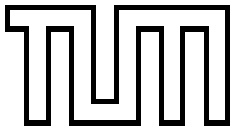


INSTITUT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

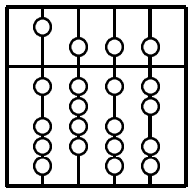
Diplomarbeit

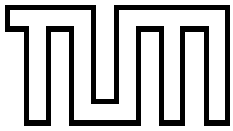
Formalisierung von Aggregationsvorschriften für Dienstinformationen

Bearbeiter: Sebastian Lange

Aufgabensteller: Prof. Dr. Heinz-Gerd Hegering

Betreuer: Vitalian Danciu
Martin Sailer





INSTITUT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Diplomarbeit

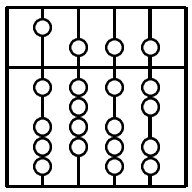
Formalisierung von Aggregationsvorschriften für Dienstinformationen

Bearbeiter: Sebastian Lange

Aufgabensteller: Prof. Dr. Heinz-Gerd Hegering

Betreuer: Vitalian Danciu
Martin Sailer

Abgabetermin: 15. Mai 2006



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

München, den 15. Mai 2006

.....

(Unterschrift des Kandidaten)

Zusammenfassung

Gegenwärtig angewandtes Netz- und System-Management stützt sich meist nur auf die Beobachtung und Überwachung einzelner Ressourcen und Netzwerk-Komponenten. In der Zeit der Internet-Dienste und Webservices ist dies jedoch nicht mehr ausreichend. Wo Vereinbarungen und Verträge mit Kunden über Eigenschaften eines Dienstes bzw. die Dienstgüte getroffen werden und nicht über die einzelnen Komponenten, ist eine umfassendere Sicht notwendig, als die auf seperate Bestandteile. Diese Dienstsicht setzt allerdings voraus, daß ein Dienst als Ganzes beobachtet werden kann, um Aussagen über ihn treffen zu können. Aus diesen Gründen geht die Entwicklung vom bisherigen Ressourcenmonitoring weg zum Dienstmonitoring, wobei einzelne Ressourcendaten zu dienstbezogenen Datenaggregationen zusammengefasst werden müssen.

In der vorliegenden Diplomarbeit wird ein Teil einer Monitoring Architektur vorgestellt, welche solch eine Dienstsicht bieten kann. Es sind theoretische und praktische Ansätze vorgestellt, wie einzelne Ressourcendaten zu einer Aggregation vereint werden können, um somit ein Dienstmerkmal zu beschreiben. Zu diesem Zweck wird eine Sprache für die Spezifikation von Datenaggregationen entwickelt und in den Entwurf der Monitoring Anwendung eingefügt. Die Grammatik der Sprache und die prototypische Implementierung der Anwendung sollen dabei die praktische Realisierbarkeit der erdachten Konzepte untermauern.

Inhaltsverzeichnis

1 Einführung **1**
- 1.1 Aufgabenstellung . . . 2
- 1.2 Vorgehensweise der Arbeit . . . 3
- 1.3 Gliederung der Arbeit . . . 5

2 Szenario **6**
- 2.1 Beschreibung der Szenarioumgebung . . . 6
  - 2.1.1 Bisheriger Ansatz . . . 8
  - 2.1.2 Dienstmonitoring . . . 10
  - 2.1.3 Allgemeine Anforderungen . . . 11
- 2.2 Zusammenfassung . . . 12

3 State of the Art **13**
- 3.1 Begriffsbestimmung und Grundlagen zu Netzmanagement und Monitoring . 13
  - 3.1.1 Informationsmodellierung . . . 14
  - 3.1.2 Kommunikationsmodelle . . . 18
- 3.2 Untersuchung vorhandener Spezifikationskonzepte . . . 20
  - 3.2.1 Resource Description Framework . . . 20
  - 3.2.2 Object Constraint Language . . . 21
  - 3.2.3 Web Service Description Language . . . 22
- 3.3 Vergleich bestehender Monitoring Architekturen . . . 22
  - 3.3.1 Nagios . . . 23
  - 3.3.2 HP OpenView . . . 23
  - 3.3.3 Management Architektur des Lehrstuhls . . . 24
- 3.4 Zusammenfassung . . . 26

4 Anforderungsanalyse **27**
- 4.1 Anforderungen an die Spezifikationssprache . . . 27
- 4.2 Anforderungen an die Architektur . . . 29
- 4.3 Anforderungskatalog . . . 31

