfolgt werden, bleiben davon unberührt.

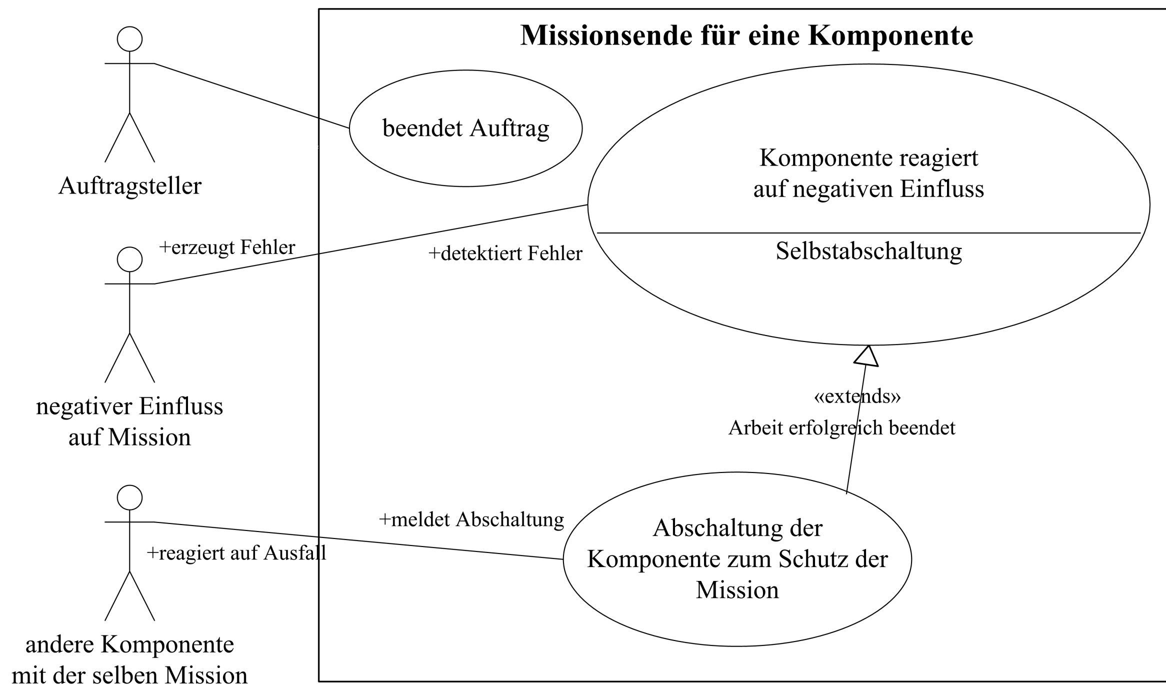

Abbildung 3.5.: Eine Mission wird beendet

Anwendungsfall: Beendet Auftrag

Kurzbeschreibung Die Mission wird regulär beendet. Dies kann vom Auftraggeber, aber auch von einer vorgesehenen Terminierungsbedingung ausgelöst werden. Siehe auch *„Anwendungsfall: Ergebnisse werden zurück-, Ressourcen freigegeben“*.

Vorbedingung Die Mission läuft.

Nachbedingung Die Mission ist beendet.

Primärszenario Die Komponente wird angewiesen, die Mission ordentlich zu beenden, wie in *„Anwendungsfall: Ergebnisse werden zurück-, Ressourcen freigegeben“* beschrieben.

Sekundärszenario Die Arbeit kann nicht ordnungsgemäß beendet werden, weshalb die *„Anwendungsfall: Fehlerbehandlung“* ausgeführt wird.

Anwendungsfall: Komponente reagiert auf negativen Einfluss

Kurzbeschreibung Die Komponente reagiert auf einen negativen Einfluss, von außen und innen. Erweitert *„Anwendungsfall: Abschaltung der Komponente zum Schutz der Mission“*. Siehe auch *„Anwendungsfall: Fehlerbehandlung“*.

Vorbedingung Komponente bearbeitet eine Mission, bezüglich der ein schädlicher Einfluss vorliegt.

Nachbedingung Die Mission läuft ohne die Komponente weiter.

Primärszenario Ein Sensor innerhalb oder außerhalb der Komponente liefert einen Wert, der jenseits der Missionsspezifikation liegt. Auch kann eine andere Komponente auf einen Fehler hinweisen, der eine Abschaltung erfordert. Eine entsprechende Reaktion wird mit der Erweiterung *„Anwendungsfall: Abschaltung der Komponente zum Schutz der Mission“* behandelt. Dies soll verhindern, dass der Fehler der Mission selbst Schaden zufügen kann.

Sekundärszenario Die Mission ist immer noch nicht gesichert, daher werden auch andere Komponenten zu einer Reaktion aufgefordert.

Anwendungsfall: Abschaltung der Komponente zum Schutz der Mission

Kurzbeschreibung	Reagiert auf schädliche Einflüsse, die die korrekte Fortsetzung der Mission im Ganzen gefährden.
Vorbedingung	Die Mission ist durch die Komponente bedroht.
Nachbedingung	Falls die Komponente noch arbeitsfähig ist, geht sie wieder in Bereitschaft. Die Mission besteht weiter, andere Komponenten sind nicht negativ beeinflusst.
Primärszenario	Eine für den gegebenen Fehlerfall festgelegte Reaktion wird ausgeführt und ins Protokoll geschrieben. Benachbarte Komponenten werden über den Fehler informiert. Wenn dies die Mission nicht wieder erfüllbar macht, werden die gebundenen Ressourcen freigegeben und falls möglich, Teilergebnisse zurückgeliefert; die Komponente stellt sich neuen Aufgaben zur Verfügung.
Sekundärszenario	Die Mission ist weiterhin gefährdet, weitere Komponenten reagieren entsprechend.

3.4. Anforderungen

Aus den vorgestellten Anwendungsfällen ergeben sich einige Anforderungen an das Framework, welches eine Zelle mit dem Ziel der Missionserhaltung steuert.

Um Missionen zu überwachen, soll auch deren Definition mit einbezogen werden. Am einfachsten lassen sich mit einem Computer Zahlen vergleichen, die einer Metrik folgen. Auch ob ein Element in einer Menge ist, kann zum Beispiel mit Hashfunktionen effizient ermittelt werden. Ein korrektes Interpretieren von natürlichsprachlichem Text hingegen ist immer noch ein großes Forschungsthema für die Computerlinguistik und die theoretische Informatik. Da Sensoren kaum Text liefern, erscheint es sinnvoll, die Verarbeitung von Informationen in Textform im Framework nicht zu unterstützen.

Eine Mission ist also vorwiegend durch numerische Parameter beschrieben. Diese können um eine Variable, die deren Semantik repräsentiert, erweitert und mit einer einheitlichen Syntax gespeichert werden. Auf diese Weise ist eine generische Implementierung eines Frameworks möglich, die einige Vorteile bringt. Neben der Wiederverwendbarkeit von bereits getestetem Code, reduziert sich außerdem die einfache Wart- und Erweiterbarkeit, die Anzahl von Fehlern und die Zeit, die für die Realisierung des Kontrollsystems benötigt wird. Dank einheitlicher Schnittstellen und Datenformaten ist die Vernetzung mehrerer Frameworks ohne größeren Aufwand machbar. Da jedes Framework einer Zelle zugeordnet ist, die ihrerseits für eine Komponente mit einer bestimmten Mission steht, lässt sich so die Interoperabilität der einzelnen Teile in einem Netz auch auf der Steuerungsebene nachbilden. Eine Zelle entscheidet sich immer noch selbstständig, ob und wie sie weiter arbeitet, kann aber den Zustand in ihrer Umgebung mit einbeziehen sowie sich ihren Nachbarn mitteilen.

Dies erfolgt ebenfalls mittels einfacher Werte mit einer beigefügten Semantik. Dabei kann das Datenpaket an sich bereits die Nachricht sein, gerade dann, wenn es eine Zustandsänderung in der Zelle bedeutet. Auch dessen Ausbleiben kann im Falle von *Keep-Alive*-Nachrichten Reaktionen auslösen. Die meisten Pakete enthalten aber Messwerte von Sensoren, dem Prozess, der überwacht wird oder von anderen Zellen. Zusätzlich muss jedes Paket den Namen seiner Quelle enthalten, damit davon abhängig Entscheidungen getroffen werden können. So kann der Name einer Zelle auch Rückschlüsse auf deren Typ zulassen. Terminiert sich die Zelle selbst, kann eine Nachbarzelle gleichen Typs an der Mitteilung erkennen, dass sie als Ersatz dienen kann. Oft können auch zusätzliche Informationen, wie die Einheit eines Messwertes nützlich sein. Damit könnte sogar eine Plausibilitätsprüfung bei der Initialisierung des Frameworks vorgenommen werden, indem eine Warnung beim Kombinieren zweier unvereinbarer Einheiten ausgegeben wird.

Der Name des Ursprungs einer Nachricht sowie deren Einheit sind auch beim Zurückverfolgen von Abläufen hilfreich. Werden diese Informationen zusammen mit einem Zeitstempel und den eingeleiteten Maßnahmen in eine Logdatei oder eine Datenbank geschrieben, können Zustand und Verhalten eines Netzes überwacht werden. Dies hilft bei Optimierungen und dem Finden von Fehlern, sowohl in den Komponenten, die terminiert worden sind, als auch bei Fehlentscheidungen eines Frameworks. Gleichzeitig lassen sich die Daten mit einem Verzeichnis der Ressourcen abgleichen, um die Verfügbarkeit einer Komponente festzustellen.

3.4.1. 1. Anforderung: Informationen sammeln

Um Entscheidungen über den Zustand eines Systems treffen zu können, muss dieses so gut wie möglich erfasst werden. Dabei ist es wichtig, dass die benötigten Informationen in einer maschinenlesbaren Form vorliegen und automatisiert abgefragt werden können. Das auswertende Framework muss dementsprechende Methoden zur Verfügung stellen, die Daten einsammeln und für die interne Weiterverarbeitung anpassen. Auch sollten Schnittstellen bereitgestellt werden, über die neue Regeln und Daten an das Framework übergeben werden können. Nicht zu vergessen sind Fehlermeldungen, die ebenfalls an das Framework zur weiteren Verarbeitung vorgelegt werden müssen.

3.4.2. 2. Anforderung: Wissen aus den Informationen ableiten

Immer wenn sich etwas in dem zu überwachenden System ändert, ist dieses auf eine weitere Gültigkeit im Sinne der Mission zu prüfen. Hierzu muss das Framework alle Werte, die sich verändert haben und die mit diesen verbundenen Informationen erneut auswerten. Es müssen verschiedene Werte mittels geeigneter Funktionen zusammengeführt werden können. Dabei kann es notwendig sein, diese vorher zu filtern und zu gewichten. Auch eine Auswertung über die Zeit ist vielfach hilfreich. Zum Bewerten von Informationen können Vergleiche mit einer geeigneten Metrik eingesetzt werden, besonders um boolesche Werte zu erzeugen, die eine Entscheidung über das weitere Vorgehen ermöglichen.

Um diese Anforderung zu erfüllen, sollte tatsächlich eine Wissensgenerierung stattfinden. So können Methoden der *Knowledge Discovery in Databases* dazu beitragen, einen umfassenderen Überblick über den Status des verteilten Systems und dessen Fähigkeit, seiner Mission weiterhin nachkommen zu können, zu gewinnen. Das Betrachten des Kontextes und das Einbeziehen der Daten anderer Komponenten sollen helfen, mehr zu erfahren, als die einzelnen Informationen mitteilen können.

3.4.3. 3. Anforderung: Reaktionen bestimmen und durchführen

Wenn genügend Informationen vorliegen, um eine Entscheidung zu treffen, werden diese bezüglich der weiteren Reaktionen ausgewertet, was schon mit einfachen Vergleichen erfolgen kann. Die Reaktionen werden dem Framework vorgegeben, könnten aber auch dynamisch mit Methoden der künstlichen Intelligenz und auf Basis eines detaillierten Modells konstruiert werden. Eine Reaktion sollte immer die Lebendigkeit der Mission als Ganzes garantieren, auch wenn dies zu Lasten der zugeordneten Komponenten erfolgt. Dabei ist der Kontext zu beachten, in dem sich eine Komponente befindet. Ist sie für die Mission überlebenswichtig, kann sie nicht einfach abgeschaltet werden. Daher ist auch immer das lokale Umfeld zu benachrichtigen, dass ein Problem vorliegt.

Fehler zu verhindern, reicht als Reaktion im Sinne der Anforderung nicht aus. Zusätzlich sind Maßnahmen zu treffen, dass das Versagen sich nicht negativ auf den Erfolg der Mission auswirkt. Auch Reaktionen, die die Mission unterstützen, sollten eingeplant werden. Diese Anforderung ist erfüllt, wenn durch Reaktionen Fehler nicht nur vermieden, sondern, soweit wie möglich, auch ihre Folgen gemindert werden.

3.4.4. 4. Anforderung: Mit anderen Zellen kommunizieren

Es ist wichtig, dass sich die Komponenten eines verteilten Systems absprechen. Wenn eine Reaktion aufgrund eines Ereignisses erfolgt, ist es durchaus wahrscheinlich, das gleichartige Zellen zu einem ähnlichen Schluss kommen und ähnlich reagieren. Dies darf nicht zu kritischen Engpässen oder widersprüchlichen Reaktionen führen. Auch kann es sein, dass eine andere Zelle besser reagieren kann. Somit sollte eine Zelle zu einer erneuten Bewertung ihrer Situation angeregt werden können. Wenn eine Komponente ihre Arbeit einstellt, ist es oft notwendig, für Ersatz zu sorgen.

Das kontrollierende Framework muss also eine Kommunikation mit anderen Instanzen von sich implementieren. Dies kann in einer verteilten Art und Weise geschehen, um der zugrunde liegenden Dezentralisierung des Systems zu entsprechen und damit bereits bestehende Infrastruktur nutzen zu können. Neben den sich daraus

ergebenden Synergien ist ein weiterer Vorteil, dass ein *single Point of Failure* vermieden wird. Die Absprache der Zellen untereinander erfolgt dabei auch über System- und Zuständigkeitsgrenzen hinweg.

3.4.5. 5. Anforderung: Verzeichnis- und Protokollplattform

Oft gibt es eine zentrale Komponente in einem verteilten System, die Befehle von außen entgegen nimmt und intern weiter verteilt. Sie kann auch Protokoll führen, damit die Ereignisse im System nachvollziehbar sind. Sie dient auch als zentrales Verzeichnis für bereitstehende Zellen, nimmt die Ergebnisse der Arbeit entgegen und reicht Warn- und Fehlermeldungen weiter. Diese zentrale Komponente kann auch dezentral organisiert sein und sogar von dem Framework selbst bereitgestellt werden, doch ist es oft einfacher, diese dediziert und notfalls redundant auszuführen. Dies macht die Administration einfacher und das Netz weniger komplex. Das Framework aber muss einen derartigen Commandserver anbinden, oder mindestens ein Verzeichnis enthalten, mit dem benachbarte Zellen identifiziert und angesprochen werden können.

3.4.6. 6. Anforderung: Eine Zelle kennt die Mission

Das Framework soll in der Lage sein, eine einfache Beschreibung der Mission bei seinen Aufgaben zu berücksichtigen. Die Mission ist dabei für das gesamte System definiert und in reduzierter Form auf die Aufgabe der einzelnen Zelle angepasst. Diese Anforderung ist erfüllt, wenn die Zelle im Sinne der Mission handelt. Das Framework sollte also nicht nur auf negative Einflüsse von außen reagieren, sondern auch den positiven Verlauf der Mission im Fokus behalten können. Allerdings ist Korrektheit nur schwer beweisbar, weshalb es nur selten möglich ist, explizit die Mission auf sie zu prüfen.

3.5. Zusammenfassung

Die in Kapitel 2 beschriebenen Szenarien stellen an das Framework die in Tabelle 3.1 markierten Anforderungen. Dabei hängen einige von der Implementierung der Szenarien ab, so dass eine konkrete Bewertung nicht immer einfach ist. Nur teilweise passende Anforderungen sind daher mit einem (✓) gekennzeichnet, die weitgehend zutreffenden mit einem ✓ohne Klammern.

Die ***1. Anforderung:*** *Informationen sammeln* (3.4.1) und die ***3. Anforderung:*** *Reaktionen bestimmen und durchführen* (3.4.3) werden von allen Szenarien vorausgesetzt. Dies ist einleuchtend, da wenigstens ein boolescher Schalter eingelesen werden muss, der die Reaktion auslöst und ohne eine Reaktionsmöglichkeit wären die eingelesenen Daten weitgehend nutzlos. Lediglich beim **2. Szenario:** Verfalldatum bei Informationen kann es sein, dass nur ein Verfalldatum fest vorgegeben ist und sonst keine weiteren Informationen gesammelt werden müssen.

Vielfach ist auch die ***2. Anforderung:*** *Wissen aus den Informationen ableiten* (3.4.2) gefragt, da ein Kontext ermittelt werden soll. Dies geht einher mit der ***4. Anforderung:*** *Mit anderen Zellen kommunizieren* (3.4.4), da die anderen Zellen als Informationsquellen herangezogen werden. Dabei werden aber häufig die Systemgrenzen nicht überschritten, was zu einer kleineren Datenbasis führt. Auch ist in keinem der Systeme vorgesehen, Reaktionen in einer fremden Domäne auszulösen, um die Integrität der gemeinsamen Mission zu bewahren.

Da die Szenarien 1, 4, 5, 6 und 7 allesamt mehrere Daten für eine Entscheidung aggregieren, ergibt sich für diese automatisch die 2. Anforderung. Im Computergrid, in der Industrie und bei intelligenten Stromnetzen werden viele verschiedene Messpunkte überwacht, die teilweise sehr unterschiedliche Daten liefern. Bei einem Computerprozess sind dies unter anderem CPU-Auslastung, Speicherverbrauch und geöffnete Kommunikationsverbindungen, in der Industrie Füllstände, Temperaturen, Geschwindigkeiten und Drücke. Smart Grids sammeln von Millionen Verbrauchern und Hunderten Kraftwerken Informationen wie Spannung, Stromstärke, Phase und Frequenz. Aus all diesen Daten muss ein Bild generiert werden, das möglichst genau die Realität widerspiegelt und demnach einer Komponente hilft, ihre Lage und ihr Umfeld richtig einzuschätzen. Dies ist auch bei Sensornetzen wichtig, die deshalb ebenso die 2. Anforderung stellen. Selbst die Überwachung einer Heizung benötigt neben der Temperatur auch einen Temperaturverlauf, Uhrzeit und Wochentag sowie den Modus, in dem sich die Steuerung befindet. Bei der Zugriffskontrolle auf Informationen trifft dies nur bedingt

Szenario	gestellte *Anforderungen* (3.4)					
	1. Anforderung: Informationen sammeln	**2. Anforderung:** Wissen aus den Informationen ableiten	**3. Anforderung:** Reaktionen bestimmen und durchführen	**4. Anforderung:** Mit anderen Zellen kommunizieren	**5. Anforderung:** Verzeichnis- und Protokollplattform	**6. Anforderung:** Eine Zelle kennt die Mission
1. Szenario: Missionen im Grid Computing	✓	✓	✓	✓	✓	✓
2. Szenario: Verfalldatum bei Informationen	(✓)	(✓)	✓			
3. Szenario: Spionagesoftware	✓		✓		✓	(✓)
4. Szenario: Sensornetze	✓	✓	✓	✓	(✓)	✓
5. Szenario: Heizung	✓	✓	✓	(✓)	(✓)	(✓)
6. Szenario: Sicherheit in der Industrie	✓	✓	✓	✓	✓	✓
7. Szenario: Smart Grid	✓	✓	✓	✓	✓	✓

Tabelle 3.1.: Anforderungsanalyse der Szenarien aus Kapitel 2

zu, je nachdem wie komplex der Kontext ist, in dem sie sich befinden. Gerade das *Digital Rights Management* stellt manchmal einige Bedingungen an die zugreifende Instanz, um die Daten freizugeben.

All diese Systeme sind stark verteilt und vernetzt, weshalb eine Kommunikation mit anderen Teilen die Einschätzung der Situation verbessern kann. Gerade bei Computersystemen wird mit Redundanz und Virtualisierung versucht, bei einem Ausfall einer Komponente auf eine andere, gleichwertige umzuschalten [BDF+ 03]. Sensornetze wie Smart Dust bestehen sowieso aus großen Mengen identischer Hardware, die als Ersatz einspringen können. Redundanz zur Ausfallsicherheit spielt auch bei Industrieanlagen eine wichtige Rolle und Energieverbundnetze wurden geschaffen, um Ausfälle kompensieren zu können. Daher ist bei allen Anlagen die Kommunikation der Komponenten untereinander möglich. Um diese zu verbessern, legen Energieversorger parallel zu den Stromtrassen Glasfaserleitungen. Sogar Heizungssteuerungen sind mittlerweile mit integriertem Funkmodul erhältlich und werden vermehrt in intelligente Haussteuerungssysteme eingebunden. Allerdings kann ein Regler nicht das Ventil eines anderen steuern, wenn jener versagt, weshalb hier nur ein (✓) vergeben wird. Hinzu kommt noch, dass eine Standardisierung zwischen Produkten verschiedener Hersteller leider nicht selbstverständlich ist. Dieses Problem gibt es auch beim Cloud Computing. So ist es mitunter sehr schwierig, von einem Anbieter zu einem anderen Cloud-Dienst zu migrieren [AFG+ 09]. Daher wird beim Grid Computing eine Middleware wie *Globus Tool Kit* eingesetzt, um anbieterunabhängig Jobs definieren zu können [FKT 01].

Die ***5. Anforderung:*** *Verzeichnis- und Protokollplattform* (3.4.5) ist teilweise durch einen Commandserver oder ein Logsystem gegeben, zentrale Ressourcenverzeichnisse finden sich bereits in verteilten Systemen, die die Techniken des Grid Computing nutzen. Hier gibt es mit OGSA [FKS+ 05] bereits ein Konzept, dass die Verteilung und Auffindung von Ressourcen in einem Grid organisiert. Auch ist die Schnittstelle für eine bessere Interoperabilität von verschiedenen Zellen spezifiziert, so dass die ***4. Anforderung:*** *Mit anderen Zellen kommunizieren* (3.4.4) leicht implementiert werden kann. Eine dezentrale Absprache der einzelnen Zellen untereinander, auch über Systemgrenzen hinweg, die eine autonome Reaktion in einer Zelle auslösen kann, ist aber oft nicht verwirklicht.

Das Protokollieren der Entscheidungen eines Kontrollsystems in einer Komponente ist vor allem in der Trainingsphase bei einem selbstlernenden System sinnvoll, um Fehlleistungen besser entdecken zu können. Auch beim Entwurf der Missionsbeschreibung können so Fehler im Design leichter gefunden werden. Sollte im Betrieb eine Fehlentscheidung getroffen werden, kann die Ursache schnell ermittelt und behoben werden.

Alle Szenarien, die ein komplexes, verteiltes System beschreiben, sollten daher über eine Protokollführung verfügen. Die gilt eingeschränkt auch für die Heizung, sofern ihre Regelungstechnik in ein Smart Home integriert ist.

Über ihre Aufgabe wissen die meisten Komponenten meistens nur so weit Bescheid, wie es für die Erledigung ihrer Arbeit notwendig ist. Die ***6. Anforderung:*** *Eine Zelle kennt die Mission* (3.4.6) erfüllt das kaum, da auch die Mission des Gesamtsystems notwendig ist, um folgerichtige Entscheidungen bezüglich eines globalen Missionserhalts treffen zu können. Wenn eine Zelle nicht weiß, wie ihre Reaktionen die anderen Komponenten beeinflusst, sind die Folgen nicht abzuschätzen, womit der Sinn der Reaktionen zur Stabilisierung der Mission hinfällig würde. Da jedoch eine derartige Abschätzung umfangreiche Simulationen oder eine Art Verständnis der Mission voraussetzt, ist eine Umsetzung dieser Anforderung nur schwer und in sehr eingeschränktem Rahmen möglich. Daher empfiehlt es sich, einer Zelle klare Regeln vorzugeben, an denen sie ihre Missionstreue ungefähr ablesen kann.

Da Informationen selbst nicht intelligent sind, wissen sie nichts über ihre Bestimmung. Sie sollen sicher vorgehalten werden und ausschließlich berechtigte Zugriffe erlauben. Schutzsysteme sind daher regelbasiert [oD 85] [FeKu 92], eine tatsächliche Missionsbeschreibung der Informationen selbst ist nicht gegeben. Sonst könnten Daten feststellen, dass sie widerrechtlich benutzt werden und sich selbst löschen. Obwohl es viele Bestrebungen gibt, dies durch Kopierschutzmechanismen und *Digital Rights Management* umzusetzen, waren die Bemühungen stets über kurz oder lang erfolglos.

Eine Heizung hingegen kennt ihre Mission teilweise, da sie systeminhärent ist: ein Raum soll immer die vorgegebene Temperatur haben, ohne dass unnötig Energie verbraucht wird, weil zum Beispiel ein Fenster offen steht. Allerdings weiß eine Heizungssteuerung nichts von den Aufgaben der anderen Komponenten in einem vernetzen intelligenten Haus.

Alle anderen Szenarien stellen die Anforderung jedoch voll, da eine Beschreibung der Mission, sowohl für die einzelne Komponente, als auch gesamt, benötigt wird, um den richtigen Kontext und damit die erfolgversprechendste Reaktion, die den Missionserhalt begünstigt, zu bestimmen. Allerdings muss aus praktischen Gründen der Umsetzung diese globale Definition für die einzelnen Zellen deutlich reduziert und angepasst werden. Daher ist es häufig nur möglich die Kausalitäten, die gegen eine Reaktion sprechen, zu bedenken. Dies lässt sich häufig mit weitgehend klaren Regeln erreichen, so dass die Auswertung der Daten nicht zu lange dauert und auch auf weniger potenten Maschinen stattfinden kann.

Bevor in Kapitel 6 ein Framework besprochen wird, das diese Anforderungen umsetzt und dabei möglichst generisch implementiert wird, folgt im nächsten Kapitel 4 eine Betrachtung von bestehenden Konzepten, die den Erfolg einer Mission sicherstellen und Gefahren für ein System abwehren sollen.

4. Bestehende Konzepte zum Missionserhalt

Komplexe Systeme bieten aufgrund ihres Umfanges und ihrer vielschichtigen Formen eine große Angriffsfläche. Oft bestehen sie aus vielen weitgehend unabhängig entwickelten Modulen, deren Schnittstellen sensibel auf Ungenauigkeiten reagieren. Allein durch die Kompliziertheit und die schiere Größe erhöht sich die Wahrscheinlichkeit für kritische Fehler enorm. Daher muss ein solches System robust auf Fehler reagieren. Schon beim Entwurf sollte dies Beachtung finden, auch um das Ausnutzen dieser Fehler durch Schadcode zu verhindern. Dabei hilf der Einsatz von Programmiersprachen mit einer strengen Prüfung der Typen und Arraygrenzen, das Vermeiden gefährlicher Funktionen und der Einsatz von Bibliotheken, die potentiell schädliche Zugriffe auf das System verbieten. So neigen auch heute noch in C geschriebene Programme zum Buffer Overflow, wenn die Funktion die Länge der Parameter nicht prüft, so wie bei dem dafür berüchtigten `strcpy()`.

Durch eine hohe Verteilung von Ressourcen ist eine Überwachung und schnelle Reaktion noch schwieriger. So müssen für die Gegenreaktionen alle Seiteneffekte bedacht werden, was ein tiefes Verständnis des Systems voraussetzt. Heterogene Hard- und Software erschweren die Wartung und den Planungsaufwand noch einmal enorm. Daher behilft man sich mit standardisierten Schnittstellen, modularen und in Schichten aufgeteilten Designs und zusätzlicher Sicherheitssoftware.

4.1. Angriffsprävention in IT-Systemen

Es gibt bereits viele Produkte auf dem Markt, um sich vor bösartiger Software zu schützen. So laufen auf den Rechnern Virenscanner, die WAN-Schnittstelle ist durch eine Firewall geschützt und Intrusion Detection Systeme sollen verdächtigen IP-Datenverkehr melden. Noch wichtiger ist aber die Vorsorge, damit Schädlinge es nicht so einfach haben, ein System zu infizieren. Dazu gehören die Sicherheitsaktualisierungen, die mittlerweile bei fast allen Produkten regelmäßig über das Internet ausgeliefert werden. Auch sollte man die Anzahl der potentiellen Sicherheitslücken gering halten, indem nur die benötigten Programme und Dienste laufen und diese nur mit den minimal notwendigen Rechten ausgestattet sind. Auch Verschlüsselung und eine strenge Authentifizierung sollen den unautorisierten Zugriff auf wertvolle Ressourcen verhindern. Den besten Schutz bietet eine gute Schulung der Nutzer, damit diese den verantwortungsvollen Umgang mit der Technik beherrschen.

4.1.1. Virenscanner

Die wohl bekannteste und verbreitetste Art der Schutzsoftware für Computer stellen die Virenscanner dar. Diese wurden erstmals notwendig, nachdem die Ausbreitung von Schadsoftware in den 90er Jahren stark zugenommen hatte. Dabei ist die Idee von sich selbst replizierenden und verbreitenden Computerprogrammen schon sehr viel älter. Doch erst mit dem Erfolg der in Massenproduktion hergestellten Heimcomputern, die eine einheitliche Soft- und Hardware aufweisen, wurde es möglich, derartige Programme zu schreiben. Die ersten Viren und Würmer waren dabei meist noch recht einfach und mit einigen Programmierfehlern behaftet [G Da 12a]. Sie ärgerten zwar ihr Opfer, konnten sich aber aufgrund der noch gering ausgebauten Vernetzung nur langsam ausbreiten. In den Neunzigern wurde die Malware immer perfider. Sie wurde polymorph und begann sich zu tarnen und gezielt Systeme anzugreifen, um dem Virenautor Ruhm oder Geld zu bescheren. Die ersten Makroviren entstanden, die sich auch über Systemgrenzen hinweg in harmlos erscheinenden Dokumenten verbreiteten [G Da 12b]. Zusätzlich bereitete der Siegeszug von Microsoft Windows auf den nun auch im Massenmarkt immer beliebter werdenden Desktopcomputern eine Monokultur, welche die Entwicklung von Schädlingen für eine möglichst große Anzahl von Systemen stark vereinfachte. Da Windows auf

möglichst vielen Plattformen Verwendung finden sollte, also einer Vielzahl von Ansprüchen gerecht werden musste, wurde es so komplex, dass Schwachstellen und Sicherheitslöcher deutlich zunahmen. Privatleute, die keine technische Ausbildung haben, sind selten in der Lage, ein System zu warten, abzusichern und Gefahren rechtzeitig zu erkennen. Durch das Internet konnten sich Schädlinge endlich schnell und unkompliziert ausbreiten, um auf von der modernen Technik überforderte und gutgläubige Nutzer zu treffen. Somit bildete Malware einen neuen Weg, um auf kriminelle Weise an viel Geld zu kommen.

Die Anzahl der Sicherheitslücken in modernen Softwaregroßprojekten ist vielfach unüberschaubar und wächst mit deren Komplexität und Grad an Vernetzung. Sie werden daher wohl niemals alle geschlossen werden können, auch weil jede neue Funktion potenziell neue Lücken aufreißt. Es bewegen sich auch immer mehr technisch unversierte Menschen sowie immer mehr autonome Geräte im Internet, die sich nicht ständig bei inzwischen fast 7000 neuen Viren am Tag auf dem neuesten Stand halten können. So gab es im ersten Halbjahr 2011 laut G Data 1.245.403 neue Schädlinge [G Da 11]. Somit ist die Notwendigkeit für automatisierte Wächter auf den Systemen zu erklären, die wir in Form der Virenscanner auf den meisten Computern finden. Anfangs suchten diese mit unregelmäßig erneuerten Virensignaturen nach Malware, die sich auf dem Computer eingeschlichen hat [Szor 05]. Dieses Prinzip nennt man *reaktiv* und es bildet auch heute noch das Rückgrat der meisten Virenscanner. Doch auch die bei Kaufversionen übliche stündliche Aktualisierung der Signaturen reicht inzwischen nicht mehr aus. Um diesen Erkennungsmechanismus zu umgehen, wurden polymorphe Viren entwickelt, die ihr Aussehen ständig verändern. Da die Semantik des Schadcodes sich jedoch nicht ändert, werden auch Verfahren eingesetzt, die die Aufgabe eines Codes und nicht nur dessen Gestalt untersuchen [CJS+ 05]. Dies geht in die Richtung des von den meisten aktuellen Virenscannern und Sicherheitssuiten zusätzlich eingesetzten *proaktiven* Verfahrens, welches eine bessere Erkennung von Schädlingen erreichen soll. Dabei werden Heuristiken eingesetzt und in einer Sandbox ausgeführte Programme auf potentiell schädliches Verhalten geprüft. So sollen auch unbekannte Viren, Würmer und trojanische Pferde erkannt werden, allerdings steigt auch die Anzahl falscher Infektionswarnungen. Außerdem verbraucht dieses Vorgehen mehr Systemressourcen, so dass der Computer langsamer werden kann. Da gute proaktive Verfahren aufwendig und teuer sind, finden diese deutlich weniger Anwendung als reaktive.

Virenscanner erfüllen, da sie nur die Schädlinge und deren Verhalten kennen, nicht jedoch die Aufgabe, die das System erfüllen soll, alle *Anforderungen* (3.4) bis auf die ***6. Anforderung:*** *Eine Zelle kennt die Mission* (3.4.6). Auch die Kommunikation mit anderen Zellen ist eingeschränkt. So teilen moderne Virenscanner dem Hersteller das Auftreten eines neuen Schädlings mit, reden aber ansonsten kaum mit anderen Komponenten des verteilten Systems. Kauft man eine komplette Sicherheitssuite, so werden Warnungen auch an andere Teile dieser Software weitergegeben. Dabei handelt es sich um Schutzfunktionen wie White- und Blacklists von Programmen und URIs, Warnungen bei verdächtigen Webseiten, Kinderschutzfilter und Echtzeitscanner. Eine weitere Hauptkomponente dieser Suiten ist die Firewall.

4.1.2. Firewall

Eine Firewall dient in erster Linie dazu, das lokale Netzwerk vor Gefahren aus dem Internet zu schützen. Hierfür werden nicht oder nur für den internen Gebrauch genutzte Ports geschlossen. Bei Verbindungen wird auf eine korrekte Form so wie auf den Initiator geachtet. Wenn eine Verbindung von einem internen System aufgebaut worden ist, gilt diese in der Regel als unverdächtig. Auch eine einfache Zugangskontrolle findet statt, beispielsweise die von DSL-Routern bekannten MAC-Adressen-Filter. Hierbei ist es essentiell, dass die Firewall korrekt konfiguriert ist, da sonst Sicherheitslücken aufgerissen oder legitimer Datenverkehr fälschlicher Weise unterbunden wird. Wenn das Setup zu unübersichtlich wird, helfen nur noch eine gute Strukturierung und automatische Prüfsysteme für die Einstellungen der Firewall [OLKT].

Deshalb und auch wegen der extremen Zunahme des Datenverkehrs werden neue Ansätze untersucht, die auf einer verteilten Firewall basieren [IKBS 00]. Dabei soll auch das Problem gelöst werden, dass die meisten Firewalls die inneren Endpunkte als vertrauenswürdig einstufen. Auch verschlüsselter Ende-zu-Ende-Verkehr kann durch eine zwischengeschaltete zentrale Firewall nicht analysiert werden. Oft reicht schon ein unbekanntes Protokoll, um die Firewall ungehindert zu passieren.

Ein großes Problem sind halboffene Verbindungen, da Angreifer gerne mehr davon aufbauen, als die Firewall überwachen kann. Dem Angreifer kostet dies kaum Speicher, da er nur Anfragen abschickt, ohne sich für das tatsächliche Zustandekommen der Kommunikation zu interessieren. Das angegriffene System aber muss

immer von einem ernst gemeinten Verbindungswunsch ausgehen und zumindest kurzzeitig die halboffenen Verbindungen im Speicher behalten. Als Abhilfe für diesen als SYN-Flooding bekannten Angriff wird daher eine zufällige Sequenznummer im TCP-Paket zurückgeschickt, mit der die Firewall weitere Pakete der damit verbundenen Quelle identifizieren kann. Oft wird diese Nummer mit einer Hashfunktion aus den Daten des Anfragenden, wie IP-Adresse, Port und einem Zeitstempel generiert. So müssen keine Informationen über SYN-Anfragen lokal vorgehalten werden.

Bei Firewalls zeigt sich ein ähnliches Bild wie bei den Virenscannern (4.1.1), wenn es um die Erfüllung der *Anforderungen* (3.4) geht. Zusätzlich können noch Anwendungen freigegeben und Ports geöffnet werden. Auch Art und Quelle der Verbindungen, die durch die Firewall hindurchgehen, werden für die Entscheidung bezüglich einer Blockierung des Datenverkehrs herangezogen. Dies erfolgt zumeist aufgrund eines klar vorgegebenen Regelsatzes, eine Analyse der Daten nach der ***2. Anforderung:*** *Wissen aus den Informationen ableiten* (3.4.2) findet kaum statt. Auch die Mission einer Komponente ist der Firewall nicht klar, da sie oft eigenständig und nicht Teil der zu schützenden Infrastruktur ist.

4.1.3. IDS und IPS

Intrusion Detection Systeme (IDS) dienen der Entdeckung von Eindringlingen und arbeiten genau wie Firewalls häufig auf dem Datenstrom. Sie sollen Angriffe von außen auf das lokale Netz erkennen, indem auffällige Internetpakete analysiert werden, aber im Gegensatz zu Firewalls auch Angriffe innerhalb des geschützten Netzes erkennen und melden. Hierbei kommen ähnliche Verfahren wie bei den Virenscannern zum Einsatz, allerdings mit der Anforderung, in Echtzeit zu arbeiten. Neben Signaturen, die schädlichen Code schon bei der Übertragung identifizieren können, werden auch Heuristiken angewendet, die abnormale Kommunikation ausfindig machen sollen [Szor 05]. Dabei müssen über bestehende, noch mehr aber über sich im Aufbau befindende Verbindungen, Metainformationen vorgehalten werden. So ist die Nutzung von ungewöhnlichen Ports oder Verbindungsanforderungen vom WAN aus grundsätzlich als verdächtig einzustufen. Zudem finden historische Daten aus Logdateien Anwendung, genauso wie Honeypots, die ein lohnendes Ziel für Würmer simulieren, um diese zu fangen.

Ist ein möglicher Angriff beziehungsweise eine Gefahr durch Malware detektiert worden, triggert dies einen Alarm, der andere Sicherheitssoftware aktiviert oder einen Administrator informiert. Je nach Einstellung kann das IDS zusätzlich den Datenaustausch blockieren, um eine Infektion des zu schützenden Systems zu verhindern. Ein Problem ist auch die steigende Dynamik in der Netzstruktur. So werden immer mehr mobile Geräte in Netze integriert, deren klassische Schutzsysteme auf eine weitgehend statische Architektur ausgelegt sind. Um dem zu begegnen, werden verteilte Agentensysteme entwickelt, die die mobilen Computer beim Anschluss ans Netz in ihre Auswerte- und Schutzverfahren integrieren [KrTo 02]. Sie dienen fortan dem IDS als Sensoren, die das Entdecken der Angriffe verbessern sollen. Weiterentwickelte *Intrusion Detection Systeme* versuchen Anomalien auch auf Anwendungsebene zu erkennen, wobei unter anderem statistische Verfahren zum Einsatz kommen.

Wenn das IDS selbst Gegenmaßnahmen bei einem vermuteten Angriff vornimmt, zum Beispiel, indem es die Verbindung kappt, spricht man von einem *Intrusion Prevention System* (IPS). Es gibt sogar IPS, die selbstständig einen Gegenangriff einleiten, um die Infrastruktur des Angreifers unbrauchbar zu machen.

Das korrekte Erkennen eines Angriffs ist allerdings oft nicht möglich [Ande 01]. Viele Schadsoftware kommt über legitime Schnittstellen in des System. Wenn ein Nutzer einen infizierten Anhang einer E-Mail öffnet oder einen vom Angreifer ausgelegten USB-Stick in einen Firmenrechner steckt, ist ein IDS kaum in der Lage, dem zu begegnen. Will man einem Konkurrenten ein trojanisches Pferd unterschieben, bedient man sich eher einer Eigenentwicklung, die man auf eine CD-ROM zusammen mit einem Angebot packt. Da für diesen bis dahin unbekannten Schädling keine Virensignatur vorliegt und der Angriff von einem legitimen Computer im Firmennetz aus erfolgt, schlagen meistens sämtliche Schutzmechanismen fehl.

Da ein Angreifer unerkannt bleiben möchte, wird er auch mehr Sorgfalt walten lassen als ein Anwendungsentwickler mit einer harten Deadline oder ein überforderter Administrator. So kann es durchaus sein, dass sich schädlicher Code besser an die Konventionen hält, als ein berechtigtes Programm. Das Ergebnis sind viele falsche Alarme oder ein weniger sensibles IDS sowie ein unentdeckt gebliebener Angriff.

IDS und IPS sind deutlich intelligenter als eine Firewall, da sie komplexe Angriffsmuster erkennen können sollen. Besonders wenn sie dezentral auf den zu schützenden Maschinen ausgeführt werden, erfüllen sie die *Anforderungen* (3.4) zu einem großen Teil. Allerdings kümmern sie sich nicht um die tatsächliche Mission, solange kein Angriff erkannt worden ist, und kommunizieren nur mit anderer Sicherheitstechnik, nicht jedoch mit den Kontrollsystemen der eigentlich arbeitenden Prozesse oder dem Ressourcenmanagement.

4.1.4. DEP und ASLR

Ist eine Schadsoftware erst einmal in ein gesichertes System eingedrungen, versucht sie typischerweise eine legitime Anwendung mit hohen Rechten dazu zu bringen, maliziösen Code auszuführen. Daher werden oft Rücksprungadressen überschrieben oder NOP-Rutschen angelegt, die zu einem Datenbereich des Speichers führen, in dem ausführbarer Code steht. Um diese Angriffe zu erschweren, gibt es einige Methoden, darunter ***D**ata **E**xecution **P**revention* und ***A**ddress **S**pace **L**ayout **R**andomization* [Blaz]. Bei DEP werden Datenbereiche im Speicher mit einem NX-Bit versehen. Dieses **N**o e**X**ecution-Bit teilt dem Prozessor mit, dass Code in diesem Teil des Speichers nicht ausgeführt werden darf [Micr 06]. Wird dies dennoch versucht, wird ein Angriff vermutet, was die Terminierung der ausführenden Anwendung zur Folge hat. Weiterhin verteilt ASLR den allozierten Speicher der Programme zufällig, wodurch das Vorhersagen einer Sprungadresse einer geschützten Funktion durch die Schadsoftware verhindert werden soll [BDS 03]. Doch auch diese Technik wurde bereits erfolgreich angegriffen, bietet aber dennoch einen Schutz gegen einfache oder ältere Viren und Würmer.

DEP und ASLR sind Sicherheitsverfahren, die alleine die Nutzung einiger Angriffsvektoren durch Malware erschweren sollen. Sie erfüllen daher kaum die ***1. Anforderung:*** *Informationen sammeln* (3.4.1) sowie die ***3. Anforderung:*** *Reaktionen bestimmen und durchführen* (3.4.3).

4.2. Physische Gefahren erkennen und vermeiden

Die bisher vorgestellten Sicherheitsmaßnahmen schützen nur gegen digitale Bedrohungen. Doch die Erfüllung einer Aufgabe in einem verteilten System hängt auch von der physischen Integrität der eingesetzten Hardware ab. Der populäre Begriff der Cloud stellt eine Abstraktion von einer konkreten, physikalischen Implementierung der Dienste dar, die ein Anbieter zu erbringen verspricht. Die Cloud ist damit ein Sinnbild für ein nicht greifbares und unbeschränktes Gebilde, das jederzeit und überall verfügbar ist. Dabei darf man aber nie vergessen, das irgendwo ein Rechenzentrum steht, das auch ganz realen Gefahren ausgesetzt ist. Wenn die Datensicherung ungenügend ist und auf teure Redundanzen verzichtet wird, kann ein Teil dieser Cloud nicht mehr verfügbar sein. Daher sind diverse Maßnahmen zu treffen, um ein Rechenzentrum gegen physische Gefährdungen zu schützen, die schon bei der Planung in Betracht gezogen werden müssen. Das Bundesamt für Sicherheit in der Informationstechnik (BSI) stellt daher einen umfangreichen Katalog mit Richtlinien zum Schutz von IT-Infrastruktur zur Verfügung [fSidI 09]. Doch können diese Maßnahmen die in Abschnitt 3.4 gestellten Anforderungen und damit den Wunsch nach Missionserhaltung auch beim Versagen von Systemen erfüllen?

4.2.1. Schutz vor Eindringlingen

Infrastruktur ist wichtig und muss daher geschützt werden. Firmengelände, Geschäfte, Banken sowie öffentliche Gebäude und Fahrzeuge werden videoüberwacht, nachts verschlossen, mit Alarmanlagen versehen und teilweise durch einen privaten Wachschutz gesichert. Doch in Zeiten von hohen Rohstoffpreisen wird nicht nur der Tankwart um seine Bezahlung geprellt, es werden sogar unter Strom stehende Kupferleitungen abmontiert und an Schrotthändler verkauft. Dabei sollte man meinen, 100.000 Volt seien ein genügend guter Diebstahlschutz.

Rechenzentren enthalten neben der teuren Hardware viel wertvollere Daten, deren Wichtigkeit und Sensibilität oft erst richtig eingeschätzt werden, wenn sie verloren oder unrechtmäßig kopiert worden sind. So konnte in der Mainframe-Ära bei einem Diebstahl manchmal nur der Wert des Speicherbandes vor Gericht geltend gemacht werden, die Daten darauf wurden jedoch nicht berücksichtigt.

Meistens ist es einfacher, von außen und auf digitalem Weg an die Daten zu gelangen, allerdings gibt es auch besonders gesicherte Zentren. Gerade staatliche Sicherheitskräfte beschlagnahmen daher gleich ganze Server mittels Durchsuchungsbeschluss, auch wenn nur ein einzelner Dienst Stein des Anstoßes ist [Alth 11]. Dabei gehen die Maßnahmen aufgrund von fehlendem technischen Sachverstand der Gerichte und Polizisten manchmal unnötig weit, wodurch unbescholtenen Dritten unverhältnismäßig großer Schaden entsteht. Auch könnten staatliche Befugnisse durch Geheimdienste missbraucht werden, um an Geschäftsgeheimnisse ausländischer Konkurrenten zu gelangen.

Natürlich gibt es weniger gut geschützte Server, die besonders zum Diebstahl einladen [dpa 11]. In den gestohlenen Computern gespeicherte Informationen sind dabei oft nur Beifang. Daher legen gewerbliche Rechenzentren großen Wert auf Schutzmaßnahmen, die den Zugang zu den Computern stark beschränken. Man findet nicht selten große Serverfarmen in alten Bunkern, mit Ein-Mann-Schleusen, Fingerabdruckscannern und manchmal sogar bewaffneten Wachen. Dies ist auch wichtig, um seinen Kunden neben der Datensicherheit auch die Datenintegrität garantieren zu können.

Es ist all diesen Sicherheitsmaßnahmen zu eigen, dass sie ein unberechtigtes Eindringen verhindern sollen. Was allerdings passiert, wenn ein Einbruch gelingt, ist all zu oft nicht eingeplant. Immer wieder finden sich in der Presse Berichte, dass sensible Daten abhanden gekommen sind und diese nicht einmal verschlüsselt waren. In Filmen sieht man immer wieder, wie Daten hektisch gelöscht werden, nachdem ein Alarm ausgelöst wurde, um den Zugriff durch den Gegner zu verhindern. Sofern man ein nicht lokales Backup hat, ist dies sicher eine brauchbare Lösung. Verfolgt man den Gedanken weiter, könnte man Datenspeicher bauen, die sich selbst vernichten, sobald sie aus einem vorgegebenen Kontext entfernt werden. In geringem Umfang gibt es bereits derartige Verfahren. So kann man sein Smartphone ferngesteuert sperren und den internen Speicher löschen. Auch gibt es verschlüsselte externe Datenträger, die bei mehrmaliger Falscheingabe des Passworts den Schlüssel entfernen. Dank GPS kann man sogar Autos bauen, die sich automatisch oder ferngesteuert verriegeln, stehen bleiben und die Polizei informieren, wenn sie eine Grenze passieren.

Diese Sicherheitssysteme erfüllen die ***1. Anforderung:*** *Informationen sammeln* (3.4.1) und die ***3. Anforderung:*** *Reaktionen bestimmen und durchführen* (3.4.3), sowie meistens noch die ***5. Anforderung:*** *Verzeichnis- und Protokollplattform* (3.4.5). Bei komplexeren Systemen, gerade bei Hochsicherheitsanlagen, findet man auch die ***2. Anforderung:*** *Wissen aus den Informationen ableiten* (3.4.2). Ein verteiltes Zusammenarbeiten von Sicherheitssystemen, wie von der ***4. Anforderung:*** *Mit anderen Zellen kommunizieren* (3.4.4) gefordert, findet eher nicht statt. Dies wäre zum Beispiel, dass ein gestohlenes Smartphone sich über WLAN an andere, fremde Sicherheitssysteme wie Überwachungskameras in einem Kaufhaus oder eine automatische Schließanlage wendet und um Hilfe bittet. So könnte ein Bild vom Dieb gemacht und er sogar eingesperrt werden, bis die Polizei vor Ort ist. Natürlich muss ein derartiges System vor Missbrauch geschützt sein, sowohl technisch, als auch juristisch. Sonst könnte man bequem seinem Ehepartner nachspionieren, indem man dessen Mobiltelefon in den *„Gestohlenmodus“* versetzt.

4.2.2. Schutz vor Elementarschäden

Die in der Antike beschriebenen Elemente Feuer, Wasser, Wind und Erde können die Arbeitsfähigkeit von Computersystemen bedrohen, wenn sie in ungezügelter Gewalt auftreten. Um diesen Gefahren begegnen und damit Mensch und Hardware schützen zu können, werden für viel Geld und mit erheblichem Aufwand technische, regulative und bauliche Sicherheiten eingesetzt.

Feuer

Offenes Feuer gehört nicht in ein Rechenzentrum. Daher gibt es strenge Vorschriften, um ein Ausbrechen eines Feuers zu verhindern. Oft wird auch der Sauerstoffanteil in der Luft künstlich herabgesetzt, wodurch Flammen schon bei der Entstehung umgehend ersticken. Sollte dennoch ein Brand ausbrechen, kann der Serverraum mit einem Edelgas wie Argon geflutet werden.

> *„Rauchansaugsysteme detektieren bereits äußerst geringe Mengen von Rauchaerosolen, wie sie in der frühesten Phase der Brandentstehung freigesetzt werden. Dies schafft den Zeitgewinn, der für*

> *das Einleiten von Gegenmaßnahmen wie weiches Herunterfahren, Datenauslagerung, selektive Abschaltung oder eventuelle gezielte Objektlöschung benötigt wird*“ [tB 08].

Dieser Ansatz erfüllt bereits alle Anforderungen aus Abschnitt 3.4 zumindest teilweise. Inwieweit die ***2. Anforderung:*** *Wissen aus den Informationen ableiten* (3.4.2) umgesetzt ist, hängt von der Implementierung der Brandschutzanlage ab. Die Entscheidung, welche Maßnahme umgesetzt werden soll, hängt aber kaum von den einzelnen Komponenten ab, sofern diese feingranularer als ein *„Rechenzentrum entspricht einer Zelle“* sind. Die ***4. Anforderung:*** *Mit anderen Zellen kommunizieren* (3.4.4), und damit ein verteilter Ansatz, dürfte kaum Verwendung finden. Wenn die Notfallpläne fehlerhaft sind, könnte aufgrund einer fehlenden Absprache der Zellen untereinander eine Art *„Datenpanik“* entstehen, die die Leitungen aus dem Rechenzentrum mit massenhaften Kopiervorgängen verstopfen.

Wasser

Moderne Rechenzentren kühlen die Server mit warmen Wasser, da dieses eine etwa 4000-fach höhere Wärmekapazität als Luft hat und sogar zum Beheizen angrenzender Gebäude genutzt werden kann [ETH 10]. Natürlich muss dieses Wasser von der Elektronik ferngehalten werden, um Kurzschlüsse zu vermeiden. Doch von diesem Kühlwasser geht eine viel kleinere Gefahr für die Computer aus, als von Überflutungen durch Rohrbrüche, eindringendem Starkregen oder Dammbruch bei einem nahe gelegenen Gewässer. Dabei kann nicht nur durch Wasserschäden an der Hardware, sondern vor allem auch durch Schäden an der umliegenden Infrastruktur der Betrieb empfindlich gestört bis unmöglich werden. Tatsächlich können in einer globalisierten Welt sogar in weit entfernten Gebieten stattfindende Überflutungen kritisch für die Versorgung von IT-Systeme werden. Die Überschwemmungen in Thailand im Herbst 2011 stellten nicht nur eine humanitäre Katastrophe dar, sie sorgten auch für eine länger anhaltende weltweite Verknappung von Festplatten [Kirs 11]. Äquivalent zu den eben beschriebenen Brandschutztechniken und Feuerbekämpfungsmaßnahmen könnten intelligente Abwehrfunktionen die Evakuierung der Dienste und Daten in weniger gefährdete Gebiete einleiten.

Wind

Wind wird einem Rechenzentrum in der Regel wenig gefährlich. Zur Kühlung durch bewegte Luft wird sogar viel Energie eingesetzt, die unter anderem auch durch Windkraft bereitgestellt werden kann. Doch Stürme, vor allem in Kombination mit anderen Elementen, können die Infrastruktur beschädigen, die zum Funktionieren eines Rechenzentrums benötigt wird. Dann müssen die Notfallsysteme greifen. Wenn nötig und möglich, kann ein automatischer Umzug von wichtigen Diensten und Daten in ein anderes Rechenzentrum erfolgen, wenn das überwachende System derartige Maßnahmen zulässt.

Erde

Es muss nicht immer gleich ein Erdbeben sein, das für IT-Systeme wichtige Infrastrukturen lahm legt. Oft sind Erdarbeiten, bei denen ein Bagger ein Glasfaserkabel durchtrennt, die Ursache für Störungen im Telefon- und Fernsehkabelnetz. Aber auch schon eine Rentnerin mit einem Spaten kann ganze Regionen vom Internet isolieren [lis/ 11]. Die Folgen durch ein Erdbeben sind natürlich viel essentieller und haben einen lang anhaltenden Ressourcenmangel. Wenn die Notversorgung ebenfalls nicht mehr gewährleistet werden kann, könnten Maßnahmen ähnlich den bei der Brandbekämpfung erfolgen.

4.2.3. Ressourcenmangel

Wenn dem Menschen Nahrung vorenthalten wird, geht sein Körper in einen Energiesparmodus, um die vorhandenen Reserven zu strecken. Auch andere Lebensformen können in Krisenzeiten ihren Stoffwechsel herunterfahren, um zu überleben. Technische Systeme verbrauchen ebenfalls Ressourcen, ohne die sie nicht mehr funktionieren. Wird die Hardware richtig gewartet, kann sie durchaus lange Zeit zuverlässig ihre Arbeiten leisten.

Doch auch Maschinen brauchen ständig neue Nahrung, die sie in Form von elektrischem Strom erhalten. Das Arbeitsklima ist ebenso wichtig, damit ein Gerät möglichst lange hält. Zusätzlich benötigen Maschinen noch die Rohstoffe, die sie verarbeiten sollen, um ihrer Bestimmung nachgehen zu können. Im Falle eines Rechenzentrums sind das Daten und die Kommunikationskanäle nach außen, durch die es Informationen mit der Welt austauscht.

Sicherung der Stromversorgung

Nach dem Erdbeben in Japan und dem Unfall im Atomkraftwerk bei Fukushima wurde in einigen Gebieten Japans der Strom knapp. Um einen Totalausfall der Versorgung mit elektrischer Energie zu verhindern, wurden Teile des Stromnetzes im ständigen Wechsel abgeschaltet. Diese Praxis des *Rolling Blackouts* wird auch in anderen Teilen der Welt angewendet, um Überlastungen der Stromnetze und Kraftwerke vorzubeugen. Meistens ist aber keine Naturkatastrophe der Grund für eine Überlastung, sondern extremes Klima. Während im Winter immer wieder Strommasten und -leitungen den Schneemassen nachgeben und gleichzeitig der Bedarf an elektrischer Energie zum Heizen ansteigt, nimmt der Verbrauch im Sommer durch den Einsatz von Klimaanlagen bei einer Hitzewelle zu. Außerdem müssen manchmal Atomkraftwerke heruntergefahren werden, da das Kühlwasser zu warm und zu wenig ist.

So finden in Nordamerika immer wieder Stromausfälle bei Hitzewellen statt. Trotz diverser Vorkehrungen, wie dem Aufteilen des Stromnetzes in weitgehend unabhängige Teilnetze, die reihum abgeschaltet werden können, kommt es immer wieder zu großen Blackouts, die nicht selten mehrere Staaten umfassen. Der bisher umfangreichste Stromausfall in Nordamerika begann am 14. August 2003. Er betraf den Nordosten der USA und den Südosten Kanadas, und damit über 50 Millionen Menschen. Eine eigens gegründete Untersuchungskommission der Energieministerien der USA und Kanadas veröffentlichte im April 2004 einen Bericht [U.S. 04], der die Ursache und mögliche Gegenmaßnahmen aufzeigt.

Nachdem schon zwei wichtige Kraftwerke nahe Cleveland heruntergefahren worden waren, verursachte eine automatische Abschaltung von Eastlake 5 um 13:31 EDT in Ohio eine starke Last auf Versorgungsleitungen, die Strom von anderen Teilen des Landes in die sonst unterversorgten Gebiete bringen sollen. Als dann auch noch eine 345kV-Linie wegen eines Baumkontakts um 14:02 EDT ausfiel, wurde das Netz überlastet, was zu weiteren Schutzabschaltungen führte. Ein noch in der Testphase befindliches System, das den Zustand des Stromnetzes abschätzen sollte, war nach Wartungsarbeiten nicht wieder auf Automatik gesetzt worden. Als der Fehler bemerkt wurde, waren einige Informationen nicht korrekt dem System mitgeteilt worden, so dass auch bei einer Reaktivierung ein falscher Systemzustand gemeldet wurde. Zu diesem Zeitpunkt war es bereits zu spät, um eine Überlastung des Stromnetzes zu verhindern. Auch fielen nacheinander der Überwachungscomputer und dessen Backup aus, ohne den Bedienmannschaften einen Alarm zu signalisieren. Das Ergebnis war ein kaskadierender Stromausfall, der sich über acht Staaten der USA und den Südosten Kanadas erstreckte. Für bis zu zwei Tage waren 50 Millionen Menschen ohne Elektrizität, neben etwa vier bis zehn Milliarden Dollar Schaden waren auch elf Tote zu beklagen.

Die von der Task Force für den Stromausfall gefundenen Gründe sind im Abschlussbericht der Untersuchung in vier Gruppen unterteilt:

1. Ungenügendes Verständnis des Systems
2. Unzureichendes Situationsbewusstsein
3. Mangelhaftes Stutzen der Bäume unter den Stromleitungen
4. Schlechte Unterstützung der Echtzeitdiagnose

Dies zeigt auf, dass die Fehler, die kumuliert zu einer Kaskade von Stromausfällen führten, auf die hohe Komplexität und unzureichendes Detailwissen der Operatoren zurückzuführen sind. Immer mehr Informationen müssen in Echtzeit bewertet werden, um folgerichtige Entscheidungen treffen und Schlimmeres verhindern zu können. Menschen alleine verlieren schnell den Überblick und die assistierenden Computer arbeiten teilweise auf veralteten, falschen oder ungenügend großen Datensätzen. Da auch benachbarte Netze in einem fremden Administrationsbereich Einfluss auf die Stabilität der eigenen Domäne haben, sind deren wichtige Daten unter Umständen nur schwer oder langsam zu besorgen. Da die Verbundnetze auch immer dichter und immer öfter

an ihrer Lastgrenze betrieben werden, steigt die Komplexität stark an, wodurch die Skalierbarkeit zu einem Problem werden kann.

Eine Lösung könnte eine möglichst detaillierte Erfassung relevanter Daten sein. So wurden seit dem Vorfall 14.08.2003 viele Sensorpunkte in die Stromnetze eingebaut und deren Erfassung und Auswertung deutlich verbessert. Als nächster Schritt wird weltweit die Schaffung von Smart Grids diskutiert. In Teilen der USA, aber auch in Europa, werden seit einiger Zeit intelligente Stromzähler, auch Smart Meter genannt, zum Kauf angeboten. Doch deren Verbreitung schreitet nicht so rasch voran wie die Energieerzeuger und Politiker angenommen haben [Wike 11]. Dem offerierten Einsparpotenzial bei Strom und Geld stehen noch ungeklärte Fragen nach Sicherheit und Datenschutz, aber auch unattraktive Tarifmodelle gegenüber. Weiterhin ist der Nutzen für den Kunden meist wenig offensichtlich. Dies könnte sich ändern, wenn alle möglichen in einem Haushalt zu findenden Elektrogeräte in das Smart Grid integriert werden können. Das Automatisieren der Wohnungen und intelligente Häuser etablieren sich nur langsam, da einheitliche Standards fehlen und man oft auf teure proprietäre Lösungen angewiesen ist. Erst wenn alle Geräte, auch einfache Verbrauchsware bis hin zur Glühbirne, in ein *Home Grid* eingebunden werden können, dürfte ein smartes Leben in einem smarten Stromnetz Realität werden.

Auch die durch die Förderung von regenerativen Energiequellen verstärkte Dezentralisierung der Stromerzeugung soll durch Smart Grids einfacher zu verwalten sein. Dabei könnten verteilte selbständig arbeitende Steuerungssysteme, die sich auch über die Grenzen der Kraftwerks- und Netzbetreiber hinweg absprechen, Skalierungsproblemen vorbeugen und schnellere Reaktionen auch beim Ausfall einzelner Komponenten gewährleisten. Nebenbei könnte so auch das Risiko des gläsernen Kunden vermindert werden, das einige Verbraucher von der Anschaffung eines Smart Meters abhält [SK 10]. Wenn dieses selbst anhand vom Stromnetz bekanntgegebener Daten handelt, ließe sich die datenschutzrechtliche kritische Übertragung von Verbrauchsdaten an einen zentralen Server des Anbieters ersetzten.

Kompensation fehlender Kühlung

Ein Stromausfall impliziert in der Regel einen Ausfall der Kühlung in einem Rechenzentrum, denn das Kühlen der Systeme verursacht bis zur Hälfte des Stromverbrauchs. Doch auch mit genügend elektrischer Energie kann eine Kühlung nicht mehr hinreichend arbeiten, zum Beispiel, wenn Maschinen ausfallen. Bis eine ausreichende Kühlung wieder hergestellt ist, müssen die Prozessoren langsamer getaktet oder teilweise bis gänzlich abgeschaltet werden. Dies wird auch als *„Browndown“* bezeichnet [CATV 01]. Danach sollten sie aber selbstständig wieder hochfahren und ihre Arbeit ohne Datenverlust fortsetzen können.

Bereitstellung von WAN

Rechenzentren brauchen neben großen Mengen elektrischen Stroms nur noch wenige externe Ressourcen. Die wichtigste davon ist der Anschluss an ein *Wide Area Network*, zumeist mit redundanten Lichtwellenleitern und in Form des Internets. Sollten alle Verbindungen nach außen zusammenbrechen, ist ein zweckmäßiges Arbeiten des Rechenzentrums in den meisten Fällen nicht mehr möglich. Server, die nicht mehr von außen erreichbar sind, können abgeschaltet werden, um Energie zu sparen. Alternativ könnten die Rechner in der Downtime auch mit Wartungsarbeiten beauftragt und Backups gefahren werden. In einem Grid vernetzt arbeitenden Zentren im Bereich *High Performance Computing* ist unter Umständen die gesamte Mission gefährdet. Wenn andere Rechenzentren den Verlust von Rechenleistung nicht kompensieren können, kann es sinnvoll sein, die Mission zu suspendieren und andere Aufgaben vorzuziehen.

Maßnahmen bei Ressourcenmangel

Um die Situation beurteilen zu können, müssen Informationen gesammelt (3.4.1) und analysiert (3.4.2) werden. Dann wird entsprechend reagiert (3.4.3). Die Aktionen werden außerdem protokolliert (3.4.5). Dabei ist auch ein Verzeichnis der zur Verfügung stehenden und der fehlenden Ressourcen gegeben.

4.2.4. Redundanz

Mit steigender Anzahl der Komponenten in einem komplexen System steigt auch die Ausfallwahrscheinlichkeit. Bei der Hardware wird daher, wenn möglich, eine Reserve vorgehalten, die die Zeit bis zur Reparatur oder zum Austausch des kaputten Moduls überbrücken soll.

Am bekanntesten in der IT ist der Einsatz von Backups, um einen umfangreichen Datenverlust beim Ausfall eines Datenträgers zu vermeiden. Bei volatileren Datenbeständen reicht das nicht mehr aus, weswegen RAID-Systeme mit Ersatzplatten eingesetzt werden. Ist ein Fehler besonders kritisch, werden gleich die Server oder gar ganze Rechenzentren gespiegelt.

Redundanz ist essentiell in Bereichen, bei denen ein Fehler lebensgefährlich werden könnte. Im Kleinen findet man diese zum Beispiel in Fahrzeugen. Da diese vermehrt mit Mikroprozessoren gesteuert werden und dank Fahrerassistenssystemen sogar teilweise autonom verkehren können, müssen hier fehlerhafte Daten möglichst vermieden werden. Darum werden über die internen Datenbusse Informationen mehrfach gesendet, um Glitches durch gekippte Bits zu kompensieren. Gerade wenn viele Menschenleben beim Versagen eines Systems betroffen wären, gelten besondere Sicherheitsmaßnahmen. In Zügen sind mehrere unabhängige Bremsen installiert, ein Flugzeug muss auch mit einem ausgefallenen Triebwerk landen können und die Notstrom- und Kühlaggregate in einem Atomkraftwerk müssen mehrfach Vorhanden sein.

Um die Redundanz nutzen zu können, wird eine Verbindung zwischen den Produktiv- und den Ersatzsystemen benötigt, die deren Zustände synchron hält. Wenn ein System ausfällt, erreicht das andere kein Lebenszeichen mehr, welches normalerweise in sehr kurzen Abständen generiert und ausgetauscht wird. Daraufhin springt das bis dahin passiv mitgelaufene System ein, um den Betrieb beinahe nahtlos zu übernehmen. So kann man sogar einen Videostream ohne Unterbrechung auf eine redundante Quelle umschalten, ohne dass der Zuschauer etwas merkt [Schm 11].

Die Komponenten müssen also ständig überprüfen, ob der Partner noch aktiv ist. Hierzu müssen untereinander Daten ausgetauscht werden, wobei dies meistens nur zwischen gleichwertigen Zellen in derselben Domäne geschieht (3.4.4). Um ein Versagen früh entdecken zu können, ist ein Erfassen von Informationen notwendig (3.4.1), wie sie bei Festplatten zum Beispiel das S.M.A.R.T. System liefert. Die darauf erfolgenden Reaktionen sind relativ beschränkt (3.4.3). Ausfälle werden gemeldet und protokolliert (3.4.5), um die Wartung zu vereinfachen.

4.3. Autonomic Computing

Der Begriff des *Autonomic Computing* wurde Anfang des Jahrhunderts von IBM geprägt [JK 03]. Dabei handelt es sich um ein Langzeitforschungsprojekt zur selbstständigen Wartung von Computersystemen. So soll es folgende Konzepte durch Automatisierung ermöglichen und verbessern:

- Selbstkonfiguration
- Selbstoptimierung
- Selbstheilung
- Selbstschutz

Bei der Selbstkonfiguration sollen sich neue Komponenten selbstständig in einem Netz installieren, in dem sie sich den anderen Teilnehmern bekannt machen. Daraufhin passen sie sich dahingehend an, dass sie die gemeinsame Mission des Systems mit tragen können. Ein Ziel ist dabei, dass ein neues Element von einem bereits im Verbund Integrierten alten Element lernt, welche Aufgabe es wie zu erfüllen hat. Grundsätzlich soll einer Komponente immer nur mitgeteilt werden, was von ihr erwartet wird, das Wie soll sie selbst erlernen. Dies ist vergleichbar mit dem integrieren einer neuen Zelle in einen Körper.

Wenn so eine Komponente läuft, soll sie sich selbst optimal einstellen, um die Mission möglichst gut zu erfüllen. Dies kann analog zum Vorgang des Lernens eines Gehirns gesehen werden. Eine Schwierigkeit bei der Einstellung ist das Beachten von Seiteneffekten im restlichen System. Ein lokales Optimum muss noch

lange kein globales Optimum sein. Im Gegenteil, wenn eine Komponente überangepasst ist und das gesamte System kanibalisiert, stellt sie eine echte Bedrohung für die Mission dar.

Gegen derartige und andere Missstände soll die Selbstheilungsfähigkeit der autonomen Komponenten helfen. Fehler werden selbstständig erkannt und die Ursache nach Möglichkeit beseitigt. Eine Analysesoftware vergleicht die Ein- und Ausgaben von Maschinen. Sollten sich hierbei Diskrepanzen ergeben, lässt das auf einen Fehler schließen. Die betroffenen Systeme werden dann auf eine ältere Version zurück gesetzt. Falls vorhanden, installiert die Software Patches und testet sich erneut. Ansonsten alarmiert sie menschliche Programmierer, damit die sich des Problems annehmen können. Allerdings ist eine Lösung des Problems nicht immer mit einer Reparatur der kaputten Komponente gleichzusetzen. Denn der einfachste Fix ist das Austauschen eben dieser Komponente, ohne die eigentliche Ursache für das Versagen zu bestimmen.

Zuletzt soll sich eine Komponente selbst schützen können. Neben dem Einsatz von den bereits erwähnten Sicherheitssystemen wie Firewalls und Virenscanner ist auch eine Überwachung von Sensoren vorgesehen, die Missstände in der Komponente oder dem gesamten System rechtzeitig erkennbar machen. So sollen Attacken gegen das System als Ganzes abgewehrt, aber auch kaskadierende Fehler verhindert werden.

Mit derartigen Maßnahmen soll die Installation neuer und die Wartung alter Systeme vereinfacht und letztendlich vollständig automatisiert werden. Dabei orientiert sich die Forschung unter anderem an biologischen Zellen, die sich ebenfalls selbstständig regulieren können. Hierzu gehört auch die Apoptose, die im nachfolgenden Kapitel 5 näher beschrieben ist.

Autonomic Computing erfüllt die *Anforderungen* (3.4) bereits vollständig, befasst sich aber hauptsächlich mit dem Einbringen neuer Komponenten in ein verteiltes System. In dieser Arbeit wird jedoch der entgegengesetzte Fall untersucht, nämlich die Optionen, die ein selbstständiges Entfernen einer Komponente aus einem System für den Missionserhalt bietet. Dies entspricht einem Teil des *Autonomic Computing*, der noch viele ungeklärte Fragen aufwirft, auf die in der restlichen Arbeit eingegangen werden wird.

4.4. Synergiediskussion

Alle gezeigten Verfahren überwachen die Systeme bezüglich ungewöhnlichem und schädlichem Verhalten, das korrekte Arbeiten wird aber nur am Rande betrachtet. Bei der Prävention von Angriffen wird böswillige Software definiert, die Aufgabenstellung der guten Programme jedoch nicht berücksichtigt. Dabei kann auch durch ein Fehlverhalten der Software Gefahr für das Gesamtsystem drohen. Ein Einzelsystem überprüft nicht, wie seine Aktionen auf seine Nachbarn wirken. Auch ist eine Umverteilung von Aufträgen für eine bessere Systemstabilität nur in seltenen Fällen vorgesehen. Da durch die immer feinmaschigere Vernetzung unvorhergesehene Effekte auftreten können, versucht man durch eine Abschottung der Teilnetze zueinander den Umfang der zu betrachtenden Komponenten zu verringern. Die Chancen, die sich durch die Verbindung kleiner Teile zu einem großen Organismus ergeben, werden kaum genutzt.

4.4.1. Chancen in IT-Grids

Als Vorbild können hierbei wissenschaftliche Netze wie das D-Grid gesehen werden. Hierbei gilt, gemeinsam sind wir stark. Doch kommt es zur Verwaltung dieser aus vielen unterschiedlichen Domänen stammenden Teilnetze, so gibt es noch immer einige Bürokratie und Handarbeit, um ein unkompliziertes schnelles Handeln zu ermöglichen. Auf technischer Seite konnten schon viele Fortschritte bei der Interoperabilität gemacht werden, indem Middlewares eingeführt wurden. Diese ermöglichen einen standardisierten und plattformübergreifenden Austausch von Diensten und Daten. Wenn die bei der Erteilung eines Jobs übermittelten Beschreibungen erweitert würden, könnte man daraus durchaus Rückschlüsse auf die Mission ziehen. Damit wäre ein Normalverhalten definiert, das überwacht werden kann. Sollte eine Abweichung von der Norm festgestellt werden, so können geeignete Gegenmaßnahmen eingeleitet werden.

Eine derartige Überwachung bei einem verteilten System, das auch noch ständig erweitert und umgebaut wird, ist zentralistisch nur schwer einzurichten. Zu viele unterschiedliche Komponenten und deren Eigenarten müssen berücksichtigt werden und ein Ausfall dieser zentralen Steuereinheit kann ebenfalls großen Schaden anrichten.

4.4.2. Power Grids

Doch es gibt auch klassische verteilte und vernetze Systeme, bei denen man nicht zuerst an Gefahren durch Software denkt. Verbundnetze für Gas, Wasser und elektrischen Strom sehen sich ebenfalls mit einer wachsenden Zahl von Gefahren aufgrund ihrer steigenden Komplexität konfrontiert. Dank immer größerem Energiebedarf und einer neuen Produktionskultur, die durch die Energiewende hin zu erneuerbaren Energien entstanden ist, haben sich die Anforderungen gegenüber früher drastisch geändert. Hat man bisher einfach in die Nähe der Verbraucher ein Großkraftwerk mit Kohle-, Gas- oder Atombetrieb gebaut, so findet die Stromproduktion immer dezentraler statt. Die Menschen wollen keine umweltverschmutzenden Kraftwerke in ihrer Nachbarschaft und werden dank Subventionen und steigender Strompreise mittels Photovoltaikanlagen, Biogas- oder Blockheizkraftwerken selbst zu Produzenten. Wasserkraft kann nur an Flüssen oder der Küste genutzt werden, Windparks werden in großem Umfang offshore errichtet.

Da hierdurch die Stromproduktion immer abhängiger von Standort und Wetter wird, muss ein modernes Hochspannungsnetz über weite Strecken Unterversorgungen und Überschüsse ausgleichen können. Erzeugungs- und Verbrauchsgebiete sind teilweise räumlich sehr weit voneinander entfernt, klassische Verbraucher speisen plötzlich ihre mit Solarzellen gewonnenen Überschüsse ein und die Ausfallsicherheit ist aufgrund unserer enormen Abhängigkeit von elektrischer Energie wichtiger denn je.

Um einem drohenden Kontrollverlust zu begegnen und mittels Optimierungen noch das Letzte aus den Netzen heraus holen zu können, werden automatisierte Computersysteme und Techniken aus der IT angewendet. Dabei handelt man sich auch deren Probleme ein, was bisher oft bei der Planung kaum berücksichtigt wurde. So sind die technischen Anlagen gegen Naturkatastrophen, Unfälle und menschliches Versagen mehr oder weniger gut gewappnet. Doch der Einfluss von Schadsoftware, Hackern und Datenverlust auf die Sicherheit der Steuerungstechnik wurde bisher unterschätzt.

4.4.3. Synergien nutzen

Für die Sicherheit eines Grids, egal welcher Art dieses ist, sind immer beide Welten zu beachten, die logische, die dem System inhärent ist, und die physische, in der sich das Grid befindet. In der IT mussten von Anfang an physische Fehler bei der Entwicklung beachtet werden, allerdings versucht man meistens die digitale Welt von der realen zu abstrahieren. Gerade Cloudcomputing zeigt das deutlich, da sich der Begriff Cloud von der Wolkendarstellung unspezifizierter Teile eines technischen Diagramms herleitet. Doch darf die physische Seite nicht vernachlässigt werden, wenn Sicherheit gefragt ist.

Konstrukteure klassischer Grids, wie es ein Stromnetz ist, haben die informationstechnischen Gefahren und Sicherheitsmechanismen hingegen häufig vernachlässigt. Doch da die Steuerungssysteme von kritischer Infrastruktur wegen Fernwartung und Kostensenkung vermehrt ans Internet angeschlossen werden, stellen Standardpasswörter [Zett 10], veraltete Software und fehlende Sicherheitsprogramme eine ernst zu nehmende Gefahr dar. Dies zeigen vermehrte Berichte über (angebliche) Angriffe auf Industrieanlagen [CIS 11] durch kriminelle sowie staatliche Hacker oder hochentwickelter und -spezialisierter Malware [Rieg 10].

Um nun die Erfüllung der Aufgabe, also gewissermaßen die Mission des Gesamtsystems, sicherzustellen, ist es manchmal unerlässlich, sich von einem Teil dieses Systems zu trennen, wenn dieses mehr schadet als nützt. Dabei sind sowohl Signale aus der digitalen als auch Messwerte aus der physischen Welt hilfreich, eine fundierte Entscheidung zu treffen, ob ein Teilverlust für das Überleben des Ganzen besser wäre. Da große Systeme kaum noch zentral zu überwachen sind, könnte ein dezentraler Ansatz bessere Ergebnisse erzielen. Daher bietet es sich an, dass sich jeder Teilnehmer im Netz selbst überwacht. Wenn irgendwelche Unregelmäßigkeiten festgestellt werden, kann sich so eine Zelle selbst beenden, falls sie davon überzeugt ist, dass dies dem Wohl aller dient. Den selbst eingeleiteten Zelltod nennt man in der Natur *Apoptose*. Dieses Konzept kann auch bei verteilten technischen Systemen angewendet werden, um die Durchführung der Mission eines derartigen *„künstlichen Organismus“* sicher zu stellen.

4.5. Bewertung der Konzepte bezüglich der Anforderungen aus Abschnitt 3.4

Wie sich der Tabelle 4.1 entnehmen lässt, erfüllt nur das *Autonomic Computing* (4.3) alle *Anforderungen* (3.4). Alle anderen Systeme lassen die eine oder andere Eigenschaft vermissen. Sie sammeln Daten (3.4.1) und reagieren entsprechend (3.4.3), wobei eine intelligente Verarbeitung der Informationen (3.4.2) deutlich seltener stattfindet. Bis auf *DEP und ASLR* (4.1.4) führen auch alle Konzepte ein Protokoll, meistens aber ohne ein Verzeichnis der anderen Komponenten (3.4.5). Eine Kommunikation zwischen den einzelnen Zellen (3.4.4) findet nur in einigen Fällen statt, dabei werden andersartige oder gar sich in fremden Netzen befindende Komponenten kaum berücksichtigt. Über die tatsächliche Mission wissen nur die wenigsten Zellen Bescheid (3.4.6). Wenn doch, dann kennen sie in der Regel nur ihre eigene Aufgabe, haben aber keine Ahnung, wie sich ihr Verhalten auf das ganze System auswirkt.

	gestellte *Anforderungen* (3.4)					
Konzept	**1. Anforderung:** Informationen sammeln	**2. Anforderung:** Wissen aus den Informationen ableiten	**3. Anforderung:** Reaktionen bestimmen und durchführen	**4. Anforderung:** Mit anderen Zellen kommunizieren	**5. Anforderung:** Verzeichnis- und Protokollplattform	**6. Anforderung:** Eine Zelle kennt die Mission
Virenscanner	✓	✓	✓	(✓)	✓	
Firewall	✓		✓	(✓)	✓	
IDS und IPS	✓	✓	✓	(✓)	✓	(✓)
DEP und ASLR	(✓)		(✓)			
Schutz vor Eindringlingen	✓	(✓)	✓		✓	
Schutz vor Elementarschäden	✓	(✓)	✓		✓	
Maßnahmen bei Ressourcenmangel	✓	✓	✓		✓	(✓)
Redundanz	✓		✓	(✓)	✓	
Autonomic Computing	✓	✓	✓	✓	✓	✓

Tabelle 4.1.: Auswertung bestehender Konzepte bezüglich den Anforderungen aus 3.4

Lediglich das noch hauptsächlich in der Forschung befindliche *Autonomic Computing* (4.3) bietet weitgehend die Eigenschaften, die von dem Framework verlangt werden. Doch betrachtet es zum großen Teil die Probleme des Erlernens der Aufgabe bei einer neuen Zelle, weniger deren Ausscheiden aus einem System zum Wohle des restlichen Gebildes.

Genau wie ein natürliches Lebewesen hat sich eine Komponente nicht dafür entscheiden müssen, in diese Welt gesetzt zu werden. Doch ist sie erst einmal *„geboren“*, kann sie sich durchaus fragen, ob die Welt ohne sie nicht besser dran wäre. Derartige *„Gedanken“* widersprechen dem Selbsterhaltungstrieb, den wir auch technischen Geräten mitgeben. Schließlich handelt es sich dabei um Investitionen, die sich möglichst lange auszahlen sollen. Doch können Situationen auftreten, in denen eine Komponente mehr stört als nützt und sich daher aus einer Mission zurückziehen können soll. Dieses Konzept des apoptotischen Verhaltens von Zellen wird im folgenden Kapitel 5 genauer betrachtet.

5. Apoptose als Lösungskonzept

In den Anwendungsfällen in Kapitel 2 finden sich viele Szenarien, die für den Einsatz des Konzepts der Apoptose sprechen. In der Natur wird Apoptose von Körperzellen genutzt, um negative Auswirkungen auf andere Zellen, und damit den ganzen Organismus, zu verhindern. Nur so ist das Überleben eines aus Milliarden Einzelzellen bestehenden Lebewesens möglich. Frei nach dem Motto, *„etwas Schwund ist immer“*, ist der Ausfall schon eingeplant. Keine Zelle im menschlichen Körper ist unersetzlich, nicht einmal die Millionen Gehirnzellen, die bei einem Vollrausch absterben. Dank Redundanz und Zellteilung kann schnell für Ersatz gesorgt werden. So werden besonders strapazierte Stellen ständig nachgebildet. Da die Haut die empfindlichen Innereien vor den Gefahren der Umwelt schützt, wächst diese besonders schnell nach, wie man an den Fingernägeln oder bei einem Sonnenbrand sehen kann.

Auch bei technischen Systemen ist der Ausfall von Teilen eingeplant. Um keine wichtigen Daten zu verlieren, werden Festplatten mittels RAID verbunden. Sollte eine Platte nicht mehr laufen, können dank zusätzlicher Speicherung der Paritäten die fehlenden Informationen wiederhergestellt und auf vorbereitete Ersatzplatten kopiert werden. Bei kritischen Systemen werden Server oder gar ganze Rechenzentren vorgehalten, um bei einem Ausfall einzuspringen.

Auf den Ausfall von Komponenten eines komplexen Systems ist man bei einer umsichtigen und vorausschauenden Planung in der Regel gut vorbereitet. Doch was passiert im Zweifelsfall bei einer Fehlfunktion? Im Gegensatz zu einem Ausfall können die Auswirkungen einer Fehlfunktion auf noch arbeitende Teile des Systems kaum vorhergesehen werden. Auch in der Natur gibt es das Problem von kranken Zellen, die den Erreger in den Körper streuen und so gesunde Zellen infizieren können. Dabei kann eine Krankheit durch Eindringlinge von außen in Form von Viren, Bakterien, Pilzen oder Parasiten induziert werden. Eine andere Form ist die Missbildung einer Zelle, genauer eine Mutation des Erbmaterials. Breiten sich diese kranken Gene aus, wobei sie ein unkontrolliertes Wachstum verursachen, sprechen die Mediziner von Krebs. Doch in den meisten Fällen greifen in der DNA verankerten Sicherheitsschalter und die Zelle stirbt. Dies nennt man Apoptose.

In der Natur findet sich das Konzept der *„Selbstabschaltung“*, um das Gesamtsystem schadfrei zu halten [Burb 07]. Hierdurch soll ein Überleben des großen Ganzen sichergestellt werden, auch wenn es das eigene Opfer verlangt. Dieses Konzept hat sich im laufe von Milliarden Jahren für viele Lebensformen als nützlich erwiesen, um die eigene Art zu erhalten.

Bereits einzellige Organismen können sich selbst töten, um das Überleben benachbarter Einzeller mit beinahe gleichem Gencode zu gewährleisten. Als sich Einzeller zu mehrzelligen Organismen zusammen schlossen, wurde es um so wichtiger, den Einzelnen für das wohl des Ganzen opfern zu können. Nur so ist überhaupt ein längerfristiges Bestehen eines komplexen Organismus zu erklären, da Krankheiten und Alter zu einem unkontrollierten Zelltod führen würden. Durch die Nekrose aber würde sich ein Körper sehr schnell selbst vergiften.

Bei der Apoptose kommunizieren die Zellen mittels chemischer Botenstoffe. Reichern sich diese über einen Grenzwert hinaus an, beginnen andere chemische Reaktionen die Zelle aufzulösen, sofern nicht eine blockierende chemische Nachricht dies verhindert. Apoptose kann von außerhalb und von innerhalb der Zelle ausgelöst werden. Die Gründe und der genaue Vorgang sind noch immer ein Forschungsthema mit mehr Fragen als Antworten. Doch grundsätzlich weiß man, dass unter anderem Stressfaktoren des endoplasmatischen Reticulums von innen heraus zur Auflösung einer Zelle führen können [ES 03]. Von außen gibt es ebenfalls viele Gründe. So können gesunde Zellen eine entartete Zelle zur Apoptose anregen, doch auch infizierte oder Krebszellen nutzen diesen Mechanismus, um die Abwehrzellen des Körpers auszuschalten [fBuF 12].

Apoptose führt aber auch im Laufe der Entwicklung eines Organismus zu Metamorphosen. Ist dies nicht der Fall, können bei einem Lebewesen Überreste der Evolution festgestellt werden, wie Schwimmhäute zwischen den Zehen beim Menschen oder Augen mit undurchsichtigen Glaskörpern. Ohne Apoptose wäre auch die

Entwicklung einer Kaulquappe zu einem Frosch nicht möglich. Dies liegt auch an der zunehmenden Spezialisierung von Zellen bei komplexer werdenden Lebewesen. Einige Zellen sind dann verzichtbar oder werden nur in bestimmten Situationen gebraucht. Würden diese bestehen bleiben, könnte dies einen unnötigen Ressourcenverbrauch und einen Nachteil beim Überleben bedeuten.

Dank Apoptose ist es Körperzellen möglich, sich in Zeiten eines Mangels oder einer Gefährdung selbst zu töten, um dem gesamten Organismus das Überleben zu sichern. So sterben unwichtige, unterversorgte Körperteile in Notsituationen zuerst ab. Aber auch Zellen, deren DNA mutiert ist, also deren Programmcode fehlerhaft ist, können Suizid begehen, um Krebs, und damit eine Gefährdung des ganzen Körpers, zu verhindern. In der Natur sehen wir ständig Apoptose. Immer im Herbst welken die Blätter der Laubbäume und Pflanzen sterben an der Oberfläche ab, um in der Wurzel und dem Stamm ihre Lebensenergie zu konservieren. Im Frühling wird diese Energie wieder frei und führt zu neuem Leben, die Krisenzeit Winter ist überstanden und das System als Ganzes hat in den Kernen der meisten Individuen überlebt.

In der Technik wird dieses Konzept, das zum Beherrschen von komplexen und verteilten Systemen essentiell ist, nur eingeschränkt angewendet. So können sich Computer selbst herunterfahren, um Schaden für die Hardware zu vermeiden. Dies geschieht bei Überhitzung oder einem Stromausfall. Mit S.M.A.R.T. soll ein baldiges Versagen von Festplatten rechtzeitig erkannt werden, damit eine Reserveplatte noch vor dem Ausfall übernehmen kann. Beim Ausbleiben von *Keep-Alive*-Nachrichten zwischen gleichartigen Komponenten wird eine Fehlfunktion des Partners angenommen und dessen Aufgabe übernommen. Letzteres ist ein Konzept, das man auch bei natürlichen Zellen findet. Diese tauschen ebenfalls *Keep-Alive*-Signale mittels chemischer Botenstoffe aus. Allerdings können besagte Botschaften auch bedeuten, dass die Zelle selbst nicht mehr richtig arbeitet. So nutzen Zellen, die von keiner anderen ihrer Nachbarn ein Lebenszeichen erhält die Apoptose, um sich selbst zu terminieren. Auf die Weise werden keine wertvollen Ressourcen mehr für eine fehlerhafte Zelle verschwendet, es wird Platz für eine neue, funktionierende Zelle geschaffen und mögliche schädliche Auswirkungen durch eine falsch arbeitende Zelle auf den Organismus verhindert.

Bei künstlichen Organismen, die von Menschen erdacht und geschaffen werden, ist diese Art der Gefahrenabwehr noch kaum implementiert. Die meisten Komponenten werden als unentbehrlich angesehen, ein kontrolliertes Teilabschalten ist oft nicht vorgesehen. Technik soll einen Zweck erfüllen und wird daher so gut wie möglich gebaut. Sollte doch mal etwas schief gehen, muss der Mensch eingreifen. Ist das System allerdings zu komplex, verstehen das die Verantwortlichen selbst nicht mehr in allen Einzelheiten und können nicht richtig oder schnell genug reagieren. In extremen Fällen, wie bei einer Naturkatastrophe oder einer Kernschmelze, ist eine Handlungsfähigkeit der Ingenieure und Rettungskräfte nicht gewährleistet oder nur unter Risiko für Leib und Leben zu schaffen.

Aber es muss ja nicht immer gleich eine Katastrophe sein, die durch Apoptose verhindert werden soll. In den allermeisten Fällen geht es darum, die Mission nicht zu gefährden. Diese heißt in der Natur Überleben, wenn dies auch nicht für jedes Individuum oder jede Spezies gilt. Im Gegensatz zu Stammzellen können sich die meisten Zellen nicht ewig teilen. Dies liegt unter anderem in ihrer Spezialisierung begründet. Bei primitiven Lebewesen dagegen ist theoretisch ein ewiges Leben möglich, solange deren DNA nicht mutiert.

5.1. Sammeln von Informationen

Um folgerichtig bei Gefahren reagieren zu können, sind möglichst viele relevante Informationen wichtig. Diese müssen alle Fälle abdecken, die zu einer Abschaltung führen können. Wenn also auf einen Feueralarm reagiert werden soll, müssen die Feuermelder, Rauchmelder oder Temperatursensoren abgefragt werden können, die für eine Auswertung einbezogen werden sollen.

5.1.1. Arten von Daten

Es gibt viele verschiedene Arten von Daten, die relevante Informationen liefern können. Da Computer im Prinzip nur rechnen können, kommen an und für sich nur Zahlen in Frage, da nur diese sich sinnvoll in eine Metrik eingliedern lassen. Auch das Parsen von Zeichenketten führt wieder zu Zahlen, die sich beispielsweise

lexikographisch ordnen oder vergleichen lassen. Text spielt in automatisierten Überwachungs- und Steuerungssystemen, abgesehen von der Nutzeroberfläche und bei Logdateien, kaum eine Rolle, weshalb hierauf nicht näher eingegangen wird.