5 Die Service Information Specification Language **32**
- 5.1 Grundlegende Sprachkonzepte . . . 32
  - 5.1.1 Mathematische Funktionen . . . 34
  - 5.1.2 Bedingungen . . . 35
  - 5.1.3 Referenzierbarkeit . . . 36

5.2 Grammatik der Sprache . . . 37
5.2.1 Anmerkung zur Verwendung der Sprache . . . 37
5.2.2 Vereinbarungen und Hilfsproduktionen . . . 38
5.2.3 Die Aggregation . . . 41
5.2.4 Das Element <ressource> . . . 43
5.2.5 Das Element <function> . . . 46
5.2.6 Das Element <condition> . . . 48
5.2.7 Anwendungsbeispiel . . . 53
5.3 Zusammenfassung . . . 55

6 Entwurf des Prototypen 56
6.1 Architektur des Prototypen . . . 56
6.1.1 Aktivitäten der Anwendung . . . 57
6.1.2 Übersicht über die verschiedenen Komponenten . . . 59
6.1.3 Verteilung der Ressourcenwerte . . . 62
6.1.4 Adaptermanagement . . . 63
6.1.5 Fehlerbehandlung bei den Adaptern . . . 65
6.1.6 Übersicht der Funktionalitäten . . . 65
6.2 Datenaggregation . . . 67
6.2.1 Das Objekt Aggregation . . . 67
6.2.2 Das Ressourcenobjekt . . . 68
6.2.3 Das Funktionenobjekt . . . 70
6.2.4 Das Conditionobjekt . . . 71
6.2.5 Dereferenzierung von Objekten . . . 73
6.3 Komponenten zur Auswertung . . . 74
6.3.1 Funktionenberechnung . . . 74
6.3.2 Bedingungsauswertung . . . 75
6.4 Zusammenfassung . . . 77

7 Richtlinien zur Implementierung 78
7.1 Spezifikation einer Datenaggregation in XML . . . 78
7.1.1 Überblick zu XML . . . 79
7.1.2 XML-Schema der Spezifikationssprache . . . 80
7.1.3 Erzeugen der Aggregationsobjekte . . . 81
7.2 Kommunikationsschnittstellen des Prototypen . . . 83
7.2.1 Planung der Schnittstellen des Prototypen . . . 84
7.2.2 Definition der Schnittstellen . . . 85
7.3 Einbindung entfernter Methodenaufrufe . . . 88
7.3.1 Common Object Request Broker Architecture . . . 89
7.3.2 Simple Object Access Protocol . . . 91
7.3.3 Remote Method Invocation . . . 93
7.4 Funktionsweise des Prototypen . . . 94
7.5 Zusammenfassung . . . 95

8 Zusammenfassung und Ausblick 97

A Sprachdefinition in EBNF **100**

B Sprachdefinition in XML **103**

Abbildungsverzeichnis

1.1 Am Lehrstuhl Hegering vorgeschlagene Monitoring Architektur 2
1.2 Vorgehensweise der Arbeit 4

2.1 Dienstsicht auf das Beispielszenario 7
2.2 Szenario Detail 8
2.3 Ressourcen Monitoring 9
2.4 Dienst Monitoring 10

3.1 Baumstruktur einer Management Information Base [HAN 99] 15
3.2 Service mit zwei Sichtweisen bei SID [Bitt 05] 18
3.3 Das Prinzip der SNMP-Kommunikation 19
3.4 RDF-Graph einer einfachen Objektbeschreibung 20
3.5 Die Monitoring Architektur des Lehrstuhls 25

4.1 Anforderungen an die dienstorientierte Datenkomposition 28
4.2 Generierung eines möglichen Rich Events 29

5.1 Aufbauskizze der Sprache 33
5.2 Referenzen einzelner Elemente 36
5.3 Bedeutung von `<amount>` am Beispiel einer Werteliste einer Ressource . 40
5.4 Unterschied zwischen *timeout*{} und *interval*{} 50