Analoge Werte

Bei analogen Daten handelt es sich um reine Zahlen, die in der Regel von einem Sensor geliefert werden. Bei einem Temperatursensor könnte ein Wert in der Einheit Kelvin zurück geliefert werden. Doch da nicht alle Sensoren bereits fertige digitale Werte ausgeben, werden die Messwerte durch eine Spannung oder durch einen Strom repräsentiert. Ein elektrischer Temperaturfühler ist somit nichts anderes, als ein sich mit der Temperatur verändernder Widerstand, den man mit einem AD-Wandler messen kann. In Industrieanlagen werden Messwerte meistens mit einer Strommodulation transportiert, da der Strom sich bei langen Leitungen im Gegensatz zur Spannung nicht verändert. Bei SPS-Systemen liegen die Stromwerte meist zwischen 4 mA und 20 mA. Somit kann eine 0 durch 4 mA ausgedrückt von einer gestörten Verbindung (gleich 0 mA) unterschieden werden.

Für die weitere Verarbeitung ist es aber unerheblich, ob der Wert eines Signals schon eine Semantik hat, also schon eine durch Nullen und Einsen dargestellte gemessene Zahl ist, oder ob die Zahl die Messung mit einem AD-Wandler als Ursprung hat. Die Syntax ist immer die gleiche, denn ein AD-Wandler kann wiederum als Voltmeter begriffen werden.

Neben einem vorgegeben Intervall, in dem die Messung erfolgt, spielt auch die Granularität eine Rolle. Während ersteres von Spezifikationen und Standards vorgegeben ist, hängt letztere von der Auflösung des AD-Wandlers in Bits ab. So kann ein 10-Bit Wandler ausschließlich ganzzahlige Werte zwischen 0 und 1023 annehmen. Auch eine endliche Zahl mit Dezimalbruch lässt sich in eine Ganzzahl verwandeln, indem die Einheit kleiner wird. Intern ließe sich also mit einem Integer in Millivolt rechnen, auch wenn später Volt mit Nachkommastellen angezeigt werden soll.

Boolesche Werte

Boolesche Werte, also Falsch oder Wahr, werden in einem Computer meistens durch ein einzelnes Bit dargestellt, das wiederum als 0 oder 1 interpretiert werden kann. Einige Programmiersprachen erlauben das Rechnen mit booleschen und nicht booleschen Zahlen in einer Gleichung, da intern `False` mit *0* und `True` mit *nicht 0* übersetzt wird. Daher unterscheiden sich die binären Darstellungen von 1 teilweise, so werden bei `BASIC` alle Bits gekippt, womit `True` gleich -1 ist.

Unabhängig von der Darstellung kann ein boolescher Wert somit als Spezialfall von einem analogen Wert, beziehungsweise dessen digitaler Repräsentation, gesehen werden. Viele Methoden für die Verarbeitung von Ganzzahlen lassen sich unverändert auf Boolesche Werte anwenden, auch wenn die Bedeutung variieren kann. Eine Summe entspricht dann einem logischen *Oder* und eine Multiplikation einem logischen *Und*. Die Invertierung des Wertes, also ein logisches *Nicht*, könnte mit einer Gewichtung von -1 erreicht werden, wenn alle Werte größer 0 als `True`, alle anderen als `False` betrachtet werden.

Daten ohne Metrik

Die bisher vorgestellten Messwerte sind entweder boolesche, oder sie folgen einer Metrik. Doch kann auch die Auswertung von Daten ohne Metrik interessant sein. So ist die Frage, ob ein Wert in einer vorgegebenen Menge von Werten ist, kaum mit rein arithmetischen Mitteln und Vergleichen zu klären. Hierzu werden Mengenoperationen benötigt, die einen booleschen Wert zurück liefern, der angibt, ob das eingegebene Element in einer Menge enthalten ist oder nicht. Dieser lässt sich dann wie jeder andere boolesche Wert weiterverarbeiten. Mit einer derartigen Operation lässt sich zum Beispiel prüfen, ob ein geöffneter Port zu der Menge der erlaubten Ports gehört.

Keep-Alive-Nachrichten benachbarter Zellen

Keep-Alive-Nachrichten können zur Überwachung einer Verbindung oder der Lebendigkeit einer Nachbarzelle eingesetzt werden. Dies lässt sich auf zweierlei Wegen erreichen. Bleibt eine beliebige Nachricht aus oder überschreitet eine maximale Wartezeit, wird eine Fehlernachricht getriggert, die dem nachfolgenden Auswertenetz einen Timeout signalisiert. Die andere Methode ist das gezielte Verschicken von Pings, die im einfachsten Fall ein boolesches `True` beinhalten. Dieser Ping wird wie alle sonstigen booleschen Werte im Netzwerk des Frameworks ausgewertet. Hierbei muss zusätzlich ein boolesches `False` weitergeschickt werden, wenn ein Ping ausbleibt.

Fehlerfall

Sollte eine Aktivität im auswertenden Framework einen Fehler feststellen, kann sie eine Fehlernachricht triggern und an alle nachfolgenden Aktivitäten weiterreichen. Eine typische Fehlermeldung ist der eben erwähnte Timeout einer Verbindung. Fehlermeldungen unterscheiden sich von normalen Signalen durch eine andere Einheit. Diese ist sinnvollerweise ein vordefinierter Fehlertyp.

Mitteilung über Apoptose

Wenn eine Zelle sich selbst abschaltet, so sollte sie das üblicherweise mitteilen, damit das System den Ausfall registrieren und für Ersatz sorgen kann. Diese Nachricht kann auch direkt an benachbarte Zellen gerichtet werden. Daher muss eine Zelle in der Lage sein, eine derartige Mitteilung empfangen und verarbeiten zu können.

Intern verhält sich die Information wie jedes andere Signal auch. Es unterscheidet sich allerdings im Datentyp und der Wert kann je nach Implementierung keine oder eine bestimmte Bedeutung haben. Wichtiger ist der Name der sendenden Zelle, da dieser über den Ursprung und den Typ der Zelle Aufschluss gibt, was interessant für eine angemessene Reaktion sein kann.

Aufforderung zur Apoptose

Eine Zelle, die anderen Komponenten in einem System Probleme bereitet, kann hierüber durch die negativ beeinflussten Nachbarn in Kenntnis gesetzt werden. Dies kommt einer Aufforderung zur Selbstabschaltung gleich. Genau wie in der Natur wird diese Nachricht intern verarbeitet und führt nur dann zu einer Terminierung, falls kein apoptosehemmendes Ereignis eintritt.

Es kann sinnvoll sein, die Apoptose erst nach dem Überschreiten einer Schwelle durch genügend Abschaltbefehle in kurzer Zeit einzuleiten. Dies soll voreilige Reaktionen vermeiden. Diese könnten auch durch eine kranke Zelle ausgelöst werden, die gesunde Zellen eliminieren will. Ein derartiges Verhalten, mit dem Ziel, das Immunsystem zu schwächen, ist auch in der Natur zu beobachten [ES 03]. Dem lässt sich teilweise entgegenwirken, indem man verlangt, dass ein Terminierungsbefehl von verschiedenen Nachbarzellen kommen muss, damit eine Art Mehrheitsvotum entscheidet, welche Zelle tatsächlich die Gefahr darstellt.

5.1.2. Erfassen von Daten

Die Daten, die für die Entscheidung, ob Apoptose angewendet werden sollte, benötigt werden, müssen meistens von Quellen außerhalb des verarbeitenden Frameworks eingesammelt werden. Diese Quellen können Sensoren, Metadaten eines Betriebssystems oder von darauf laufenden Prozessen, entfernte Computer in einem Netz oder eine andere Instanz des Frameworks sein. Die Daten können eingesammelt, von außen eingebracht oder im Framework selbst erzeugt werden. Ebenso können Informationen beziehungsweise Entscheidungen des Frameworks an andere Systeme weiter gereicht oder zum Abruf bereit gestellt werden.

Zeitgesteuertes Abfragen über ein Netz

Am einfachsten ist der Abruf von auf anderen Systemen bereit gestellter Daten über einen Socket. Dies sollte zeitgesteuert in regelmäßigen Abständen erfolgen. Dabei erzeugt eine große Frequenz der Anfragen eine hohe Netzlast, eine zu geringe kann rechtzeitige Reaktionen verhindern.

(Interuptgesteuertes) Lauschen auf einem Port

Im Gegensatz zu einem abfragenden Erfassen der Daten, kann ein lauschender Socket geöffnet werden, der es entfernten Systemen ermöglicht, gezielt und nur bei Bedarf Informationen mitzuteilen und eine Verarbeitung selbiger zu triggern. Dies erzeugt weniger Netzlast bei sich selten ändernden Signalen. Auch die Anzahl der Auswertungen kann so reduziert werden, was Rechenzeit spart. Andererseits binden mehrere lauschende Sockets wieder Systemressourcen. Allerdings reicht ein einzelner Socket, wenn anhand der Quellenadresse diese identifiziert und somit das richtige Datenpaket im Framework erzeugt werden kann.

Manuelle Eingabe, RPC

Eine weitere Möglichkeit der Eingabe kann über eine entsprechende Funktion erfolgen. Dies kann mit einem Mechanismus wie der in Unixsytemen gebräuchlichen Pipe, einem Argument beim Programmstart, dem regelmäßigen Auswerten einer Datei oder einem *Remote Procedure Call* (RPC) bewerkstelligt werden. Im Framework erfolgt dann die Auswertung äquivalent zu den oben besprochenen Datenerfassungsmethoden.

5.1.3. Weitergabe von Ergebnissen

Die Ergebnisse der Auswertung durch das Framework müssen natürlich auch weitergeleitet werden. Dies kann ebenfalls über die eben aufgezählten Wege erfolgen. Gerade Sockets bieten sich wegen ihrer Universalität und breiten Verwendung in verteilten Systemen an. Aber auch der Aufruf lokaler Methoden oder Programme kann sinnvoll sein, vor allem wenn das Ergebnis eine Apoptose auslösen soll. Ein Beispiel wäre das Senden eines Abschaltbefehls an das Betriebssystem, beziehungsweise an das BIOS/EFI durch das *Advanced Configuration and Power Interface* (ACPI).

Zwecks Wartbarkeit und der Nachvollziehbarkeit von Entscheidungen ist ein Logging der Daten ein wichtiger Punkt. Dies kann durch ein einfaches Ausschreiben der Informationen in eine lokale Textdatei erfolgen. Allerdings kann diese im Fehlerfall oder nach der Anwendung von Apoptose beschädigt oder nicht mehr erreichbar sein. Daher ist es häufig sinnvoller, in eine entfernte Datenbank zu schreiben.

5.1.4. Normalisierung

Die übermittelten Daten können noch gewichtet werden, um zwischen Signalen mit verschiedener Priorität zu differenzieren oder um die Information zu invertieren. Vorher müssen die eingelesenen Daten aber zu meist auf den internen Wertebereich umgerechnet werden, denn unterschiedliche Signalquellen können verschiedene Repräsentationen eines Wertes liefern. Ein einfacher Temperatursensor modifiziert die Spannung, so dass diese erst auf eine Temperaturskala abgebildet werden muss. Andere Thermometer hingegen liefern bereits ein digitales Signal. Dieses kann direkt die gemessene Temperatur in Grad Celsius, Grad Fahrenheit oder Kelvin, aber auch ein binärer codierter Wert einer eigenen Skala sein, der wiederum einer Prozentangabe entspricht.

Auch sind Vergleiche verschiedener Sensorarten im Framework denkbar, so dass ein einheitlicher interner Werteraum unerlässlich ist. Um die Datenverarbeitung einfach zu halten, ist eine Vielzahl von Datentypen wenig konstruktiv, da dies bei den Vergleichen zu Umkonvertierungen führen würde. Boolesche Werte lassen sich wie oben gezeigt auch leicht als Integer ausdrücken. Auch Rasterisierer wie Analog-Digital-Konverter liefern Ganzzahlwerte. Integer haben außerdem den Vorteil, schneller als Fließkommazahlen verarbeitet werden zu können. Um den Fehler durch Rundungen beim Arbeiten mit den Werten gering zu halten, sollten diese um zusätzliche Dezimalstellen erweitert werden, was einer Fixkommadarstellung entspricht.

Noch vor der Projektion der externen auf die interne Skala ist eine Transposition vorzunehmen, denn nicht alle Sensoren liefern eine 0 für eine 0. Gerade in Industrieanlagen wird eine 0 mit einer Spannung oder Stromstärke ungleich 0 Volt beziehungsweise 0 Ampere über die Leitungen transportiert. Nimmt ein Signal tatsächlich einmal 0 Volt oder 0 Ampere an, ergibt dies intern eine negative Zahl, die eine Reaktion wie die Apoptose bedingen kann.

Somit ergibt sich für die Normalisierung folgende Gleichung:

$$f_{normalize}(x) = (x - i_{min})\frac{n_{max} - n_{min}}{i_{max} - i_{min}}w + n_{min}$$

x Eingangswert

$f(x)$ Normalisierter interner Wert

$[i_{min}, i_{max}]$ Intervall der Eingangswerte

$[n_{min}, n_{max}]$ Internes Intervall des normalisierten Wertes

w Gewichtung des Wertes.

Die Funktion $f_{normalize}$ hat als einzigen Parameter x, von dem zunächst die untere Grenze des Werteintervalls abgezogen wird. Somit sollte eine logische 0 auf die tatsächliche 0 eines Integers abgebildet werden. Jetzt kann mit dem Bruch das Eingangsintervall $[i_{min}, i_{max}]$ auf das interne Intervall $[n_{min}, n_{max}]$ skaliert werden. Nun wird eine Gewichtung w vorgenommen. Da eine 1 intern auch durch eine andere Zahl k dargestellt werden kann, um auf Fließkommazahlen verzichten zu können, gilt $w = \frac{j}{k}$, wobei $w = 1$ wenn $j = k$ mit $j, k \in \mathbb{N}$. Danach wird das ganze Intervall um n_{min} transponiert, wodurch auch intern der Wertebereich nicht bei 0 beginnen muss.

5.1.5. Aggregation

Da sich Entscheidungen in der Regel auf mehrere Daten stützen, wird ein Zusammenführen der verschiedenen Signale notwendig. Da die Werte alle den selben Datentyp haben und im Vorfeld normalisiert wurden, ist dieses mit einfachen mathematischen Funktionen, bei denen meistens über alle relevanten Signale iteriert wird, zu bewerkstelligen. Schwieriger ist es, eine geeignete Funktion zu bestimmen, die einen aussagekräftigen Wert zurück liefert. Vergleicht man mehrere Temperatursensoren in einer Maschine, so kann ein Durchschnittswert durchaus sinnvoll sein:

$$\frac{1}{N}\sum_{j=1}^{N} j$$

Noch interessanter ist die Durchschnittsfunktion aber, wenn man ein Signal über die Zeit normalisieren will. Dies kommt dann einem Integral über die gerasterten Werte gleich, dessen Ergebnis man, abhängig von der weiteren Verarbeitung, mittels Division durch die Anzahl der Messwerte normalisieren kann. Unter Umständen ist eine Eliminierung des kleinsten und des größten Wertes sinnvoll, so wie es bei Juryergebnissen oft gehandhabt wird. Damit lassen sich Ausreißer entfernen und Messschwankungen ausgleichen. Der Ausgangswert steigt somit nur langsam an, Reaktionen werden nur verzögert vorgenommen, ein vorschnelles Handeln wird ausgeschlossen. In der Elektrotechnik wird dieses Verfahren in einfacher Form zum softwareseitigen Entprellen von Tasten eingesetzt.

$$\frac{1}{N-2}((\sum_{j=1}^{N} j) - j_{min} - j_{max})$$

Werte verschiedener Semantik

Doch wie vergleicht man Werte mit unterschiedlicher semantischer Bedeutung? Für einige Größen gibt es physikalische Umrechnungsfunktionen, die die Semantik erhalten oder eine neue, sinnvolle Bedeutung haben. Will man einen elektrischen Transformator überwachen, sind Größen wie die Spannung und der Strom auf Primär- und Sekundärseite wichtig. Hier ist eine einfache Multiplikation der Werte möglich, da das Ergebnis $Volt \times Ampere = Watt$ in der Physik ebenfalls mit einer Semantik versehen ist. Wenn man aber auch noch die Temperatur des Kühlmittels in die Entscheidung mit einbeziehen möchte, ob ein Abschalten des Systems notwendig ist, sprich Apoptose stattfinden soll, wird ein einfacher Kumulator, wie eine Multiplikation, absurd. Was soll $Volt \times Ampere \times Kelvin$ sein? Zwar stehen Leistung und Temperatur in einem physikalischen Zusammenhang, doch ist eine mathematische Operation über Werte dieser Einheiten nicht sonderlich aussagekräftig.

In den meisten Fällen ist es daher ratsam, erst Werte mit einer gemeinsamen Semantik zusammen zu fassen, das Ergebnis mittels Vergleichsoperationen in boolesche Form zu bringen und diese Werte mit Funktionen der booleschen Algebra zu verknüpfen. Wenn `True` als *Zahl größer Null* definiert wird, sind Invertierungen der Werte durch einen negativen Faktor wie `-1` äquivalent zu einem booleschen `NOT`.

$$\neg b \equiv -b$$

$$mit \quad b, \in B, \quad F_B(x) = \begin{cases} True, & x > 0 \\ False, & sonst \end{cases}$$

Vergleiche

Signale lassen sich nicht nur mit arithmetischen, sondern auch mit vergleichenden Operatoren vereinen. Eine einfache Implementierung ist das Ermitteln von Minimum und Maximum der Eingangswerte. Die Ausgabe entspricht dann dem niedrigsten beziehungsweise höchsten Wert. Nachfolgende Funktionen müssen anhand der ursprünglichen Signalquelle entscheiden, ob der übermittelte Wert für sie relevant ist. Daher ist es notwendig, den Signalen nicht nur einen Wert, sondern auch den Namen der Quelle mitzugeben. Schwierig ist es, ein aggregiertes Signal zu benennen, da dabei die Beziehung zu den vorhergegangenen Signalquellen verloren geht.

Folglich ist es einfacher, zwei Werte mit Vergleichsoperatoren zusammenzufügen. Dies hat auch den Vorteil, auf Gleichheit prüfen zu können. Denn wenn zwei Werte gleich sind und immer das Maximum oder das Minimum weitergereicht wird, welches Signal soll das sein? Bei einem Vergleich mit einem Operator hingegen wird einfach ein boolescher Wert weitergereicht. Dadurch ist auch ein späteres Abfragen des Namens der Signalquelle nicht mehr notwendig. Natürlich ist so kein Vergleich von mehr als zwei Werten möglich, da das boolesche Ergebnis nur zwei Zustände annehmen kann.

5.2. Aus- und Bewerten von Daten

Zuletzt erfolgt die Entscheidung, ob Apoptose angewendet werden soll. Dazu müssen alle Integer in boolesche Werte umgewandelt werden, was über Vergleichsfunktionen geregelt werden kann. Verglichen wird mit voreingestellten Werten. Hierbei empfiehlt es sich, zu prüfen, ob ein Wert innerhalb eines Intervalls liegt. So können mit der Angabe von zwei Werten a und b sowie der Vergleichsoperatoren $\geq$ und $\leq$ alle benötigten Vergleiche vorgenommen werden.

$$x \in [a, b], \quad x, a, b \in \mathbb{N}$$

kann demnach mit

$$F_{a,b}(x) = x \geq a \wedge x \leq b$$

geprüft werden. Setzt man $\infty = -\infty$, dann entspricht dies einem Variablenüberlauf bei einem endlichen Integer. Folglich lässt sich der Zahlenraum als Ring sehen, womit folgende Vergleiche angewendet werden können:

$$\begin{aligned}
x \notin [a,b] &\equiv x \in]b,a[\quad = \quad x \in [b+1, a-1] \\
x = a &\equiv x \in [a,a] \\
x \neq a &\equiv x \in [a+1, a-1] \\
x \geq a &\equiv x \in [a, \infty[\\
x \leq a &\equiv x \in]-\infty, a]
\end{aligned}$$

5.2.1. Entscheidungsbaum

Finden nun mehrere Vergleichsoperationen dieser Art statt, erhält man einen Entscheidungsbaum. Dabei wird für jede Abzweigung ein Intervall bestimmt, das die Wahl des nächsten Zweigs beeinflusst. Hierbei dürfen sich die Intervalle einer Abzweigung nicht überschneiden, da sonst die weitere Abfolge nicht eindeutig bestimmt werden kann.

Ein Entscheidungsbaum stellt somit eine sehr einfache und damit effiziente und Ressourcen schonende Umsetzung eines Abwägeprozesses dar. Auch eine Anpassung an neue Parameter ist möglich. Sollen die Grenzen eines Intervalls verschoben werden, damit ein bestimmter Ast weniger oder öfter genutzt wird, müssen lediglich die Grenzen der benachbarten Intervalle korrigiert werden. Sollen jedoch Entscheidungen in der Höhe des Baums verschoben werden, ist eine Änderung des Frameworks an dieser Stelle notwendig, da sich folglich die Signalwege ändern. Dies zu automatisieren ist leider nicht trivial und hängt stark von der Implementierung des Frameworks ab.

5.2.2. Klassifizierung und Clustering

Außer Entscheidungsbäumen gibt es noch andere Klassifikationsverfahren wie die k-nächsten Nachbarn. Dabei wird die multidimensionale Umgebung eines unbekannten Elements betrachtet. Die Annahme ist hierbei, dass die Klasse dieses Elements der Mehrheit seiner k nächsten Nachbarn entspricht. Dabei sind die zu bildenden Klassen zusammen mit bereits klassifizierten Elementen vorgegeben.

Die teilweise recht ähnlichen Clusterisierungsalgorithmen hingegen sind meistens ergebnisoffen. Hierbei sollen dicht beieinander liegende Elemente zusammengefasst werden. Schwierig ist die Bestimmung einer scharfen Grenze zwischen den einzelnen Clustern.

Für dieses Framework eignen sich beide Methoden. Bei der Klassifikation gibt es nur zwei oder maximal drei Klassen: Apoptose *ja*, *nein* oder optional *vielleicht*. Bei Letzterem handelt es sich um eine Klasse auf der Grenze zwischen den beiden anderen, wo eine eindeutige Entscheidung nicht getroffen werden kann. Bei einer Clusterisierung entfällt das Initialisieren mit bekannten Trainingsdaten. Jede Missionsbeschreibung gibt einen Startpunkt vor, von dem aus der zu der Mission gehörende Cluster ermittelt wird. Kann dies bei einem Datenpunkt trotz Toleranzen nicht getan werden, so ist dieser ein Kandidat für Apoptose. Sogar das Fehlen eines Clusters oder das Ausbleiben neuer Datenpunkte ist ein interessantes abnormales Verhalten.

5.2.3. Outliner Detection

Outliner sind Punkte, die sich in einem multidimensionalen Featureraum nicht sinnvoll zuordnen lassen. Im Falle von Clustering sind das Punkte mit großem Abstand zu anderen, sie liegen also in einer Zone geringer

Dichte. Auch gibt es Algorithmen, die speziell zum Auffinden solcher Outliner entwickelt wurden. Outliner sind immer ein guter Anhaltspunkt für den Apoptoseeinsatz, doch können diese wiederum in Clustern auftreten, zum Beispiel wenn ein Schadprogramm die Kontrolle übernommen hat. Dann werden viele eigentlich ungewünschte Datenverbindungen geöffnet, so dass ein entsprechender Outliner nicht mehr alleine ist und somit nicht mehr als Fehler wahrgenommen wird. Jedoch dürfte ein plötzlich auftretender Cluster von Outlinern ebenfalls als Gefahr gesehen werden.

5.2.4. Neuronale Netze

Im Rahmen der Forschung zur künstlichen Intelligenz werden häufig neuronale Netze eingesetzt. Sie sollen das aus Neuronen aufgebaute menschliche Gehirn nachempfinden, um ähnliche Denkprozesse auch bei Computern zu ermöglichen. Dabei haben Neuronen den Vorteil des selbstständigen Lernens. Sie sind in der Lage, sich automatisch auf die eingehenden Daten einzustellen und immer bessere Ergebnisse zu liefern. Künstliche Neuronen haben in der Regel mehrere gewichtete Eingänge, die zusammengerechnet werden. Wird dabei ein Schwellwert überschritten, gibt das Neuron ein Signal aus.

Prinzipiell eignen sich künstliche neuronale Netze gut für die Entscheidung über Apoptose, da sie hochgradig anpassungsfähig sind. Sie haben aber einige Probleme, so sind wie bei den Klassifikatoren geeignete Trainingsdaten erforderlich und das Finden eines Optimums ist sehr zeitaufwendig. Außerdem sind schnelle und flexible Lösungen leider oft sehr teuer [Wind 11].

5.3. Angemessene Reaktion bestimmen und ausführen

Letztendlich steht am Schluss jeder Auswertung die Frage, ob Apoptose angewendet werden soll. Somit werden alle Eingaben immer auf einen booleschen Wert reduziert, egal welche der oben genannten Methoden angewendet wurde. Nachdem eines der in Abschnitt 5.2 dargelegten Verfahren den Wert `true` zurück geliefert hat, folgt als letzter Schritt eine Reaktion, die eine Verbesserung der Situation des überwachten Systems herbeiführen oder mindestens eine Verschlechterung verhindern soll.

5.3.1. Mögliche Reaktionen

Apoptose ist hierbei aber nicht immer mit dem kompletten und endgültigen Abschalten von dem sich selbst überwachenden System gleich zu setzen. Oft können auch weniger drastische Reaktionen ausreichend sein, um die Stabilität des Gesamtsystems wieder herzustellen. Dabei bietet es sich an, gestaffelte Aktionen durchzuführen, um die Gefahr möglichst minimalinvasiv zu beseitigen. Teilweise können auch mehrere Handlungen gleichzeitig sinnvoll sein, so das Absetzen einer Warnung und das der Situation angemessene Herabsetzen der eigenen Leistungsfähigkeit.

In aufsteigender Reihenfolge finden sich viele Reaktionen, die je nach Situation ausgewählt werden können.

Ressource ist unverzichtbar

Dieser Fall erscheint zunächst trivial. Eine unverzichtbare Ressource muss natürlich erhalten bleiben, koste es, was es wolle. Das Framework kann aber für die Überwachung der Ressource eingesetzt werden. Allerdings erschließt sich die Unverzichtbarkeit einer Ressource meist nur aus dem Kontext, in dem sie sich zur fraglichen Zeit befindet.

So gibt es auch eine vorübergehende Unverzichtbarkeit. Wenn alle Alternativen ausgefallen sind oder sich selbst verabschiedet haben, sollte die letzte überlebende Zelle des selben Typs erhalten bleiben. Überprüft werden kann dies, indem die Nachbarzellen Lebenszeichen von sich geben. Bleiben alle diese Zeichen aus, kann dies zweierlei bedeuten: Die Zelle ist die letzte Überlebende und unverzichtbar oder der fehlende Kontakt zu allen anderen Zellen weist auf eine Isolation hin, deren logische Konsequenz die Selbstabschaltung ist, da eine unerreichbare Ressource nur Energie verbraucht, ohne eine Gegenleistung liefern zu können.

Eine Zelle ist demnach nur unverzichtbar, wenn alle gleichartigen Zellen ausgefallen sind, aber noch Kontakt zu mindestens einer weiteren Ressource besteht. Schwierig wird es allerdings, wollte man einen kompletten Abhängigkeitsgraph modellieren, um auch unwichtige Teilnetze zu erkennen und in Gänze abschalten zu können. Dies ließe sich möglicherweise mit Verfahren aus der Deadlock-Erkennung erreichen, die bei vielen verteilten Ressourcen aber schlecht skalieren oder zu viel Overhead produzieren, um praktikabel zu sein.

Verlegung der Aufgabe

Ein gerne gepriesener Vorteil von Cloud Computing ist die Möglichkeit, Aufgaben dynamisch zu verteilen. Sollte eine Zelle Probleme ausmachen, kann sie demnach ihre Aufgabe an eine andere Zelle im Grid übertragen. Da häufig Server in virtuellen Maschinen laufen, ist es nicht weiter schwierig, gleich die ganze Zelle an einen anderen Ort zu verlegen, wo sie die Stabilität des Systems weniger belastet. Dies kann in einem Grid sogar den virtuellen Umzug eines ganzen Rechenclusters in ein anderes Rechenzentrum bedeuten. So kann im Falle einer Katastrophe die Sicherheit von Datenbeständen und Forschungsarbeiten erhalten bleiben, wenn ein Supercomputer aufgegeben werden muss. Natürlich ist nur ein vertrauenswürdiges Ziel geeignet, die Aufgaben weiterzuführen.

Temporäre oder teilweise Suspendierung

In der Natur findet jährlich Apoptose statt, um Kraft zu sparen. Im Herbst bilden sich die Blätter der Bäume zurück, werden welk und fallen ab. Wenn im Frühjahr die Wärme und die Zahl der Sonnenstunden zunimmt, treiben die Pflanzen neu aus. Dieses Konzept lässt sich in schlechten Zeiten auf Zellen eines Grids übertragen. Wenn die Kapazitäten eng oder teuer werden, kann es sinnvoll sein, weniger wichtige Aufgaben zurück zu stellen, bis der Engpass überwunden ist. Bei Prozessoren werden bereits Taktrate und Spannung gesenkt, wenn die Last niedrig ist. Auch können immer mehr Multicoreprozessoren Teile von sich abschalten, um andere, stark genutzte Kerne extra hoch zu takten. Der umgekehrte Weg wäre aber auch denkbar. Statt der Abschaltung einiger Kerne, um mit den anderen die Thermal Design Power durch Übertaktung ausreizen zu können, ließe sich die Anforderung senken, um das teilweise Abschalten des Prozessors ohne Übertaktung zu ermöglichen. Hierdurch werden elektrischer Strom und Kühlkapazitäten eingespart. Somit ließe sich ein Totalausfall eines Rechenzentrums im Krisenfall vermeiden.

Zurücksetzen auf einen unkritischen Zustand

Wurde vor dem Auftreten eines kritischen Fehlers ein Backup oder ein Snapshot gemacht, kann ein Zurücksetzten auf einen früheren Zustand der Maschine eine gute Alternative sein. Bei Datenbanksystemen finden sich Mechanismen wie inkrementelle Sicherungskopien, Logdateien und Undo/Redo-Funktionen. Sollte ein Fehler ein konsistentes Fortsetzen der Arbeit unmöglich machen, kann damit einem größeren Datenverlust vorgebeugt werden. Bei virtuellen Maschinen gibt es mit Schnappschüssen des laufenden Systems die Möglichkeit, einen kurzen Zeitsprung vor dem Versagen vorzunehmen. Änderungen, die zu dem Fehler geführt haben, lassen sich nun anders handhaben. Auch moderne Betriebssysteme erstellen vor gefährlichen Systemeingriffen einen Sicherungspunkt, um ein komplettes Neuinstallieren unnötig zu machen.

Bei dieser Form der Apoptose ist ein Datenverlust nicht ausgeschlossen, doch der Schaden ist deutlich geringer als bei der nächsten Methode.

Neustart

Sollte kein Backup oder Sicherungspunkt bestehen oder dieser unbrauchbar sein, bleibt oft nur ein kompletter Neustart des Systems. Sollte auch dies den Fehler nicht beheben, ist eine Neuinstallation meist nicht mehr zu umgehen. Bekannt ist der *„Blue Screen Of Death“* von Microsofts Windows, der häufig auf einen fehlerhaften Treiber oder gar einen Hardwaredefekt zurückzuführen ist. Ein standardmäßig automatisch durchgeführter Neustart hat keine bessernde Wirkung, so dass sich der Computer in eine endlose Rebootschleife begibt.

Oft erfahren Computer durch einen Neustart eine wundersame Heilung. Daher ist eine beliebte Einstiegsfrage bei Computerproblemen aller Art durch IT-Personal „did you try turning it off and on again?“, die besonders durch die britische Fernsehserie „The IT-Croud“ zum geflügelten Wort wurde [The]. Auch bei einfacheren Systemen wie Sensoren, Mikrocontrollern oder Telefonen kann ein Reset helfen. Dadurch wird das Gerät dazu veranlasst, den fehlerhaften Zustand zu verlassen und mit weitgehend sicheren Initialwerten in einen frischen Normalbetrieb überzugehen. Hierdurch können auch für andere Teilnehmer eines Netzes blockierte Ressourcen frei werden. So kann der Neustart eines Telefons eine reservierte Leitung freigeben, wodurch andere Teilnehmer wieder telefonieren können.

Endgültige Terminierung

Wenn auch ein Neustart keine Verbesserung bringt, sollte sich ein Gerät selbst ausschalten. Auf die Weise werden keine zum Betrieb benötigte Ressourcen, wie elektrischer Strom, verschwendet. Auch wird die Sicherheit der Computer erhöht. Ausgeschaltete Systeme können nicht gehackt oder von Malware befallen werden, andere Geräte infizieren oder im Betrieb stören und sind gegen schädliche Umwelteinflüsse besser gefeit.

Eine Terminierung ist aber nicht zwangsweise eine Gefahrenabwehrmaßnahme. Wie in der Natur kann eine (technische) Zelle ein geplantes Lebensende erreichen. Ist ihre Aufgabe erledigt, gibt es keinen Grund mehr, ungenutzt weiter zu laufen. Auch kann eine Unverfügbarkeit nach Ablauf einer gewissen Zeit gewollt sein. Es gilt als quasi unmöglich, einmal im Internet veröffentlichte Inhalte wieder zu löschen. Deshalb werden Konzepte wie der digitale Radiergummi angeboten. Hierbei soll durch das begrenzte Bereithalten eines Schlüssels eine chiffrierte Information nur für eine vorher definierte Zeitspanne lesbar sein.

Anfordern von menschlicher Hilfe

Wenn das Framework keine eindeutige Entscheidung treffen kann, welches die beste Erwiderung auf eine Gefahr für das Gesamtsystem ist, oder wenn eine Notabschaltung vorgenommen worden ist, bleibt als letztes Mittel nur noch das Eingreifen von Menschen. Auch wenn immer mehr Roboter immer mehr Arbeit übernehmen können, so sind menschliche Intelligenz und Feingefühl noch lange nicht ersetzbar. Es gibt zwar Roboter, die kaputte Festplatten selbstständig wechseln können und automatisierte Einkaufssysteme sollen rechtzeitig Ersatz ordern, doch schon einen simplen Fehler wie ein lockeres Kabel zu finden und zu beheben ist für eine Maschine kaum machbar.

Bei besonders kritischer Infrastruktur ist ein vollständig autonomes Handeln oft gar nicht gefragt. Neben einem durchaus berechtigten Misstrauen gegenüber Maschinen ist auch die Haftungsfrage im Schadensfall ein bedeutendes Hindernis. Da Maschinen nur so gut wie ihre Erbauer und Programmierer sind und Irren menschlich ist, können Fehler niemals ausgeschlossen werden. Dies ist auch wieder ein Grund, warum Apoptose eine Möglichkeit darstellen könnte, um die Auswirkungen des Versagens eines Einzelteils auf ein großes System zu verhindern. Deshalb müssen die Maßnahmen bei einer Entscheidung für Apoptose gründlich geprüft werden. Lieber wird einmal zu viel als zu wenig nachgefragt.

Das Konzept der Apoptose entstammt der Natur, die auf das Schicksal eines Individuums im Gegensatz zum Menschen keine Rücksicht nimmt. Sie kennt weder eine Schuld- noch Haftungsfrage, weder Versicherungen noch Juristen. Doch wer haftet, wenn Menschen durch eine fehlerhaft durchgeführte Apoptose zu Schaden kommen: der Entwickler des Systems, der Verkäufer, der Betreiber oder der Nutzer? Obwohl heutzutage Fahrzeuge vollkommen selbstständig einparken könnten, gibt es nur Einparkhilfen, bei denen der Fahrer Gas und Bremse bedienen muss. Auch in ein Flugzeug ohne Piloten würden sich die meisten Leute wohl kaum setzen, auch wenn im militärischen Bereich seit längerem autonome Flugdrohnen erfolgreich eingesetzt werden. Aber auch diese müssen in kritischen Situationen von Menschen gesteuert werden.

Somit sollte Apoptose eher bei Systemen eingesetzt werden, deren Fehler schlimmer wären als deren Ausfall. In Situationen, in denen Apoptose mehr schaden als nützen könnte, sollte immer ein Mensch entscheiden müssen.

5.3.2. Loggen und Bewerten der Entscheidungen

Auf jeden Fall sind alle relevanten Aktionen zu protokollieren und an die angrenzenden Zellen weiter zu reichen. Nur so kann die Entscheidungsfindung nachvollzogen und verbessert werden. Auch wenn ein Vorfall untersucht wird, ist ein genaues und in einigen Fällen sogar gerichtsverwertbares Log wichtig. Dieses sollte nicht nur auf dem betreffenden System geführt, sondern an einer anderen Stelle vorgehalten werden. Denn ist eine Apoptose erfolgt, ist in der Regel eine Zelle nicht mehr ansprechbar. Auch könnte die Apoptose durch einen Unfall wie einem Brand oder Wasserschaden ausgelöst worden sein, der natürlich lokale Daten unwiederbringlich zerstören kann. Weiterhin hat die Zelle eigentlich eine andere Aufgabe, als sich permanent selbst zu überwachen und zu beschreiben. Ein einfacher Sensor beispielsweise hat gar nicht die Speicherkapazität, um ein umfangreiches Protokoll vorhalten zu können. Bei größeren Systemen, die aus vielen Zellen bestehen, bietet sich ein Log mit einem Datenbankmanagementsystem an, da ein Datenbankserver oft schon vorhanden ist und umfangreiche Anfragen ermöglicht.

In jedem Fall ist immer zu speichern, was die Apoptose wann und in welcher Form ausgelöst hat. Daher bietet es sich an, jedem Signal, das im Framework verarbeitet wird, einen eindeutigen Namen zu geben. Hierdurch sind Signalwege zurückverfolgbar und die Ursache des Problems recht genau eingrenzbar. Auch die Meldungen an benachbarte Zellen sind zu speichern, da diese ihrerseits Gründe für weitere Abschaltungen sein können. Ein Protokoll führender Server, der eine größere Gruppe von Zellen überwacht, hat den Vorteil, einen Überblick über das Verhalten seiner Klienten bieten zu können.

Somit können Abschaltkaskaden der Zellen rechtzeitig gesehen und gestoppt werden. Diese können auftreten, wenn eine Apoptose einer Zelle die Lage in einem Teilnetz verschlechtert und dadurch andere Zellen ebenfalls zur Apoptose veranlasst. Als Begegnung eignet sich ein Mix aus dem Verbot weiterer Apoptose und dem Wiedereinschalten der bereits terminierten Zellen. Anhand der Aufzeichnungen kann unter Umständen auch eine automatische Anpassung der Parameter vorgenommen werden, die zu der Entscheidung über Apoptose geführt haben. Da man eine derartige Aktion als Apoptose der Apoptose betrachten kann, lässt sich folgern, dass sich auch diese Aufgabe mit einem generischen Apoptoseframework bewerkstelligen lässt. So ein Teilnetz kann wiederum als Zelle eines größeren Systems gesehen werden. Und letztlich ist es zu Wartungszwecken einfacher, nur **einen** Rechner ansprechen zu müssen, der seinerseits die Aufgaben an die Subzellen verteilt.

Der Ansatz der Subzellen kann auch in der Natur wiedergefunden werden. Der Körper einer Ameise ist aus einzelnen Zellen aufgebaut, die Ameise selbst kann wiederum als Zelle eines größeren Organismus, nämlich des Ameisenstaates, betrachtet werden. Dieser wiederum ist Teil eines Ökosystems, in dem schädliche Einflüsse oft gnadenlos eliminiert werden.

Natürlich stellt ein zentrales Element wie ein Logserver einen *single Point of Failure* da. Doch durch Redundanz oder einen virtuellen Server, der wiederum auf die Zellen verteilt ist, können Schäden vermieden werden. Andererseits existieren mit Knotenrechnern in Supercomputern, Zentralen bei Energieversorgern oder einem Router beziehungsweise einem Smart Meter in einem privaten Haushalt bereits Geräte, die diese Aufgabe für das entsprechende Teilnetz übernehmen könnten.

5.4. Adaptive Apoptose

Teile und herrsche ist ein altbewährtes Verfahren, um bei großen Projekten nicht den Überblick zu verlieren und die Technik kontrollierbar zu machen. Auch Apoptose zählt dazu, da jede Zelle nur sich selbst überwacht, wodurch die Komplexität der Kontrolle gering gehalten werden soll. Hierdurch wird auch die Anzahl von Fehlern geringer, leider aber nicht gleich Null. Zum einen kann die Implementierung der Apoptose fehlerhaft sein, zum anderen kann sich die Aufgabe der Zelle dahingehend ändern, dass die Entscheidung zur Apoptose auf falschen Annahmen beruht und daher unzulänglich ist. Ein bekanntes Beispiel hierfür ist der Verlust der Adriane-5-Rakete der ESA am 4. Juni 1996 [Glei 96]. Aufgrund eines Variablenüberlaufs beim Konvertieren einer 64 Bit Gleitkommazahl in einen 16 Bit Integer wurde die Selbstzerstörung der Rakete eingeleitet. Diese Apoptose war unnötig, da der Flug einwandfrei verlief. Doch das von der Vorgängerin Ariane 4 kopierte Programm war für die höheren Geschwindigkeiten der Ariane 5 nicht ausgelegt und verursachte eine Explosion von 500 Millionen Dollar Hardware.

Dieser Fall zeigt, dass auch eine korrekt arbeitende Überwachung fälschlicherweise eine Apoptose einleiten kann, wenn sich der Kontext geändert hat. Die Aufgabe, die eine Zelle erfüllen soll, kann man auch als Job oder Mission bezeichnen.

5.4.1. Die Mission einer Zelle

In der Natur gibt es zwei Arten von Zellen: die normalen, auf eine Aufgabe spezialisierten Zellen, und die generischen Stammzellen. Obwohl alle Zellen der selben Eizelle entsprangen und den vollständigen genetischen Code enthalten, können nur die generischen Zellen sich unbegrenzt vermehren und verschiedene Gestalt annehmen. Letztere sind für die Biologie und die Medizin viel interessanter, aber auch wesentlich schwerer zu handhaben. Bei technischen Zellen verhält es sich ähnlich.

Spezialisierte Zellen

Soll ein Gerät nur eine für immer gleiche, von Anfang an festgelegte Zahl von Funktionen bereitstellen, so ist die Definition der Mission oft nicht allzu schwierig. Dementsprechend kann ein extra angepasstes oder entwickeltes Framework mit geringem Aufwand erstellt werden. Meistens kann auf das Framework auch gänzlich verzichtet werden, da die überwachenden Funktionen direkt in die Firmware integrierbar sind. Dabei bieten Exceptions einen guten Weg, eine Fehlerbehandlung einzubauen. Allerdings berücksichtigt dieser Ansatz weder das Überwachen des korrekten Ablaufs eines Programms im Sinne der Aufgabenstellung, noch den Einfluss, den die Zelle auf das Gesamtsystem haben könnte. Sollte ein Gerät in einem Netz arbeiten, so ist demnach auch die Komplexität desselben bei der Entwicklung zu beachten. Hierdurch ergeben sich ähnliche Anforderungen an das Framework, wie es generische Zellen tun.

Generische Zellen

Früher gab es für jede Aufgabe dedizierte Maschinen, die explizit für diese entwickelt wurden. Die Zahl der Funktionen war meistens gering genug, dass ein Mensch sie alle erfassen und die Maschine dagegen testen konnte. Heutzutage sind viele Geräte mit Funktionalität überladen, um diese möglichst weit auf die Bedürfnisse des Kunden anpassen zu können. Aus Kostengründen werden hierzu bereits an sich komplexe und vielseitige Standardbauteile eingekauft und in Massenfertigung verbaut.

Möglich wurde dies Entwicklung erst mit dem Mikroprozessor, der mittels Software eine große Platine mit elektronischen Bauteilen ersetzen kann. Um nicht für jede Einsatzmöglichkeit einen neuen Mikrocontroller konstruieren zu müssen, werden diese immer generischer gebaut. Natürlich werden die Prozessoren auf Fehler geprüft, doch unter seltenen Umständen finden sich dann meistens doch welche. Ähnlich ist es mit der Software; diente diese früher nur einem Zweck, so muss sie heute viele Schnittstellen nach außen bieten, um für zukünftige, noch nicht näher spezifizierte Anwendungsfälle gerüstet zu sein.

Ein Framework, das eine generische Zelle überwachen soll, muss daher ebenso generisch für die jeweilige Mission angepasst werden. Ändert sich die Aufgabe, so muss auch die Entscheidung für oder gegen Apoptose auf angepassten Daten und Auswerteverfahren beruhen. Ein weiterer Vorteil eines generischen Frameworks besteht in der Test- und Wartbarkeit. So muss immer nur die korrekte Funktion einer einzigen Implementierung gezeigt und neue Funktionen nur einmal geschrieben werden.

Schwieriger ist es jedoch, diesem generischen Framework die Mission zu erklären. Wie kann normales und abnormales Verhalten bestimmt und beschrieben werden? Formale Sprachen, Diagramme und Pseudocode sollen die Verständigung zwischen Mitgliedern eines Teams erleichtern und das Projekt dem Menschen verständlich machen. Doch Maschinen begreifen diese Modellierungen nur eingeschränkt. Verschärfend wirken sich große Änderungen der Regeln bei einem Jobwechsel aus. Werte, die eben noch im tief grünen Bereich waren, können auf einmal eine Gefahr bedeuten. Am meisten Probleme machen allerdings schleichende Anpassungen und die Wiederverwendung einer Aufgabenbeschreibung für neue, nur geringfügig anders definierte Missionen. Dann kann ein Programm für die Vorgängerversion einer Rakete Daten falsch verarbeiten und eine Selbstzerstörung auslösen. Doch wie kann man auf schleichende Veränderungen geeignet reagieren? Soll man die Parameter

langsam nachführen, da sich die Aufgabe langsam ändert, oder besser Alarm geben, weil eine harte Grenze überschritten wurde, die immer einzuhalten ist? Wann erfüllt eine Zelle ihre Mission nicht mehr?