6.1 Ausschnitt der Gesamtarchitektur 57
6.2 Workflow der Monitoring Anwendung 58
6.3 Komponenten der Anwendung 60
6.4 Vorgang der Zuordnung der Datensätze 63
6.5 Das Adapter Respository 64
6.6 Repräsentaion einer Aggregation 67
6.7 Repräsentaion einer Ressource 68
6.8 Repräsentaion einer Funktion 70
6.9 Repräsentation einer Bedingung 72
6.10 Schachtelung der Objekte in einer Bedingung 73
6.11 Sequenzdiagramm einer Funktionsberechnung 75
6.12 Beispiel eines Bedingungsbaums mit Auswertung 76

7.1 Ausschnitt eines DOM-Baumes 83
7.2 Schnittstellen des Prototypen zu anderen Komponenten 84
7.3 Grundprinzip der Kommunikation mit CORBA [Zimm 00] 90

7.4 Prinzipieller Ablauf des IDL-Compilers [Zimm 00] 90
7.5 Kommunikation über einen ORB am Beispiel des Prototypen 91
7.6 Remote-Methode für eine Beschreibung aufrufen 93
7.7 Darstellung des *aggregation*-Package 94
7.8 Darstellung des *core*-Package 94
7.9 Bearbeitung einer Datenaggregation 96

Tabellenverzeichnis

4.1 Überblick der Anforderungen 31
6.1 Überblick der Funktionenalitäten 66
7.1 Überblick der Schnittstellen des Prototyps 88

Kapitel 1

Einführung

Computer sind aus unserer heutigen Geschäftswelt nicht mehr weg zu denken. Fast jedes Unternehmen ist „im Internet" zu finden, oder hat eine Homepage, auf der seine Produkte angeboten werden. Auch öffentliche Institutionen oder Universitäten nutzen täglich das Internet, sei es, um zu informieren, im Netz selbst nach Informationen zu suchen oder Daten auszutauschen. Ein weiteres Beispiel dafür wäre, dass die Email neben dem Telefon das populärste Kommunikationsmedium geworden ist und den Brief längst abgelöst hat.

Doch dem normalen Benutzer bleibt verborgen, dass hinter „dem Internet" viel mehr steckt, als er sich vorstellen kann, wenn er eine Homepage anklickt. Internet Service Provider (ISP) bieten verschiedene technische Leistungen an, die für die Nutzung des Internets erforderlich sind. Dazu gehört der Zugang zum Internet genauso wie Registrierung von Domains, Vermietung von Webservern usw. Der ISP muss sich wiederum auf andere externe Dienstanbieter verlassen können, die er selbst nur nutzt. Hinter dieser Kette von Abhängigkeiten stehen unzählige Geräte, die für den reibungslosen Ablauf so eines Dienstes sorgen sollen. Webserver, Datenbankserver, Switches und Router müssen ständig korrekt funktionieren, damit vertraglich festgelegte Abmachungen eingehalten werden können.

Für fast alle Unternehmen ist ein funktionierendes und fehlerfreies Computernetzwerk überlebenswichtig. Doch nicht selten kommt es vor, dass etwas nicht so funktioniert, wie es soll. Dienste sind entweder überhaupt nicht oder nur schlecht verfügbar, Geräte fallen aus oder haben „urplötzlich" zu lange Wartezeiten. Das Netzwerk- und Systemmanagement der Anbieter soll deswegen dafür sorgen, dass solche Ausfälle nicht vorkommen. Und falls doch, in kürzester Zeit Abhilfe geschaffen werden kann. Zum Beispiel sind nicht selten Mindestlaufzeiten vertraglich festgelegt, und deren Nichteinhalten kann für den Anbieter teure Folgen haben. Bei Fehler- oder Performancemanagement sollte man also „sein Netz" genau beobachten, um immer einen genauen Überblick darüber zu haben, ob noch alles in Ordnung ist, oder gerade etwas schief läuft. Schließlich sollte man eventuelle Ausfälle zuerst bemerken, bevor die Kunden es tun.

Unter Netz-Monitoring versteht man die Überwachung und regelmäßige Kontrolle von Netzen, deren Hardware (z.B. Server, Router, Switches) und Diensten (z.B. Webserver, DNS-Dienste, E-Mail-Dienste). Es werden entweder aktiv Informationen von entsprechenden Geräten/Diensten in periodischen Zeitabständen geholt, oder die Ressourcen schicken selb-

ständig ihre Statusmeldungen. Bei einem größeren Netz mit mehreren, u.U. voneinander abhängigen, Geräten schickt natürlich jedes einzelne seine eigenen Werte. Und nicht selten hat der zuständige Administrator riesige Zahlentabellen vor sich liegen. Wenn jetzt ein spezielles Gerät ausfällt, kann er schnell in der betreffenden Tabelle nachschauen, was dort nicht stimmt. Ist allerdings ein komplexer Dienst unzureichend verfügbar, wo mehrere Ressourcen mit hinein spielen und miteinander verknüpft sind, kann der Administrator sich leicht in der Masse an Werten verlieren. So wird die Reaktionsfähigkeit stark eingeschränkt und es kann nur unzureichend schnell bis überhaupt nicht reagiert werden.