Kranke Zellen

Wenn eine Maschine ihre Aufgabe nicht mehr erfüllt, ist sie kaputt. Eine Steigerung davon ist, wenn sich die Arbeit unberechtigter Weise geändert hat, so dass ein Gerät jetzt gegen die eigentliche Mission vorgeht. Eine Zelle, die nicht mehr nach ihrer DNA, sondern nach der RNA eines Virus handelt, wird als infiziert bezeichnet. Auch bei Computern spricht man von einer Infektion, wenn schädliche Software auf diesem zur Ausführung gekommen ist. Das Gefährliche an kranken Zellen ist ihre Fähigkeit, andere anzustecken.

Doch wie erkennt man eine kranke Zelle? Neben dem Einsatz von Virenscannern, die Schadcode erkennen und eliminieren sollen, kommt noch die Heuristik zum Einsatz. Aber auch diese prüft nur auf fehlerhaftes Verhalten, nicht jedoch auf eines im Sinne der Mission.

5.4.2. Selbstregulierende Überwachung

Das Ziel muss es demnach sein, die Mission so zu definieren, dass ein Computer automatisch sich selbst auf deren Einhaltung hin überprüfen kann. Die Parameter dürfen weder zu streng sein, da das viele falsch-positive Meldungen ergeben würde, aber auch nicht zu lasch, um abnormales Verhalten erkennen zu können. Auch wäre es wünschenswert, würde sich die Konfiguration an veränderte Bedingungen anpassen.

Ein solcher Parameter könnte demnach die Zeit sein, die eine Zelle hat, um eine Aufgabe zu erledigen. Wird diese im Voraus falsch geschätzt oder berechnet, so kann die Zelle nicht erfolgreich abschließen oder zu lange fehlerhaft arbeiten. Wenn diese Zelle noch von einem ausgefallenen Nachbarn die Arbeit mit übernehmen muss, so sind alle Zeitvorgaben hinfällig. Eine neue Berechnung dieser Zeitvorgaben unter Einbeziehung der neuen Aufgaben ist nötig.

Andere Parameter können CPU-Last, Speicherverbrauch und geöffnete Kommunikationsverbindungen sein. Ein Number Cruncher sollte keine E-Mails verschicken, ein Prozess sich nicht unendlich oft und in schneller Abfolge vervielfältigen oder ein aufgelegtes Telefon weiterhin Aufnahmen des Mikrofons übertragen. All dies deutet auf eine Infektion oder einen Fehler hin, sofern dieses Verhalten nicht ausdrücklich gewünscht wurde. Andere interessante Daten bei der Überwachung eines Computers sind Temperatur und Stromverbrauch, aber auch boolesche Daten wie ein Feueralarm. Während die vorhin genannten Daten stark variabel und missionsabhängig sind, ist die zulässige maximale Betriebstemperatur von Hardware eher fix. Insbesondere bei kritischen Schaltern wie Feuer-, Flut- oder Erdbebenalarm ist eine Anpassung des Frameworks ungewünscht. Derartige Missionsparameter lassen sich darum auch gut modellieren.

5.5. Zusammenfassung

Die Natur scheint mit der Apoptose ein Konzept entwickelt zu haben, das den in Abschnitt 3.4 vorgestellten Anforderungen genügt. Sie ist eine interessante Methode, um den Bestand einer Mission in einem großen verteilten System sicherzustellen. Es gibt bereits viele Analogien zwischen technischen Zellen und natürlichen Zellen, wie zum Beispiel die Infektion durch einen Virus (4.1.1). Es liegt also nahe, bei vergleichbaren Problemen nach vergleichbaren Lösungen zu suchen.

Um die Apoptose in einem Grid untersuchen zu können, bedarf es zunächst eines Frameworks, das die in diesem Kapitel vorgestellten Verfahren implementiert. Damit können dann die besten Reaktionen auf mögliche Gefahren bestimmt und durchgeführt werden. Das folgende Kapitel 6 stellt ein derartiges Programm vor, bevor in Kapitel 7 dessen Implementierung und sein Einsatz beschrieben werden.

6. Konzept eines Frameworks zum Missionserhalt durch Apoptose

Bevor eine Zelle sich mittels Apoptose aus einer Mission zurückzieht, ist eine Evaluierung des Zustands der Zelle notwendig. Dies erfolgt ähnlich der Signalverarbeitung in einer natürlichen Zelle: Botenstoffe aus Nachbarzellen reichern sich an und blockierende Moleküle fehlen, so dass chemische Prozesse in Gang gesetzt werden, die die Apoptose der Zelle zur Folge haben.

Auch in einer technischen Umgebung ist ein derartiges Verhalten umsetzbar. Benachbarte Zellen tauschen Nachrichten aus, die in einer Zelle weiterverarbeitet werden. Dies soll durch ein Framework erfolgen, dessen Eigenschaften in diesem Kapitel beschrieben werden. Eine mögliche Reaktion, die dieses Framework auslösen kann, ist die Apoptose.

6.1. Paradigmenwahl

Der Ablauf der Signalverarbeitung, wie er in dem Aktivitätsdiagramm Abbildung 6.1 dargestellt ist, ähnelt stark dem Signalfluss in einer klassischen elektronischen Schaltung. Auch hier gibt es Signalquellen, wie Taster, Drehregler oder Sensoren. Diese reichen in Form von Spannungspegeln Informationen weiter an passive und aktive Bauteile, die zum Beispiel die Tastenanschläge entprellen oder die Eingangsspannung glätten, verstärken oder abschwächen. Nach dieser Modifikation werden mehrere Signale oft zusammengeführt. Hierfür bedarf es spezieller Bauteile mit mehreren Eingängen, wie AND-, OR- oder XOR-Gatter für digitale Daten. Analoge Signale können mittels Komperatoren verglichen und mit Multiplexern selektiert werden. Um diese Signale in digitale Werte zu konvertieren, werden häufig Schmitt-Trigger eingesetzt [Schm 38]. Zuletzt folgt die Schaltung der Reaktion, im einfachsten Fall ein Relais, das die Stromzufuhr zu dem zu kontrollierenden Gerät unterbricht.

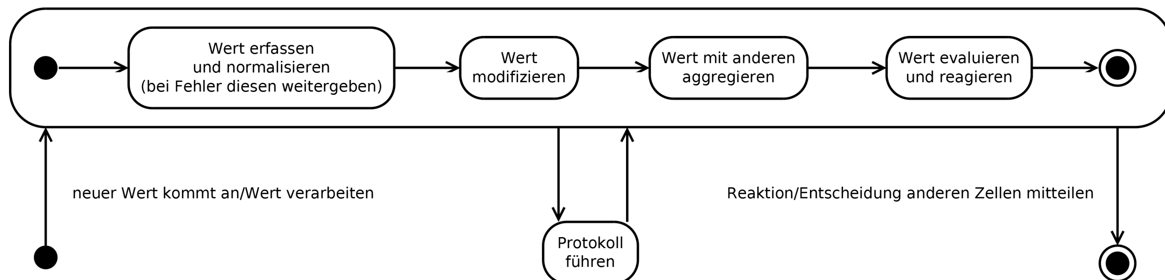

Abbildung 6.1.: Signalverarbeitung im Framework

Natürlich werden derartige Schaltungen heutzutage in Software geschrieben, wofür es extra Programmiersprachen und -umgebungen gibt. In der Industrie finden vielfach *Speicherprogrammierbare Steuerungen* (SPS) Anwendung, die mit einer maschinennahen Sprache wie *Step 5* oder in einer grafischen Umgebung wie *Step 7* programmiert werden. Bei Letzterer wird in Software ein Logikschaltplan nachgebildet.

Im Grunde entspricht also ein Aktivitätsdiagramm aus der *Unified Modeling Language* (UML) einem Signalflussdiagramm, wie es auch für die Modellierung von elektronischen Schaltungen verwendet wird. Der Aufbau des Frameworks zur Kontrolle einer Zelle orientiert sich daher an beiden Formen.

Aus der Abbildung 6.1 lassen sich vier grundsätzliche Schritte bei der Verarbeitung von Informationen in dem Framework einer Zelle erkennen: zunächst werden die Daten eingeholt, bei Bedarf modifiziert und mit anderen

Daten aggregiert. Zuletzt erfolgt eine Auswertung der Information und eine Reaktion, falls diese als sinnvoll erachtet wird.

Eine Möglichkeit der Implementierung wäre ein hart kodierter Ablauf, der für jede Art von Zelle und Mission erneut erstellt werden müsste. Durch die Wiederverwendung von häufig genutzten Modulen und den Einsatz von sich selbst modifizierenden Skriptsprachen lässt sich der Entwicklungsprozess vereinfachen, doch ist diese Methode wenig elegant. Eine generische Lösung, die mit einer standardisierten Beschreibung der Aufgabe initialisiert wird, erscheint deutlich geeigneter.

6.1.1. Prolog

Da ein Logikplan umgesetzt werden soll, bietet sich zunächst eine Logikprogrammiersprache wie Prolog an. Bei dieser wird vom Ergebnis ausgegangen, das der Computer berechnet. Prolog setzt auf Regeln, die mathematischen Gleichungen aus der booleschen Algebra entsprechen. Diese versucht der Computer nun durch das Einsetzen von geeigneten Werten in die freien Variablen zu erfüllen. Dazu verwendet er einen Backtrackingalgorithmus, der nacheinander alle möglichen Permutationen testet. Findet er mindestens eine Lösung, so gibt er `yes` aus, andernfalls `no`. Dies würde einer Antwort auf die Frage entsprechen, ob bei der gegebenen Datenlage eine Apoptose erfolgen soll.

Prolog wird hauptsächlich im akademischen Umfeld eingesetzt, vor allem beim automatisierten Beweisen oder bei der Forschung im Bereich Sprachverarbeitung und künstliche Intelligenz. Prolog ist nicht für den produktiven Gebrauch optimiert, so handelt es sich um eine Interpretersprache, die nicht für zeit- und ressourcenkritische Anwendungen bestimmt ist. Auch müsste die Missionsbeschreibung in Form von Prologregeln vorliegen. Ein generisches Framework ist so nicht sonderlich komfortabel umsetzbar.

6.1.2. Skript mit funktionalen Ansätzen

Ähnlich verhält es sich mit einem generischen Skript, das mit den zu verarbeitenden Daten und den zu verwendenden Funktionen gefüttert wird. Dabei entspricht eine Funktion einer Aktivität, die wiederum als Parameter in die sie aufrufende Aktivität übergeben wird. Soll ein einzelnes Datum in einer immer gleich bleibenden Reihenfolge verarbeitet werden, ist dies sicher eine elegante Lösung. Doch was ist, wenn die Anzahl der Arbeitsschritte variiert? Das Skript müsste eine dynamische Liste mit Funktionen verarbeiten können, die sich auch noch abhängig von Zwischenwerten in Länge und Reihenfolge verändern kann.

So könnte es sein, dass bei einem Wert erst zwei Modifikationen durchgeführt werden sollen, bevor eine Auswertung erfolgt, bei einem anderen Wert erst eine Modifikation, dann eine Auswertung und dann abhängig vom Ergebnis zwei weitere Modifikationen. Die zu verwendenden Funktionen stehen also beim Aufruf der Methode, der sie als Parameter zusammen mit dem zu verarbeitenden Wert übergeben werden, noch gar nicht fest. Auch die Verzweigung von Signalpfaden führt zu komplexen Konstruktionen.

Ein anderes Problem entsteht, wenn mehrere Werte verglichen werden sollen. So müssen immer alle Werte in die Startmethode eingegeben werden, zusammen mit den Funktionen, die deren Verarbeitung bestimmen. Dies führt zu generischen Methoden, die eine beliebige Anzahl von Werten und Funktionen als Parameter akzeptieren müssen. Das größte Problem sind aber Schleifen im Programmcode. Wenn also ein Zwischenergebnis an einem früheren Punkt in der Abarbeitungskette für einen Vergleich herangezogen werden soll, ergibt sich eine Rekursion bei der Parameterübergabe. Dies ist nicht mehr trivial aufzulösen, auch wird der Code extrem unübersichtlich.

6.1.3. Objektorientierte Lösung

Für ein generisches Framework empfiehlt sich ein modularer Aufbau. Jedes Modul hat eine spezifische Art, die eingegebenen Daten zu verarbeiten. Diese interessiert die anderen Module aber nicht, wichtig ist für sie nur eine gemeinsame Schnittstelle, die die Weitergabe der Daten ermöglicht, sowie ein einheitliches Datenformat. Das weist wiederum eine Verwandtschaft mit der Elektrotechnik auf, bei der jedes Bauteil gekapselt

seine Arbeit verrichtet und sich nach außen standardkonform gibt. So ist die Spannung und die Deutung der verschiedenen Potentiale vorgegeben, auch das Raster der Platine ist einheitlich.

Es liegt also nahe, die signalverarbeitenden Module als Objekte auszuführen. Die Funktionalität lässt sich nach außen kapseln und mit einer gemeinsamen Schnittstelle können die Objekte beliebig zueinander angeordnet und untereinander ausgetauscht werden. Entweder implementieren alle Objekte das gleiche Interface oder sie erben von einer gemeinsamen Basisklasse. Der zweite Ansatz hat den Vorteil, dass die Basisklasse für alle Objekte gemeinsame Funktionalitäten zur Verfügung stellen kann, wodurch diese nur einmal implementiert werden müssen. So kann sich die Basisklasse um die Anbindung des Objekts an die anderen kümmern, während die Spezialisierung nur das vorgegebene Codegerüst mit Funktionalität füllen muss.

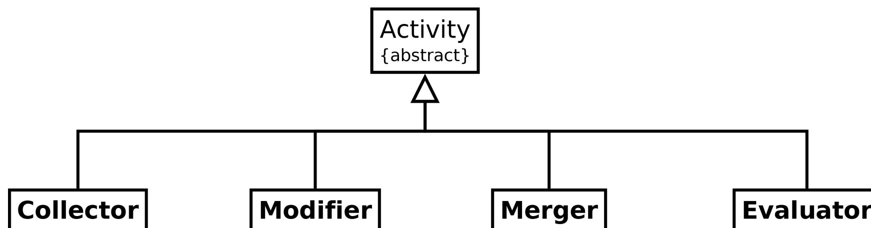

Abbildung 6.2.: Vereinfachtes Klassendiagramm der Aktivitäten

Orientiert man sich an dem zuvor betrachteten Aktivitätsdiagramm in Abbildung 6.1, ergeben sich vier Varianten von Spezialisierungen der Basisklasse `Activity`. Abbildung 6.2 zeigt das entsprechende Vererbungsdiagramm.

Von `Activity` erben `Collector`, `Modifier`, `Merger` und `Evaluater`. Dies sind wiederum die Basisklassen für die nächste Ebene. Sie stellen die grundlegenden Funktionen bereit, die von Aktivitäten benötigt werden, die die entsprechende Aufgabe ausführen sollen. Sie bestimmen also das *Was*, während ihre Kinder das *Wie* implementieren.

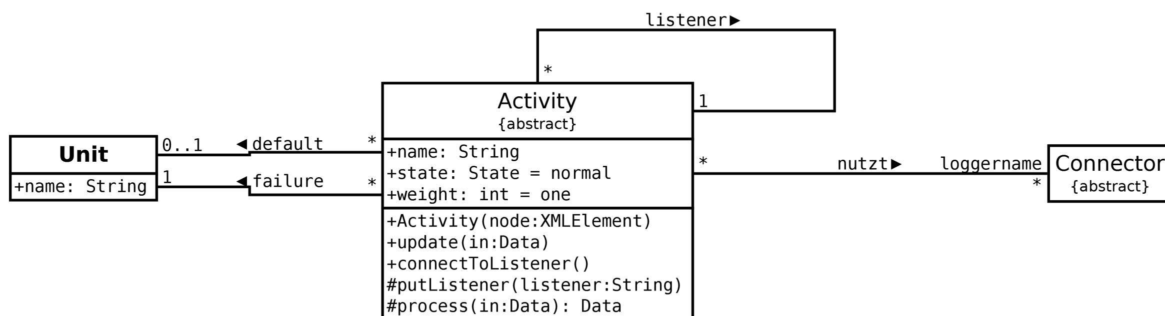

Abbildung 6.3.: Vereinfachte Ansicht der Basisklasse *Activity*

Abbildung 6.3 zeigt die wichtigsten Argumente und Operationen der abstrakten Klasse `Activity`, die eine Aktivität des Frameworks mitbringen muss. Sie wird mit einem Element aus einer XML-Datei (***E**xtensible* ***M**arkup* ***L**anguage* [BPSM+ 06]) initialisiert, welches alle relevanten Daten beinhalten muss. Einige dieser Werte sind optional, für diese muss ein geeigneter Defaultwert angegeben werden. Jeder Konstruktor der Subklassen liest die von ihm benötigten Daten aus und reicht das XML-Element zur Superklasse weiter.

Da die Referenzen auf eine `Activity` erst zur Laufzeit bestimmt werden, aber schon im Vorfeld die Verbindungen der Aktivitäten untereinander angegeben werden müssen, ist es erforderlich, jeder Aktivität einen eindeutigen Namen zu geben. Mit diesem können sich die Aktivitäten gegenseitig finden und miteinander verbinden. Dabei lauscht eine `Aktivity`, mit Ausnahme von `Merger`, immer genau einer anderen Aktivität. Diese wird mit dem Wert `listensto` identifiziert.

Eine Aktivität kann beliebig viele *Listener* haben, die sie über Änderungen des ihr zugeordneten Datums informiert. Dies erfolgt über die Methode `update()`, der als Argument das modifizierte Datum mitgegeben wird. Die nachfolgenden Aktivitäten erhalten so nacheinander das neue Datum, das sie ihrerseits verarbeiten

und weiterreichen. Je nach Implementierung kann es sinnvoll sein, den Updateprozess asynchron auszuführen, also jede Aktivität in einem eignen Thread laufen zu lassen.

Die Methode `update()` ruft ihrerseits `process()` auf, die einzige abstrakte Methode in `Activity`. Sie muss deshalb von den erweiternden Klassen implementiert werden. Ihr wird das Argument vom Typ `Data` von `update()` unverändert übergeben. Der Rückgabewert ist ein neues `Data`-Objekt. Jenes kann entweder die gleiche Einheit haben, wie das Eingangsdatum, oder es hat eine dem Objekt vorgegebene Defaulteinheit. Diese ist insbesondere für das Erzeugen eines neuen Datums notwendig. Außerdem kann auch eine von der globalen Defaulteinheit für einen Fehler unterschiedliche `failureUnit` bestimmt werden, die bei einem Fehler weitergereicht werden soll. Ist die Einheit eines Datums `null`, so wird die Weitergabe des Signals an nachfolgende Aktivitäten gestoppt.

Die Methode `connectToListener()` wird nur initial benötigt, um die Aktivitäten miteinander zu verbinden. Sie ruft ihrerseits die Methode `putListener()` von der zuvor in `listensTo` benannten Aktivität auf und übergibt den eigenen Namen als Argument. So wird eine Aktivität als Listener der vorhergehenden Aktivität eingetragen. Dies kann erst erfolgen, wenn bereits alle Aktivitäten geladen wurden, da erst dann alle Referenzen bekannt sind.

Jede Aktivität hat weiterhin ein optionales Gewicht `weight`, mit dem der Wert des Eingangsdatums multipliziert wird. Auch kann eine Aktivität einen mit `state` definierten Zustand haben. Ist dieser auf `blocked` gesetzt, werden von der Aktivität keine Daten weitergereicht. Dies ist nötig, um eine Auswertung nur dann vorzunehmen, wenn eine andere Bedingung erfüllt ist. So könnte man sich eine Reihe von Daten vorstellen, deren Bearbeitung und die daraus folgende Reaktion erst bei einem aktivien Feueralarm durchgeführt werden soll. Auch wenn selbst lernende Algorithmen im Framework zum Einsatz kommen sollen, ist es sinnvoll, während der Lernphase Reaktionen zu unterbinden. Das Lernziel kann zusätzlich in `state` angegeben werden.

Zuletzt bindet eine Aktivität noch einen `Connector` ein, der selbst eine abstrakte Basisklasse ist. Er dient der Protokollierung und Übermittlung von Ereignissen an andere Zellen oder sonstige Teile des Systems. `Connectoren` repräsentieren damit eine entfernte Zelle gegenüber dem Framework und vereinfachen so die Kommunikation und abstrahieren den Loggingmechanismus für die Aktivitäten. Ein weiterer Vorteil besteht darin, dass wenn viele Aktivitäten auf die Eingabe von Daten über ein Netz warten, nicht jede ihren eigenen Port öffnen muss, sondern dies an einer zentralen Stelle von einem einzelnen `Connector` gemacht werden kann. Empfängt dieser Daten, reicht er sie abhängig von der Quelladresse an die Aktivitäten weiter, die sich entsprechend registriert haben.

6.2. Umsetzung der Anforderungen aus Abschnitt 3.4

Da sich die zuvor ermittelten Anforderungen teilweise direkt auf den Komponenten des Frameworks abbilden lassen, werden diese nun mit der möglichen objektorientierten Umsetzung erläutert.

6.2.1. 1. Anforderung: Informationen sammeln

Um Daten zu erhalten, auf deren Basis man eine Entscheidung bezüglich einer Reaktion zwecks Missionserhalt treffen kann, müssen diese zunächst gesammelt werden. Das genaue Vorgehen ist in Abbildung 6.4 beschrieben.

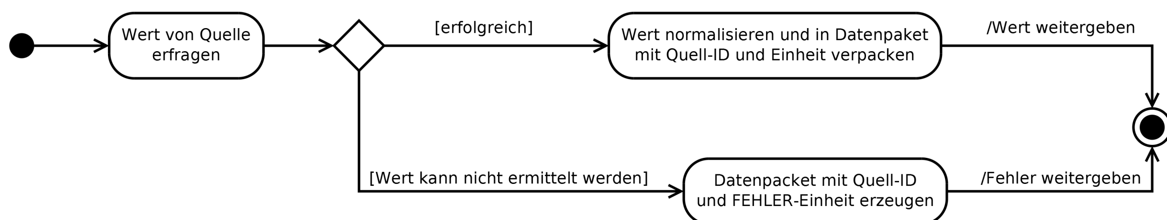

Abbildung 6.4.: Sammeln von Informationen

Zunächst wird der Wert von der Datenquelle gelesen. Der einfachste Fall ist bereits in der Klasse `Collector` (Abbildung 6.5) implementiert. Wird die Variable von `startvalue` bei der Initialisierung mit einem Wert belegt, wird dieser als Konstante verwendet. Beim Aufruf von `process()` wird dieser dann weitergereicht. Soll der Wert von einer externen Quelle eingelesen oder nach einer komplexeren Methode intern erzeugt werden, so muss in einer `Collector` erweiternden Klasse `process()` überladen oder `update()` mit dem neuen Datum aufgerufen werden. Dies kann zum Beispiel durch einen Thread erfolgen, der dieses periodisch erzeugt.

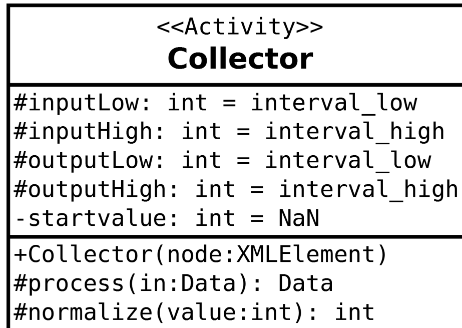

Abbildung 6.5.: Die Klasse *Collector*

Abbildung 6.5 zeigt weiterhin die Verwendung zweier Intervalle in der Klasse `Collector`: `inputLow` und `inputHigh` geben das Minimum und das Maximum der einzulesenden Werte an, `outputLow` und `outputHigh` den intern verwendeten Wertebereich. Beides wird von der Funktion `normalize()` benötigt, die die Eingabe aus unterschiedlichen Quellen auf ein einheitliches Niveau bringt und somit die Weiterverarbeitung im Framework vereinfacht.

`Collector` verpackt den neu gewonnen Wert in ein neues `Data`-Objekt, dessen Klasse in Abbildung 6.6 dargestellt ist. Dieses wird nun an die Aktivitäten weitergereicht, die sich beim `Collector` als Listener registriert haben.

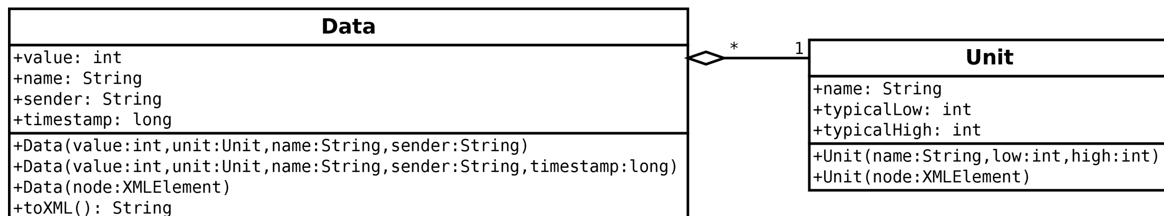

Abbildung 6.6.: Die Klassen *Data* und *Unit*

Ein `Data`-Objekt hat neben dem numerischen Wert auch einen Namen, der die Herkunft angibt. Somit kann der Weg eines Datums nachvollzogen werden. Zusätzlich enthält es eine Referenz auf eine `Unit`, die ihrerseits einen sie beschreibenden Namen enthält. Dieser kann zum Beispiel *„boolean“* sein, falls es sich um einen binären Wert handelt, oder auch *„Volt“*, wenn eine Spannung gemessen wurde. Zusätzlich kann eine Einheit noch ein typisches Messintervall enthalten, dass bei der Auswertung hilfreich sein kann.

Bei einem TTL-Analog-Digital-Wandler (*Transistor-Transistor-Logik*) könnte dies $[0, 5]$ Volt sein. Hat der Wandler eine Auflösung von 10 Bit, ergeben sich Werte von 0 bis 1023, wobei $0 = 0V$ und $1023 = 5V$. Demnach wäre das Eingangsinterval für die Normalisierung in `Collector` gleich $[0, 1023]$. Will man dies auf einen Prozentwert abbilden, der intern weiter verwendet werden soll, kann das Ausgangsintervall gleich [0, 100] gesetzt werden, womit letztendlich $0V = 0\%$ und $5V = 100\%$ ist. Die dabei verwendete Normalisierungsmethode entspricht der in Unterabschnitt 5.1.4 vorgestellten Formel:

$$f_{normalize}(x) = (x - i_{min})\frac{n_{max} - n_{min}}{i_{max} - i_{min}}w + n_{min}$$

x Eingangswert

$f(x)$ Normalisierter interner Wert

$[i_{min}, i_{max}]$ Intervall der Eingangswerte

$[n_{min}, n_{max}]$ Internes Intervall des normalisierten Wertes

w Gewichtung des Wertes.

Dabei ist immer zu bedenken, dass durch die Multiplikation großer Faktoren ein zu klein dimensionierter Integer schnell überlaufen kann. Intern kann daher ein größerer Integer (zum Beispiel 64 Bit) Anwendung finden, als nach außen angegeben wird, um auf jeden Fall genug Platz für ein Produkt aus mehreren kleineren Integern (32 Bit) zu haben.

Sollte die Abfrage des Wertes scheitern, zum Beispiel weil die Netzverbindung nicht hergestellt werden konnte, ist es oft sinnvoll, eine Fehlermeldung weiterzugeben. Dies geschieht, indem ein `Data`-Objekt erzeugt wird, das als `Unit` eine Fehlereinheit hat. Deren Name sollte selbstbeschreibend sein.

6.2.2. 2. Anforderung: Wissen aus den Informationen ableiten

Nachdem ein Datum erzeugt wurde, wird es weiter verarbeitet. Das Ziel ist es, dass Wissen generiert wird. Dies geschieht aufgrund der eingesetzten Aktivitäten. Derer gibt es zwei Sorten: `Modifier` bearbeitet immer genau ein Datum, `Merger` vereint mehrere `Data`-Objekte mit einer geeigneten Operation.

Modifizieren von Daten

Die simpelste Modifikation ist keine Modifikation, also *No Operation* (NOP). Diese wird von der Klasse `Modifier` bereits implementiert (Abbildung 6.7). Subklassen, die eine komplexere Operation durchführen sollen, müssen lediglich `process()` überladen.

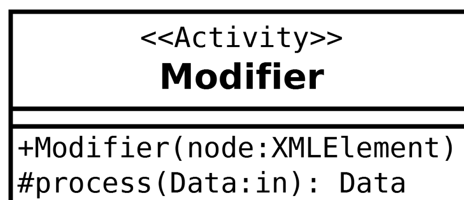

Abbildung 6.7.: Die Klasse *Modifier*

Der genaue Ablauf ist daher denkbar einfach, wie Abbildung 6.8 zeigt. Selbst bei einer NOP kann der Wert noch neu gewichtet werden. Damit kann die NOP auch zum Invertieren verwendet werden, indem das Gewicht gleich -1 gesetzt wird. Es ergibt sich eine Invertierung oder NOT-Operation, die auch für geeignete Darstellungen von binären Werten angewendet werden kann.

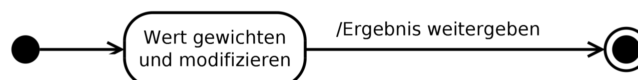

Abbildung 6.8.: Modifikation der Information

Komplexere Modifikationen eines Wertes sind eine Integration über die Zeit oder das Überprüfen, ob ein Wert in einer Menge von vorgegebenen Werten enthalten ist.

Zusammenführen von Daten

Sollen mehrere Daten miteinander verrechnet oder verglichen werden, so muss ein `Merger` eingesetzt werden. Diese Klasse hat, wie in Abbildung 6.9 gezeigt, mehrere `Ports`.

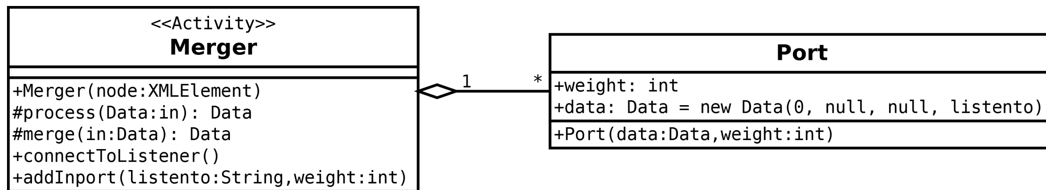

Abbildung 6.9.: Die Klasse *Merger*

Ein `Port` hält im `Data`-Objekt neben dem Wert, der an ihm anliegt, auch den Namen der Aktivität, die diesen liefert. Somit kann ein neu eingehender Wert dem richtigen Eingangsport zugewiesen werden. Jeder Port kann außerdem ein eigenes Gewicht erhalten.

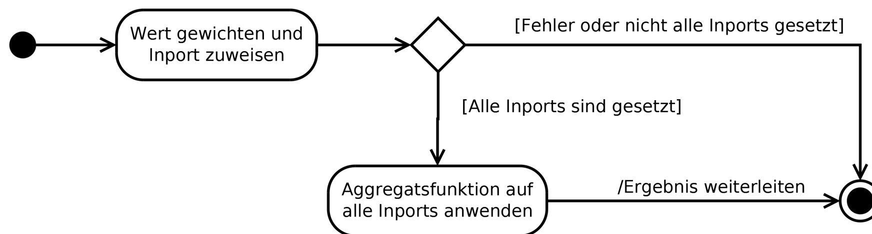

Abbildung 6.10.: Zusammenführen von mehreren Informationen

Wie Abbildung 6.10 zeigt, wird bei jeder Änderung an einem `Port` eine vorgegebene Aggregatsfunktion ausgeführt, sofern alle Eingänge mindestens einmal einen Wert erhalten haben, sich also in einem wohldefinierten Zustand befinden. Dann wird `merge()` aufgerufen, was die eigentliche Zusammenführung vornimmt. Wird diese Methode nicht von einer Subklasse überladen, liefert sie immer das zuletzt erhaltene Datum zurück.

`Merger` sollte die Methode `process()` mit einem Monitor synchronisieren, damit nicht mehrere Aktivitäten gleichzeitig Daten liefern können, wodurch nichtdeterministische Ergebnisse vermieden werden sollen.

6.2.3. 3. Anforderung: Reaktionen bestimmen und durchführen

Das letzte Element in der Verwertungskette in einem Framework ist die Evaluierung der Daten, so wie in Abbildung 6.11 dargestellt. Zunächst kann der Wert gewichtet werden, bevor geprüft wird, ob er sich innerhalb oder außerhalb eines vorgegebenen Intervalls befindet. Sollte das übermittelte Datum eine Fehlereinheit haben, so wird statt der Auswertung eine Fehlerbehandlung aufgerufen.

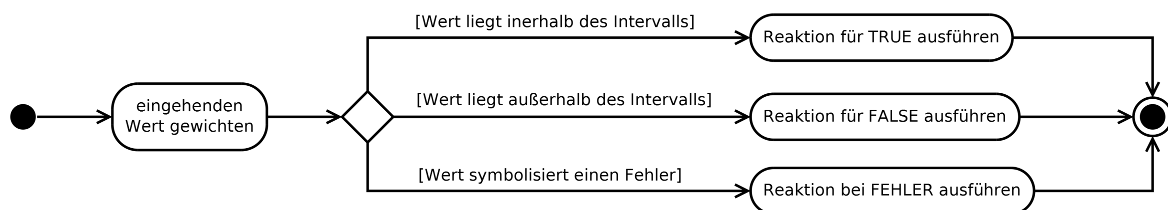

Abbildung 6.11.: Auswerten der Informationen mit entsprechender Reaktion

Die grundlegende Funktionalität der letzten Aktivität wird durch die Klasse `Evaluator` bereit gestellt. Sie erbt wie alle anderen von `Activity` und wird in Abbildung 6.12 gezeigt. Bei der Initialisierung sollten die Werte der Variablen `triggerLow` und `triggerHigh` gesetzt werden, um die Grenzen des Vergleichsintervalls abzustecken.

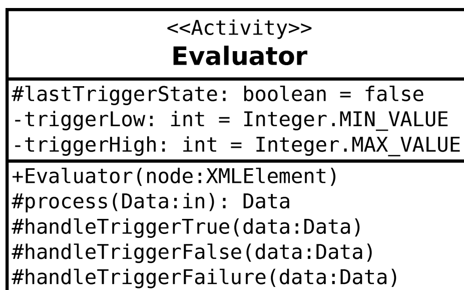

Abbildung 6.12.: Die Klasse *Evaluator*

Die Evaluierung erfolgt in der Methode `process()`. Zuerst wird geprüft, ob das Datum einen Fehler repräsentiert. Sollte das so sein, wird `handleTriggerFailure` aufgerufen, zusammen mit dem Datum als Argument. Selbiges ist auch der Rückgabewert von `process()` und damit von der gesamten `Activity`.

Bei der Auswertung kommen die Regeln aus Abschnitt 5.2 zum Tragen. Ist der gewichtete Eingangswert größer oder gleich `triggerLow` und kleiner oder gleich `triggerHigh`, dann liegt er im gegebenen Intervall und die Methode `handleTriggerTrue` wird ausgeführt. Ihr wird noch das eingelesene Datum mitgegeben, falls dieses in der Auswertemethode benötigt wird. Der Rückgabewert entspricht einem neuen booleschen `Data`-Objekt mit dem Wert für `TRUE`.

Im anderen Fall wird `handleTriggerFalse` aufgerufen, `process()` liefert demnach ein Datum mit der Einheit `boolean` und dem Wert für `FALSE`.

Wird die Klasse `Collector` nicht erweitert, so entspricht sie einem Schmitt-Trigger. Sie erzeugt aus analogen oder digitalen Werten einen booleschen Wert, führt ansonsten aber keine weiteren Reaktionen aus. Ein `Elevator` kann demnach vor eine boolesche Aggregation gesetzt werden, um auch analoge Werte mit einbeziehen zu können.

Falls eine Subklasse `handleTriggerTrue`, `handleTriggerFalse` oder `handleTriggerFailure` überlädt, können hier Reaktionen implementiert werden. Falls es hierfür benötigt wird, ist auch das vorherige Ergebnis in der Variable `lastTriggerState` hinterlegt.

Dadurch, dass immer das Ergebnis der Auswertung an potentiell nachfolgende Aktivitäten weitergereicht wird, ist ein Chaining möglich, wodurch mehrere Reaktionen einfach aneinander gereiht werden können. Sollen die Reaktionen sich nach dem Wertebereich unterscheiden, also zum Beispiel beim Erreichen eines Maximums eine Warnung und beim Überschreiten eine Notabschaltung durchgeführt werden, so können einfach zwei `Evaluator`-Objekte mit aufeinanderfolgenden Intervallen der vorhergehenden Aktivität lauschen.

6.2.4. 4. Anforderung: Mit anderen Zellen kommunizieren

Ein zentraler Punkt ist die Kommunikation von mehreren Zellen untereinander. Dies kann entweder indirekt über einen gemeinsamen Austauschpunkt oder direkt miteinander erfolgen. Letzteres ist schwieriger zu implementieren, kann zu zusätzlichen Nachrichten führen und erfordert eine höhere Intelligenz in der einzelnen Zelle. Dafür skaliert ein derartiges System besser als ein zentralistisches und weist keinen *single Point of Failure* auf. Um die Nachteile dieses Ansatzes zu verringern, kann man die Zentrale redundant auslegen oder gleich auf mehrere Komponenten verteilen. Im Extremfall sprechen die Zellen mit einer einzelnen Plattform, die ihrerseits auf alle Zellen aufgeteilt ist. Dies wird seit einiger Zeit erfolgreich bei Peer-to-Peer Systemen eingesetzt, um einen zentralen Indexserver zu vermeiden [SMK+ 01][RoDr 01].

Die Schwierigkeit für die Kommunikation der Zellen untereinander liegt in der Auffindung des richtigen Ansprechpartners. Hierfür bedarf es einer Lokalisierungstechnik, die unabhängig vom konkreten Standort einer

Komponente deren Nachbarn ermittelt. Ein Nachbar ist in diesem Fall eine andere Zelle, die die gleiche Mission wie die anfragende Zelle hat und in der Lage ist, die benötigten Ressourcen zu liefern. Dabei kann es sich um einen Messwert handeln, aber auch um die Fähigkeit, als Ersatz für eine ausfallende Zelle einzuspringen. Auch das Lokalisieren einer störenden Komponente ist ein wichtiger Punkt. Ist der Gefahrenherd für eine Mission eingekreist, kann er eine Aufforderung erhalten, die negative Beeinträchtigung des System zu unterlassen.

Es werden nun drei Nachrichtenarten betrachtet, die zwischen den Zellen ausgetauscht werden. Dabei handelt es sich um *Keep-Alive-Pakete*, die eine funktionierende Verbindung zwischen den Zellen sicherstellen, *Apoptosebenachrichtigungen*, die das System über den (baldigen) Ausfall der Zelle informieren, und um Meldungen, die eine andere Zelle zur Apoptose anregen sollen. Alle drei Mitteilungsarten kommen in ähnlicher Form auch bei natürlichen Zellen vor [ES 03].

Keep-Alive-Signal

Wie bereits in Unterunterabschnitt 5.1.1 beschrieben, kann das Versenden von regelmäßigen Signalen die Lebendigkeit einer Nachbarzelle oder einer Verbindung überwachen. Bleiben zu viele Pings aus, deutet dies auf einen Fehler hin, der unter Umständen weitere Maßnahmen erfordert. Tabelle 6.1 listet die Daten auf, die in einem solchen Ping enthalten sein sollten.

Elementname	**Datentyp**	**Beschreibung**
Name	URI bzw. String	Die Bezeichnung des Datums (Ping)
Sender	URI bzw. String	Der Name der sendenden Zelle
Einheit	Unit bzw. String	Gibt an, dass es ein Keep-Alive-Paket ist
Wert	Integer	Ein optionaler Wert, z.B. eine Sequenznummer oder Priorität
Zeitstempel	Timestamp bzw. Long	Zeitpunkt des Absendens

Tabelle 6.1.: Inhalt eines Keep-Alive-Pakets

Das *Keep-Alive-Paket*, beziehungsweise dessen Ausbleiben, ist bereits die gewünschte Information. Eine Reaktion ist nur dann erforderlich, wenn sich zu viele Ausfälle ereignen. Die Wahl der Sendefrequenz sollte sorgsam getroffen werden: zu lange Pausen können zu einer zu späten Entdeckung einer kritischen Situation führen, eine zu hohe Frequenz sorgt für einen unnötig hohen Datenverkehr.

Um auch Timingprobleme aufspüren zu können, sollte ein Zeitstempel mitgesendet werden. Dieser kann einer logischen oder einer absoluten Uhr entspringen, was eine ausreichende Synchronisierung der Uhrzeiten im gesamten System voraussetzt. Auch der Einsatz von Sequenznummern wäre denkbar. Weiterhin ist der Name der Quelle des Pings wichtig. Dieser sollte bereits in der empfangenden Zelle bekannt sein, wenn das Versenden der Pings bidirektional erfolgt.

Sollte ein Ping ausbleiben, muss in den meisten Fällen nicht umgehend reagiert werden. Erst wenn zu viele Signale aus- und Rückfragen unbeantwortet bleiben, sind weitere Maßnahmen erforderlich. So kann eine Zelle einen Ausfall melden, sich nach Alternativen für die fehlende Ressource umsehen, selbst als Ersatz einspringen oder gar die eigene Arbeit einstellen, falls sie nicht mehr in der Lage ist, ihre Mission zu erfüllen.

Die zu überwachenden Verbindungen sind meistens schon bei der Initialisierung der Zelle bekannt. Doch wenn alte Verbindungen abreißen und neue aufgebaut werden, ergibt sich eine Dynamik, über die die Zelle informiert werden muss. Entweder werden nicht mehr erreichbare Nachbarn aus einer entsprechenden Liste in der Zelle aus- und neue eingetragen, oder die Referenzierung ist so abstrakt, dass sie ohne Veränderung auf den Ersatz verweist. Wie sich Zellen finden können, wird im folgendem Unterunterabschnitt 6.2.4 und in Unterunterabschnitt 6.2.5 genauer erläutert.

Benachrichtigung über Ausfall

Die zweite wichtige Nachricht, die Zellen miteinander austauschen können, ist die Mitteilung über eine baldige Apoptose, wie auch schon in Unterunterabschnitt 5.1.1 beschrieben. Dadurch werden Nachbarzellen und das System gewarnt, dass sie bald ohne die ausfallende Zelle auskommen müssen. Eine wahrscheinliche Reaktion ist die Beschaffung von Ersatz.

Elementname	Datentyp	Beschreibung
Name	URI bzw. String	Die Bezeichnung des Datums (Apoptosemitteilung)
Sender	URI bzw. String	Der Name der sendenden Zelle
Einheit	Unit bzw. String	Der Grund für die Apoptose
Wert	Integer	Ein optionaler Wert, z.B. eine Dringlichkeit
Zeitstempel	Timestamp bzw. Long	Zeitpunkt des Absendens

Tabelle 6.2.: Inhalt einer Apoptosemitteilung

Eine typische Meldung einer Zelle, die kurz vor der Terminierung steht, könnte wie in Tabelle 6.2 gezeigt, aussehen. Neben dem Namen des Datums und des Senders finden sich auch der Absendezeitpunkt, eine Einheit und ein Wert. Damit entspricht eine derartige Nachricht ebenso wie ein *Keep-Alive-Signal* (6.2.4) dem Datenpaket, das in Form eines `Data`-Objekts auch innerhalb des Frameworks einer Zelle eingesetzt wird. Durch diese Wiederverwendung vereinfacht sich die Implementierung, da ein einheitlicher Parser eingesetzt und das Ergebnis ohne großen Konvertierungsaufwand weiterverarbeitet werden kann. Weiterhin findet bei der Weitergabe ins Framework kein Informationsverlust statt und neue Pakete lassen sich einfach aus `Data`-Objekten erstellen.

Es gibt viele Verfahren, um die Meldung zu versenden. Drei verschiedene Ansätze werden nun genauer anhand in UML erstellter Sequenzdiagramme besprochen.

Verteilte Kommunikation findet zwischen den einzelnen Zellen direkt mittels Broadcast oder Multicast statt, wie es in Abbildung 6.13 dargestellt ist. Nachdem die Zelle A einen Wert von einem Sensor abgerufen hat, der ihr eine Gefahr signalisiert, beschließt sie aufgrund der Auswertung durch das Framework, sich aus der Mission zurückzuziehen, um diese schadfrei zu halten.

Da ihre Arbeit noch nicht abgeschlossen ist, versucht die Zelle Ersatz für sich zu organisieren. Sie erfragt von einem Verzeichnis der Mission zugeordnete Zellen, die noch freie Kapazitäten haben könnten. Diesen teilt sie ihre Absicht mit, sich zu terminieren, zusammen mit dem Grund für die Apoptose. Die angesprochenen Zellen entscheiden nun ihrerseits, ob sie von der Gefahr betroffen sind und ebenfalls Maßnahmen ergreifen müssen, oder ob ihre Ressourcen und Fähigkeiten ausreichen, die Aufgabe von Zelle A zu übernehmen.

In dem hier dargestellten Fall reagiert Zelle B nicht, während Zelle C sich bereit erklärt, die Arbeit von Zelle A weiterzuführen, was sie umgehend an das Zellverzeichnis meldet. Dieses kann nun den Status von Zelle C ändern und auch Referenzen, die im Rahmen der Mission auf Zelle A zeigen, auf Zelle C ausrichten. Nachdem Zelle C die Aufgabenstellung von Zelle A zusammen mit eventuell fertigen Zwischenergebnissen erhalten hat, führt sie die Mission fort. Dieser Jobtransfer kann direkt von Zelle A zu Zelle C erfolgen, aber auch von einem Kontrollzentrum abgerufen werden. Die erste Variante ist allerdings nicht mehr möglich, wenn Zelle A nicht mehr fähig ist, die Daten zu übertragen.