1.1 Aufgabenstellung

In dieser Arbeit wird ein Teil einer generischen Monitoring Architektur bearbeitet, die bereits am Lehrstuhl von Prof. Hegering vorgeschlagen wurde [DaSa 05]. Um sie kurz vorzustellen, ist sie schon einmal in Abbildung 1.1 gezeigt.

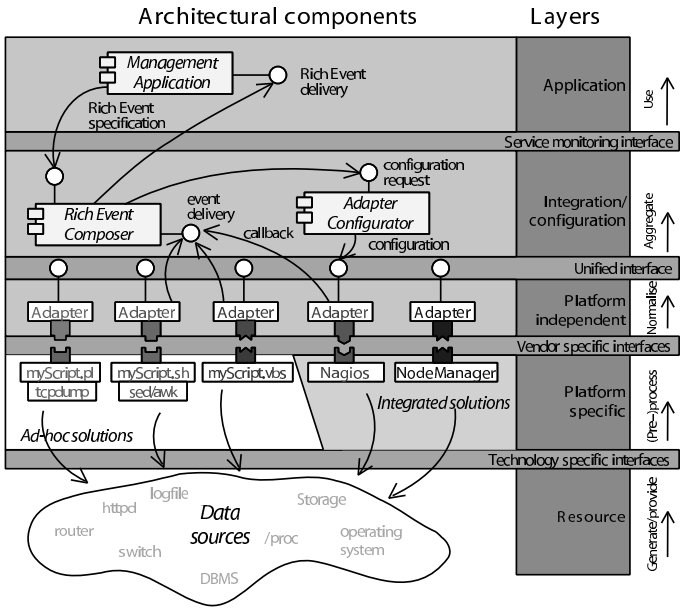


Abbildung 1.1: Am Lehrstuhl Hegering vorgeschlagene Monitoring Architektur

Vorrausgesetzt wird hier ein heterogenes Netzwerk mit unterschiedlichsten Ressourcen und Quellen. Diese unterscheiden sich alle im Format und je nach Plattform wird unterschiedlich auf sie zugegriffen. Diesem heterogenen Netzwerk soll eine einheitliche Architektur aufgesetzt werden, die eine einheitliche Zugriffsschicht ermöglicht. Im Bild wird dies durch die Adapter symbolisiert. Aus den reinen Ressourceninformationen sollen dann aggregierte und korrelierte Datensätze, sogenannte „Rich Events", erstellbar sein, die später z.B. höheren Managementanwendungen oder Auslösern für Policies als Eingabe dienen können.

Um solche Datensätze erzeugen zu können, muss erst einmal ein Dienst in verschiedene Bestandteile gegliedert und maschinenverständlich beschrieben werden können. Außerdem

muss von vorn herein ein einheitliches Format festgelegt werden, mit dem solch eine Datenaggregation repräsentiert wird. Dazu wird eine Spezifikationssprache entwickelt, die es dem Benutzer erlaubt, verschiedene u.U. im Netzwerk verteilte Ressourcen zu einem Datensatz zusammen zu fassen. Außerdem müssen auch flexible Bedingungen festlegbar sein, nach welchen Regeln Ereignisse ausgelöst und verschickt werden sollen. Dabei müssen diese Bedingungen frei formulierbar und kombinierbar bleiben, und sollen nicht nur an Datenwerte sondern auch an die Zeit geknüpft sein können.

Ferner wird der obere Teil der Monitoring Architektur, welcher die Sprache versteht und die Rich Events zusammenstellt, prototypisch implementiert. Es muss sichergestellt werden, dass mehrere Rich Events gleichzeitig bearbeitet werden können. Außerdem muss der Benutzer in der Lage sein, schnell zu sehen, welche Ressourcen überhaupt angefragt werden können. Die Anwendung sollte selbst erkennen, wenn die gleichen Ressourcenwerte von mehreren Rich Events verlangt werden, und somit unnötige Redundanz vermeiden. Mittels CORBA sollen die Anfragen an die einzelnen Ressourcen geschickt werden können. In dieser Diplomarbeit wird die einheitliche Zugriffsschicht zwar vorrausgesetzt, aber nicht behandelt, da dies den Rahmen sprengen würde.

1.2 Vorgehensweise der Arbeit

In Abbildung 1.2 wird die Vorgehensweise in dieser Arbeit graphisch dargestellt. Ausgehend von einem Szenario werden grobe Anforderungen an eine mögliche Monitoring Architektur gestellt und mit den bisher verfügbaren Lösungen verglichen. Die daraus resultierenden Defizite werden analysiert, woraus dann ein konkreter Anforderungskatalog für das eigene Projekt hervorgeht. Anhand dieser verfeinerten Anforderungen wird die Spezifikationssprache entworfen und in einer Grammatik definiert, während die Monitoringanwendung prototypisch implementiert wird. Die Arbeit ist in drei Phasen aufgeteilt.

- Analysephase:
  Zunächste wird mit dem *Szenario* ein möglicher Anwendungsfall für die neue Architektur vorgestellt. Es wird gezeigt, dass die bisherigen Methoden für die gewünschte Art von Aufgabe nicht ausreichend sind, und wo derzeitige Probleme zu finden sind. Nachdem die Situation im Szenario dargestellt wurde, können anhand des Problems *allgemeine Anforderungen* jetzt grob umschrieben werden.

  Bei *State of the Art* werden bereits vorhandene Lösungen für das im Szenario gestellte Problem auf ihre Tauglichkeit hin untersucht. Dabei werden sowohl existierende Sprachen als auch Monitoringanwendungen betrachtet. Es wird versucht, die vorhandenen *Defizite* dieser Lösungen aufzuzeigen.

  Anhand dieser Defizite können nun *detaillierte Anforderungen* für den eigenen Anwendungsentwurf formuliert werden. Dabei werden die allgemeinen Anforderungen konkretisiert und verfeinert, damit sie eine gute Entwicklungsgrundlage für die eigene Lösung bieten.

- Entwurfsphase:
  Ausgehend von den Anforderungen werden nun parallel zwei Entwürfe angefertigt.

Zum einen werden Konzepte für die Spezifikationssprache entwickelt, die fortlaufend mit den Anforderungen verglichen werden. Die entstandene Sprache kann danach in einer *Grammatik* formuliert werden.

Zum zweiten wird die Monitoring Anwendung in ihren funktionalen Einzelheiten geplant. Auch hier wird anhand der detaillierten Anforderungen kontrolliert. Ein *konzeptioneller Ansatz* zeigt dann, wie solch eine Anwendung realisiert werden könnte.

- Realisierungsphase:
  Nachdem sowohl Grammatik als auch konzeptioneller Ansatz vorhanden sind, kann die Implementation des Anwendungsprototypen beginnen. Der *Prototyp* muss diese Sprache einlesen, und aufgrund der so übergebenen Informationen die korrekten Datenaggregationen liefern können. Nach Möglichkeit sollten mit dem abschließenden Prototypen alle zu Anfang festgelegten Anforderungen erfüllt sein.