Je nach Umsetzung des Zellverzeichnisses kann ganz auf eine zentrale Komponente mit all ihren Nachteilen verzichtet werden. Allerdings muss eine Zelle immer in der Lage sein, noch vor dem Ausfall ihre Nachfolge zu regeln.

Ein gemeinsamer Speicher bietet eine Plattform, über die sich die Zellen untereinander austauschen können. Sie funktioniert dabei wie ein schwarzes Brett, auf dem eine nicht mehr arbeitsfähige Zelle ihre Mission zusammen mit den benötigten Fähigkeiten ausschreibt. Alle anderen Zellen, die noch freie Kapazitäten

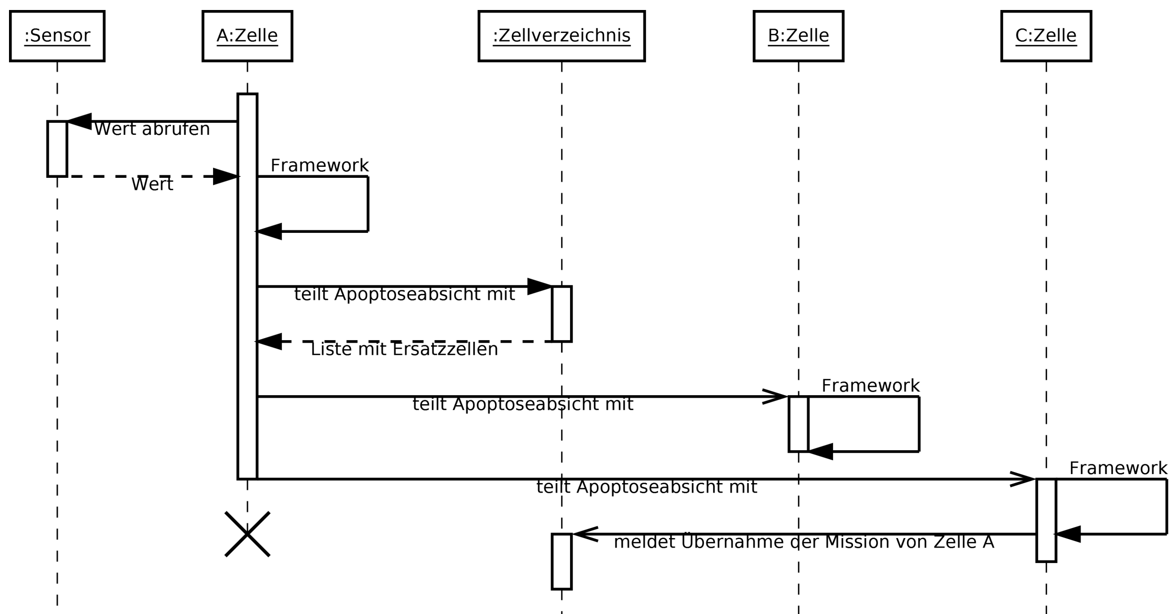

Abbildung 6.13.: Broadcast der Apoptose von Zelle A

aufweisen, sehen in regelmäßigem Abstand nach, ob eine für sie geeignete Arbeit ausgeschrieben ist, die sie gleich über- und vom schwarzen Brett abnehmen. Dieser Vorgang wird von Abbildung 6.14 gezeigt. Zelle A teilt ihre Terminierungsabsicht ausschließlich dem Zellverzeichnis mit. Sowohl Zelle B als auch Zelle C fragen regelmäßig bei diesem Index nach, ob neue Arbeit verfügbar ist. Da dies Zelle C im richtigen Augenblick macht, erhält sie den Job von Zelle A, Zelle B geht leer aus.

Der Ablauf erfordert kaum Intelligenz in den Zellen. Allerdings finden auch im problemfreien Fall viele Anfragen durch die unbeschäftigten Zellen statt. Wenn der gemeinsame Speicher eine zu große Anzahl von Komponenten versorgen muss oder das Netz unterdimensioniert ist, kann der Overhead das Versenden von wichtigen Nachrichten oder Nutzdaten blockieren.

Ein Broker ist eine weitere Methode, die Aufgabe von A auf eine andere Zelle zu übertragen. Dabei informiert Zelle A nur das Zellverzeichnis, welches eine Liste mit potentiell verfügbaren Zellen vorhält. Wird ein geeigneter Kandidat gefunden, erhält er die Zuteilung des Jobs. Dies ist in Abbildung 6.15 dargestellt.

Die Zellen B und C registrieren sich im Zellverzeichnis, das als Broker fungiert, als arbeitsfähig. Nachdem die Zelle A dem Verzeichnis ihre Apoptose angekündigt hat, wird ihre Aufgabe an Zelle C übertragen. Auch die Zelle B käme in Frage, doch Zelle C hat sich früher beim Broker eingetragen. Diese Methode hat den Vorteil einer geringen Anforderung an die Intelligenz der Zellen, ohne die hohen Zugriffszahlen des gemeinsamen Speichers zu benötigen. Allerdings erkauft man sich diesen Vorteil mit einem aufwändigen Zellverzeichnis und dem höheren Ausfallrisiko aufgrund der Verwendung eines zentralen Elements.

Aufforderung zur Apoptose

Wie schon in Unterunterabschnitt 5.1.1 erwähnt, bedeutet die Aufforderung zur Terminierung durch eine oder mehrere Nachbarzellen nicht unmittelbar die Selbstabschaltung einer Zelle. Sie ist vielmehr als Hinweis zu verstehen, dass bei der angeschriebenen Komponente etwas im Argen liegt. Abbildung 6.16 zeigt drei Zellen, die zusammen an einer Mission arbeiten. Zelle A und C erhalten von der mittleren Zelle B regelmäßig Werte. Zelle B wird von ihren Nachbarn zur Apoptose aufgefordert, falls die von ihr übermittelten Werte

- fehlerhaft sind,
- auf ein Problem in Zelle B hindeuten,

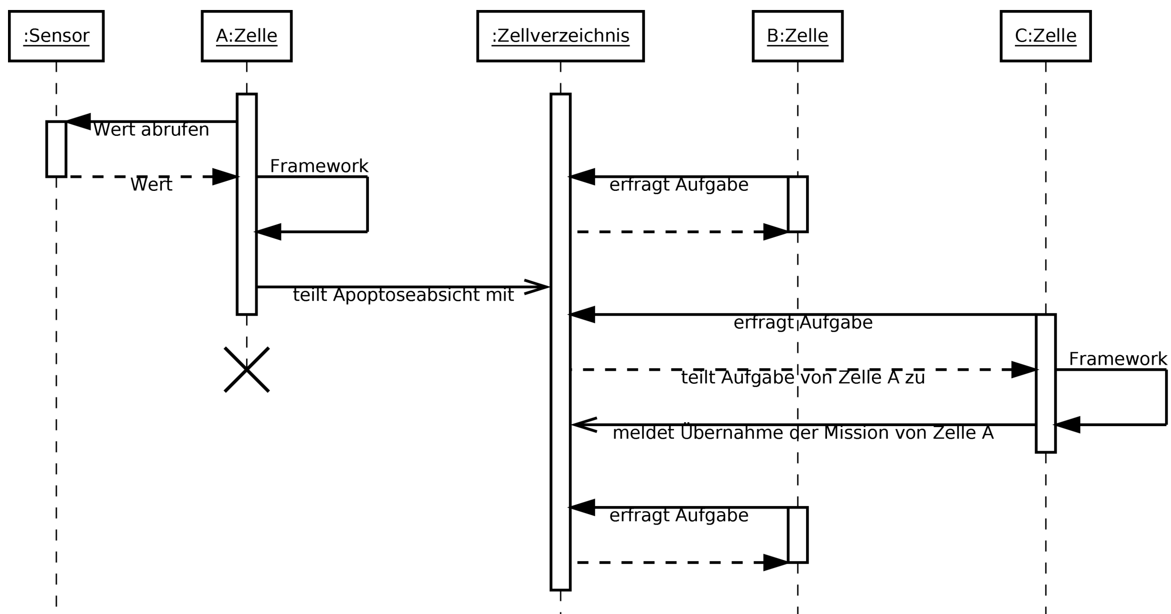

Abbildung 6.14.: Zentrales Ablegen der Meldung der Apoptose von Zelle A

- gar nicht gesendet werden sollten,
- andere Zellen an der Missionserfüllung hindern,
- nicht wie gefordert ankommen.

Dies muss nicht sofort geschehen, so wartet Zelle B erst einen wiederholten Fehler ab. Auch reagiert Zelle B nicht umgehend auf den Apoptosebefehl. Erst wenn sich die Beschwerden von mehreren verschiedenen Zellen häufen, fällt im Framework die Entscheidung zur Terminierung.

Sollte Zelle B aufgrund einer Anordnung zur Apoptose den Fehler beheben können, oder dieser von selbst verschwinden, kann die Mission von Zelle B weitergeführt werden. Entweder löscht sie alle bis dahin erhaltenen Meldungen, oder diese werden mit der Zeit von alleine reduziert. Auch eine Antiapoptosemeldung von den Zellen A und C ist denkbar, um die vorhergegangenen Aufforderungen zu annullieren.

Die verzögerte Reaktion einer Zelle hat auch den Vorteil, dass falsche Apoptoseaufforderungen wirkungslos bleiben. Wie in Abbildung 6.17 gezeigt, versendet die Zelle B unbegründete Befehle zur Apoptose an ihre Nachbarn. Dies kann verschiedene Ursachen haben:

- Zelle B reagiert aufgrund falscher Informationen,
- Zelle B ist mit einer Malware infiziert,
- Zelle B ist fehlerhaft konfiguriert,
- Zelle B interpretiert (ausbleibende) Signale falsch.

Da die Zellen A und C nur von Zelle B auf einen Fehler ihrerseits aufmerksam gemacht werden, andere Zellen aber mit ihrer Arbeit zufrieden sind, liegt der Schluss nahe, dass die Zelle B den eigentlichen Defekt aufweist. Wenn auch eine Überprüfung des eigenen Status durch das jeweilige Framework keine Anhaltspunkte auf ein Versagen liefert, senden die Zellen A und C ebenfalls eine Apoptoseaufforderung an Zelle B.

Zelle B wird nun von mehreren Seiten auf das falsche Verhalten aufmerksam gemacht, was nach einer Erwägung von schonenderen Maßnahmen zur Apoptose führen kann.

Das dabei versendete Datenpaket kann wie in Tabelle 6.3 aussehen. Der Unterschied zu den anderen Mitteilungen liegt dabei wie auch schon zuvor in der verwendeten Einheit, sowie dem Namen des Datenpakets.

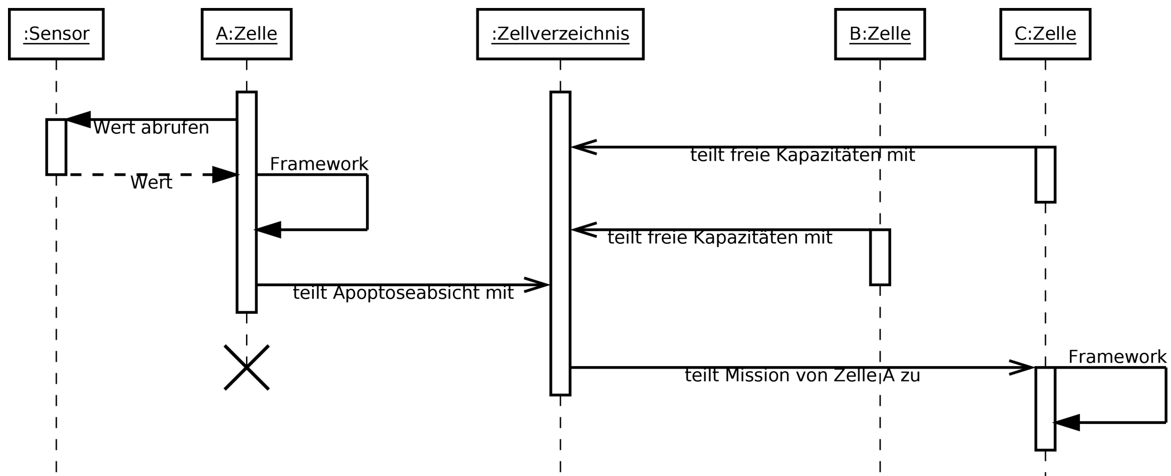

Abbildung 6.15.: Broker reagiert auf die Apoptose von Zelle A

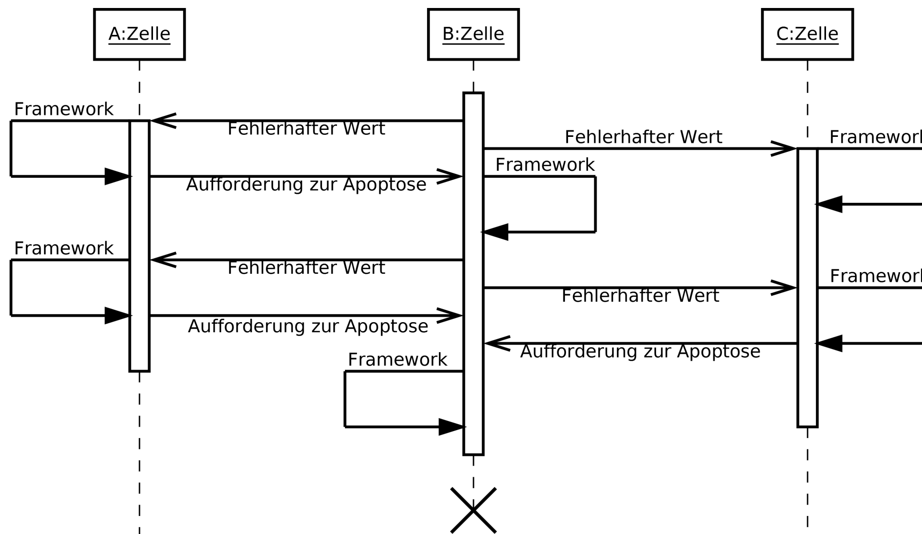

Abbildung 6.16.: Eine nicht mehr korrekt funktionierende Zelle wird zur Apoptose aufgefordert

So können bereits implementierte und erprobte Verbreitungs- und Verarbeitungsstrategien wiederverwendet werden.

6.2.5. 5. Anforderung: Verzeichnis- und Protokollplattform

Eine nicht zu unterschätzende Anforderung ist die Fähigkeit des Frameworks, seine Arbeit zu protokollieren. Einerseits hilft dies bei der Optimierung der Entscheidungsfindung und der Wahl der richtigen Reaktion. Andererseits können dadurch andere Zellen die bisherige Entwicklung im System betrachten und ihr Verhalten besser anpassen. Hierfür sollte die Protokollierung unbedingt maschinenlesbar sein.

Die zweite Aufgabe des Protokollservers ist das Vorhalten einer Registratur für die Zellen. Diese ergibt sich bereits teilweise aus den Protokollmeldungen. Wenn eine Zelle die Aufnahme der Arbeit an einer Mission bekannt gibt, lässt sie sich auch gleich in das Register eintragen, zusammen mit der zugeteilten Mission und den gebundenen sowie freien Ressourcen. Muss sich eine Zelle beenden, teilt sie auch dies inklusive dem Grund der Reaktion mit. Somit ist bekannt, welche Zelle noch fähig und bereit für neue Aufgaben ist.

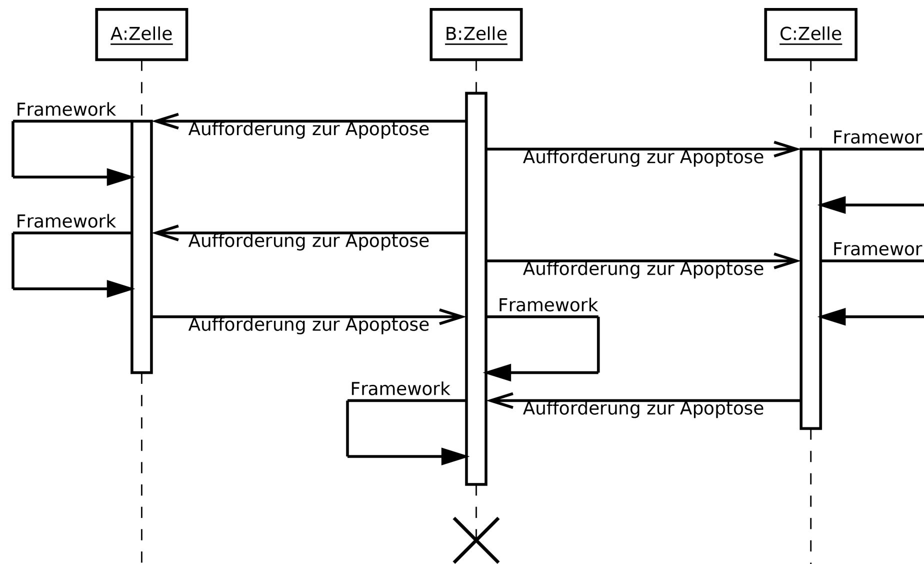

Abbildung 6.17.: Eine Zelle verschickt unangebrachte Aufforderungen zur Apoptose

Elementname	**Datentyp**	**Beschreibung**
Name	URI bzw. String	Die Bezeichnung des Datums (Apoptosebefehl)
Sender	URI bzw. String	Der Name der sendenden Zelle
Einheit	Unit bzw. String	Der Grund für den Befehl zur Apoptose
Wert	Integer	Die Dringlichkeit
Zeitstempel	Timestamp bzw. Long	Zeitpunkt des Absendens

Tabelle 6.3.: Inhalt eines Befehls zur Apoptose

Protokollieren wichtiger Ereignisse

Das Protokollieren von Ereignissen kann bei der Verfolgung von Fehlern sehr hilfreich sein. Dies gilt insbesondere in der Entwurfsphase, in der eine gute Missionsbeschreibung noch nicht gefunden worden, oder das Framework noch in der Trainingsphase ist, sofern ein selbst lernender Ansatz implementiert wird. Tabelle 6.4 listet die wichtigsten Werte auf, die bei der Überwachung der Aktivitäten in einem Framework mitprotokolliert werden können. Hinzu kommen jeweils eine eindeutige ID der Logzeile sowie ein Zeitstempel. Auch der Name des Frameworks, das die Aktivität enthält, also im Prinzip auch der Name der Zelle, sollte angegeben werden, um das Ereignis richtig einordnen zu können.

Eine Datenbank - mit oder ohne SQL - bietet sich bei großen Datenmengen an, da sie die Synchronisation, die Verarbeitung der Anfragen und die Optimierungen und Sicherungen, die ein Datenbanksystem ausmachen, bereits mitliefert. Aber auch CSV-Dateien mit kommaseparierten Werten, XML-Dokumente oder Dateien in einem proprietären Format sind möglich. Wenn das Framework gegenüber den Aktivitäten eine Abstraktionsschicht verwendet, die den Anschluss des Logsystems transparent gestaltet, können bei der Initialisierung des Frameworks individuelle Anweisungen erteilt werden, wie und wo die Informationen zu protokollieren sind.

Lokalisierung von Zellen

Es ist notwendig, den Zellen eine eindeutige ID zu geben. Dies kann zum Beispiel in Form einer URI (Uniform Resource Identifier) geschehen, einem Format, das explizit für die Identifizierung von Ressourcen spezifiziert wurde [BLFM 05]. URIs sind weit verbreitet und können in allen Systemen genutzt oder gespeichert werden, die mindestens Strings verarbeiten können. Die URI ist auch das Mittel der Wahl, um im semantischen Web

Spaltenname	Datentyp	Eigenschaften	Kommentar
id	unsigned int	primary key, not null, auto inc	Zeilen-ID
timestamp	timestamp	not null, CURRENT_TIMESTAMP	Zeitstempel
cellname	varchar	null	Name der Zelle
name	varchar	null	Name der Aktivität
state	varchar	null	Zustand der Aktivität
weight	integer	null	Gewichtung
inname	varchar	null	Name des Eingangsdatums
invalue	integer	null	Wert des Eingangsdatums
inunit	varchar	null	Einheit des Eingangsdatums
insender	varchar	null	Name des Senders des Datums
intimestamp	timestamp	null	Erzeugungszeitpunkt des Datums
outname	varchar	null	Name des Ausgangsdatums
outvalue	integer	null	Wert des Ausgangsdatums
outunit	varchar	null	Einheit des Ausgangsdatums
outsender	varchar	null	Name des Senders des Datums
outtimestamp	timestamp	null	Erzeugungszeitpunkt des Datums
comment	varchar	null	Ein zusätzlicher Kommentartext

Tabelle 6.4.: Beispiel für eine Log-Tabelle mit SQL für die Aktivitäten (A.11)

unter der Verwendung vom *Resource Description Framework* (RDF) auf Ressourcen zu referenzieren [OL 99].

Es empfiehlt sich, die URIs, die auf eine Zelle verweisen, mit einer Semantik zu versehen, die über den Typ und die Verwendungsmöglichkeiten einer Ressource Aufschluss geben. Dadurch ist es für eine Zelle möglich, durch das Abtrennen der letzten Segmente der URI eine allgemeinere Beschreibung der Eigenschaften von sich zu erhalten. Mit diesen können benachbarte Zellen, mit gleichen oder ähnlichen Fähigkeiten, leicht gefunden und angesprochen werden.

Angenommen, es soll auf eine Zelle referenziert werden, die einen Job in einem Rechenzentrum für High Performance Computing bearbeitet. Die Zelle ist Teil des Grid *„D-Grid“*, einer virtuellen Organisation *„vo0815“* zugeordnet und mit dem Job *„mission4711“* betraut. Es handelt sich bei ihr um den Prozess *„pid42“*, der in einer CPU im Knoten *„node23“* läuft. Die zugehörige URI könnte so aussehen:

```
cluster64bit.example.com/d-grid/vo0815/mission4711/node23/pid42
```

Fällt nun der Knoten *„node23“* aus, streicht die Zelle von ihrem Namen die letzten Segmente, bis eine Auswahl an alternativen Ressourcen verfügbar wird. Die URI könnte nun so aussehen:

```
cluster64bit.example.com/d-grid/vo0815/mission4711
```

Dies hat vielleicht diese Liste als Ergebnis:

Name	Status
`cluster64bit.example.com/d-grid/vo0815/mission4711/node11`	frei
`cluster64bit.example.com/d-grid/vo0815/mission4711/node17`	besetzt
`cluster64bit.example.com/d-grid/vo0815/mission4711/node19`	besetzt
`cluster64bit.example.com/d-grid/vo0815/mission4711/node23`	unerreichbar

Da der Knoten *„node11“* verfügbar ist, kann der Prozess an diesen übertragen werden. Der Knoten wird durch eine andere Zelle mit dem Namen

```
cluster64bit.example.com/d-grid/vo0815/mission4711/node11
```

repräsentiert, die den Job von Knoten *„node23“* weiterführt, während jener sich selbst abschaltet.

Ist der Knoten *„node11“* jedoch ebenfalls belegt, dann fehlt der Mission eine benötigte Ressource. Diese kann unter Umständen zusätzlich für die Mission *„mission4711“* eingebunden werden, sofern die virtuelle Organisation *„vo0815“* die Zugriffsrechte für weitere Rechenknoten hat, die die Aufgabe übernehmen können. Sollte auch dieser Versuch scheitern, kann an die nächsthöhere Instanz *„d-grid“* eine automatische Aufforderung gestellt werden, die Rechte von *„vo0815“* zu erweitern, um zusätzliche Kapazitäten einbinden zu können. Die Art der benötigten Ressource lässt sich aus der Unterdomäne *„cluster64bit“* ablesen, die in diesem Fall für einen CPU-Cluster steht.

Eine andere Art von Ressource könnte mit dieser URI beschrieben sein:

```
storage.example.com/d-grid/vo0815/mission4711/san3/raid5/
```

Dabei handelt es sich um ein RAID-System mit ID *„raid5“*, das zum *Storage Area Network „san3“* gehört. Fällt es aus, kann es durch ein anderes SAN ersetzt werden, sofern jenes verfügbar ist und die VO die geforderten Zugriffsrechte besitzt.

Da die Zelle mit einer derartigen URI als Name nur einen Speicher repräsentiert, kann sie nicht die Aufgaben der zuvor besprochenen Prozessorzellen übernehmen und umgekehrt. Aus dem Namen einer Zelle sind demnach recht einfach ihre Fähigkeiten und Zugehörigkeiten abzulesen. Dabei bezieht sich eine URI immer auf die Mission, in deren Kontext die Zellen referenziert werden sollen. Daher ist einer Abstraktion von den zugrundeliegenden Netzen und Administrationsdomänen gegeben, was die domänübergreifende Kommunikation vereinfacht.

Durch diese Formatierung des Namens einer Zelle ist es auch einfach, andere gleichartige Zellen in der Nachbarschaft anzusprechen. Um so länger die Ziel-URI ist, um so ähnlicher und kleiner ist die Gruppe von Zellen, denen die Nachricht gilt. Somit lassen sich netzweite Broadcasts vermeiden, die zu einem erheblichen Overhead führen und dementsprechend die Netze unnötig belasten würden.

Auch kann durch die Verwendung des *Domain Name System* (DNS) schnell das zuständige Zellenverzeichnis ermittelt werden, in dem eine Zelle registriert ist. Jenes kann seinerseits als Zelle implementiert sein. Das führt zu einem hybriden Ansatz bei der Kommunikation: die Zellen verständigen sich untereinander direkt, finden sich aber über einen gemeinsamen Indexserver. Dieser stellt somit einen *single Point of Failure* dar.

In Peer-to-Peer-Netzen wie Gnutella wird daher häufig mit *Supernodes* gearbeitet, die besonders zuverlässig und teilweise sogar redundant ausgeführt sein sollten. Diese verwalten jeweils eine kleinere Gruppe von normalen Knoten oder Ressourcen [LZL+ 05]. Um effizient in einem verteilten System suchen zu können, wurden auf Hashwerten der Ressourcen basierende Algorithmen entwickelt. Diese sollen robust gegenüber dem Verlust von Indexknoten sein, wodurch eine hohe Fluktuation der Knoten erst möglich wird und Suchanfragen schnell und ohne große Netzlast erfolgen können. Auch eine gute Skalierbarkeit ist wichtig, um große Netze mit vielen tausend, teilweise sogar mehreren hunderttausend Knoten zu unterstützen. *Distributed Hash Tables* (DHT) haben sich als recht zuverlässig und gut skalierend erwiesen und ermöglichen große vollständig verteilte Netze. Bekannte Implementierungen sind dabei *Chord* [SMK+ 01] und *Pastry* [RoDr 01].

Methoden wie DHT lassen sich auch für die Auffindung von Zellen in einem Grid verwenden. Dazu muss lediglich die URI einer Zelle ganz oder in Teilen mittels einer geeigneten Hashfunktion in eine Zahl gewandelt werden, die eine Zelle oder eine Gruppe gleichartiger Ressourcen referenziert. Mittels einer DHT lässt sich dann die gesuchte Zelle lokalisieren und ansprechen.

Da Gridressourcen in der Regel recht beständig und zuverlässig sind, ist ein derartiger Aufwand oft gar nicht nötig. Doch wenn ein andersartiges verteiltes System, wie beispielsweise ein Smart Grid, vorliegt, können *Distributed Hash Tables* die geeignetere Wahl sein.

6.2.6. 6. Anforderung: Eine Zelle kennt die Mission

Die am schwierigsten umzusetzende Anforderung ist die Missionsbeschreibung. Am schönsten wäre es, könnte man dem Computer seine Aufgabe wie einem Menschen erklären, wie man das zum Beispiel bei klassischen Spielen machen würde.

Spiele haben meistens ein Ziel, das es für die Spieler zu erreichen gilt. Es gibt ein Regelwerk, in dem neben der Mission auch Grenzen definiert werden, die alle Teilnehmer einhalten müssen. Betrachtet man nun ein verteiltes Computersystem als Spielfeld, so entsprechen die Zellen den Mitspielern und die Missionsbeschreibungen den Spielregeln. Es gibt dabei meist komplexe Spielfelder, die für genau ein Spiel geschaffen wurden, deren Gestaltung also unmittelbar mit den Regeln zusammenhängt. Entsprechend gibt es auch Systeme, deren Ausgestaltung durch ihre konkrete Mission bestimmt wird, so wie es im ***5. Szenario:*** *Heizung* (2.4.1) oder im ***7. Szenario:*** *Smart Grid* (2.4.3) der Fall ist. Andere Spiele benutzen generisches Material, beispielsweise Spielkarten. Erst die Regeln legen fest, ob damit Poker, Black Jack, Rommé oder Mau Mau gespielt wird. Genau so gibt es auch generische Systeme, deren Mission erst die Art der Nutzung bestimmt. Ein Beispiel hierfür ist das ***1. Szenario:*** *Missionen im Grid Computing* (2.3.1).

Um die Aufgabe eines Systems für einen Menschen verständlich modellieren zu können, wurde die *Unified Modeling Language* (UML) definiert, die auch in dieser Arbeit bisher Verwendung gefunden hat [RQZ 07]. UML ist sehr gut geeignet, um das Design eines Systems anschaulich darzustellen, macht aber kaum Aussagen über korrektes oder fehlerhaftes Verhalten. Daher wurde eine zusätzliche textuelle Beschreibung der Bedingungen, die von einem Objekt erfüllt werden müssen, in Form der *Object Constraint Language* (OCL) entwickelt [RiGo 98]. Mit OCL werden Vor- und Nachbedingungen formal spezifiziert, auf deren Einhaltung eine Softwarekomponente geprüft werden kann. Dies erfolgt zum Beispiel mit *Unit Tests*, die automatisch das Kompilat gegen zuvor im Code getroffene Annahmen (*Assertions*) prüfen [ZHM 97].

All diese Ansätze sollen helfen, einen verständlichen, zuverlässigen und möglichst fehlerfreien Entwurf zu erstellen und umzusetzen. Da jedoch trotz einer großen Anzahl von Werkzeugen die Implementierung weiterhin viel Handarbeit erfordert und aufgrund von Zeitdruck, Unverständnis des Systems und Sparzwang ein umfangreiches und weitgehend vollständiges Testen unterbleibt, kann man sich in einem verteilten System mit einer Vielzahl unterschiedlicher Geräte von diversen Produzenten nur selten auf die Einhaltung einer effektiven Qualitätssicherung verlassen.

Dies hat zur Folge, dass Teile eines Systems sich nicht an die Regeln halten, vor allem wenn Komponenten in Beziehungen zueinander gestellt werden, die beim Entwurf nicht bekannt waren, und damit auch nicht bei den Tests berücksichtigt wurden. Eine Verbesserung der Missionssicherheit könnte eine einheitliche selbstständige Überwachung der Komponenten zur Laufzeit sein. Um Fehler abzufangen, werden bereits im Code, aber auch in Laufzeitumgebungen, *Exceptions* abgefangen und behandelt, sofern dies möglich ist. Weiterhin werden Monitorprogramme und -hardware eingesetzt, die auf Systemebene einer Zelle diese auf die Einhaltung einer grundsätzlichen Spezifikation der Komponente kontrollieren.

Die Aufgabe, die die Zelle im Verbund zu erfüllen hat, also die Einhaltung der Missionsdefinition, wird dabei nur kaum bis gar nicht berücksichtigt. Es gibt einige Ansätze dies zu ändern, was wegen der hohen Komplexität und dem Einfluss durch teilweise unbekannte Komponenten alles andere als trivial ist. So wurde 1998 an der *University of Pennsylvania* das *Monitoring and Checking* (MaC) Framework entwickelt [LBAK+ 98] und in Java implementiert [LKK+ 99] [SSD+ 03], das genau dies leisten soll. Das MaC-Framework benutzt dabei zwei Definitionssprachen: die PEDL (*Primitive Event Definition Language*) bestimmt die Erfassung der Messwerte und die MEDL (*Meta Event Definition Language*) spezifiziert die zu erfüllenden Anforderungen. Es ist auch eine Erweiterung für die Überwachung von Komponenten in einem verteilten System vorgesehen.

Allerdings ist aufgrund der starken Integration des MaC-Frameworks in die virtuelle Maschine von Java eine einfache Portierung auf andere Systeme, vor allem nicht computerspezifische, recht aufwendig. Eine Verknüpfung von physischen und digitalen Komponenten liefert einen viel größeren Einzugsbereich von relevanten Daten sowie eine vielseitigere Kontrollplattform. So kann ein generisches Framework nicht nur für Computergrids, sondern auch für andere Netze, wie Smart Grids, verwendet werden. Auch der Einsatz von zwei Sprachen, die eine hohe Ähnlichkeit zu Javacode aufweisen, wirken nicht unbedingt konsistent. Auf die Anwendung von Apoptose und die daraus resultierenden Anforderungen wird ebenfalls nicht eingegangen

Daher ist das Framework dieser Arbeit zur Durchführung von Apoptose etwas anders aufgebaut. Das Ziel ist ein generisches Kontrollsystem mit dem Schwerpunkt auf verteilte Systeme und Missionserhalt auf einem globalen Level. Um dem Framework die Mission der zu überwachenden Komponente zu beschreiben, ist eine maschinenlesbare Definitionssprache unumgänglich. Diese ist aus diversen Gründen in XML-Syntax verfasst.

XML ist eine wohl spezifizierte, weit verbreitete und sehr erprobte Auszeichnungssyntax, um die Semantik eines Textes für einen Computer verständlich zu machen [BPSM+ 06]. Dabei werden Textstellen, die im

einfachsten Fall einem einzigen Wert entsprechen, mit *Tags* eingefasst, die die Bedeutung des Textabschnitts ausdrücken. Derartige Elemente können ihrerseits durch *Tags* ausgezeichnete Unterelemente haben, so dass das gesamte XML-Dokument eine Baumform annimmt, die als *Document Object Model* (DOM) bezeichnet wird. Diese Baumstruktur entsteht auch beim Parsen des Dokuments. Zusätzlich kann jedes XML-Element beliebige, benannte Argumente annehmen, zum Beispiel einen eindeutigen Namen des Elements. XML hat eine hohe Verbreitung, da es generisch verwendet werden kann und es für beinahe jede Programmiersprache leicht zu nutzende Parser gibt, was den Datenaustausch sehr erleichtert. Auch im Grid Computing wird es für die Beschreibung von Schnittstellen von Diensten eingesetzt. Daher wird auch einer Zelle die Beschreibung ihrer Aufgabe in XML-Syntax mitgeteilt.

Die konkrete Missionsbeschreibung für eine Zelle kann entweder manuell erstellt oder aus einer globalen Missionsdefinition abgeleitet werden. Während der erste Fall die Problematik der Konvertierung einer teilweise abstrakten Definition in eine für die Zelle angepasste Darstellung umgeht, muss immer auch die globale Mission berücksichtigt werden. Das kann leicht zu Fehlern führen, die die Stabilität des Systems mehr gefährden als schützen. Im anderen Fall ist eine nicht triviale Umsetzung der Arbeit des Gesamtsystems auf die Aufgaben und Fähigkeiten einer einzelnen Komponente zu entwickeln. Dafür ist das nachträgliche Modifizieren der Mission und das Ausrollen auf alle betroffenen Ressourcen, ohne deren Spezifikation im Einzelnen zu kennen, leichter möglich, wenn ein Automatismus die Missionsdefinition für eine Zelle umschreibt.

Eine Vereinfachung kann ein hybrider Ansatz bewirken, bei dem jede Zelle eine Reihe von Regeln mitbringt, welche Ressourcen sie benötigt und welche Informationen sie liefern kann, zusammen mit der jeweiligen Formatierung. Wenn also eine Zelle eine Temperaturüberwachung von sich anbietet, kann sie den Namen und den Wertebereich des Sensors, die Darstellung eines Messwerts auf Bitebene und einen Normwert für die Zelle angeben. Auch die Auswertung von Messwerten externer Sensoren kann eine Zelle schon mitliefern, wobei die Spezifikation der Nutzung vom Sensor bereitgestellt werden sollte. Äquivalent kann eine Zelle die Schnittstelle beschreiben, über die sie Informationen für andere Komponenten bereit stellt.

Derartige Definitionsschnipsel können ihrerseits abstrakt für eine Missionsbeschreibung herangezogen werden. Man gibt also nur noch an, dass eine Zelle innerhalb einer bestimmten Temperatur arbeiten soll. Der Konverter setzt für diese Anforderung die Zugriffs- und Auswerteregeln für den Temperatursensor der jeweiligen Zelle ein.

Abbildung 6.18.: Hierarchie bei der Konvertierung der Missionsdefinitionen

Wie Abbildung 6.18 zeigt, können noch weitere Abstraktionsebenen eingesetzt werden, um möglichst viele Konvertierungsregeln und -module wiederverwenden zu können. Außerdem reicht es bei lokalen Änderungen meistens, eine weitgehend vorberechnete Missionsbeschreibung von nur einer Ebene höher an die jeweilige Zelle anzupassen.

Sollten alle Missionsbeschreibungen, angefangen bei der abstrakten globalen bis hin zur konkreten für eine Zelle, in XML-Syntax vorliegen, bringt das den Vorteil, eine Transformationssprache wie XSLT (***Ex**tensible **S**tylesheet **L**anguage **T**ransformations*) einsetzen zu können [Kay 07]. Diese besteht aus XSL-Regeln in XML-Syntax, die XML-Dokumente in andere XML-Dokumente überführen können. Zellen und Systeme könnten

derartige Regeln bereits mitbringen, die eine übergeordnete Abstraktionsebene auf die eigenen Bedürfnisse anpasst. Mit geeigneten Werkzeugen könnten sogar in UML beschriebene Missionen in ein XML-Dokument übersetzt werden, das der obersten Ebene in Abbildung 6.18 entsprechen würde.

Derartige Werkzeuge und XSLT-Regeln sind aber nicht Teil dieser Arbeit. Daher wird davon ausgegangen, dass eine Missionsbeschreibung bereits in einem für die konkrete Zelle angepassten XML-Dokument vorliegt. Dieses entspricht einer Abfolge von Aktivitäten, wie in Abbildung 6.1 am Anfang des Kapitels bereits gezeigt worden ist.

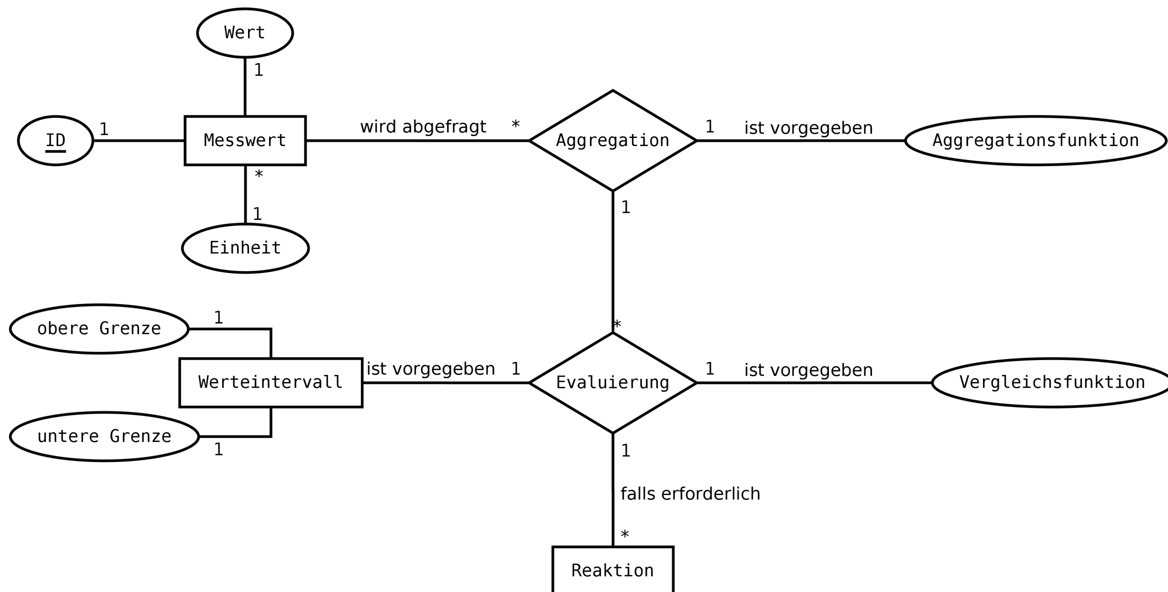

Abbildung 6.19.: Datenmodell der Beschreibung einer Mission

Eine genauere Spezifikation muss dem Relationenmodell in Abbildung 6.19 entsprechen. Im Prinzip wird eine Abfolge von Aktivitäten angegeben, die Messwerte einlesen, verarbeiten und aggregieren, um letztendlich eine Entscheidung bezüglich Apoptose zu treffen und umzusetzen. Es können beliebig viele Messwerte, die neben dem eigentlichen Wert noch über eine ID und eine Einheit verfügen, mittels zu spezifizierender Aggregationsfunktionen zusammengeführt werden. Die Ergebnisse werden beliebig oft evaluiert, wobei eine Vergleichsfunktion und ein Werteintervall angegeben werden müssen. Daraufhin kann eine Reaktion erfolgen. Es ist auch möglich, einzelne Zwischenschritte wegzulassen oder mehrfach zu wiederholen, um geeignete Informationen zu gewinnen.

Konkret beinhaltet das XML-Dokument neben diversen Einstellungen und der Angabe von Kommunikationsschnittstellen die Namen und Initialisierungsparameter von Klassen, die von den oben erläuterten Aktivitäten `Collector`, `Modifier`, `Merger` und `Evaluator` abstammen. Beim Einlesen in das Framework einer Zelle werden diese in die entsprechenden Objekte umgesetzt, die untereinander wie angegeben verknüpft werden. Außerdem baut das Framework die geforderten Kommunikationswege auf und bindet die erzeugten Aktivitäten an diese an.

6.3. Kleines Beispiel für ein Framework

Eine typische Kette von Aktionen, wie sie von einem Framework implementiert und genutzt wird, könnte aus der Abfrage von Temperatur, CPU-Last und Datenverkehr im Netz bestehen. Damit kann bereits ungewöhnliches Verhalten eines Computers ermittelt werden. Hinzu können noch Stromverbrauch und binäre Signale, wie ein Feueralarm, kommen. Abbildung 6.20 zeigt ein Minimalbeispiel für ein Framework, das eine Zelle überwacht. Jene hat den Namen *„servercell“* erhalten und läuft mit einem *verbose*-Level von 8, was eine recht hohe Anzahl von Logeintragungen nach sich zieht. Das Logsystem ist hier allerdings nicht dargestellt.

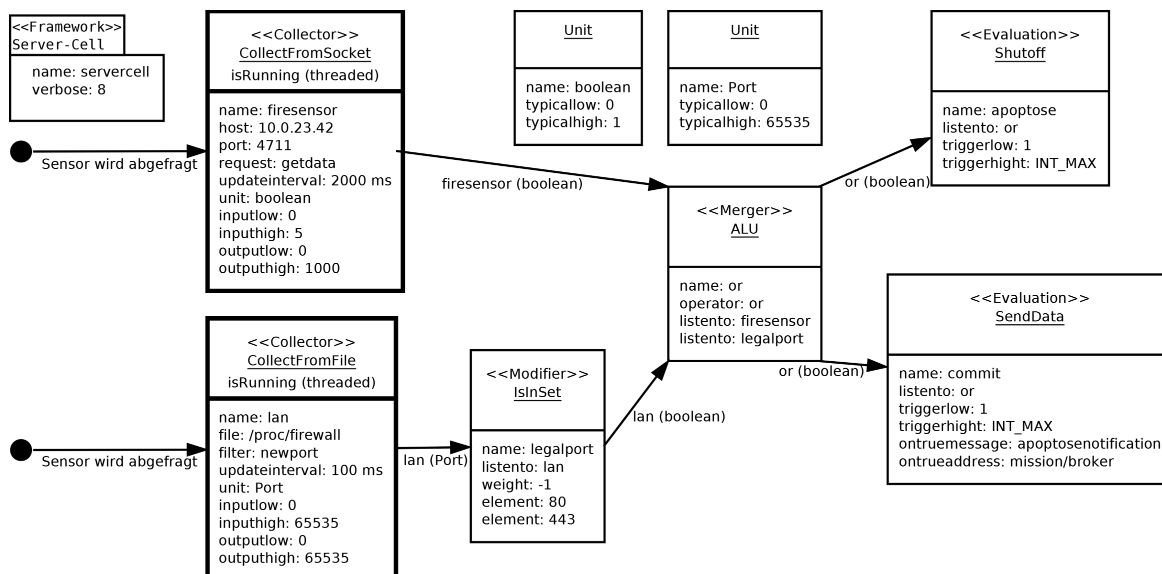

Abbildung 6.20.: Ein minimalistisches Beispiel für ein Framework

Bei der Zelle handelt es sich um einen reinen Webserver. Es werden zwei Werte überwacht: zum einen wird ein boolesches Signal eines externen Feuermelders alle zwei Sekunden via eines TCP/IP-Sockets abgefragt, zum anderen wird ein Messwert aus einer virtuellen Datei direkt auf dem Server gelesen, was dem Unixcredo *„alles ist eine Datei“* folgt. Diese kann zum Beispiel von einer Firewall bereitgestellt werden und enthält geöffnete Ports.

Die Abfrage der Messpunkte erfolgt durch zwei Instanzen von `Collector`, die jeweils in einem eigenen Thread laufen, regelmäßig neue Anfragen stellen und die Ergebnisse weiterreichen. Während `firesensor` eine Spannung von 0 Volt als eine 0 interpretiert und bei 5 Volt die interne Darstellung eines booleschen `True` in Form eines Integers mit dem Wert 1000 erzeugt, enthält das `Data`-Objekt, das von `lan` gebildet wird, die Nummer des eingelesenen geöffneten Ports. Durch die zusätzlich angegebene Einheit lässt sich von nachfolgenden Aktivitäten die korrekte Interpretation des Wertes bestimmen.

Da beide Signale miteinander durch ein logisches Oder aggregiert werden sollen, muss zunächst der von `lan` gelieferte Port in einen booleschen Wert überführt werden. Dies geschieht mittels einer Erweiterung von `Modifier` mit dem Namern `legalport`. Hier wird überprüft, ob der Eingangswert in einer vorgegebenen Menge enthalten ist. Das Ergebnis wird aufgrund des Gewichts -1 invertiert, bevor es an den `Merger or` weitergereicht wird. `or` erhält demnach ein `True` für jeden Port, der nicht in der genehmigten Liste steht, sowie den Wert des Feuermeldesensors. Da mit `operator` die Aggregatsfunktion `or` angegeben ist, reicht es, wenn nur einer der beiden Eingangswerte wahr wird, um die zwei Reaktionen zu triggern.

Die erste Reaktion heißt `apoptose` und erweitert `Evaluation`. Erhält sie einen positiven Wert, wird die entsprechende hinterlegte Aktion ausgeführt, die zur Apoptose führen kann. Eine weitere Instanz von `Evaluation (SendData)` teilt der angegebenen Adresse, in diesem Fall einem Broker, die Absicht der Zelle mit, sich aus der Mission mittels Apoptose zurückzuziehen.

Im Anhang findet sich eine XML-Datei (Listing A.1), die der Konfiguration des eben beschrieben Beispielframeworks entspricht.

Im nachfolgenden Kapitel 7 werden zwei deutlich komplexere Szenarien vorgestellt und anhand einer prototypischen Implementierung des Frameworks untersucht.

7. Prototypische Implementierung der Apoptose

Um das im vorausgegangenen Kapitel 6 erarbeitete Konzept auf Umsetzbar- und Sinnhaftigkeit zu prüfen, wird in diesem Kapitel eine prototypische Implementierung vorgestellt. Dazu werden zwei Beispielszenarien in einer stark vereinfachten Form beschrieben, die die breite Verwendungsmöglichkeiten der Apoptose aufzeigen sollen. So wird in Abschnitt 7.3 ein Szenario ähnlich dem in Unterabschnitt 2.3.1 Dargestellten untersucht, bevor mit Abschnitt 7.2 das Szenario aus Unterabschnitt 2.4.3 aufgegriffen und konkretisiert wird, wobei auf die Möglichkeiten des Konzeptes der Apoptose eingegangen wird.

7.1. Ein Framework für die Steuerung von Apoptose

Die Umsetzung des Frameworks ist teilweise gegenüber dem eben vorgestellten Konzept reduziert. Dies ist der Konzentration auf den für die Umsetzung der Apoptose relevanten Teil geschuldet. Dennoch ist das Framework weitgehend generisch einsetz- und leicht erweiterbar gehalten. So wird zwecks der Vereinfachung angenommen, dass immer nur eine Mission vorgegeben ist, die von dem verteilten System bearbeitet wird. Deshalb ist das Framework, wobei eine Instanz immer einer Zelle entspricht, nicht auf die Betreuung von mehreren Missionen ausgelegt. Dies liegt am Fehlen des Speicherschutzes, der die Missionen voneinander isolieren würde. Allerdings kann das Framework durchaus für mehrere Missionen gleichzeitig eingesetzt werden, wenn bei seiner Beschreibung auf sich überschneidende Namen für die einzelnen Aktivitäten verzichtet wird.

Eine Anpassung des Frameworks für die gleichzeitige Bereitstellung von mehreren Zellen mit nur einer Instanz kann einfach realisiert werden, indem die Aktivitäten Zellenobjekten zugeordnet werden. Hierfür ist aber zunächst zu klären, ob eine Aktivität von mehreren Zellen genutzt werden darf. Dies würde eine Art Referenzzähler implizieren, um zu ergründen, welche Zellen noch von einer Aktivität Gebrauch machen. Einfacher und deutlicher dem Zellengedanken entsprechend ist es deshalb, für jede Mission auf einer Komponente eine neue Instanz des Frameworks zu starten. Diese können echt unabhängig voneinander operieren und dennoch, falls gewünscht, sich wie jede andere Zelle über Sockets austauschen.

Der Prototyp kann nur wachsen und nicht schrumpfen. Es ist demnach nicht möglich, einzelne Aktivitäten wieder zu löschen, auch wenn dies durchaus mit etwas Aufwand implementierbar wäre. Doch für den Prototypen gilt, dass es einfacher ist, eine neue Instanz des Frameworks mit einer veränderten internen Struktur zu starten. Wenn eine Zelle die Apoptose einleitet und das Framework demnach nicht mehr gebraucht wird, kann es einfach beendet werden, ohne einen direkten Einfluss auf andere Instanzen auf der gleichen Komponente zu nehmen.

7.1.1. Programmiersprache und Entwicklungsumgebung

Da das Konzept für das Framework objektorientiert ist, wurde als Programmiersprache Java in der Version 7 für die Implementierung des Prototypen gewählt [Orac 12]. Sie ist weit verbreitet und wird von embedded Systemen wie Smart Cards bis hin zum Grid Computing verwendet. Im Gegensatz zu den Sprachen C und C++ hat Java eine automatische Speicherverwaltung und kann von sich aus Klassen per Namen ansprechen und vom Dateisystem oder über das Netz nachladen. Diese Fähigkeit zur *Reflection* wird verwendet, um die in der XML-Konfigurationsdatei angegeben Objekte im Framework zu erzeugen. Das gilt für alle Aktivitäten im Paket `framework.activities`, mit Ausnahme der abstrakten Klasse `Activity`. Auch die im Paket `framework.connectors` hinterlegten Connectoren werden mittels *Reflection* geladen.

Als Entwicklungsumgebung dient die populäre und mächtige IDE Eclipse in der Enterprise Edition in der Version Indigo im Service Release 1 [Foun 12]. Eclipse bietet viel Komfort bei der Entwicklung von Java-Anwendungen, auch da es durch unzählige Plug-Ins erweiterbar ist. Außerdem ist es in Java geschrieben und läuft somit auf beinahe allen Maschinen, die eine vollwertige Java Runtime bieten. Dies trifft auf einen Großteil der modernen Betriebssysteme und Plattformen zu, was mit zur Entscheidung für Java als Programmiersprache des Prototypen beigetragen hat.

Wegen der in Unterabschnitt 6.2.6 dargelegten Gründe kommt für die Konfiguration des Frameworks XML zum Einsatz [BPSM+ 06]. Dieses lässt sich mit Bordmitteln von Java einfach und flexibel parsen, aber auch außerhalb des Frameworks mit einer breiten Palette von Werkzeugen bearbeiten. Des weiteren ist XML sowohl von Maschinen, als auch vom Menschen mit einem einfachen Texteditor schreib- und lesbar. Eine Schemadefinition durch eine *Document Type Definition* (DTD) wird nicht angewendet, um den Prototypen einfach zu halten.

Die wichtigen Klassen des Frameworks sind mit Javadoc in englischer Sprache kommentiert. In den Kommentaren zu den Konstruktoren der Aktivitäten und Connectoren werden auch die XML-Tags und mögliche Werte angegeben, die in dem als Argument übergebenen XML-Element enthalten sein können oder müssen.

7.1.2. Aufbau des Projekts

Der gesamte Quellcode für ein Framework in einer Zelle befindet sich im gleichnamigen Paket `framework`. Es enthält zwei Unterpakete: in `activities` finden sich alle Klassen, die von `Activity` erben und mit *Reflection* geladen werden; in `connectors` sind alle von `Connector` erbenden Klassen. Auch diese können von einem XML-Dokument mittels *Reflection* geladen und initialisiert werden.

Das Paket `framework`

`framework` enthält fünf Klassen, die projektweit verwendet und nicht durch *Reflection* geladen werden. Der Einstiegspunkt in das Framework befindet sich in der Klasse `Main`.

Main ist die Hauptklasse, die die Initialisierung beim Programmstart vornimmt. Dies geschieht in der statischen Methode `main()`, die als Argumente von der Kommandozeile Pfade zu XML-Dateien übergeben bekommt, die die Konfiguration des Frameworks bereitstellt. Der genaue Aufbau der Konfigurationsdateien wird in 7.1.3 erläutert.

Neben dem Namen des Frameworks und dem Ausgabelevel der Logmeldungen hält `Main` auch eine Reihe von systemweit genutzten Konstanten, so dass diese nur an diesem Punkt geändert werden müssen. Sie werden mit einem Schlüsselwort über die Methode `getInt` erfragt. Durch diesen indirekten Zugriff auf die Integer kann man die Werte in der XML-Datei angeben und somit die Defaulteinstellung überschreiben. Das Framework gibt bereits die Werte in Tabelle 7.1 vor.

Name	Defaultwert	Bedeutung
`verbose`	6	Geschwätzigkeit des Frameworks in den Logs
`false`	0	Wert für *Falsch* im Datentyp `boolean`
`true`	100000	Wert für *Wahr* im Datentyp `boolean`
`null`	0	Legt den internen Nullpunkt fest
`one`	1000	Interner Wert für eine Eins (Fixkommaverschiebung)
`interval_low`	0	Unteres Ende des internen normalisierten Intervalls (0)
`interval_high`	100000	Oberes Ende des internen normalisierten Intervalls (100)

Tabelle 7.1.: Defaultwerte in `Main`

Weiterhin werden zwei Objekte der Klasse `Unit` erstellt, die von dem Framework immer benötigt werden, ein Defaultfehlertyp und eine boolesche Einheit:

```
new Unit("failure", Main.getInt("null"), Main.getInt("null"));
new Unit("boolean", Main.getInt("false"), Main.getInt("true"));
```

Diese können auch durch die XML-Dateien überschrieben werden, die gleich im Anschluss geparst werden. Sind alle Objekte erzeugt, werden die Aktivitäten miteinander verbunden. Zum Schluss folgt ein Aufruf von `update(new Data(0, null, null, null))`, damit alle Aktivitäten die Gelegenheit erhalten, sich zu initialisieren, meistens jedoch ohne die Daten weiterzugeben.

XMLParser dient als Parser für die XML-Dokumente, die vom Framework verarbeitet werden. Um ein XML-Dokument zu verarbeiten, wird es als Argument an die Methode `parse()` weitergereicht. Dabei kann es vom Typ `java.io.File`, `java.io.InputStream` oder `String` sein, wobei Letzterer der URI des XML-Dokuments entspricht. Der Klasse wird im Konstruktor eine Instanz des Interfaces `XMLParser.ICallback` übergeben. Diese enthält nur eine Methode, die von `XMLParser` für jedes neue XMLElement aufgerufen und zusammen mit seinem Elternelement übergeben wird.

```
public interface ICallback {
    public void processNode(XMLElement node, XMLElement parentNode);
}
```

DefaultHandlerICallback ist die Standardimplementierung der Schnittstelle `XMLParser.ICallback`. Sie wird von `Main` genutzt, um das Framework mit Objekten und Klassen zu füllen, kann aber auch von anderen Klassen genutzt oder erweitert werden.

XMLElement ist ein geparster Knoten eines XML-Dokuments. Es wird an den Konstruktor der meisten Komponenten des Frameworks übergeben und enthält die jeweiligen Initialisierungswerte, die über das jeweilige Schlüsselwort abgefragt werden können. Der Name des XML-Elements steht im String `tag`, Argumente werden mit der Methode `getAttributeString()` abgefragt, die bei einem Fehler eine Exception wirft, sofern kein Defaultwert angegeben wurde.

Nach dem gleichen Verfahren gelangt man auch an die Werte der Kindelemente. Zur Bequemlichkeit soll beitragen, dass es für die Datentypen `String`, `Integer`, `Long`, `Float`, `Double` und `Boolean` eigene Abfragemethoden gibt, die die Konvertierung vornehmen. So liefert `getChildInteger()` einen Integer zurück. Sollte dies nicht möglich sein, wird eine Exception geworfen, falls kein Defaultwert mitgegeben worden ist.

Sollte es mehrere Kindelemente mit gleichem Namen geben, so kann über diese in einer Schleife iteriert werden. Der Zähler wird bei jedem Aufruf der Methode `nextChild()` inkrementiert. Wenn es keinen weiteren Wert für das angegebene Schlüsselwort gibt, liefert sie `false` zurück und setzt den Zähler wieder auf 0. Der Zähler gilt auch für Kindelemente mit anderem Namen, weshalb beim Zugriff auf diese innerhalb der Schleife sichergestellt sein sollte, dass ihre Anzahl der des übergebenen Schlüsselwortes entspricht. Eine Schleife kann demnach so aussehen:

```
ArrayList<Integer> set = new ArrayList<Integer>();

node.nextChild(null);
do {
    set.add(node.getChildInteger("element"), 0);
} while (node.nextChild("element"));
```

Data und Unit sind genau so aufgebaut, wie in Unterabschnitt 6.2.1 beschrieben. Um Variablenüberläufe bei Produkten mit großen Faktoren zu vermeiden, wird `value` intern als `long` gespeichert. Weiterhin bieten `Data` und `Unit` eine sinnvolle Beschreibung, die von `toString()` zurück gegeben und vom Logsystem genutzt wird. Da alle Werte in den Klassen `final` sind, müssen diese mit dem Konstruktor initialisiert werde,

eine Änderung ist dann nicht mehr möglich. Hierdurch soll verhindert werden, dass eine Aktivität ein Datum modifiziert, das von einer weiteren `Activity` genutzt wird, die sich auf dessen Integrität verlässt.

`Unit` bietet außerdem noch eine statische `HashMap` mit dem Namen `units`, die einen Zugriff auf alle Einheiten mit ihren Namen anbietet, die innerhalb des Frameworks und in den XML-Dokumenten als Referenz dienen.

Das Paket `framework.activities`

Dieses Paket enthält alle Aktivitäten. Sie werden bei ihrer Erzeugung in der abstrakten Klasse `Aktivity` hinterlegt, um sie über ihren Namen ansprechen zu können. Ihre Funktionalität entspricht weitgehend dem vorgestellten *Konzept eines Frameworks zum Missionserhalt durch Apoptose* (6). Allerdings wird jeder Aktivität maximal ein Connector für Logausgaben unter dem Bezeichner `loggername` zugeordnet, da dies für den Prototypen ausreichend ist. Wird kein `Connector` angegeben, findet die Ausgabe der Meldungen auf der Kommandozeile statt. Eine Übersicht über alle Klassen des Pakets findet sich in Tabelle 7.2, weiterführende Informationen können dem Javadoc auf beiliegender (*CD-ROM* (A.3)) entnommen werden. Die obersten beiden Stufen der Klassenhierarchie sind wie in Unterabschnitt 6.1.3 beschrieben umgesetzt (Abbildung 6.2).

Klasse	**Basisklasse**	**Kurzbeschreibung**
`Activity`		Abstrakte Basisklasse
`Collector`	`Activity`	Sammelt Daten ein, Kinder sind meistens threaded
`Modifier`	`Activity`	Modifiziert das Datum
`Merger`	`Activity`	Aggregiert mehrere Daten
`Evaluator`	`Activity`	Führt Reaktionen aus, falls der Wert diese bedingt
`ALU`	`Merger`	Aggregiert mit: add, sub, mul, div, and, or, xor
`AverageMerge`	`Merger`	Ermittelt den Durchschnitt
`CollectFromSocket`	`Collector`	Ruft Daten von einem TCP-Socket ab (threaded)
`CollectMessage`	`Collector`	Erhält Daten via eines Connectors
`ComparatorMerge`	`Merger`	`if/else`, Vergleicht zwei Werte mit: eq, ne, gt, lt, ge, le
`Delay`	`Modifier`	Verzögert die Weitergabe um die angegebene Zeitspanne
`IsInSet`	`Modifier`	Ermittelt, ob der Wert Element einer Menge ist
`MaximumMerge`	`Merger`	Liefert das Maximum der Werte
`MinimumMerge`	`Merger`	Liefert das Minimum der Werte
`ProvideData`	`Evaluator`	Stellt über `XMLServerSocket` Daten bereit
`RequestMessage`	`Collector`	Erfragt von einem remote `XMLServerSocket` Daten
`SendData`	`Evaluator`	Sendet Daten an andere Zellen, falls die Bedingung erfüllt ist
`Shutoff`	`Evaluator`	Reaktion, die eine Komponente ausschaltet
`Smoother`	`Modifier`	Mittelt über mehrere Werte
`UnitPass`	`Modifier`	Lässt nur Daten mit passender Einheit durch

Tabelle 7.2.: Klassenübersicht vom Paket `framework.activities`

Eine `Activity` kann sich in mehreren Zuständen befinden, die durch den Enumerator `Activity.State` ausgedrückt werden. Tabelle 7.3 listet alle Zustände auf. Durch einen entsprechenden Wert können Kombinationen der ersten beiden Zustände mit den drei anderen gebildet werden, womit die Weitergabe von Daten während des Trainings geblockt werden kann (`state = blocked & yellow`).

Zustand	Wert	Bedeutung
normal	0	Keine Blockierung der Ausgabe
blocked	1	Die Weitergabe der Daten wird blockiert
red	2	Trainiere Aktivität, bis (beinahe) keine Reaktion erfolgt
yellow	4	Trainiere Aktivität, bis Reaktionen ungefähr 50/50 erfolgen
green	8	Trainiere Aktivität, bis (fast) immer eine Reaktion erfolgt

Tabelle 7.3.: Der Enumerator `Activity.State`

Das Paket `framework.connectors`

Connectoren sollen die Kommunikation zu anderen Zellen oder sonstigen Ressourcen im Netz bereitstellen. Sie abstrahieren die Verbindungen gegenüber den Aktivitäten. Dies hat den Vorteil, dass auch bei mehreren Aktivitäten, die die gleiche entfernte Ressource ansprechen wollen, zu jener nur eine tatsächliche Verbindung gehalten werden muss. Außerdem werden die Connectoren mit einem passenden Namen angesprochen, so dass sie in den XML-Konfigurationsdateien einfach referenziert werden können. Tabelle 7.4 zeigt alle Klassen im Paket `framework.connectors`.

Klasse	Basisklasse	Kurzbeschreibung
Connector		Abstrakte Basisklasse mit statischen Zugriffsmethoden.
ConsoleLogger	Connector	Gibt Logmeldungen auf der Konsole aus
SQLClient	Connector	Bindet eine MySQL-Datenbank an
TCPConnector	Connector	Verbindet über TCP, z.B. mit (`PrintTCP`)
XMLClientSocket	Connector	Sendet Daten in XML kodiert an andere Zellen
XMLServerSocket	Connector	Lauscht nach Daten und hält Daten zum Abruf bereit

Tabelle 7.4.: Klassenübersicht vom Paket `framework.connectors`

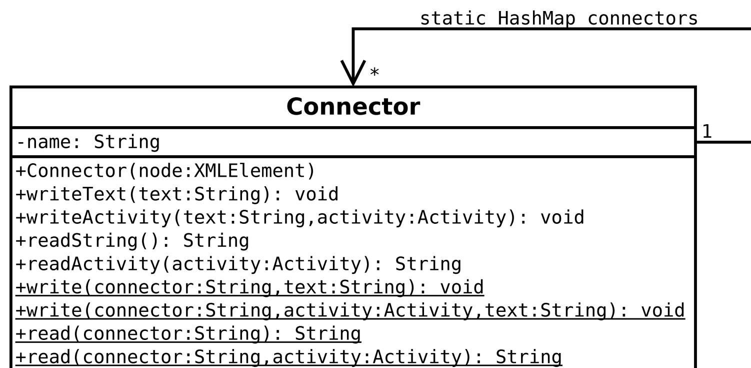

Abbildung 7.1.: Die Klasse `Connector`

Alle Connectoren, die die abstrakte Klasse `Connector` erweitern, müssen die Methoden `writeText()`, `writeActivity()`, `readString()` und `readActivity()` implementieren. Diese werden wiederum von den statischen Methoden von `Connector` aufgerufen. Abbildung 7.1 zeigt weiterhin, dass über eine statische HashMap alle Connectoren mit ihrem Namen als Schlüsselwort ermittelt werden können. So reicht es für eine `Activity` aus, sich diesen Namen zu merken, mit dem sie auf `Connector` lesend und schreibend zugreifen kann. Die Funktionalität und die möglichen Parameter bei der Initialisierung eines `Connectors` können den Javadoc-Kommentaren auf der (*CD-ROM* (A.3)) entnommen werden.

Das `Defaultpaket`

Zusätzlich zum Framework liegen auch noch einige Klassen im `Defaultpaket`. Diese gehören nicht direkt zum Framework, sondern bieten Funktionen, um jenes zu testen. Sie stehen alle für sich, implementieren also die statische Methode `main()`. Allerdings verwenden sie teilweise Klassen des Frameworks, so zum Beispiel `Data` und `Unit`. Die Klassen sind weitgehend selbsterklärend und menügesteuert, die Tabelle 7.5 gibt einen kurzen Überblick. Sie werden für die Ausführung der Beispielszenarien verwendet, die später in diesem Kapitel erläutert werden.

Klasse	**Kurzbeschreibung**
`Computergrid`	Steuert das Beispielszenario *Apoptose im Smart Grid* (7.2)
`DataTest`	Dient zum menügesteuerten Verfassen und Senden von Nachrichten
`Powergrid`	Steuert das Beispielszenario *Apoptose beim Grid Computing* (7.3)
`PrintTCP`	Gibt den Inhalt ankommender TCP-Pakete auf der der Konsole aus

Tabelle 7.5.: Klassen jenseits des Frameworks

7.1.3. Aufbau einer XML-Konfigurationsdatei

Die Konfiguration eines Frameworks erfolgt mit XML-Dokumenten. Grundsätzlich können diese auf verschiedenen Wegen und zu jeder Zeit an das Framework übergeben werden, doch der Prototyp verwendet nur die beim Programmstart als Argumente geladenen XML-Dateien. Es gilt dabei, dass Elemente mit gleichem Namen sich überschreiben, also nur der zuletzt definierte Parameter Anwendung findet. Geben also mehrere XML-Dateien den Tag `verbose` in den `settings` an, so wird der Verboselevel auf den Wert des zuletzt geladenen Dokuments gesetzt. Entspricht der Name eines Tags einer Klasse des Frameworks, so ist ein eindeutiger Name mit dem Attribut `name` anzugeben. Wird ein Name mehrfach für Aktivitäten vergeben, so ist lediglich das zuletzt erzeugte `Activity`-Objekt im Framework ansprechbar.

```
<framework name="Name_der_Zelle">
    <settings>
    </settings>
    <connectors>
    </connectors>
    <units>
    </units>
    <activities>
    </activities>
</framework>
```

Listing 7.1: Grundgerüst einer Konfigurationsdatei

Das Listing 7.1 zeigt das Grundgerüst einer XML-Konfigurationsdatei. Das Attribut `name` muss nicht angegeben werden, sollte aber in wenigstens einer Konfigurationsdatei für ein Framework stehen, um der Zelle eine Bezeichnung zu geben, mit der diese im Netz und ihre Einträge in einem Log identifiziert werden können. Alle Kinder von `framework` sind optional, es können also die Einstellungen in einem anderen XML-Dokument stehen, als die Aktivitäten. Weiterhin können die einzelnen Gruppen über mehrere XML-Dateien verteilt sein. So lässt sich ein hoher Grad an Modularisierung erreichen, wodurch die Konfiguration übersichtlicher und generischer wird.

In `settings` werden optionale globale Einstellungen aufgelistet. Dabei entspricht immer der Variablennamen dem XML-Tag und der Wert dem Inhalt des Elements. Die Einstellungen sind die einzige Gruppe, bei der aus den Kindelementen keine Objekte erzeugt werden, weshalb sie nicht weiter parametrisiert wird. Alle anderen Gruppen beinhalten XML-Elemente, die einen eindeutigen Bezeichner im Attribut `name` haben müssen, mit dem sie im Framework referenziert werden. Ein Element entspricht einem im Framework zu erzeugenden Objekt, dessen Initialisierungswerte in weiteren Unterelementen enthalten sind. Bei Letzteren gilt wieder,

der Parametername entspricht dem XML-Tag. Die Gruppe `connectors` enthält Connectoren, die von den Aktivitäten mit dem jeweiligen Namen angesprochen werden. Ihnen wird meistens eine Adresse mitgegeben, zu der sie die Verbindung aufbauen sollen. In `units` werden alle verwendeten Einheiten aufgeführt. Je Einheit wird ein XML-Element `unit` mit einem eindeutigen Bezeichner im Attribut `name` angelegt. Es enthält zwei optionale Unterelemente mit den XML-Tags `typicallow` und `typicalhigh`, die einen typischen Wertebereich für diese Einheit abstecken. Dieses Intervall kann im Framework für die Auswertung oder für eine bessere Darstellung des Wertes in einer grafischen Benutzeroberfläche (GUI) herangezogen werden. Die letzte Gruppe `activities` definiert die Aktivitäten, die von dem Framework ausgeführt werden sollen. Je gelisteter Aktivität wird ein entsprechendes Objekt erzeugt, dessen Klasse dem verwendeten XML-Tag entspricht. Soll dieses das Ergebnis einer anderen Aktivität weiterverarbeiten, muss es einen `listento`-Tag setzen, dessen Wert dem Namen einer anderen Aktivität entspricht. Je verwendeter Klasse können weitere Parameter mitgegeben werden, zum Beispiel eine Gewichtung des Eingangswertes mit dem Tag `weight`. Bei diesem ist darauf zu achten, dass die Fixkommadarstellung innerhalb des Frameworks berücksichtigt wird. Wenn intern eine 1 einer 1000 entspricht, so muss man für ein Gewicht von 1 ebenfalls eine 1000 angeben. Eine Gewichtung mit dem Faktor -1,5 entspricht folglich einer -1500. Die vollständige Konfigurationsdatei kann im Anhang eingesehen werden (Listing A.2), mögliche Parameter der XML-Elemente können der Javadoc der entsprechenden Klasse entnommen werden (A.3).

```
<Merger name="merge">
    <listento>col</listento>
    <weight>1000</weight>
    <listento>mod</listento>
    <weight>2000</weight>
</Merger>
```

Listing 7.2: Ein `Merger` mit zwei Eingängen

Bei allen Elementen gilt generell, dass die Reihenfolge beliebig ist, sofern keine Werte überschrieben werden. Allerdings kann die Ordnung bei Elementen, die mehrere unterschiedliche gleichnamige Parameter verarbeiten können, Unterschiede in der Auswertung bedeuten. Wird ein `Merger` initialisiert, so werden mehrere Aktivitäten angegeben, bei denen er sich als Listener registriert. Sollen die Eingangsports, die jeder `Aktivity`, der gelauscht wird, entsprechen, gewichtet werden, so müssen die `weight`-Tags in der selben Reihenfolge stehen, wie die `listento`-Tags (Listing 7.2). Auch werden Listener in der Reihenfolge abgearbeitet, in der sie sich bei einer Aktivität registrieren.

Das Einlesen der Konfigurationsdateien in das Framework erfolgt mit einem SAX-Parser. Dieser verarbeitet den Datenstrom noch während des Transfers. Sobald ein XML-Element fertig übertragen wurde, steht es dem Framework zur Verfügung. Da ein Stack verwendet wird, dessen Höhe der Tiefe des aktuell verarbeiteten Elements im Baum des DOMs entspricht, ist vergleichsweise wenig Arbeitsspeicher von Nöten. Wenn es die CPU zulässt, kann das Framework in der Geschwindigkeit gefüllt werden, in der die Konfiguration übertragen wird. Hierfür müssen unter Umständen noch einige kleinere Optimierungen im Framework erfolgen.

7.1.4. Datenaustauschformat zwischen den Zellen

Auch zur Kommunikation zwischen den Zellen wird XML eingesetzt, wie es in Unterabschnitt 6.2.4 erläutert worden ist. Dabei handelt es sich um XML-Darstellungen von `Data`-Objekten, die neben normalen Signalen auch die in Unterunterabschnitt 6.2.4 beschriebenen Mitteilungen enthalten. Ein derartiges XML-Element für ein Datum wird in Listing 7.3 gezeigt.

```
<message>
    <data name="Name_des_Datums">
        <sender>Name_der_sendenen_Zelle</sender>
        <value>42</value>
        <unit>Everything</unit>
        <timestamp>1234567890</timestamp>
    </data>
</message>
```

Listing 7.3: Ein `Data`-Objekt kodiert in XML

7.1.5. Weitere Eigenschaften des Prototyps

Für jede `Activity` und jeden `Connector`, der im XML-Dokument angegeben ist, muss eine passende Klasse im entsprechenden Paket vorhanden sein. Diese wird dann mit dem Classloader von Java geladen, um die verlangten Objekte zu bilden. Der Prototyp setzt voraus, dass diese Klassen lokal und im entsprechenden Unterverzeichnis vorliegen. Dank der umfangreichen Klassenladestrategien von Java wäre es mit geringen Anpassungen problemlos möglich, dass eine Zelle benötigte aber fehlende Klassen über das Netz nachlädt. Das würde das Deployment weiter vereinfachen und den Speicherbedarf in der Zelle reduzieren. Bei besonders leistungsschwachen Komponenten wäre sogar ein Auslagern der Funktionalität des Frameworks an eine andere Netzwerkressource denkbar, die einen Zugriff mit *Remote Procedure Calls* gestattet. Schon jetzt ist dank der TCP-basierten Kommunikation des Frameworks mit anderen Zellen eine dedizierte Maschine realisierbar, die als Kontrollinstanz für mehrere Komponenten dient, wodurch jene entlastet würden.

Dadurch, dass das Framework nur bei neu eingelesenen Daten aktiv wird und ansonsten ruht, ist der Rechenzeitverbrauch gering. Auch die Datenmenge im Netz kann mit einer optimierten Konfiguration recht klein gehalten werden. Die Connectoren helfen die Zahl der Sockets zu beschränken. Falls zu viele Aktivitäten in einem Framework angelegt werden, könnte durch den Einbau von Serialisierung eine Auslagerung der nicht ständig genutzten Objekte auf einen Festspeicher oder gar auf eine Netzressource erfolgen, was den Arbeitsspeicherverbrauch minimiert. Um die Systemanforderungen so gering wie möglich zu halten, wurde bei der Kozeptionierung des Frameworks auf den ausschließlichen Einsatz von Integern (bzw. Longs) geachtet, um eine effiziente Berechnung auf einfachen Mikrocomputern zu ermöglichen. Auch werden hierdurch die wertvollen Fließkommaeinheiten der Hochleistungscomputer für die eigentliche Mission freigehalten. Da die Datenauswertung in einem Framework nur bei Bedarf stattfindet, reduziert sich die benötigte Rechenleistung, besonders, wenn nur einfache Aggregatsfunktionen und Fallunterscheidungen zur Anwendung kommen. Für die Kommunikation über das Netz wird auf einfaches TCP/IP gesetzt, was einen geringen Implementierungsaufwand mit sich bringt, da es meisten bereits bestehende Infrastruktur nutzt.

Wenn der Prototyp um neue Aktivitäten oder Connectoren erweitert werden soll, muss lediglich eine neue Klasse angelegt werden, die eine passende Basisklasse erweitert. Außer dieser müssen noch die Klassen `Data`, `Unit` und `XMLElement` bekannt sein. Die erzeugte `class`-Datei wird in das passende Unterverzeichnis auf den Zielkomponenten kopiert. Sie sollte dem Framework augenblicklich zur Verfügung stehen.

7.2. Apoptose im Smart Grid

Nicht nur im IT-Sektor gibt es verteilte und daher komplexe Systeme, die als Grid bezeichnet werden. So sehen sich auch Energieversorger und Betreiber von Stromnetzen einer immer stärker werdenden Dezentralisierung ausgesetzt. Wie der bereits in Unterabschnitt 2.4.3 geschilderte Fall eines Blackouts zeigt, haben die Verantwortlichen die Übersicht schon öfters verloren. Dies liegt unter anderem an der wachsenden Zahl an Teilnehmern am Stromnetz. So gibt es immer mehr Verbraucher und immer mehr Kraftwerke. Hierdurch steigt die Leistung, die über die Stromleitungen transportiert werden muss. Diese haben kaum noch Reserven für die weiteren Belastungen, die auf sie in vielfältiger Form zukommen.

7.2.1. Probleme im Stromnetz

Hierfür gibt es viele Gründe, denen man mit Automatisierung und moderner Kommunikations- und Computertechnik begegnet. Bisher wird die Seite der Produzenten und der Zustand der Netze immer feinmaschiger mit Sensoren überwacht, um mit möglichst vielen und genauen Messwerten rechtzeitig notwendige Entscheidungen treffen und einen Stromausfall verhindern zu können. Nun soll auch auf Verbraucherseite aufgerüstet werden. Der wichtigste Grund ist, dass produzierter Strom sofort verbraucht werden muss. Andererseits muss auf eine plötzliche Lastspitze umgehend reagiert werden, um eine Unterversorgung und damit einen Zusammenbruch des Stromnetzes zu verhindern. Die Generatoren benötigen zum Anlaufen allerdings ein paar Minuten, während Verbraucher, wie zum Beispiel Glühbirnen, unverzüglich ihre volle Leistung entfalten. Wenn also viele Kunden gleichzeitig das Licht einschalten oder ein großer Verbraucher wie ein Eisenbahnzug anfährt, ist die

Stabilität des Netzes kurzzeitig gefährdet. Daher möchten die Energielieferanten mittels *Smart Metering* derartige Spitzen frühzeitig erkennen und entsprechend reagieren können. Der Kunde hingegen soll damit Energie und Geld sparen, sowie in den Genuss zusätzlichen Komforts in Verbindung mit Heimautomatisierung, auch als *Smart Home* bekannt, kommen [TK 11].

Während man die fossilen Brennstoffe für Kohle-, Gas- und Ölkraftwerke praktisch überall hin schaffen kann und Atomkraftwerke zum Betrieb lediglich ausreichend Kühlwasser in der Nähe benötigen, ist die umweltfreundliche und regenerative Stromerzeugung meistens vom Standort abhängig. Wasserkraftwerke benötigen ein Gefälle bei einem fließenden Gewässer, Windparks sollten in Gegenden mit reichlich konstantem Wind gebaut werden und Solarenergie eignet sich besonders für Orte mit viel Sonnenschein. Gerade die Küstenregionen bieten sich für Offshore-Windräder und Gezeitenkraftwerke an, auch weil hier viel Platz mit wenigen Nachbarn ist, die Einspruch gegen die Anlagen einlegen könnten.

Doch leben nicht alle Menschen an der Küste, auch im Landesinneren wird viel Strom benötigt. Daher sind leistungsfähige und verlustarme Hochspannungsleitungen gefragt, die von der Küste landeinwärts führen. Reichen deren Kapazitäten nicht aus, sind die inländischen Gebiete unterversorgt, was einen Ausgleich durch alternative Energiequellen oder Speicherkraftwerke erfordert. Gleichzeitig kann der produzierte Strom nicht verbraucht werden. Entweder wird die Produktion gedrosselt, was die Rentabilität des Kraftwerks herabsetzt, oder der Strom muss in alternative Verbrauchsgebiete verkauft oder teilweise sogar verschenkt werden. Um dies zu ermöglichen, ist ein gut ausgebautes und gewartetes Verbundnetz mit ausreichend Reserven unersetzlich. Daher sollen die Verbundnetze der Energieversorger Konzepte der Informations- und Kommunikationstechnologie übernehmen [Kuri 11]. Hier gibt es mit dem Internet ein sehr komplexes Netzwerk, das bisher gut skaliert und eine schnelle Akzeptanz sowohl bei Produzenten, als auch Konsumenten erreicht hat. Dessen paketorientierte Struktur und die damit möglichen Routingmethoden sind auch für Stromnetze interessant. Doch leider lässt sich Strom nicht in Pakete packen und an eine beliebige Adresse senden, wenn man nicht Batterien mit der Post verschicken möchte. Dennoch versucht man, durch Abstraktion und mit moderner Kommunikationstechnik, Smart Grids zu schaffen und dadurch zukünftigen Anforderungen schneller und flexibler begegnen zu können. Dabei werden die einzelnen aktiven Komponenten, wie Turbinen, Transformatoren oder Stromzähler mit Computern ausgestattet und vernetzt, wodurch sie intelligenter werden sollen.

Leider wurden bei den ersten Planungen einige Fehler aus den Anfängen des Internets wiederholt. So wurden zunächst diverse proprietäre Protokolle und Verfahren zum Datenaustausch entwickelt, was die heterogene Nutzung von Geräten unterschiedlicher Hersteller erschwert. Auch über die Absicherung der Daten bei Speicherung und Übertragung mittels Kryptographie war zunächst nicht ausreichend bedacht worden [SK 10]. Ähnlich wie bei GSM soll sich der Nutzer dem Anbieter gegenüber mit einem Äquivalent zur SIM-Karte authentifizieren. Dadurch und durch die teilweise sekundengenaue Erfassung ist das Erstellen einer personalisierten Verbrauchsstatistik möglich, die erschreckend genaue Rückschlüsse auf die Lebensgewohnheiten des Kunden ziehen lassen. So kann anhand von Schwankungen im Strombedarf eines Fernsehgeräts bei unterschiedlichem Bildinhalt das angezeigte Fernsehprogramm ermittelt werden [dab 11].

Die moderne Energieproduktion zeichnet sich durch immer weiter verstreute Standorte aus. Gegenüber Großkraftwerken hat die Verteilte Stromproduktion einige Vorteile. So soll der Strom da produziert werden, wo er verbraucht wird, wodurch Transportverluste vermieden und die bei der Erzeugung anfallende Wärme auch gleich zum Heizen genutzt werden kann. So erreichen Blockheizkraftwerke bei fossilen Brennstoffen mit die höchste Effizienz. In ländlichen Gegenden gibt es vermehrt Biogaskraftwerke und Geothermie und auf dem Meer finden sich diverse Windparks. Hinzu kommen Spartechniken wie die Energierückgewinnung beim Bremsen elektrischer Eisenbahnzüge. Kann die Bremsenergie nicht in dem Augenblick aufgenommen werden, in dem der Zug verzögert, muss sie buchstäblich mittels eines Widerstands verheizt werden.

All diese Kleinkraftwerke müssen gesteuert und in das Verbundnetz aufgenommen werden. Auch kann durch die teilweise umweltabhängigen Produktionsmethoden keine konstante Lieferung der Energie garantiert werden. Ein ausgeklügeltes System ist notwendig, um Ausfälle zu kompensieren und Produktionsspitzen sinnvoll zu nutzen. Dass die vielen Kleinerzeuger unterschiedlichen Gesellschaften angehören, macht die Lage zusätzlich unübersichtlicher. Sogar klassische Energiesenken wie Städte liefern vermehrt Strom.

Durch Dezentralisierung werden Administration und Erweiterbarkeit der Stromnetze einfacher. Neue Systeme registrieren sich selbstständig im Netz, passen ihren Bedarf oder ihre Produktion von elektrischem Strom an die Situation im Netz an, bieten ein auf Prioritäten basierendes Krisenmanagement und verhindern durch Datensparsamkeit, dass der Energieversorger zu viel über seine Kunden in Erfahrung bringt.

7.2.2. Beispielzenario

Um den Ansprüchen eines heutigen Verbundnetzes gerecht zu werden, sind Techniken aus der IT gefragt. Diese sollen die Kontrolle sichern, auch wenn trotz vorausschauender Planung kritische Ereignisse die Versorgung mit Strom gefährden. Doch wie schon in 4.2.3 gezeigt, reicht das nicht immer aus. Darum sollen die Stromnetze intelligent werden, wobei auch das Konzept der Apoptose einen wichtigen Beitrag leisten kann. Um die Auswirkung von Apoptose in einem elektrischen Stromnetz genauer betrachten zu können, empfiehlt es sich, ein kleines Beispielszenario zu entwerfen, mit dem der Vorgang der Selbstabschaltung in einem verteilten Energiesystem simuliert werden kann. Abbildung 7.2 zeigt ein derartiges Szenario.

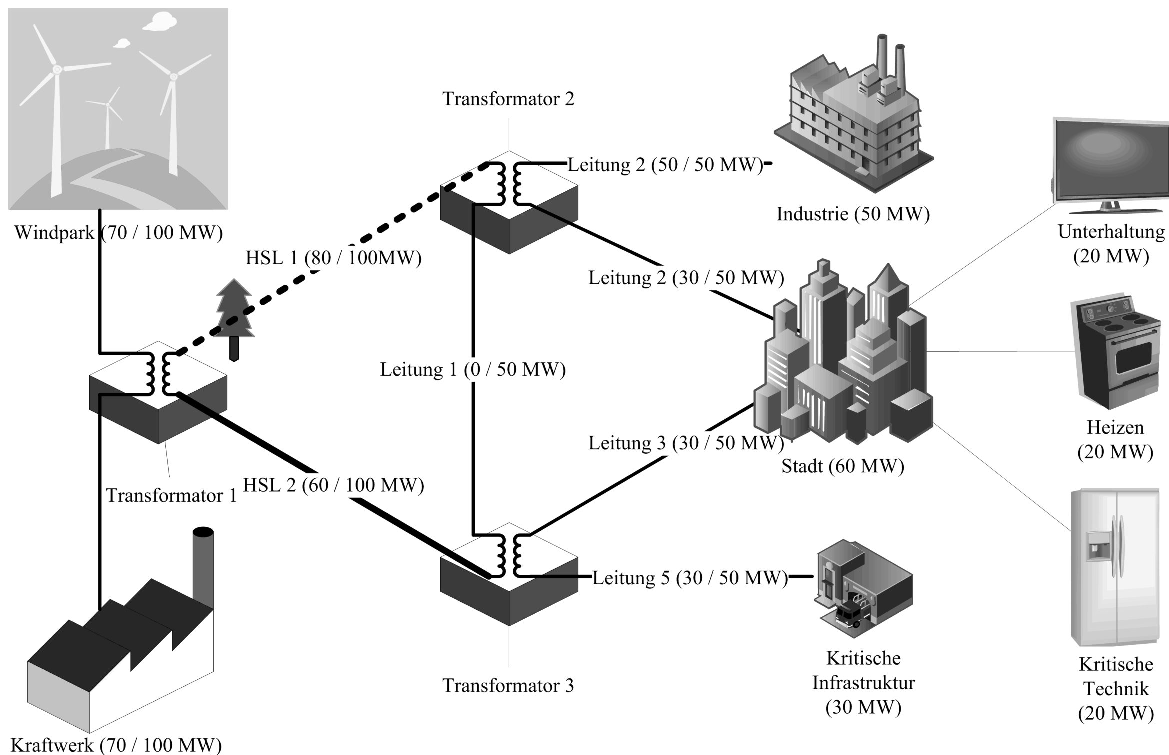

Abbildung 7.2.: Ein Beispielszenario für Apoptose im Stromnetz

Ein *Kraftwerk* und ein *Windpark* können jeweils ein Maximum von 100 MW Strom erzeugen, der von *Transformator 1* auf eine Spannung von über 100 kV gebracht und über zwei Hochspannungsleitungen zu einem lokalen Stromnetz übertragen wird. Dessen Leitungen transportieren mit einer deutlich niedrigeren Spannung jeweils bis zu 50 MW. Es besteht aus zwei weiteren Transformatoren, die die Spannung für das Ortsnetz anpassen und die Endkunden beliefern. An *Transformator 2* ist eine *Industriegegend* angebunden, die bis zu 50 MW verbraucht, *Transformator 3* versorgt *kritische Infrastruktur*, wie Feuerwehr, Krankenhäuser, Polizei, Pumpen und Verkehrswesen, was mit 30 MW zu Buche schlägt. Beide Transformatoren binden außerdem eine Stadt an, die bis zu 60 MW Strombedarf hat, unterteilt in drei Klassen von Verbrauchern. Nach aufsteigender Wichtigkeit sortiert sind das *Unterhaltungselektronik*, *Heiztechnik* und *kritische Technik* wie Kühlschränke, Aufzüge, Kommunikation und Beleuchtung. Um das Beispiel einfach zu halten, verbrauchen alle Geräteklassen jeweils 20 MW.

An normalen Tagen sind beide Energieerzeuger zu 70 % ausgelastet, die Hochspannungsleitung *HSL 1* transportiert 80 MW, *HSL 2* 60 MW. Der Bedarf der Verbraucher kann voll gedeckt werden. Für den folgenden Ablauf gilt, dass jede Stromleitung einer Zelle entspricht. Auch die Kraftwerke und die Verbraucher stellen jeweils eine Zelle dar, wobei Haushalte, die mit Smart Metern ausgestattet sind, die ihnen zugehörigen Elektrogeräte als Subzellen betrachten und kontrollieren.

Schritt 1: Ausfall von *HSL 1*

Analog zu dem in Unterunterabschnitt 4.2.3 geschilderten Vorfall erzeugt auch in diesem Beispiel Baumwuchs einen Kurzschluss in *HSL 1*. Dies erkennt *Transformator 1* an dem sprunghaften Anstieg der Stromstärke auf seiner Sekundärseite und schaltet *HSL 1* vollständig ab. Dies kann als Apoptose von *HSL 1* gesehen werden. Sie schickt eine entsprechende Meldung an *HSL 2*, die allerdings nur die Hälfte der Aufgabe von *HSL 1*, sprich 40 MW, übernehmen kann. Sie sendet daher ein weiteres Hilfeersuchen an die anderen Zellen.

Schritt 2: Drosselung der Produktion

Das *Kraftwerk* erhält die Nachricht, dass *HSL 1* ausgefallen und *HSL 2* überfordert ist. Da keine weiteren Alternativwege zur Verfügung stehen und es flexibler als der Windpark ist, drosselt es seine Produktion um 40 MW. Es sendet eine Mitteilung an die Verbraucher, die jene zum Energiesparen auffordert.

Schritt 3: Reduktion des Konsums

Die Konsumenten erhalten eine Modifikation ihrer Mission, die einer Reduktion des nutzbaren Stromkontingents gleich kommt. Da die *kritische Infrastruktur* die höchste Priorität genießt, reagiert sie nicht weiter. Auch in der *Stadt* sollen Kühlschränke und Telefone weiter laufen, weshalb sie mindestens 20 MW Bedarf anmeldet. Die *Industrie* kann ihren Verbrauch senken, indem beispielsweise Berechnungen in andere Rechenzentren ausgelagert oder Teile der Produktionsanlagen stillgelegt werden.