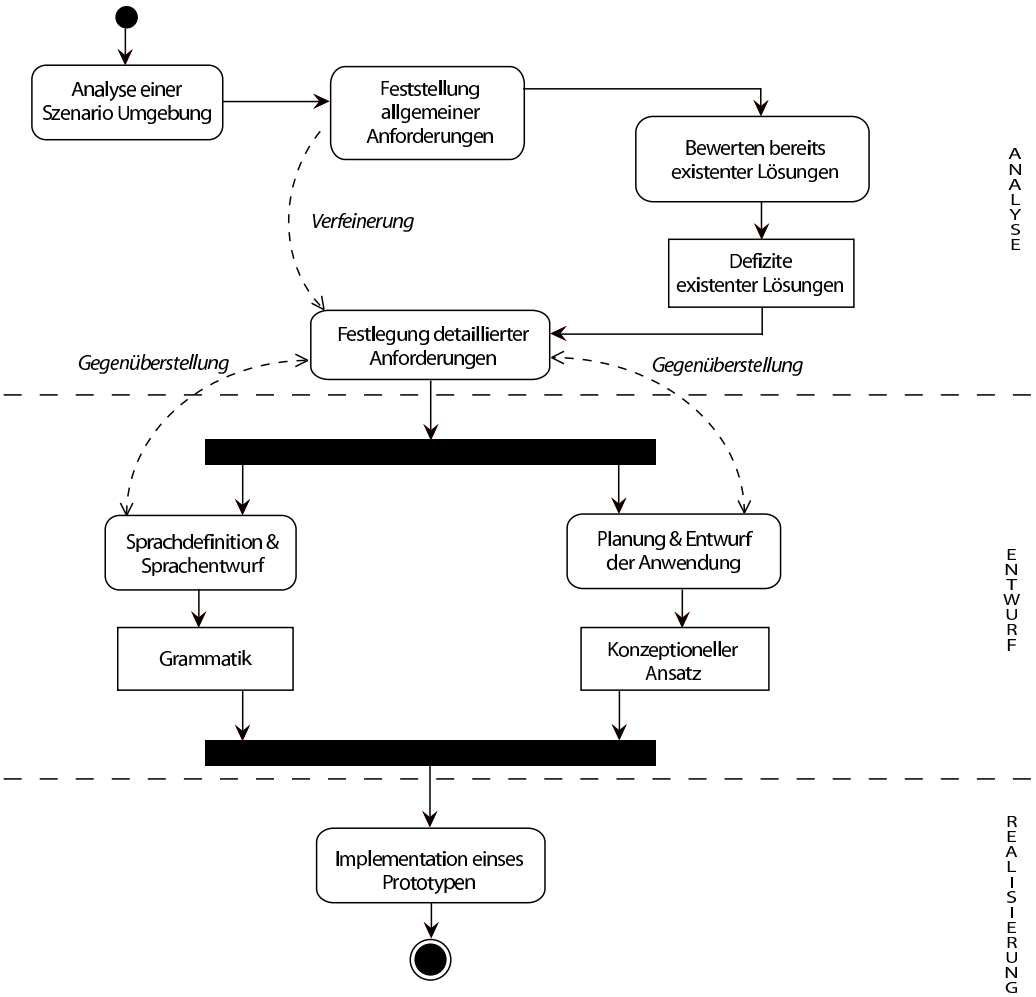


Abbildung 1.2: Vorgehensweise der Arbeit

1.3 Gliederung der Arbeit

Diese Arbeit gliedert sich in acht Kapitel. Im ersten Kapitel wird die Notwendigkeit von Netzmanagement und damit verbundenem Monitoring grob umrissen. Danach wird die Aufgabenstellung erläutert und in einen größeren Kontext gestellt. Außerdem wird die Vorgehensweise der Arbeit festgehalten.

Kapitel Zwei präsentiert ein Szenario, welches zur Motivation dienen soll und die grundsätzliche Problematik des behandelten Themas darstellt. Dabei wird der Unterschied zwischen Ressourcen-Monitoring und Dienst-Monitoring erläutert. Darüber hinaus werden allgemeine Anforderungen zur Lösung des Problems formuliert.

Das dritte Kapitel klärt zunächst grundlegende Begriffe, worum es sich beim Thema Management bzw. Monitoring handelt und stellt Standardisierungen und verbreitete Protokolle vor. Danach werden bereits existete Lösungen und kommerzielle Produkte untersucht, ob ihre Relevanz bezüglich dieser Arbeit gegeben ist. Des weiteren wird die gesamte Monitoring Architektur vorgestellt, von welcher hier nur ein gewisser Teil bearbeitet wird.

Im vierten Kapitel werden die Anforderungen an die zu entwickelnde Anwendung konkretisiert und analysiert. Dabei stellen sich zwei Teilgebiete heraus, die bearbeitet werden müssen. Zum einen die *Service Information Specification Language* und zum anderen die Monitoring Anwendung selbst, die diese Sprache verstehen muss. Ein Anforderungskatalog umreißt klar die Aufgaben der zwei Teilaspekte.

In Kapitel Nummer Fünf wird die *Service Information Specification Language* entworfen und spezifiziert. Dazu werden zunächst grundlegende Konzepte erklärt, danach wird die Grammatik der Spezifikationssprache in EBNF angegeben und anhand von Beispielen erläutert.

Das sechste Kapitel behandelt die Planung und den Entwurf des Prototypen. Zunächst werden die Architektur und die funktionalen Konzepte beschrieben. Daraufhin ist die Darstellung von Datenaggregationen innerhalb der Anwendung beschrieben, sowie die dynamischen Aspekte, die zur Auswertung bzw. Bearbeitung solch einer Aggregation notwendig sind.

Im siebten Kapitel wird die Implementierung des Prototypen vorgestellt. Es wird gezeigt, wie die Spezifikationssprache in XML umgesetzt werden kann, wie die Kommunikation zu den unteren Schichten der gesamten Monitoring Architektur realisiert wird und wie der Aufbau des Quellcodes organisiert und realisiert ist.

Das letzte, achte Kaptiel beinhaltet die Zusammenfassung der Arbeit sowie die gewonnenen Ergebnisse, die noch einmal heraus gestellt werden. Außerdem werden hier Möglichkeiten vorgestellt, wie der gewonnene Prototyp der Monitoring Architektur noch zu verbessern und zu erweitern wäre.

Kapitel 2

Szenario

Heutzutage gibt es unzählige *Dienste*, die über das Internet angeboten werden. Dies reicht von dem Bereitstellen einer Homepage des eigenen Unternehmens bis hin zu komplexen Web Services, die von verschiedensten Anbietern zu Verfügung gestellt werden. Diese Arbeit richtet ihr Augenmerk nicht auf die technisch einfachen Dienste, die vielleicht nur einen einzigen Webserver umfassen, sondern betrachtet ein umfangreicheres System.

Charakteristisch für solche Dienste ist, dass viele unterschiedliche Geräte und Anwendungen zum erfolgreichen Betrieb erforderlich sind. Die notwendigen Komponenten eines Dienstes werden hier als *Ressourcen* bezeichnet. So gelten in diesem Umfeld z.B. Server und Router genauso als Ressource wie ein DNS-Service, der benutzt wird. Soll ein Dienst fehlerfrei arbeiten, setzt dies ein einwandfreies Funktionieren der Ressourcen voraus. Der Anbieter ist daher darauf bedacht, seine einzelnen Ressourcen zu pflegen und zu überwachen. Aus dieser *Anbietersicht* stehen deswegen die einzelnen Komponenten im Vordergrund.

Allerdings ist die *Kundensicht* eine Andere. Der Kunde sieht den Dienst als eine Einheit auf die er zugreift und für die er einen bestimmten Geldbetrag bezahlt. Er trifft mit dem Anbieter gewisse Vereinbarungen über die Dienstgüte (Service Level Agreements), d.h. die Betrachtungsweise befindet sich hier auf einem viel höheren Niveau. Aus der *Dienstsicht* werden keine einzelnen Ressourcen beobachtet, sondern der Dienst als Ganzes, um überhaupt Aussagen zu Dienstvereinbarungen und Dienstgüte machen zu können.

Es gibt viele Varianten, solch ein Szenario aufzubauen, vielleicht ein Web Hosting Service oder ein Onlinespiele Anbieter. Die Möglichkeiten sind vielfältig. Um das bestehende Problem im Monitoring aufzuzeigen, wurde hier das Szenario des Onlinespiel-Anbieters gewählt.

2.1 Beschreibung der Szenarioumgebung

Was muss man sich nun bei einem Spieleanbieter vorstellen? Ein Massive Multiplayer Online Role-Playing Game (MMORPG) ist ein Computerspiel, an dem mehrere tausend Spieler gleichzeitig über das Internet spielen können. Es ist im Prinzip eine Weiterentwicklung, die

aus den Multi User Dungeons Anfang der 90er Jahre entstanden ist.[Wiki 06]

Wie auf Abbildung 2.1 zu sehen ist, stellt der Provider sein Spiel, bzw. seine Spieleserver, zur Verfügung. Jeder Benutzer kann sich nun auf dem Server einloggen und anfangen zu spielen, wenn er dem Anbieter den Preis dafür bezahlt hat. Die Leistung des Providers dagegen ist, das Spiel bereitzustellen, und zu garantieren, dass es jederzeit einwandfrei erreichbar ist, und alle spielinternen Anforderungen erfüllt sind.