Die Verbraucher sind alles einzelne Zellen, die untereinander aushandeln, wem wieviel Energie zur Verfügung steht. Um den Schock durch den Ausfall von *HSL 1* zu überwinden, werden zunächst alle unkritischen Systeme heruntergefahren, beginnend mit der niedrigsten Priorität. Das erledigen die Zellen selbst, jede entscheidet also für sich, ob es an der Zeit für Apoptose ist. Das gilt zunächst für die gesamte *Unterhaltungselektronik*, vielleicht mit Ausnahme von Radios für Nachrichtensendung, sowie ein Rechenzentrum und eine Produktionsstraße in der *Industrie* mit zusammen 10 MW Verbrauch. Die restlichen 10 MW werden durch das Abschalten von Elektroheizungen, Elektroautotankstellen und Ähnlichem gewonnen.

Der Gesamtstrombedarf liegt nun bei 100 MW: 30 MW für die *kritische Infrastruktur*, 40 MW in der *Industrie* und 30 MW in der *Stadt* für die *kritische Technik* und das *Heizen*.

Schritt 4: Dynamische Energiezuteilung

Da mittags viel gekocht wird, benötigt die *Stadt* zusätzliche 10 MW. Dies teilt sie den anderen Zellen auf der Verbraucherseite mit. Hierdurch erhöht sich die Priorität des *Heizens*, womit dieses wichtiger als Teile der *Industrie* wird. Es werden also Produktionsstätten mit einem Bedarf von 10 MW heruntergefahren, die Belegschaft kann in die Mittagspause. Ist diese vorbei, erhält die *Industrie* sogar 15 MW, da einige *Heizungen* in der *Stadt* tagsüber abgeschaltet bleiben. Auch die Straßenlaternen sind bis zum Einbruch der Dunkelheit außer Betrieb. Dafür fallen in der *Industrie* die Nachtschichten aus, um die Beleuchtung der *Stadt* zu gewährleisten.

Schritt 5: Dynamische Stromproduktion

Wenn es sehr windig ist, kann der *Windpark* die Versorgung alleine stemmen. Er hat gegenüber dem flexibleren *Kraftwerk*, das mit fossilen Brennstoffen betrieben wird, eine höhere Abnahmepriorität, um keine Windkraft zu vergeuden. Er sendet daher einen Apoptosebefehl an das *Kraftwerk*, welches sich herunterfährt. Lässt der Wind nach, so teilt dies der *Windpark* dem Kraftwerk mit, das daraufhin die Deckung der Versorgungslücke übernimmt.

Schritt 6: Wiederherstellung von *HSL 1*

Nachdem die Bäume, die den Kurzschluss verursacht hatten, gestutzt worden sind, geht *HSL 1* wieder ans Netz. Dies hat eine Erhöhung des Gesamtstromkontingents zur Folge. Kommen neue Verbraucher ans Netz, werden ihre Stromforderungen erfüllt, ohne dass andere Zellen sich abschalten müssen. Der Normalzustand ist wiederhergestellt.

7.2.3. Versuchsaufbau und -durchführung

Der Versuchsaufbau kann gegenüber dem geschilderten Szenario noch weiter vereinfacht werden. So bleibt der Verbrauch der *kritischen Infrastruktur* immer gleich und auch die *Leitungen 1-7* des Ortsnetzes sind ausreichend dimensioniert, um keinen Einfluss auf den Ausgang des Beispiels zu nehmen. In der Realität sind die *Transformatoren* die aktiven Komponenten des Stromnetzes, doch ist eine Betrachtung der *Hochspannungsleitungen* in diesem Szenario sinnvoller. Auf diese Weise kann die Modellierung von Sensoren, wie zum Beispiel die Thermometer in den *Transformatoren*, entfallen.

Da die tatsächlichen Spannungen, Stromstärken und Widerstände keine direkte Bedeutung für das Modell haben, wird nur das Produkt *Leistung*, gemessen in Megawatt, verwendet. Dies verringert den Implementierungsaufwand und damit die Komplexität der Zellen, was der Übersichtlichkeit der XML-Konfigurationsdateien und der Abläufe im Framework zu Gute kommt. Aus dem gleichen Grund wird auch auf die Unterteilung der *Stadt* in weitere Zellen verzichtet. Weil nur die produzierte, übertragene und verbrauchte Leistung in das Modell eingeht, lässt es sich leicht auf Gasnetze, Wasserversorger und ähnliche Infrastrukturen übertragen.

Zelle	Konfigurationsdatei	Mission
Windpark	*windpark.xml* (A.4)	Liefert wetterabhängig Strom, hat daher eine hohe Priorität
Kraftwerk	*kraftwerk.xml* (A.5)	Liefert flexibel Strom, hat daher eine niedrige Priorität
HSL 1	*hsl1.xml* (A.6)	Gibt Produktions- und Verbrauchskontingent vor. Partner: HSL 2
HSL 2	*hsl2.xml* (A.7)	Gibt Produktions- und Verbrauchskontingent vor. Partner: HSL 1
Stadt	*stadt.xml* (A.8)	Verbraucht Strom, mittlere Priorität, meldet und passt Bedarf an
Infrastruktur	*infrastruktur.xml* (A.9)	Verbraucht Strom, hohe Priorität, meldet konstanten Bedarf
Industrie	*industrie.xml* (A.10)	Verbraucht Strom, niedrige Priorität, meldet und passt Bedarf an

Tabelle 7.6.: Die modellierten Zellen im Szenario *Powergrid*

Tabelle 7.6 zeigt die modellierten Zellen des Szenarios *Apoptose im Smart Grid* (7.2), deren Konfigurationsdateien auf der *CD-ROM* (A.3) im Verzeichnis `Sourcen/powergrid` oder im Anhang bei den *XML-Listings* (A.1) gefunden werden können. Sie entsprechen demnach der Mission, die das System *Powergrid* zu erfüllen hat.

Alle Zellen benötigen Informationen über ihre Umwelt, aus denen sie die Situation des Systems ermitteln. Da es sich nur um eine Simulation handelt, werden diese Daten von der Klasse `Powergrid` bereitgestellt. Sie enthält Sensordaten, die über einen Dialog auf der Kommandozeile verändert werden können. Die Klasse stellt außerdem einen Server bereit, der die angeforderten Sensordaten an die anfragende Zelle via TCP ausliefert. Will die Hochspannungsleitung *HSL 1* die Last wissen, die sie aktuell zu bewältigen hat, sendet sie `get hsl1primpower`; soll die Leistung, die von ihr transportiert worden ist, ermittelt werden, sendet sie `get hsl1secpower`. Sie erhält dann jeweils die entsprechende Zahl in Megawatt. Die Differenz dieser Werte entspricht dem Leitungsverlust von `HSL 1`, der hier mit 0 MW angenommen wird.

Da keine reale Hardware existiert, die das Framework steuern könnte, werden die Steuerbefehle in Textform über das Netz an einen Server übermittelt, der diese auf der Konsole ausgibt. Hierfür dient die Klasse `PrintTCP`. Um eine Physiksimulation zu vermeiden, werden diese Befehle von einem Menschen umgesetzt, der die entsprechenden Reaktionen ausführt. Schaltet zum Beispiel ein Kraftwerk ab, so wird der zugehörige Produktionswert in der Klasse `Powergrid` auf 0 MW gesetzt, was seinerseits Aktionen in den Zellen hervorruft. Auf diese Weise kann die Simulation Schritt für Schritt durchgeführt und leicht modifiziert werden. Eine

weitere Informationsquelle für ein Framework sind Daten von anderen Zellen, die wie in Unterabschnitt 6.2.4 und in Unterabschnitt 7.1.4 beschrieben, übermittelt werden. Sollte ein manueller Eingriff erforderlich werden, können mit der Klasse `DataTest` dialoggesteuert Mitteilungen erzeugt und an eine Zelle gesendet werden.

Nachdem `PrintTCP` und `Powergrid` gestartet worden sind, können die Zellen initialisiert werden. Hierzu werden die in Tabelle 7.6 genannten XML-Dateien dem Framework beim Start als Parameter übergeben. Es kann entweder je Zelle eine Instanz erstellt oder alle Zellen in einem Framework betrieben werden. Zusätzlich muss noch das XML-Dokument *gemeinsam.xml* (A.3) als Argument bei jeder Instanz des Frameworks mitgegeben werden. Es enthält alle gemeinsamen Definitionen der Zellen, wodurch diese nur ein einziges Mal für alle Zellen angegeben werden müssen. Grundsätzlich wären noch weitere Aufteilungen der Missionsbeschreibungen möglich, sofern dies sinnvoll ist und der Übersichtlichkeit beiträgt.

Um die genauen Vorgänge besser nachvollziehen zu können, wird *„Schritt 1: Ausfall von HSL 1“* exemplarisch durchgeführt. Hierzu werden die in Tabelle 7.7 angegeben Zellen sowie die Simulationssteuerung `Powergrid` und für die Ausgabe der Reaktionen auf der Konsole `PrintTCP` in der in der Tabelle aufgeführten Reihenfolge gestartet. Jede der Klassen lauscht auf den genannten Ports auf eingehende Nachrichten, die für die jeweilige Zelle bestimmt sind.

Klasse/Zelle	**Port**	**Aufruf**
PrintTCP	42000	`java PrintTCP`
Powergrid	42100	`java Powergrid`
HSL 1	42101	`java Main gemeinsam.xml hsl1.xml`
HSL 2	42102	`java Main gemeinsam.xml hsl2.xml`
Windpark	42103	`java Main gemeinsam.xml windpark.xml`
Kraftwerk	42104	`java Main gemeinsam.xml kraftwerk.xml`
Stadt	42105	`java Main gemeinsam.xml stadt.xml`
Infrastruktur	42106	`java Main gemeinsam.xml infrastruktur.xml`
Industrie	42107	`java Main gemeinsam.xml industrie.xml`

Tabelle 7.7.: Die Zellen mit Startparametern und Port für XML-Nachrichten im Szenario *Powergrid*

Nachdem Alles läuft, können die Werte, die `Powergrid` bereit hält, verändert werden. So erfragt *HSL 1* mit „`get HSL1primpower\` die Leistung, die von *Transformator 1* an sie geliefert wird, und mit „`get HSL1secpower\` die Leistung, die sie an *Transformator 2* abgibt. Sie bildet die Differenz dieser beiden Werte, um den Leitungsverlust zu berechnen. Sollte dieser zu hoch ausfallen, also zu wenig bis gar kein Strom transportiert werden, liegt vermutlich ein Kurzschluss vor. *HSL 1* schaltet sich daraufhin ab, um weiteren Schaden zu vermeiden.

Ohne Apoptose würde nun die gesamte Leistung, die von den Kraftwerken erzeugt wird, über *HSL 2* geleitet, die damit um 40 MW überlastet würde. Dies entspricht 140 % der maximal zugelassenen Nennlast. Dies kann zum Ausfall von *HSL 2* führen, wodurch alle Verbraucher stromlos wären, während die Kraftwerke heiß laufen würden. Um dem totalen Blackout entgegenzuwirken, müssen die Generatoren auf eine Produktion von insgesamt 100 MW gedrosselt werden, die *HSL 2* noch bewältigen kann. Das Ergebnis sind unterversorgte Verbraucher und ein instabiles Stromnetz. Da die Glühbirnen in einer derartigen Situation nicht mehr mit voller Leistung strahlen, wird das Ereignis auch als *Brownout* bezeichnet. Das Netz kann nur stabilisiert werden, wenn einige Konsumenten vom Netz getrennt werden, bis sich die Lage wieder normalisiert hat. So lange bleiben demnach manche Stadtviertel dunkel. Die Apoptose kann den Strommangel nicht ausgleichen, aber das Netz dazu befähigen, sich selbsttätig zu stabilisieren, indem sich unwichtigere Verbraucher abschalten.

Zunächst teilt *HSL 1* daher *HSL 2* ihr baldiges Versagen und die zu übernehmende Strommenge mit. *HSL 2* errechnet, dass sie nur die Hälfte der zusätzlich geforderten 80 MW bewältigen kann. Sie fordert die Kraftwerke auf, zusammen 40 MW weniger zu produzieren, während sie die Verbraucher dazu aufruft, die Leistungsaufnahme um insgesamt 40 MW zu reduzieren. Da der *Windpark* weiterhin 70 MW an umweltfreundlicher Energie liefert, drosselt das *Kraftwerk* seine Produktion auf 30 MW.

Die Verbraucher handeln untereinander aus, wie viel Energie jeder Partei zusteht. Es gibt diverse Protokolle, die genau dies leisten, aber nicht Teil dieser Arbeit sind. Daher wird ein fest in den XML-Dateien vorgegebenes, auf Prioritäten basierendes Verfahren angewendet. Da die *kritische Infrastruktur* die höchste Priorität genießt, kann sie ihren Bedarf vollständig decken, womit für *Stadt* und *Industrie* noch 70 MW verbleiben. Aufgrund von internen Prioritäten und Verteilungen einigen sie sich darauf, dass die *Stadt* die Leistungsaufnahme halbiert, während die *Industrie* ihren Verbrauch um 10 MW zurückfährt.

Dank dem eben dargelegten apoptotischen Verhalten der Zellen, werden weder *HSL 1* noch die *Kraftwerke* überlastet. Der Strom fällt nicht unkontrolliert oder flächendeckend aus, sondern wird gezielt bei weniger wichtigen Verbrauchern eingespart. Die kritischen Systeme laufen weiter, wodurch die Grundversorgung aufrechterhalten bleibt. Der wirtschaftliche Schaden bleibt so gering wie möglich und der Komfort der Menschen wird deutlich weniger eingeschränkt, als dies ohne Apoptose der Fall wäre.

Die weiteren Schritte des Beispielszenarios setzen eine wesentlich umfangreichere und schwerer zu verstehende Konfiguration des Prototypen voraus, ohne einen weiteren größeren Erkenntnisgewinn zu liefern. Daher wird auf deren konkrete Implementierung und Ausführung verzichtet. Ähnlich verhält es sich mit dem folgenden Szenario: es liefert ein interessantes Experimentierfeld für konkretere Arbeiten, allerdings kaum Erfahrungen, die man nicht schon auf abstrakter Ebene aus dem eben geschilderten Ablauf entnehmen könnte.

7.3. Apoptose beim Grid Computing

Mit der voranschreitenden Vernetzung von Computersystemen erhöht sich die Komplexität immens, wie schon in Abschnitt 2.3 beschrieben worden ist. Die bisherigen in Abschnitt 4.1 dargelegten Maßnahmen tragen alle dazu bei, die Sicherheit der Mission zu erhöhen, doch sind sie fast nur auf den bedingungslosen Erhalt aller Komponenten eines verteilten Systems ausgelegt. Aber erst Konzepte wie das *Autonomic Computing* (4.3) sind dazu geeignet, die Komplexität zu verringern und ein umfangreiches Fehlermanagement in einem Grid automatisiert zu betreiben.

7.3.1. Probleme im Grid

Wenn es zu einem Schaden gekommen ist, haben die Sicherheitstechniken versagt und bieten meistens keine Lösung mehr. Doch es reicht eben nicht aus, nur den Fehlerfall zu verhindern, es muss auch das Vorgehen bei einem Defekt geplant werden, um weitreichende Gefahren von einem System abzuwenden.

Die bisher vorgestellten Gefahren für ein Rechenzentrum, und damit einen entscheidenden Teil eines Grids, sind vielfältig. Neben den normalen Alltagsschäden und Sicherheitsbrüchen kommen viele negative Einflüsse von außen, angefangen bei Dieben und Spionen bis hin zu Naturkatastrophen. Wie auch schon Abschnitt 7.2 gezeigt hat, hängt in Zukunft die Leistungsfähigkeit eines Rechenzentrums nicht nur von den benötigten digitalen Kapazitäten, sondern auch vom verfügbaren Stromkontingent ab. So könnte, abhängig von Lastspitzen im Stromnetz und den damit einhergehenden Preisschwankungen, Nutzungsmodelle interessant werden, bei denen ein Rechenzentrum nur bei günstigem Strom voll arbeitet. Auch suchen die Energiekonzerne nach schnell anzapfbaren Stromspeichern, um besagte Lastspitzen abzufangen. Die Notstrombatterien in einem großen Rechenzentrum könnten da durchaus als kurzfristiger Puffer interessant werden. Gerade in einem europaweiten Wissenschaftsgrid könnte eine Jobverteilung nach regional verfügbarem Strom kostensenkend wirken. So würden Rechenzentren im Norden Deutschlands bei kräftigem Wind von günstiger Windenergie profitieren, während bei Flaute solarbetriebene Rechenzentren im Süden die Arbeit übernehmen. Damit käme nicht mehr der Strom zum Verbraucher, der Verbraucher käme zum Kraftwerk, was das Stromnetz etwas entlasten könnte.

Gefahren für eine Mission entstehen in einem Computergrid immer dann, wenn ein Mangel einer Ressource eintritt oder eine Komponente fehlerhaft arbeitet. Hat das Problem seine Ursache außerhalb der eigenen Zuständigkeit eines Systems, ist es deutlich schwerer, zeitnah und schnell wirksam einzugreifen. Eine Alternative, um die Mission weiterführen zu können, besteht in der Suche nach einem geeigneten Ersatz.

7.3.2. Beispielszenario

Ein einfaches Beispiel eines Rechenzentrums, das Teil eines Grids ist, soll die Anwendung und den Nutzen von Apoptose verdeutlichen. Abbildung 7.3 zeigt ein derartiges Szenario. *Nutzer* in einer virtuellen Organisation nehmen über ein `Wide Area Network` (WAN) Kontakt mit einem Supercomputer auf, um Berechnungen durchführen zu lassen. Der Auftrag wird an den *Indexserver* übergeben, der diesen auf die drei *High Performance Computer HPC 1*, *HPC 2* und *HPC 3* aufteilt. Neben dem Zugriff auf das Netz benötigt jeder Computer noch die Ressource *Strom*. Da die *Nutzer* dazu berechtigt sind, Missionen an die Rechner zu senden, lässt sie die *Firewall* gewähren. Entdeckt sie einen potentieller *Angreifer*, so meldet sie dies den anderen Komponenten im Netz, so auch dem *Indexserver*. Im Rechenzentrum existieren noch eine Reihe von Sensoren und *Alarmgeber*, hier exemplarisch durch einen Feuermelder vertreten. Dieses Beispiel verfügt somit über sieben Zellen: *Firewall*, *Alarmgeber*, *Stromversorgung*, *Indexserver* und die drei *HPCs*.

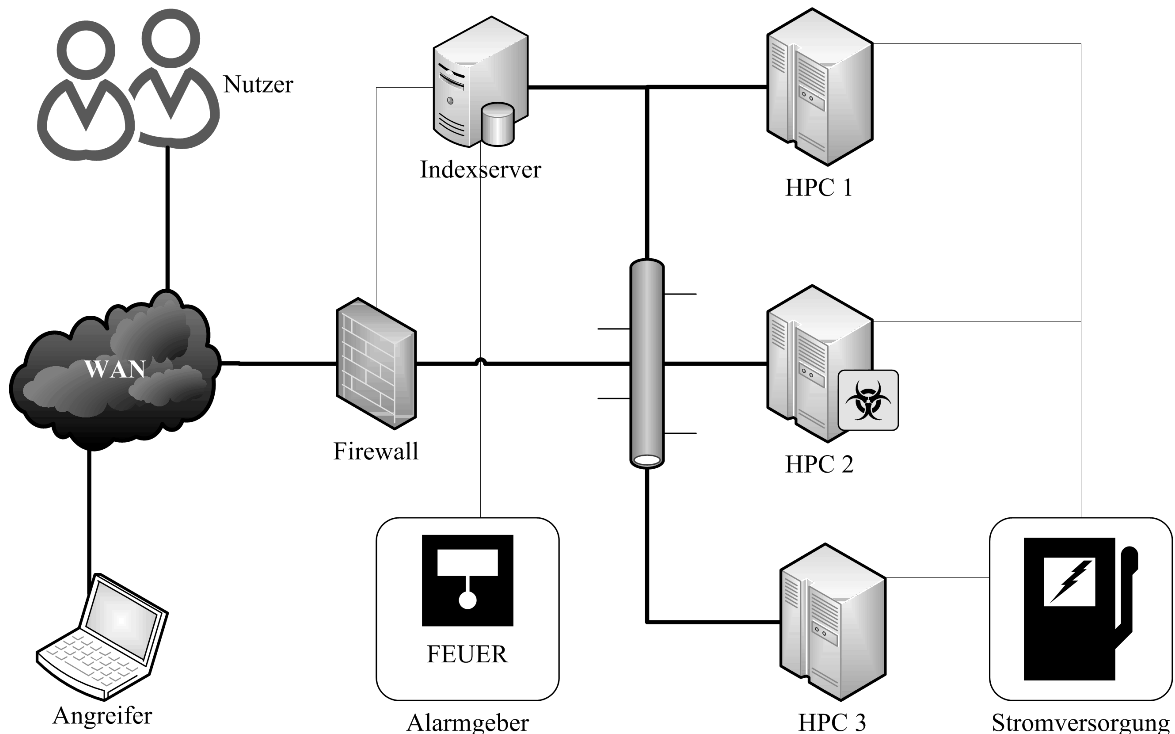

Abbildung 7.3.: Ein Beispielszenario für Apoptose im Computergrid

Fall 1: Reaktion auf einen Angreifer

Einem Angreifer ist es gelungen, eine Schadsoftware auf *HPC 2* zu installieren. Diese beginnt, große Mengen an Spam-E-Mails zu versenden. Die *Firewall* entdeckt einen massiven Anstieg des Datenvolumens durch *HPC 2* in das *WAN*, noch dazu auf dem für die Mission ungewöhnlichen Port 25. Sie teilt dies *HPC 2* durch einen *Apoptosebefehl* mit, der daraufhin sich selbst analysiert und einen geöffneten Port vorfindet, der nicht für die Mission benötigt wird. Daher muss *HPC 2* annehmen, mit einer Malware infiziert worden zu sein und beschließt, sich unter Verwendung eines früheren System-Snapshots neu zu starten.

HPC 2 teilt die Absicht, eine Form von *Apoptose* durchzuführen, den anderen Zellen im Netz mit und setzt sich auf einen früheren Zustand zurück. Der *Indexserver* trägt *HPC 2* aus der Liste der verfügbaren Rechner aus. Nach einiger Zeit kehrt der vom Virus gereinigte *HPC 2* in das Grid zurück, was umgehend vom *Indexserver* registriert wird. *HPC 2* erhält erneut seine vorherige Aufgabe und ist damit wieder ein Teil der Mission des Grids.

Fall 2: Suspendierung bei Stromknappheit

Bei Erdarbeiten wurde die Stromleitung zum Rechenzentrum unterbrochen. Daraufhin springt die *Notstromversorgung* ein, kann aber den Betrieb nicht lange genug aufrecht erhalten, um alle Jobs abzuschließen, was sie mit einer *Apoptosemeldung* dem Grid mitteilt. Der *Indexserver* sendet daraufhin *Apoptosebefehle* mit aufsteigenden Prioritäten an die drei *HPCs*. Diese suspendieren alle Aufträge mit geringerer Dringlichkeit und melden die restliche benötigte Rechenzeit zurück. Dies wiederholt sich so lange, bis genug Strom zur Verfügung steht, um alle verbleibenden Arbeiten abzuschließen.

Nachdem das Erdkabel repariert worden ist, informiert die *Stromversorgung* über die Wiederherstellung der vollen Leistungsfähigkeit. Daraufhin starten die *HPCs* ihre unterbrochenen Arbeiten wieder.

Fall 3: Evakuierung bei Feueralarm

Ein Feuer bricht aus, es wird Alarm gegeben. Dieser ist als *Apoptosebefehl* an das ganze Rechenzentrum zu werten. Die Zellen schicken nun an andere Rechenzentren im Grid eine *Apoptosemeldung* mit Bitte um Hilfe. Haben jene Kapazitäten frei, antworten sie den Zellen und übernehmen deren Aufgaben. Die Rechner melden sich beim *Indexserver* ab und fahren sich herunter. Sind alle Computer ausgeschaltet, sendet der *Indexserver* einen *Apoptosebefehl* an die *Stromversorgung*, bevor auch er sich beendet. Die *Stromversorgung* schaltet sich zuletzt ab, um die mittlerweile eingetroffene Feuerwehr nicht zu gefährden.

7.3.3. Auswertung des Szenarios

Das Beispielszenario *Apoptose beim Grid Computing* (7.3) kann nicht weiter vereinfacht modelliert werden, woraus die in Tabelle 7.8 angegebenen sechs Zellen folgen. Der *Alarmgeber* ist ein einfacher boolescher Wert, der über TCP abgefragt werden kann und daher nicht als eigene Zelle ausgeführt sein muss. Die Versuchsanordnung ist abstrakt betrachtet äquivalent zu dem zuvor beschriebenen Aufbau des *„Powergrid“* in Abschnitt 7.2. Sie unterscheidet sich lediglich in den XML-Dateien und der Klasse, die das Testsystem mit Werten versorgt und so die Umgebung simuliert. Da der Versuchsablauf den eben vorgestellten Fällen entspricht, werden lediglich die Konsequenzen erläutert, die ohne die Apoptose entstehen könnten.

Zelle	Mission
Indexserver	Verwaltet Zellen und Aufträge, Koordiniert das Grid
HPC 1	Benötigt WAN und Strom. Partner: HPC 2, HPC 3
HPC 2	Benötigt WAN und Strom. Partner: HPC 1, HPC 3
HPC 3	Benötigt WAN und Strom. Partner: HPC 1, HPC 2
Firewall	Analysiert Datenverkehr, meldet potentielle Angriffe
Stromversorgung	Liefert (Not)Strom, meldet Kapazität

Tabelle 7.8.: Die modellierten Zellen im Szenario *Computergrid*

Aus der Beschreibung des ersten Falls kann entnommen werden, dass ohne Apoptose ein zentrales Element die von einem Virus befallene Maschine isolieren muss, um die Störung des Systems zu unterbinden. Doch dies gestaltet sich als schwierig, wenn die den Fehler erkennende Firewall sich in einer anderen Domäne als der infizierte Computer befindet, da sie diesem keine Befehle erteilen kann. So gedenken einige Internetprovider, Kundenzugänge zu kappen, wenn deren Computer Symptome einer Infektion aufweisen und beispielsweise als Spamschleudern missbraucht werden. Dies soll die Besitzer zu Gegenmaßnahmen zwingen und den Rest des Internets schützen. Die Zulässigkeit einer derartigen Maßnahme ist allerdings fraglich, weshalb ein dezentraler Ansatz wie die Apoptose eine rechtlich unbedenklichere Lösung sein könnte.

Der zweite Fall greift das Thema der Stromknappheit aus Abschnitt 7.2 wieder auf, die Ergebnisse sind demnach ähnlich. Während dank Apoptose wenigsten einige Jobs fertiggestellt und die restlichen kontrolliert beendet werden, würde ohne ein apoptotisches Verhalten vermutlich bis zum kompletten Stromausfall gerechnet

werden. Deshalb fahren viele aktuelle Systeme bei Notstromversorgung automatisch herunter, um den Erhalt der Daten zu sichern. Hier wird die Apoptose also bereits angewendet, ohne sie so zu nennen.

Auch die Daten und Aufgaben der Maschinen sollten bei einem Alarm evakuiert werden, wie der dritte Fall zeigt. Oft bleibt dazu keine Zeit oder ein zentrales Element versagt, so dass ein Totalverlust erfolgt. Wenn sich die Komponenten untereinander absprechen und die Missionen mit den besten Fluchtchancen und der höchsten Priorität vorlassen, steigt die Wahrscheinlichkeit, dass wenigstens die wichtigsten Daten in Sicherheit gebracht und die dringendsten Jobs weitergeführt werden können.

7.4. Evaluierung des Prototypen

Sowohl das Konzept als auch der Prototyp des Frameworks sind anhand der *Anforderungen* (3.4) entwickelt worden. Sie erfüllen diese somit, wie die Tabelle 7.9 zeigt.

	gestellte *Anforderungen* (3.4)					
Framework zur Apoptose	**1. Anforderung:** Informationen sammeln	**2. Anforderung:** Wissen aus den Informationen ableiten	**3. Anforderung:** Reaktionen bestimmen und durchführen	**4. Anforderung:** Mit anderen Zellen kommunizieren	**5. Anforderung:** Verzeichnis- und Protokollplattform	**6. Anforderung:** Eine Zelle kennt die Mission
Konzept (Kapitel 6)	✓	✓	✓	✓	✓	✓
Prototyp (Abschnitt 7.1)	✓	(✓)	✓	✓	(✓)	(✓)

Tabelle 7.9.: Anforderungsanalyse (3.4) von Konzept (6) und Prototyp (7.1)

Allerdings weist der Prototyp einige Einschränkungen auf, was durch (✓) angedeutet ist. So fehlen die Implementierungen aus den Bereichen der künstlichen Intelligenz und der *Knowledge Discovery in Databases*, die dem Framework die Fähigkeit des selbstständigen Lernens und damit der automatischen Adaption auf veränderte Umstände verleihen würden. Dennoch sind diese Funktionalitäten bei der Entwicklung des Frameworks berücksichtigt worden, weshalb sie leicht eingebaut werden können. Dies geschieht am besten unter Verwendung der Basisklassen `Modifier` und `Merger`. Zu komplexe Algorithmen verschlechtern allerdings die Laufzeit und den Speicherverbrauch, weshalb diese für Mikrocontroller eher ungeeignet sind.

Die Daten, die für eine Entscheidung herangezogen werden, können vom Prototypen nur über TCP-Sockets abgefragt werden, was für die Beispielszenarien ausreichend ist. Das Abfragen von Informationen von einem Betriebssystem oder einer Middleware ist zu systemspezifisch, um in einem allgemein gehaltenen Prototypen angewendet zu werden. Die kann mit einer entsprechenden Erweiterung von `Collector` nachgerüstet werden.

Dasselbe gilt auch für die möglichen Reaktionen. Will man direkt auf einen Hypervisor einwirken oder einen Computer via ACPI steuern, so muss `Evaluator` erweitert werden. Da dies teilweise native API-Funktionen voraussetzt, beschränkt sich der Prototyp auf das Versenden geeigneter Mitteilungen über TCP. Für die Simulation der Beispielszenarien reicht bereits eine entsprechende Ausgabe auf der Kommandozeile, um über die Entscheidungen des Frameworks unterrichtet zu sein.

Die Kommunikation zwischen den Zellen ist über die TCP-Sockets und die XML-Repräsentation der *Data*-Objekte vollständig gegeben. Nachrichten können beliebig definiert werden, es sollte allerdings systemweit einheitlich geschehen, damit die Zellen sich gegenseitig verstehen. Besonders der Einsatz von *Unit*-

Definitionen bietet eine gute Basis für eindeutige Typen von Nachrichten und Signalen, was zur Verständigung der Zellen untereinander beiträgt.

Durch die Anbindung eines Datenbankservers ist das Logging leicht durchzuführen. Auch die Abstraktion der Verbindungen durch die Connectoren hilft, neue Logs anzulegen und zu verwenden. Mit den selben Techniken kann auch ein Zellindex, wie in Unterunterabschnitt 6.2.5 beschrieben, realisiert werden. Da die Beispielszenarien aufgrund ihrer stark beschränkten Größe weitgehend statisch sind, wurde auf eine Implementierung der Lokalisierungsfunktion im Prototypen verzichtet. Je nach gewünschter Indexstruktur findet sich ein passender Ansatz, wie zum Beispiel *Distributed Hash Tables*. Die benötigten Techniken sind bereits in unterschiedlichsten Formen im Einsatz und müssen bei Bedarf für das Framework angepasst werden. Dies ist aber nicht Teil dieser Arbeit.

Leider ist es noch nicht möglich, einem Computer eine Mission verständlich zu machen, da es hierfür einer künstlichen Intelligenz bedarf. Auch das automatische Anpassen von allgemein gehaltenen Missionen an eine konkrete Maschine ist alles andere als trivial. Da bei den beschriebenen Beispielszenarien die Aufgabe einer Komponenten designinhärent ist, also ein Transformator immer die selbe Bestimmung hat, wurden die Missionsbeschreibungen für den Prototypen von Hand erstellt. Die Missionen beschränken sich dabei darauf, das System regelbasiert bei Teilausfällen in vorgegeben Bahnen zu halten.

8. Zusammenfassung und Ausblick

Der Mensch hat lange Zeit versucht, die Natur mit technischen Hilfsmitteln zu übertreffen. Doch umso mehr geforscht wurde, desto klarer wurden die Wissensdefizite bei der Lösung von komplexen Aufgaben. Gerade wenn verteilte Strukturen mit unzähligen selbstständigen, aber dennoch eng zusammenarbeitenden Zellen vorliegen, scheitert unsere Technik oft an Problemen, die natürliche Organismen schon seit Millionen oder gar Milliarden Jahren gelöst haben. Daher wird heutzutage versucht, von der Natur zu lernen.

So ist auch das in dieser Arbeit besprochene Konzept der Apoptose ein natürliches Verfahren, das sich in vielen Situationen im technischen Umfeld nutzen lässt, um die Systeme bei der Ausführung ihrer Mission stabiler zu machen. Hierfür wurden einige Szenarien beschrieben, die nicht nur dem informationstechnischen Alltag entnommen wurden (2). Die aus diesen Beispielen abgeleiteten Anforderungen für ein Framework (3), das helfen soll, die schädlichen Auswirkungen der erkannten Fehlerquellen zu minimieren, werden von den vorgestellten Methoden zur Gefahrenabwehr nur teilweise umgesetzt 4.

Daher wurde Apoptose als ein weitgehend neues Konzept vorgestellt, das durch das selbstständige Abschalten von Komponenten eines verteilten Systems, dessen Mission vor negativen Einflüssen schützen soll. Während in der Natur chemische Botenstoffe selbst zerstörende Reaktionen in einer Zelle auslösen, werden in dem vorgestellten Konzept eines Frameworks, das einen vergleichbaren Prozess bei technischen Ressourcen in Gang setzen können soll, Nachrichten zwischen den Zellen ausgetauscht (6). Für die hierbei auftretenden Schwierigkeiten der Lokalisierung von Nachbarzellen wurden mögliche Lösungen präsentiert, die auf bestehenden und weit verbreiteten Technologien basieren, was die Umsetzung einfach gestalten soll.

Um das Konzept des Frameworks zur Apoptose genauer untersuchen zu können, wurden zwei recht unterschiedliche Szenarien aus Kapitel 2 aufgegriffen: *Apoptose beim Grid Computing* (7.3) und *Apoptose im Smart Grid* (7.2). Für beide wurde eine minimale Testumgebung spezifiziert, in der der Einsatz von Apoptose mit dem zuvor beschriebenen Framework simuliert wurde. Das Framework wurde in einer vereinfachten Form mit Java implementiert. Dieser Prototyp kann generisch mit XML-Konfigurationsdateien parametrisiert werden, wodurch eine einfache Eingabe der Testszenarien erfolgen konnte. Damit erfüllt er alle Anforderungen, wie die Evaluierung in Abschnitt 7.4 zeigt.

8.1. Schwierigkeiten bei Konzeption und Umsetzung

Eine konzeptionelle Entscheidung bei der Implementierung war die Abstraktion der Verbindungen gegenüber den Aktivitäten durch Connectoren. Diese Kapselung für alle Aktivitäten generisch umzusetzen, gestaltet sich allerdings als schwierig. Wenn eine Aktivität den Auftrag hat, Daten von einer Quelle einzusammeln und in das Framework zu überführen, ist es ihre Aufgabe, über die Formatierung und Bedeutung der Informationen Bescheid zu wissen. Daher erscheint es sinnvoll, die Abfrage von Daten in der passenden Aktivität durchzuführen. Dies kann durch Parametrisierung teilweise generalisiert implementiert werden, zum Beispiel durch die Verwendung eines vorgegebene regulären Ausdrucks.

Da die Datenquellen nicht lokal sein müssen, können einige Aktivitäten Verbindungen zu einer von ihnen benötigten Ressource aufbauen. Dies jedoch sollte eigentlich weitgehend transparent durch die Connectoren erfolgen. Jetzt gibt es zum einen die Möglichkeit, die Connectoren sehr schlank und generisch zu gestalten, was die Schnittstelle zwischen ihnen und den Aktivitäten verkompliziert, oder zum anderen die Intelligenz der Auswertung in den jeweiligen Connector zu verlagern, was aber eine Reihe hochspezialisierter Connectoren zur Folge hat. Im Gegenzug hätten die anfragenden Aktivitäten kaum eine Berechtigung, da die ganze Arbeit der Datenerfassung und Informationsauswertung von den Connectoren erledigt würde.

Analog verhält es sich mit Aktivitäten, die eine Reaktion auslösen sollen. Wenn dies nicht lokal, sondern

entfernt erfolgt, müssen sie sich wiederum der Connectoren bedienen. Dies kann aber je nach Komplexität der Nachricht die bewusst einfach gehaltene Schnittstelle zu den Connectoren überfordern, was in einer stärkeren Unübersichtlichkeit enden würde.

Daher wurde die Abstraktion der Verbindungen im Prototypen nur teilweise umgesetzt. So sind ständige Verbindungen, wie die Anbindung einer Datenbank oder das Lauschen auf einem Port, in Connectoren ausgelagert, um nicht unnötig viele redundante Ressourcen durch die Aktivitäten zu binden. Geht die Verbindung allerdings von den Aktivitäten selbst aus und hat nur ein Ziel, so ist dies in der entsprechenden Activity direkt umgesetzt. Dies gilt zum Beispiel für das Polling von Daten über das Netz. Für solche Anwendungen wäre oft UDP die bessere Wahl, doch um den Prototypen simpel zu halten, wurden nur TCP-Sockets implementiert.

8.2. Vor- und Nachteile von Apoptose

Probleme können auftreten, wenn die Sicherheit eines verteilten Systems kompromittiert wurde. In der Natur finden wir Viren, die die befallene Zelle dazu zwingen, andere Zellen, vor allem das Immunsystem, zur Apoptose anzuregen, um der eigenen Vernichtung zu entgehen. Einen vergleichbaren Vorgang sehen wir bei Computerviren, die die Schutzmaßnahmen und Virenscanner auf einem System deaktivieren, um ungestört wüten zu können. Gefälschte Befehle und Nachrichten könnten in einem verteilten System dazu führen, dass die Apoptose gegen die eigenen Interessen eingesetzt wird. Ein Schutz vor Missbrauch ist nur mit den bereits verwirklichten Maßnahmen, wie verschlüsselter Kommunikation und signierter Nachrichten, möglich. Besonders beim grenzübergreifenden Einsatz von Apoptose ist ein gemeinsamer Sicherheitsstandard unabdingbar, um das Vertrauen in die anderen Parteien in einem Grid zu rechtfertigen.

Andererseits kann Apoptose, wie in der Arbeit anhand mehrerer Szenarien gezeigt worden ist, für die Erhöhung der Sicherheit sorgen, indem sich Komponenten bei einer Infektion selbst terminieren. Bei kritischen Systemen kann es ratsam sein, das Framework in einem gesonderten und extra geschützten Bereich zu betreiben, um die Handlungsfähigkeit auch bei kompromittierten oder nicht mehr reagierenden Komponenten zu erhalten.

Die ideale Zelle in einem Grid ist somit rücksichtsvoll und bereit zur Selbstaufopferung, um die Mission des Gesamtsystems zu schützen. Dies könnten *Bullies*, also Zellen, die sich nicht an den gemeinsamen Kodex halten, zu ihrem Vorteil ausnutzen. Sie geben sich einfach selbst die höchste Priorität und schalten ihre Konkurrenten mit Aufforderungen zur Apoptose aus, ohne allerdings selbst auf solche Befehle zu reagieren. In der IT helfen da die altbekannten Mittel der Sicherheitstechnik, wie Kryptographie, Zertifikate und Rechtemanagement.

Durch den dezentralen Ansatz, dem die Apoptose folgt, und dem dadurch vermiedenen Flaschenhals eines einzelnen Kontrollorgans, werden *single Points of Failure* und Skalierungsprobleme umgangen. Weiterhin werden der Schutz der Daten und der Privatsphäre gefördert, da jede Zelle für sich entscheidet, womit das Herausgeben von sensiblen Informationen an das Netz entfällt. Dieser Punkt ist besonders bei Smart Metern im Stromnetz interessant, da so die Entstehung von gläsernen Kunden vermieden werden kann.

All dies erkauft man sich mit dem Zwang zum Einsatz von recht intelligenten Zellen, was deren Kosten erhöht und die Konstruktion sowie die Programmierung erschwert. Doch durch das effiziente Design des Frameworks sollten diese Faktoren recht gering ausfallen.

8.3. Weiterführende Arbeiten

Es konnte gezeigt werden, dass sich der Einsatz von Apoptose in einem verteilten System durchaus lohnen kann. Doch konnte diese Arbeit nur einen kleinen Einblick in die Möglichkeiten, Risiken und Schwierigkeiten geben, die durch die Nutzung von Apoptose entstehen können. So wurden die Bereiche Sicherheit, Anwenderfreundlichkeit und Methoden der Künstlichen Intelligenz weitgehend außen vorgelassen.

So ist das Framework vollkommen unabhängig von einer späteren Nutzungsumgebung oder Plattform konzipiert worden. Daher fehlen Module, die die Anbindung an reale Systeme, wie zum Beispiel an das *Globus Tool Kit*, ermöglichen. Auch eine komfortable grafische Oberfläche, mit der sich Missionen beschreiben und an die

Zellen ausliefern lassen, wäre eine hilfreiche Ergänzung. Falls bereits bestehende Definitionen der Aufgabe eines Grids vorliegen, wäre eine automatische Konvertierung und Verteilung angenehm.

Dies ließe sich mit Abstraktionsschichten, die die jeweiligen Konkretisierungen der Aufgabe transparent gestalten, erreichen. Da die Abläufe im Framework mittels XML gesteuert werden, bietet sich der Einsatz von Konvertierungsregeln in XSLT an. Diese sollten bereits von den einzelnen Komponenten mitgeliefert werden, so dass sich die Zellen die für sie relevanten Einstellungen aus der abstrakten Missionsbeschreibung extrahieren können.

Der nächste Schritt wäre das Aufgreifen von Techniken aus dem autonomen Rechnen und der Forschung zur künstlichen Intelligenz, um selbst lernende Zellen zu schaffen, die sich an neue Bedingungen im verteilten System anpassen können, wodurch die aufwendige Beschreibung der Mission stark vereinfacht würde. Auch die *Knowledge Discovery in Databases* kann im Framework mit geeigneten Aktivitäten eingesetzt werden. So wurde bei der Konzeption bereits berücksichtigt, dass die Zellen in einen Trainingsmodus versetzt werden können, in dem das Weiterleiten von Daten zwischen den Aktivitäten unterbunden werden kann, um Reaktionen auf Trainigsdaten zu verhindern.

Das ist auch hilfreich bei einer vorherigen Simulation der Auswirkungen von der Apoptose einer Zelle auf des Gesamtsystem. So könnte vor einer Reaktion überprüft werden, ob sich die Systemgesundheit durch die Terminierung einer Zelle verbessern lässt, oder ob die Nebenwirkungen zu gefährlich sind. Allerdings dürfte sich eine derartige Simulation als äußerst schwierig gestalten, da eine Zelle nie einen ausreichenden Überblick über die Mission hat, denn das widerspräche dem Konzept eines verteilten Systems mit möglichst selbst verwaltenden Komponenten. Auch sind die meisten Grids viel zu groß und oft auch verschiedenen Administrationsdomänen angehörig, was die Beschaffung der benötigten Daten unmöglich macht. Dies gilt umso mehr, als die Reaktion auf eine Gefahr kurzfristig erfolgen muss, um noch eine entsprechende Wirkung entfalten zu können. Eine umfangreiche Vorher-Nachher-Analyse ist kaum schnell genug durchzuführen.

Ein weiteres interessantes Konzept für die Missionssteuerung in einem Computergrid stellt das *Urgent Computing* dar [BBNC 07]. Hierbei geht es um die Bevorzugung von Jobs auf einer Maschine, wenn diese eine höhere Priorität haben. Auch bei der Konzeption des Frameworks zur Apoptose wurden Prioritäten berücksichtigt. Daher könnte Apoptose eine nützliche Ergänzung beim *Urgent Computing* darstellen, indem sich eine Zelle selbstständig zurückzieht, um einer wichtigeren Mission Platz zu machen.

A. Anhang

A.1. XML-Listings

A.1.1. Kleines Beispiel für ein Framework

```
<?xml version="1.0" encoding="UTF-8"?>
<framework name="servercell">
    <settings>
        <verbose>8</verbose>
    </settings>
    <units>
        <Unit name="boolean">
            <typicallow>0</typicallow>
            <typicalhigh>1</typicalhigh>
        </Unit>
        <Unit name="Port">
            <typicallow>0</typicallow>
            <typicalhigh>65535</typicalhigh>
        </Unit>
    </units>
    <activities>
        <CollectFromSocket name="firesensor">
            <host>10.0.23.42</host>
            <port>4711</port>
            <request>getdata</request>
            <unit>boolean</unit>
            <inputlow>0</inputlow>
            <inputhigh>5</inputhigh>
            <outputlow>0</outputlow>
            <outputhigh>1000</outputhigh>
            <updateinterval>2000</updateinterval>
        </CollectFromSocket>
        <CollectFromSocket name="lan">
            <file>/proc/firwall</file>
            <filter>newport</filter>
            <unit>Port</unit>
            <inputlow>0</inputlow>
            <inputhigh>65535</inputhigh>
            <outputlow>0</outputlow>
            <outputhigh>65535</outputhigh>
            <updateinterval>100</updateinterval>
        </CollectFromSocket>
        <IsInSet name="legalport">
            <listento>lan</listento>
            <weight>-1</weight>
            <element>80</element>
            <element>443</element>
        </IsInSet>
        <ALU name="or">
            <operator>or</operator>
            <listento>firesensor</listento>
```

```
            <listento>legalport</listento>
        </ALU>
        <SendData name="commit">
            <listento>or</listento>
            <triggerlow>0</triggerlow>
            <triggerhigh>2147483647</triggerhigh>
            <ontruemessage>apoptosenotification</ontruemessage>
            <ontrueaddress>mission/broker</ontrueaddress>
        </SendData>
        <Shutoff name="apoptose">
            <listento>or</listento>
            <triggerlow>0</triggerlow>
            <triggerhigh>2147483647</triggerhigh>
        </Shutoff>
    </activities>
</framework>
```

Listing A.1: Beispiel für eine Konfigurationsdatei des Frameworks aus Abschnitt 6.3

A.1.2. Aufbau einer XML-Konfigurationsdatei

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<framework name="Name_der_Zelle">
    <settings>
        <verbose>8</verbose>
        <interval_low>0</interval_low>
        <interval_high>100</interval_high>
        <!-- Hier können weitere globale Einstellungen stehen -->
    </settings>
    <connectors>
        <SQLClient name="Name_des_Collectors">
            <url>jdbc:mysql://example.com:3306/apoptosedb</url>
            <user>apoptose</user>
            <password>geheim</password>
            <sql>INSERT INTO log (cellname, comment) VALUES (?cellname, ?comment)</sql>
        </SQLClient>
        <!-- Hier können weitere Connectoren stehen -->
    </connectors>
    <units>
        <unit name="Volt">
            <typicallow>0</typicallow>
            <typicalhigh>5</typicalhigh>
        </unit>
        <!-- Hier können weitere Einheiten definiert werden -->
    </units>
    <activities>
        <Collector name="col">
            <startvalue>666</startvalue>
            <unit>Volt</unit>
            <inputlow>0</inputlow>
            <inputhigh>1023</inputhigh>
        </Collector>
        <Modifier name="mod">
            <listento>col</listento>
            <weight>-1000</weight>
        </Modifier>
        <Merger name="merge">
            <listento>col</listento>
            <weight>1000</weight>
            <listento>mod</listento>
            <weight>2000</weight>
```

```
        </Merger>
        <Evaluator name="eval">
            <listento>merge</listento>
            <triggerlow>50</triggerlow>
            <logger>sqllog</logger>
        </Evaluator>
        <!-- Hier können weitere Aktivitäten angegeben werden -->
    </activities>
</framework>
```

Listing A.2: Vollständig XML-Konfigurationsdatei von Unterabschnitt 7.1.3

A.1.3. XML-Dokumente des Beispielszenarios *Apoptose im Smart Grid* (7.2)

Alle XML-Dateien finden sich auf der *CD-ROM* (A.3) im Verzeichnis `Sourcen/powergrid`.

```
<?xml version="1.0" encoding="UTF-8"?>
<framework>
    <settings>
        <verbose>8</verbose>
    </settings>
    <connectors>
        <ConsoleLogger name="logger">
        </ConsoleLogger>
        <DummyNetwork name="szenarioconnector">
        </DummyNetwork>
        <XMLClientSocket name="connectorhsl1">
            <uri>tcp://localhost:42101</uri>
        </XMLClientSocket>
        <XMLClientSocket name="connectorhsl2">
            <uri>tcp://localhost:42102</uri>
        </XMLClientSocket>
        <XMLClientSocket name="connectorwindpark">
            <uri>tcp://localhost:42103</uri>
        </XMLClientSocket>
        <XMLClientSocket name="connectorkraftwerk">
            <uri>tcp://localhost:42104</uri>
        </XMLClientSocket>
        <XMLClientSocket name="connectorstadt">
            <uri>tcp://localhost:42105</uri>
        </XMLClientSocket>
        <XMLClientSocket name="connectorinfrastruktur">
            <uri>tcp://localhost:42106</uri>
        </XMLClientSocket>
        <XMLClientSocket name="connectorindustrie">
            <uri>tcp://localhost:42107</uri>
        </XMLClientSocket>
    </connectors>
    <units>
        <Unit name="MW">
            <typicallow>0</typicallow>
            <typicalhigh>100</typicalhigh>
        </Unit>
    </units>
</framework>
```

Listing A.3: gemeinsam.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<framework name="windpark">
    <connectors>
```

```
        <XMLServerSocket name="xmlserver3">
            <port>42103</port>
        </XMLServerSocket>
    </connectors>

    <activities>
        <RequestMessage name="produktionwindpark3">
            <listento>windpark</listento>
            <connector>szenarioconnector</connector>
            <unit>MW</unit>
            <inputlow>0</inputlow>
            <inputhigh>100</inputhigh>
            <updateinterval>5000</updateinterval>
        </RequestMessage>

        <SendData name="sendwindpark3">
            <listento>produktionwindpark3</listento>
            <triggerlow>0</triggerlow>
            <ontruemessage>produktionwindpark</ontruemessage>
            <ontrueconnector>connectorhslkraftwerk</ontrueconnector>
        </SendData>

        <CollectMessage name="produktionkraftwerk3">
            <connector>xmlserver3</connector>
            <listento>produktionkraftwerk</listento>
        </CollectMessage>
        <CollectMessage name="changeproduction3">
            <connector>xmlserver3</connector>
            <listento>changeproduction</listento>
        </CollectMessage>

        <ALU name="calcdiff3">
            <listento>changeproduction3</listento>
            <listento>produktionwindpark3</listento>
            <operator>sub</operator>
        </ALU>

        <Shutoff name="shutoff3">
            <listento>calcdiff3</listento>
            <triggerlow>-2000000000</triggerlow>
            <triggerhigh>0</triggerhigh>
            <connector>szenarioconnector</connector>
            <text>windpark</text>
            <command>set</command>
        </Shutoff>
        <Shutoff name="shutoff3l">
            <listento>calcdiff3</listento>
            <triggerlow>-2000000000</triggerlow>
            <triggerhigh>0</triggerhigh>
            <connector>logger</connector>
            <text>!!! Windpark reduziert seine Produktion</text>
        </Shutoff>
    </activities>
</framework>
```

Listing A.4: windpark.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<framework name="kraftwerk">
    <connectors>
        <XMLServerSocket name="xmlserver4">
```

```
            <port>42104</port>
        </XMLServerSocket>
    </connectors>

    <activities>
        <RequestMessage name="produktionkraftwerk4">
            <listento>kraftwerk</listento>
            <connector>szenarioconnector</connector>
            <unit>MW</unit>
            <inputlow>0</inputlow>
            <inputhigh>100</inputhigh>
            <updateinterval>5000</updateinterval>
        </RequestMessage>

        <SendData name="sendkraftwerk4">
            <listento>produktionkraftwerk4</listento>
            <triggerlow>0</triggerlow>
            <ontruemessage>produktionkraftwerk</ontruemessage>
            <ontrueconnector>connectorhslwindpark</ontrueconnector>
        </SendData>

        <CollectMessage name="produktionwindpark4">
            <connector>xmlserver4</connector>
            <listento>produktionwindpark</listento>
        </CollectMessage>
        <CollectMessage name="changeproduction4">
            <connector>xmlserver4</connector>
            <listento>changeproduction</listento>
        </CollectMessage>

        <ALU name="calcdiff4">
            <listento>changeproduction4</listento>
            <listento>produktionwindpark4</listento>
            <operator>sub</operator>
        </ALU>

        <Shutoff name="shutoff4">
            <listento>calcdiff4</listento>
            <triggerlow>0</triggerlow>
            <connector>szenarioconnector</connector>
            <text>kraftwerk</text>
            <command>set</command>
        </Shutoff>
        <Shutoff name="shutoff41">
            <listento>calcdiff4</listento>
            <triggerlow>0</triggerlow>
            <connector>logger</connector>
            <text>!!! Kraftwerk reduziert seine Produktion</text>
        </Shutoff>
    </activities>
</framework>
```

Listing A.5: kraftwerk.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<framework name="hsl1">
    <connectors>
        <XMLServerSocket name="xmlserver1">
            <port>42101</port>
        </XMLServerSocket>
    </connectors>
```

```
<activities>
    <RequestMessage name="prim1">
        <listento>hsl1prim</listento>
        <connector>szenarioconnector</connector>
        <unit>MW</unit>
        <inputlow>0</inputlow>
        <inputhigh>100</inputhigh>
        <updateinterval>5000</updateinterval>
    </RequestMessage>
    <RequestMessage name="sec1">
        <listento>hsl1sec</listento>
        <connector>szenarioconnector</connector>
        <unit>MW</unit>
        <inputlow>0</inputlow>
        <inputhigh>100</inputhigh>
        <updateinterval>5000</updateinterval>
    </RequestMessage>

    <ALU name="calcdiff1">
        <listento>prim1</listento>
        <listento>sec1</listento>
        <operator>sub</operator>
    </ALU>

    <Shutoff name="apoptosisprim1">
        <listento>calcdiff1</listento>
        <triggerlow>10000</triggerlow>
        <connector>szenarioconnector</connector>
        <command>set hsl1prim 0</command>
    </Shutoff>
    <Shutoff name="apoptosissec1">
        <listento>calcdiff1</listento>
        <triggerlow>10000</triggerlow>
        <connector>szenarioconnector</connector>
        <command>set hsl1sec 0</command>
    </Shutoff>
    <Shutoff name="apoptosis1">
        <listento>calcdiff1</listento>
        <triggerlow>10000</triggerlow>
        <connector>logger</connector>
        <text>!!! HSL1 wird abgeschaltet</text>
    </Shutoff>
    <SendData name="hsl1down">
        <listento>calcdiff1</listento>
        <triggerlow>10000</triggerlow>
        <ontruemessage>apoptosehsl1</ontruemessage>
        <ontrueconnector>connectorhsl2</ontrueconnector>
    </SendData>

    <!-- Reaktion auf Apoptose HSL2 -->
    <CollectMessage name="hsl2down1">
        <connector>xmlserver1</connector>
        <listento>apoptosehsl2</listento>
    </CollectMessage>
    <ALU name="calcfree1">
        <listento>calcdiff2</listento>
        <listento>hsl2down1</listento>
        <operator>sub</operator>
    </ALU>
    <SendData name="shrinkusage1">
```

```
            <listento>calcfree1</listento>
            <triggerlow>0</triggerlow>
            <onfalsemessage>shrinkusage</onfalsemessage>
            <ontrueconnector>connectorhslinfrastruktur</ontrueconnector>
            <ontrueconnector>connectorhslstadt</ontrueconnector>
            <ontrueconnector>connectorhslindustrie</ontrueconnector>
        </SendData>

        <MinimumMerge name="calclimit1">
            <listento>calcdiff1</listento>
            <listento>calcfree1</listento>
        </MinimumMerge>
        <SendData name="changeprodurction1">
            <listento>calclimit1</listento>
            <ontruemessage>changeprodurction</ontruemessage>
            <ontrueconnector>connectorhslwindpark</ontrueconnector>
            <ontrueconnector>connectorhslkraftwerk</ontrueconnector>
        </SendData>
    </activities>
</framework>
```

Listing A.6: hsl1.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<framework name="hsl2">
    <connectors>
        <XMLServerSocket name="xmlserver">
            <port>42102</port>
        </XMLServerSocket>
    </connectors>

    <activities>
        <RequestMessage name="prim2">
            <listento>hsl2prim</listento>
            <connector>szenarioconnector</connector>
            <unit>MW</unit>
            <inputlow>0</inputlow>
            <inputhigh>100</inputhigh>
            <updateinterval>5000</updateinterval>
        </RequestMessage>
        <RequestMessage name="sec2">
            <listento>hsl2sec</listento>
            <connector>szenarioconnector</connector>
            <unit>MW</unit>
            <inputlow>0</inputlow>
            <inputhigh>100</inputhigh>
            <updateinterval>5000</updateinterval>
        </RequestMessage>

        <ALU name="calcdiff2">
            <listento>prim2</listento>
            <listento>sec2</listento>
            <operator>sub</operator>
        </ALU>

        <Shutoff name="apoptosisprim2">
            <listento>calcdiff2</listento>
            <triggerlow>10000</triggerlow>
            <connector>szenarioconnector</connector>
            <command>set hsl2prim 0</command>
        </Shutoff>
```

```
        <Shutoff name="apoptosissec2">
            <listento>calcdiff2</listento>
            <triggerlow>10000</triggerlow>
            <connector>szenarioconnector</connector>
            <command>set hsl2sec 0</command>
        </Shutoff>
        <Shutoff name="apoptosis2">
            <listento>calcdiff2</listento>
            <triggerlow>10000</triggerlow>
            <connector>logger</connector>
            <text>!!! HSL2 wird abgeschaltet</text>
        </Shutoff>
        <SendData name="hsl2down">
            <listento>calcdiff2</listento>
            <triggerlow>10000</triggerlow>
            <ontruemessage>apoptosehsl2</ontruemessage>
            <ontrueconnector>connectorhsl1</ontrueconnector>
        </SendData>

        <!-- Reaktion auf Apoptose HSL2 -->
        <CollectMessage name="hsl1down2">
            <connector>xmlserver2</connector>
            <listento>apoptosehsl1</listento>
        </CollectMessage>
        <ALU name="calcfree2">
            <listento>calcdiff2</listento>
            <listento>hsl1down2</listento>
            <operator>sub</operator>
        </ALU>
        <SendData name="shrinkusage2">
            <listento>calcfree2</listento>
            <triggerlow>0</triggerlow>
            <onfalsemessage>shrinkusage</onfalsemessage>
            <ontrueconnector>connectorhslinfrastruktur</ontrueconnector>
            <ontrueconnector>connectorhslstadt</ontrueconnector>
            <ontrueconnector>connectorhslindustrie</ontrueconnector>
        </SendData>

        <MinimumMerge name="calclimit2">
            <listento>calcdiff2</listento>
            <listento>calcfree2</listento>
        </MinimumMerge>
        <SendData name="changeprodurction">
            <listento>calclimit2</listento>
            <ontruemessage>changeprodurction</ontruemessage>
            <ontrueconnector>connectorhslwindpark</ontrueconnector>
            <ontrueconnector>connectorhslkraftwerk</ontrueconnector>
        </SendData>
    </activities>
</framework>
```

Listing A.7: hsl2.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<framework name="stadt">
    <connectors>
        <XMLServerSocket name="xmlserver5">
            <port>42105</port>
        </XMLServerSocket>
    </connectors>
```

```
<activities>
    <RequestMessage name="bedarfstadthigh5">
        <listento>stadthigh</listento>
        <connector>szenarioconnector</connector>
        <unit>MW</unit>
        <inputlow>0</inputlow>
        <inputhigh>100</inputhigh>
        <updateinterval>5000</updateinterval>
    </RequestMessage>
    <RequestMessage name="bedarfstadtmedium5">
        <listento>stadtmedium</listento>
        <connector>szenarioconnector</connector>
        <unit>MW</unit>
        <inputlow>0</inputlow>
        <inputhigh>100</inputhigh>
        <updateinterval>5000</updateinterval>
    </RequestMessage>
    <RequestMessage name="bedarfstadtlow5">
        <listento>stadtlow</listento>
        <connector>szenarioconnector</connector>
        <unit>MW</unit>
        <inputlow>0</inputlow>
        <inputhigh>100</inputhigh>
        <updateinterval>5000</updateinterval>
    </RequestMessage>

    <SendData name="sendbedarfhigh5">
        <listento>bedarfstadthigh5</listento>
        <triggerlow>0</triggerlow>
        <ontruemessage>bedarfstadt</ontruemessage>
        <ontrueconnector>connectorhslinfrastruktur</ontrueconnector>
        <ontrueconnector>connectorhslindustrie</ontrueconnector>
    </SendData>
    <SendData name="sendbedarfmedium5">
        <listento>stadtpowermedium5</listento>
        <triggerlow>0</triggerlow>
        <ontruemessage>bedarfstadt</ontruemessage>
        <ontrueconnector>connectorhslinfrastruktur</ontrueconnector>
        <ontrueconnector>connectorhslindustrie</ontrueconnector>
    </SendData>
    <SendData name="sendbedarflow5">
        <listento>stadtpowerlow5</listento>
        <triggerlow>0</triggerlow>
        <ontruemessage>bedarfstadt</ontruemessage>
        <ontrueconnector>connectorhslinfrastruktur</ontrueconnector>
        <ontrueconnector>connectorhslindustrie</ontrueconnector>
    </SendData>

    <CollectMessage name="limit5">
        <connector>xmlserver5</connector>
        <listento>shrinkusage</listento>
    </CollectMessage>
    <CollectMessage name="bedarfinfrastrukturhigh5">
        <connector>xmlserver5</connector>
        <listento>bedarfinfrastrukturhigh</listento>
    </CollectMessage>
    <CollectMessage name="bedarfindustriemedium5">
        <connector>xmlserver5</connector>
        <listento>bedarfindustriemedium</listento>
    </CollectMessage>
    <CollectMessage name="bedarfindustrielow5">
```

```
            <connector>xmlserver5</connector>
            <listento>bedarfindustrielow</listento>
        </CollectMessage>

        <ALU name="calchighpriority5">
            <listento>limit5</listento>
            <listento>bedarfinfrastrukturhigh5</listento>
            <listento>bedarfstadthigh5</listento>
            <operator>sub</operator>
        </ALU>
        <ALU name="calcmediumpriority5">
            <listento>calchighpriority5</listento>
            <listento>bedarfstadtmedium5</listento>
            <listento>bedarfindustriemedium5</listento>
            <operator>sub</operator>
        </ALU>
        <ALU name="calclowpriority5">
            <listento>calcmediumpriority5</listento>
            <listento>bedarfstadtlow5</listento>
            <listento>bedarfindustrielow5</listento>
            <operator>sub</operator>
        </ALU>

        <Shutoff name="resethighpower5">
            <listento>calchighpriority5</listento>
            <weight>500</weight>
            <triggerlow>-2000000000</triggerlow>
            <triggerhigh>-1</triggerhigh>
            <connector>szenarioconnector</connector>
            <command>reset stadthigh</command>
        </Shutoff>
        <Shutoff name="resetmediumpower5">
            <listento>calcmediumpriority5</listento>
            <weight>500</weight>
            <triggerlow>-2000000000</triggerlow>
            <triggerhigh>-1</triggerhigh>
            <connector>szenarioconnector</connector>
            <command>reset stadtmedium</command>
        </Shutoff>
        <Shutoff name="resetlowpower5">
            <listento>calclowpriority5</listento>
            <weight>500</weight>
            <triggerlow>-2000000000</triggerlow>
            <triggerhigh>-1</triggerhigh>
            <connector>szenarioconnector</connector>
            <command>reset stadtlow</command>
        </Shutoff>
    </activities>
</framework>
```

Listing A.8: stadt.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<framework name="infrastruktur">
    <connectors>
        <XMLServerSocket name="xmlserver6">
            <port>42106</port>
        </XMLServerSocket>
    </connectors>

    <activities>
```

```
<RequestMessage name="bedarfinfrastrukturhigh6">
    <listento>infrastrukturhigh</listento>
    <connector>szenarioconnector</connector>
    <unit>MW</unit>
    <inputlow>0</inputlow>
    <inputhigh>100</inputhigh>
    <updateinterval>5000</updateinterval>
</RequestMessage>

<SendData name="sendbedarfhigh6">
    <listento>bedarfinfrastrukturhigh6</listento>
    <triggerlow>0</triggerlow>
    <ontruemessage>bedarfinfrastrukturhigh</ontruemessage>
    <ontrueconnector>connectorhslstadt</ontrueconnector>
    <ontrueconnector>connectorhslindustrie</ontrueconnector>
</SendData>

<CollectMessage name="limit6">
    <connector>xmlserver6</connector>
    <listento>shrinkusage</listento>
</CollectMessage>
<CollectMessage name="bedarfindustriemedium6">
    <connector>xmlserver6</connector>
    <listento>bedarfindustriemedium</listento>
</CollectMessage>
<CollectMessage name="bedarfindustrielow6">
    <connector>xmlserver6</connector>
    <listento>bedarfindustrielow</listento>
</CollectMessage>
<CollectMessage name="bedarfstadthigh6">
    <connector>xmlserver6</connector>
    <listento>bedarfstadthigh</listento>
</CollectMessage>
<CollectMessage name="bedarfstadtmedium6">
    <connector>xmlserver6</connector>
    <listento>bedarfstadtmedium</listento>
</CollectMessage>
<CollectMessage name="bedarfstadtlow6">
    <connector>xmlserver6</connector>
    <listento>bedarfstadtlow</listento>
</CollectMessage>

<ALU name="calchighpriority6">
    <listento>limit6</listento>
    <listento>bedarfinfrastrukturhigh6</listento>
    <listento>bedarfstadthigh6</listento>
    <operator>sub</operator>
</ALU>
<ALU name="calcmediumpriority6">
    <listento>calchighpriority6</listento>
    <listento>bedarfstadtmedium6</listento>
    <listento>bedarfindustriemedium6</listento>
    <operator>sub</operator>
</ALU>
<ALU name="calclowpriority6">
    <listento>calcmediumpriority6</listento>
    <listento>bedarfstadtlow6</listento>
    <listento>bedarfindustrielow6</listento>
    <operator>sub</operator>
</ALU>
```

```
        <Shutoff name="resetmediumpower6">
            <listento>calcmediumpriority6</listento>
            <weight>500</weight>
            <triggerlow>-2000000000</triggerlow>
            <triggerhigh>-1</triggerhigh>
            <connector>szenarioconnector</connector>
            <command>reset industriemedium</command>
        </Shutoff>
        <Shutoff name="resetlowpower6">
            <listento>calclowpriority6</listento>
            <weight>500</weight>
            <triggerlow>-2000000000</triggerlow>
            <triggerhigh>-1</triggerhigh>
            <connector>szenarioconnector</connector>
            <command>reset industrielow</command>
        </Shutoff>
    </activities>
</framework>
```

Listing A.9: infrastruktur.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<framework name="industrie">
    <connectors>
        <XMLServerSocket name="xmlserver7">
            <port>42107</port>
        </XMLServerSocket>
    </connectors>

    <activities>
        <RequestMessage name="bedarfindustriemedium7">
            <listento>industriemedium</listento>
            <connector>szenarioconnector</connector>
            <unit>MW</unit>
            <inputlow>0</inputlow>
            <inputhigh>100</inputhigh>
            <updateinterval>5000</updateinterval>
        </RequestMessage>
        <RequestMessage name="bedarfindustrielow7">
            <listento>industriemedium</listento>
            <connector>szenarioconnector</connector>
            <unit>MW</unit>
            <inputlow>0</inputlow>
            <inputhigh>100</inputhigh>
            <updateinterval>5000</updateinterval>
        </RequestMessage>

        <SendData name="sendbedarfmedium7">
            <listento>bedarfindustriemedium7</listento>
            <triggerlow>0</triggerlow>
            <ontruemessage>bedarfindustrie</ontruemessage>
            <ontrueconnector>connectorhslinfrastruktur</ontrueconnector>
            <ontrueconnector>connectorhslstadt</ontrueconnector>
        </SendData>
        <SendData name="sendbedarflow7">
            <listento>bedarfindustrielow7</listento>
            <triggerlow>0</triggerlow>
            <ontruemessage>bedarfindustrie</ontruemessage>
            <ontrueconnector>connectorhslinfrastruktur</ontrueconnector>
            <ontrueconnector>connectorhslstadt</ontrueconnector>
        </SendData>
```

```
        <CollectMessage name="limit7">
            <connector>xmlserver7</connector>
            <listento>shrinkusage</listento>
        </CollectMessage>
        <CollectMessage name="bedarfinfrastrukturhigh7">
            <connector>xmlserver7</connector>
            <listento>bedarfinfrastrukturhigh</listento>
        </CollectMessage>
        <CollectMessage name="bedarfstadthigh7">
            <connector>xmlserver7</connector>
            <listento>bedarfstadthigh</listento>
        </CollectMessage>
        <CollectMessage name="bedarfstadtmedium7">
            <connector>xmlserver7</connector>
            <listento>bedarfstadtmedium</listento>
        </CollectMessage>
        <CollectMessage name="bedarfstadtlow7">
            <connector>xmlserver7</connector>
            <listento>bedarfstadtlow</listento>
        </CollectMessage>

        <ALU name="calchighpriority7">
            <listento>limit7</listento>
            <listento>bedarfinfrastrukturhigh7</listento>
            <listento>bedarfstadthigh7</listento>
            <operator>sub</operator>
        </ALU>
        <ALU name="calcmediumpriority7">
            <listento>calchighpriority7</listento>
            <listento>bedarfstadtmedium7</listento>
            <listento>bedarfindustriemedium7</listento>
            <operator>sub</operator>
        </ALU>
        <ALU name="calclowpriority7">
            <listento>calcmediumpriority7</listento>
            <listento>bedarfstadtlow7</listento>
            <listento>bedarfindustrielow7</listento>
            <operator>sub</operator>
        </ALU>

        <Shutoff name="resetmediumpower7">
            <listento>calcmediumpriority7</listento>
            <weight>500</weight>
            <triggerlow>-2000000000</triggerlow>
            <triggerhigh>-1</triggerhigh>
            <connector>szenarioconnector</connector>
            <command>reset industriemedium</command>
        </Shutoff>
        <Shutoff name="resetlowpower7">
            <listento>calclowpriority7</listento>
            <weight>500</weight>
            <triggerlow>-2000000000</triggerlow>
            <triggerhigh>-1</triggerhigh>
            <connector>szenarioconnector</connector>
            <command>reset industrielow</command>
        </Shutoff>
    </activities>
</framework>
```

Listing A.10: industrie.xml

A.2. SQL-Listings

```
CREATE TABLE `log` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `timestamp` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  `cellname` varchar(255) DEFAULT NULL,
  `name` varchar(45) DEFAULT NULL,
  `state` varchar(20) DEFAULT NULL,
  `weight` int(11) DEFAULT NULL,
  `inname` varchar(45) DEFAULT NULL,
  `invalue` int(11) DEFAULT NULL,
  `inunit` varchar(45) DEFAULT NULL,
  `insender` varchar(45) DEFAULT NULL,
  `intimestamp` timestamp NULL DEFAULT NULL,
  `outname` varchar(45) DEFAULT NULL,
  `outvalue` int(11) DEFAULT NULL,
  `outunit` varchar(45) DEFAULT NULL,
  `outsender` varchar(45) DEFAULT NULL,
  `outtimestamp` timestamp NULL DEFAULT NULL,
  `comment` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`id`)
)
```

Listing A.11: MySQL Statement, um die Logtabelle anzulegen

A.3. CD-ROM

Der Diplomarbeit liegt auch eine CD-ROM bei, die den praktischen Teil sowie die LaTeX-Quellen dieser Arbeit enthält. Das Framework besteht aus dem Javacode mit zugehörigem Eclipseprojekt, dem Javadoc und den XML-Dateien der Beispielszenarien. Auch alle in dieser Arbeit verwendeten Grafiken und deren Projektdateien sind angefügt.

Abbildungsverzeichnis

2.1. Grundsätzliche Fehlerklassen aus [AcLR 01] . . . 11
2.2. Arten des Versagens aus [AcLR 01] . . . 13

3.1. Jede **Komponente** ist genau einer **Zelle** zugeordnet . . . 16
3.2. Jede **Zelle** ist genau einer **Mission** zugeordnet . . . 17
3.3. Definieren einer Mission . . . 19
3.4. Eine Mission wird gestartet und ausgeführt . . . 21
3.5. Eine Mission wird beendet . . . 23

6.1. Signalverarbeitung im Framework . . . 55
6.2. Vereinfachtes Klassendiagramm der Aktivitäten . . . 57
6.3. Vereinfachte Ansicht der Basisklasse *Activity* . . . 57
6.4. Sammeln von Informationen . . . 58
6.5. Die Klasse *Collector* . . . 59
6.6. Die Klassen *Data* und *Unit* . . . 59
6.7. Die Klasse *Modifier* . . . 60
6.8. Modifikation der Information . . . 60
6.9. Die Klasse *Merger* . . . 61
6.10. Zusammenführen von mehreren Informationen . . . 61
6.11. Auswerten der Informationen mit entsprechender Reaktion . . . 61
6.12. Die Klasse *Evaluator* . . . 62
6.13. Broadcast der Apoptose von Zelle A . . . 65
6.14. Zentrales Ablegen der Meldung der Apoptose von Zelle A . . . 66
6.15. Broker reagiert auf die Apoptose von Zelle A . . . 67
6.16. Eine nicht mehr korrekt funktionierende Zelle wird zur Apoptose aufgefordert . . . 67
6.17. Eine Zelle verschickt unangebrachte Aufforderungen zur Apoptose . . . 68
6.18. Hierarchie bei der Konvertierung der Missionsdefinitionen . . . 72
6.19. Datenmodell der Beschreibung einer Mission . . . 73
6.20. Ein minimalistisches Beispiel für ein Framework . . . 74

7.1. Die Klasse `Connector` . . . 79
7.2. Ein Beispielszenario für Apoptose im Stromnetz . . . 84
7.3. Ein Beispielszenario für Apoptose im Computergrid . . . 89

Tabellenverzeichnis

3.1. Anforderungsanalyse der Szenarien aus Kapitel 2 27

4.1. Auswertung bestehender Konzepte bezüglich den Anforderungen aus 3.4 40

6.1. Inhalt eines Keep-Alive-Pakets 63
6.2. Inhalt einer Apoptosemitteilung 64
6.3. Inhalt eines Befehls zur Apoptose 68
6.4. Beispiel für eine Log-Tabelle mit SQL für die Aktivitäten (A.11) 69

7.1. Defaultwerte in `Main` 76
7.2. Klassenübersicht vom Paket `framework.activities` 78
7.3. Der Enumerator `Activity.State` 79
7.4. Klassenübersicht vom Paket `framework.connectors` 79
7.5. Klassen jenseits des Frameworks 80
7.6. Die modellierten Zellen im Szenario *Powergrid* 86
7.7. Die Zellen mit Startparametern und Port für XML-Nachrichten im Szenario *Powergrid* 87
7.8. Die modellierten Zellen im Szenario *Computergrid* 90
7.9. Anforderungsanalyse (3.4) von Konzept (6) und Prototyp (7.1) 91

Listings

7.1. Grundgerüst einer Konfigurationsdatei 80
7.2. Ein `Merger` mit zwei Eingängen 81
7.3. Ein `Data`-Objekt kodiert in XML 81

A.1. Beispiel für eine Konfigurationsdatei des Frameworks aus Abschnitt 6.3 97
A.2. Vollständig XML-Konfigurationsdatei von Unterabschnitt 7.1.3 98
A.3. gemeinsam.xml 99
A.4. windpark.xml 99
A.5. kraftwerk.xml 100
A.6. hsl1.xml 101
A.7. hsl2.xml 103
A.8. stadt.xml 104
A.9. infrastruktur.xml 106
A.10. industrie.xml 108
A.11. MySQL Statement, um die Logtabelle anzulegen 110

Literaturverzeichnis

[AcLR 01] AVIZIENIS, ALGIRDAS, JEAN CLAUDE LAPRIE und BRIAN RANDELL: *Fundamental Concepts of Dependability*, 2001.

[AFG+ 09] ARMBRUST, MICHAEL, ARMANDO FOX, REAN GRIFFITH, ANTHONY D. JOSEPH, RANDY H. KATZ, ANDREW KONWINSKI, GUNHO LEE, DAVID A. PATTERSON, ARIEL RABKIN und MATEI ZAHARIA: *Above the Clouds: A Berkeley View of Cloud Computing*. Technischer Bericht, UC Berkeley Reliable Adaptive Distributed Systems Laboratory, 2009.

[Alth 11] ALTHERR, FLORIAN: *servergate - Polizei beschlagnahmt Piratenpartei-Server*. Netzpolitik, 2011.

[Ande 01] ANDERSON, ROSS: *Security Engineering: A Guide to Building Dependable Distributed Systems*. ISBN 9780471389224, 2001.

[ASSC 02] AKYILDIZ, LAN F., WELLJAN SU, YOGESH SANKARASUBRAMANIAM und ERDAL CAYIRCI: *A Survey on Sensor Networks*, 2002.

[BBNC 07] BECKMAN, PETE, IVAN BESCHASTNIKH, SUMAN NADELLA und NICK TREBON C: *Building an Infrastructure for Urgent Computing. High Performance Computing and Grids in Action*, 2007.

[BDF+ 03] BARHAM, PAUL, BORIS DRAGOVIC, KEIR FRASER, STEVEN H, TIM HARRIS, ALEX HO, ROLF NEUGEBAUER, IAN PRATT und ANDREW WARFIELD: *Xen and the art of virtualization*. In: *In SOSP (2003*, Seiten 164–177, 2003.

[BDS 03] BHATKAR, EEP, DANIEL C. DUVARNEY und R. SEKAR: *Address obfuscation: an efficient approach to combat a broad range of memory error exploits*. In: *In Proceedings of the 12th USENIX Security Symposium*, Seiten 105–120, 2003.

[Blaz] BLAZAKIS, DIONYSUS: *Interpreter Exploitation*.

[BLFM 05] BERNERS-LEE, T., R. FIELDING und L. MASINTER: *RFC 3986, Uniform Resource Identifier (URI): Generic Syntax*. `http://tools.ietf.org/html/rfc3986`, 2005.

[BPSM+ 06] BRAY, TIM, JEAN PAOLI, C.M. SPERBERG-MCQUEEN, EVE MALER, FRANÇOIS YERGEAU und JOHN COWAN: *Extensible Markup Language (XML) 1.1 (Second Edition)*. W3C Recommendation, W3C - World Wide Web Consortium, September 2006, http://www.w3.org/TR/2006/REC-xml11-20060816/ .

[BSC 99] BHOJ, PREETI, SHARAD SINGHAL und SAILESH CHUTANI: *SLA management in federated environments*. In: *Computer Networks*, Seiten 293–308. IEEE Publishing, 1999.

[Burb 07] BURBECK, STEVE: *Complexity and the Evolution of Computing: Biological Principles for Managing Evolving Systems*. `http://evolutionofcomputing.org`, 2007.

[CATV 01] CHASE, JEFFREY S., DARRELL C. ANDERSON, PRACHI N. THAKAR und AMIN M. VAHDAT: *Managing Energy and Server Resources in Hosting Centers*. In: *In Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP*, Seiten 103–116, 2001.

[Chao 11] CHAOS COMPUTER CLUB: *Chaos Computer Club analysiert Staatstrojaner*. `http://ccc.de/de/updates/2011/staatstrojaner`, Oktober 2011.

[CIS 11] CIS: *Die Legende vom großen Hack*. Spiegel online, Dezember 2011.

[CJS+ 05] CHRISTODORESCU, MIHAI, SOMESH JHA, SANJIT A. SESHIA, DAWN X. SONG und RANDAL E. BRYANT: *Semantics-Aware Malware Detection*. In: *IN IEEE SYMPOSIUM ON SECURITY AND PRIVACY*, Seiten 32–46, 2005.

[CoEl 71] COFFMAN, E. G. und M. J. ELPHICK: *System Deadlocks*. Computing Surveys, 3:67–78, 1971.

[dab 11] DAB: *Smart Meter verraten Fernsehprogramm*. heise Security, September 2011.

[dpa 11] DPA: *Landratsamt Bad Hersfeld: Einbrecher klauen Computer*. Frankfurter Rundschau, 2011.

[ES 03] E. SZEGEZDI, U. FITZGERALD, A SAMALI: *Caspase-12 and ER-stress-mediated apoptosis: the story so far*. http://toxicology.usu.edu/endnote/Caspase-12-Szegezdi-ANYAS-2003.pdf, Dezember 2003.

[ETH 10] ETH ZÜRICH: *Aquasar ist in Betrieb*. http://www.ethz.ch/media/detail?pr_id=986, Mai 2010.

[fBuF 12] FORSCHUNG, BUNDESMINISTERIUM FÜR BILDUNG UND: *In tödlicher Mission: Programmierter Zelltod als Waffe gegen Krebs*. http://www.gesundheitsforschung-bmbf.de/de/536.php, 2012.

[fBuK 11] KATASTROPHENHILFE, DAS BUNDESAMT FÜR BEVÖLKERUNGSSCHUTZ UND: *Der Zentrale Bergungsort der Bundesrepublik Deutschland*. http://www.bbk.bund.de/DE/AufgabenundAusstattung/Kulturgutschutz/ZentralerBergungsort/zentralerbergungsort_node.html, 2011.

[FeKu 92] FERRAIOLO, DAVID F. und D. RICHARD KUHN: *Role-Based Access Controls*, 1992. http://csrc.nist.gov/rbac/ferraiolo-kuhn-92.pdf.

[FKS+ 05] FOSTER, I., H. KISHIMOTO, A. SAVVA, D. BERRY, A. DJAOUI, A. GRIMSHAW, B. HORN, F. MACIEL, F. SIEBENLIST, R. SUBRAMANIAM, J. TREADWELL und J. VON REICH. http://www.gridforum.org/documents/GWD-I-E/GFD-I.030.pdf, 2005.

[FKT 01] FOSTER, IAN, CARL KESSELMAN und STEVEN TUECKE: *The Anatomy of the Grid - Enabling Scalable Virtual Organizations*. International Journal of Supercomputer Applications, 15:2001, 2001.

[Foun 12] FOUNDATION, ECLIPSE: *Eclipse IDE for Java EE Developers (Indigo)*. http://www.eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/indigor, 2012.

[fSidI 09] INFORMATIONSTECHNIK, BUNDESAMT FÜR SICHERHEIT IN DER: *IT Grundschutz*. https://www.bsi.bund.de/DE/Themen/weitereThemen/ITGrundschutzKataloge/Inhalt/Massnahmenkataloge/massnahmenkataloge_node.html;jsessionid=F352DF0260DA6AAE034787E845EF78BF.2_cid244, 2009.

[G Da 11] G DATA: *G Data Malware Report - 1. Halbjahr 2011*. http://www.gdata.de/fileadmin/dam_files/Ab_Mai_2010/SecurityLab/Texte/Whitepapers/DE/G_Data_MalwareReport_H1_2011_DE.pdf, 2011.

[G Da 12a] G DATA: *Eine kleine Geschichte der Viren, Würmer und Trojaner, Teil 2*. http://www.gdata.de/virenforschung/info/malware-geschichte/1988-1994.html, 2012.

[G Da 12b] G DATA: *Eine kleine Geschichte der Viren, Würmer und Trojaner, Teil 3*. http://www.gdata.de/virenforschung/info/malware-geschichte/1996-heute.html, 2012.

[Glei 96] GLEICK, JAMES: *A Bug and a Crash*. New York Times Magazine, Dezember 1996.

[Huck 11] HUCKLE, THOMAS: *Collection of Software Bugs*. 'http://wwwzenger.informatik.tu-muenchen.de/persons/huckle/bugse.html#general', 2011.

[IKBS 00] IOANNIDIS, SOTIRIS, ANGELOS D. KEROMYTIS, STEVE M. BELLOVIN und JONATHAN M. SMITH: *Implementing a Distributed Firewall*, 2000.

[JK 03] J. KEPHART, D. CHESS: *The Vision of Autonomic Computing*. Computer Magazine, IEEE, 2003.

[Kay 07] KAY, MICHAEL: *XSL Transformations (XSLT) Version 2.0*. http://www.w3.org/TR/xslt20/, 2007.

[Kirs 11] KIRSCH, CHRISTIAN: *Festplattenproduktion könnte um mehr als ein Viertel einbrechen*. Heise online, November 2011.

[KKaFP 99] KAHN, J. M., R. H. KATZ, R. H. KATZ (ACM FELLOW und K. S. J. PISTER: *'Next Century Challenges: Mobile Networking for SSmart Dust"*, 1999.

[KrTo 02] KRÜGEL, CHRISTOPHER und THOMAS TOTH: *Flexible, Mobile Agent based Intrusion Detection for Dynamic Networks*, 2002.

[Kuri 11] KURI, JÜRGEN: *Das Internet der Energie*. heise resale, Februar 2011.

[LBAK+ 98] LEE, INSUP, H. BEN-ABDALLAH, S. KANNAN, M. KIM, O. SOKOLSKY und M. VISWANATHAN: *A Monitoring and Checking Framework for Run-time Correctness Assurance*, 1998.

[lis/ 11] LIS/AFP: *Alte Dame klaut Kabel - Georgien offline*. Spiegel online, April 2011.

[LKK+ 99] LEE, INSUP, SAMPATH KANNAN, MOONJOO KIM, OLEG SOKOLSKY und MAHESH VISWANATHAN: *Runtime Assurance Based On Formal Specifications*, 1999.

[LZL+ 05] LO, VIRGINIA, DAYI ZHOU, YUHONG LIU, CHRIS GAUTHIERDICKEY und JUN LI: *Scalable Supernode Selection in Peer-to-Peer Overlay Networks*. In: *In Proceedings of the 2nd International Workshop on Hot Topics in Peer-to-Peer Systems, La*, Seiten 18–25. IEEE, 2005.

[MB] MICHAEL BACKES, STEFAN LORENZ: *X-pire*. http://www.x-pire.de.

[Micr 06] MICROSOFT: *A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003*. http://support.microsoft.com/kb/875352/EN-US/, 2006.

[oD 85] DEFENSE, DEPARTMENT OF: *Trusted Computer System Evaluation Criteria*, Dezember 1985. http://www.fas.org/irp/nsa/rainbow/std001.htm.

[OL 99] ORA LASSILA, RALPH R. SWICK: *Resource Description Framework (RDF) Model and Syntax Specification*. http://www.w3.org/TR/PR-rdf-syntax, 1999.

[OLKT] OLIVEIRA, RICARDO M., SIHYUNG LEE, HYONG S. KIM und PORTUGAL TELECOM: *using firewall and*.

[Orac 12] ORACLE: *Java 7*. http://java.com, 2012.

[RaSt 96] RAMANATHAN, S. und MARTHA STEENSTRUP: *A Survey of Routing Techniques for Mobile Communications Networks*. MOBILE NETWORKS AND APPLICATIONS, 1:89–104, 1996.

[Rieg 10] RIEGER, FRANK: *Der digitale Erstschlag ist erfolgt*. Frankfurter Allgemeine Zeitung, September 2010.

[RiGo 98] RICHTERS, MARK und MARTIN GOGOLLA: *On Formalizing the UML Object Constraint Language OCL*. In: *Proc. 17th Int. Conf. Conceptual Modeling (ER'98*, Seiten 449–464. Springer, 1998.

[RoDr 01] ROWSTRON, ANTONY und PETER DRUSCHEL: *Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems*, 2001.

[RQZ 07] RUPP, CHRIS, STEFAN QUEINS und BARBARA ZENGLER: *UML 2 glasklar – Praxiswissen für die UML-Modellierung*. Hanser, 3 Auflage, 2007.

[Schm 38] SCHMITT, OTTO H: *A thermionic trigger*. Journal of Scientific Instruments, 15(1):24, 1938, http://stacks.iop.org/0950-7671/15/i=1/a=305 .

[Schm 11] SCHMITZ, HEINZ. http://www.cczwei.de/index.php?id=tvissuearchive&tvissueid=83#a188, 2011.

[SK 10] STEFAN KREMPL, PETER-MICHAEL ZIEGLER: *Datenschützer fordern klare Regeln bei Smart-Meter-Nutzung*. heise online, November 2010.

[SMK+ 01] STOICA, ION, ROBERT MORRIS, DAVID KARGER, M. FRANS KAASHOEK und HARI BALAKRISHNAN: *Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications*. In: *Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications*, Seiten 149–160, 2001.

[SMS+ 02] SAHAI, AKHIL, VIJAY MACHIRAJU, MEHMET SAYAL, LI JIE JIN und FABIO CASATI: *Automated SLA Monitoring for Web Services*. In: *IEEE/IFIP DSOM*, Seiten 28–41. Springer-Verlag, 2002.

[SSD+ 03] SAMMAPUN, USA, RAMAN SHARYKIN, MARGARET DELAP, MYONG KIM und STEVE ZDANCEWIC: *Formalizing Java-MaC*, 2003.

[Stol 89] STOLL, CLIFFORD: *The Cuckoo's Egg: Tracking a Spy Through the Maze of Computer Espionage*. ISBN 0-385-24946-2, 1989.

[Stra 11] STRAUBE, CHRISTIAN: *DAGA - Active Probing zur Bestimmung der Verfügbarkeit von Grid-Ressourcen*. Ludwig-Maximilians-Universität München, 2011.

[Szor 05] SZOR, PETER: *The Art of Computer Virus Research and Defense*. ISBN 0-32-130454-3, 2005.

[tB 08] BRANDSCHUTZ, BUNDESVERBAND TECHNISCHER: *Moderne Brandschutzkonzepte für Rechenzentren*. `http://www.bvfa.de/pdf-download/de-15/`, 2008.

[The] THE IT CROWD: *Have You Tried Turning It Off And On Again?* `http://www.youtube.com/watch?v=nn2FB1P_Mn8`.

[TK 11] TORSTEN KLEINZ, ANDREAS WIKENS: *Conlife: Smart Meter als Hoffnungsträger*. heise online, Juni 2011.

[U.S. 04] U.S.-CANADA POWER SYSTEM OUTAGE TASK FORCE: *Final Report on the August 14, 2003 Blackout in the United States and Canada: Causes and Recommendations*. Technischer Bericht, U.S.-Canada Power System Outage Task Force, April 2004.

[VVKH 00] VIDAL, BY EDWARD, EDWARD VIDAL, VLADIK KREINOVICH und HUANG HAITAO: *'Geombinatorics Of ßmart Dust"*, 2000.

[Wike 11] WIKENS, ANDREAS: *Nutzer zeigen intelligenten Stromzählern die kalte Schulter*. Heise online, November 2011.

[Wind 11] WINDECK, CHRISTOF: *Modulare Hardware für neuronale Netze*. heise online, November 2011.

[Zett 10] ZETTER, KIM: *SCADA System's Hard-Coded Password Circulated Online for Years*. Wired, Juli 2010.

[ZHM 97] ZHU, HONG, PATRICK A. V. HALL und JOHN H. R. MAY: *Software unit test coverage and adequacy*. ACM Computing Surveys, 29:366–427, 1997.