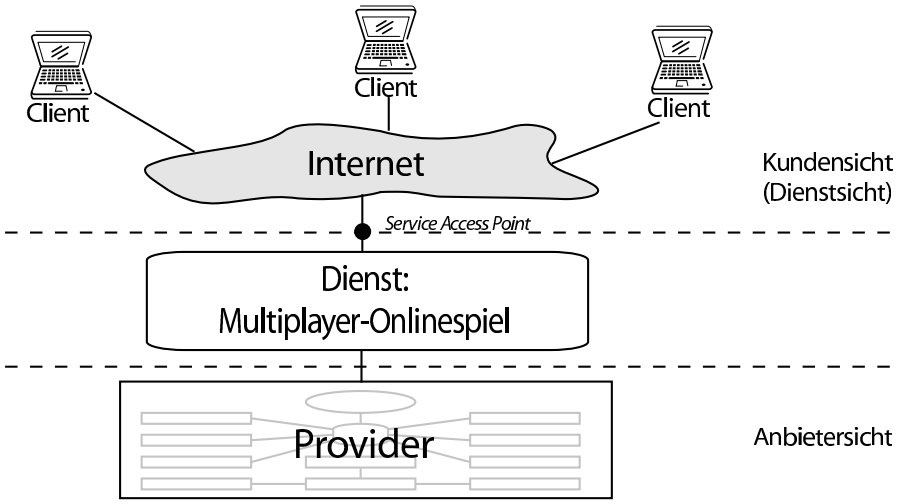


Abbildung 2.1: Dienstsicht auf das Beispielszenario

Diese Darstellung ist an das MNM-Dienstmodell [GHKR 01] angelehnt. Der Dienst selbst stellt demnach für die Benutzer die Zugriffsschicht auf die im Hintergrund angewandten Ressourcen dar. Ein einzelner Spieler kann überhaupt nicht wissen, wie der Anbieter sein Netzwerk intern aufgebaut hat. Muss er auch nicht, schließlich bezahlt er für einen Dienst, der funktioniert. Auch der Provider selbst hat unter Umständen nicht den genauen Überblick, da seine Ressourcen irgendwo physisch verteilt sein können. Innerhalb seiner Architektur hat der Anbieter vielleicht verschiedene Spezialisten, die sich nur um einen kleinen Bereich des Ganzen kümmern, und sich genau dort gut auskennen. (vgl. Abbildung 2.2) Die Sicht des Benutzers auf den angebotenen Dienst ist in diesem Fall identisch mit der des Providers. Sollten Fragen auftauchen, betreffen sie den Dienst an sich, und nicht irgendwelche einzelnen Netzkomponenten, die intern für das erfolgreiche Funktionieren notwendig sind.

So interessiert zum Beispiel sowohl den Kunden als auch den Anbieter, ob der Dienst überhaupt läuft. *„Ist er denn über die normale URL erreichbar, oder gibt es da nur Fehlermeldungen?" „Kann man die Homepage zum Spiel erreichen?" „Ist die Performance in Ordnung, oder kann wegen zu schlechtem Durchsatz der Dienst nur mangelhaft genutzt werden?"* Fragen in dieser Form werden gestellt, und daran wird auch die Qualität des Dienstes bemessen. Es interessiert keinen Kunden, wieviele Pakete über Port X des ersten Routers gehen, oder mit welcher Auslastung einer von 20 Servern läuft. Der Kunde vereinbart mit dem Anbieter keine Switchauslastungen sondern Verzögerungszeiten des gesamten Angebots. Der primäre Blick von Kunde und Anbieter geht zu den Informationen der Dienstgüte, nicht zu den einzelnen Interna (siehe auch Abschnitt 2.1.2).

In allgemeiner Formulierung bezeichnet Dienstgüte die Gesamtheit der Qualitätsmerkmale des Dienstes. So stellen sich Fragen zur Verfügbarkeit, Antwortzeit, Auslastung, Fehlerrate, Durchsatz, etc. Um Aussagen zu solchen Punkten machen zu können, müssen natürlich Daten und Zustände aller beteiligten Ressourcen gesammelt, beobachtet und entsprechend aggregiert werden. Die bisherige Herangehensweise an eine solche Fragestellung wird nun genauer im folgenden Abschnitt beschrieben.

2.1.1 Bisheriger Ansatz

Um Daten der zuständigen Ressourcen zu sammeln, reicht die Betrachtungsweise von Abbildung 2.1 nicht mehr aus. Der Dienst muss aufgeschlüsselt werden in alle beteiligten Geräte und Anwendungen. Abbildung 2.2 zeigt deshalb das „Innere" des Beispieldienstes.

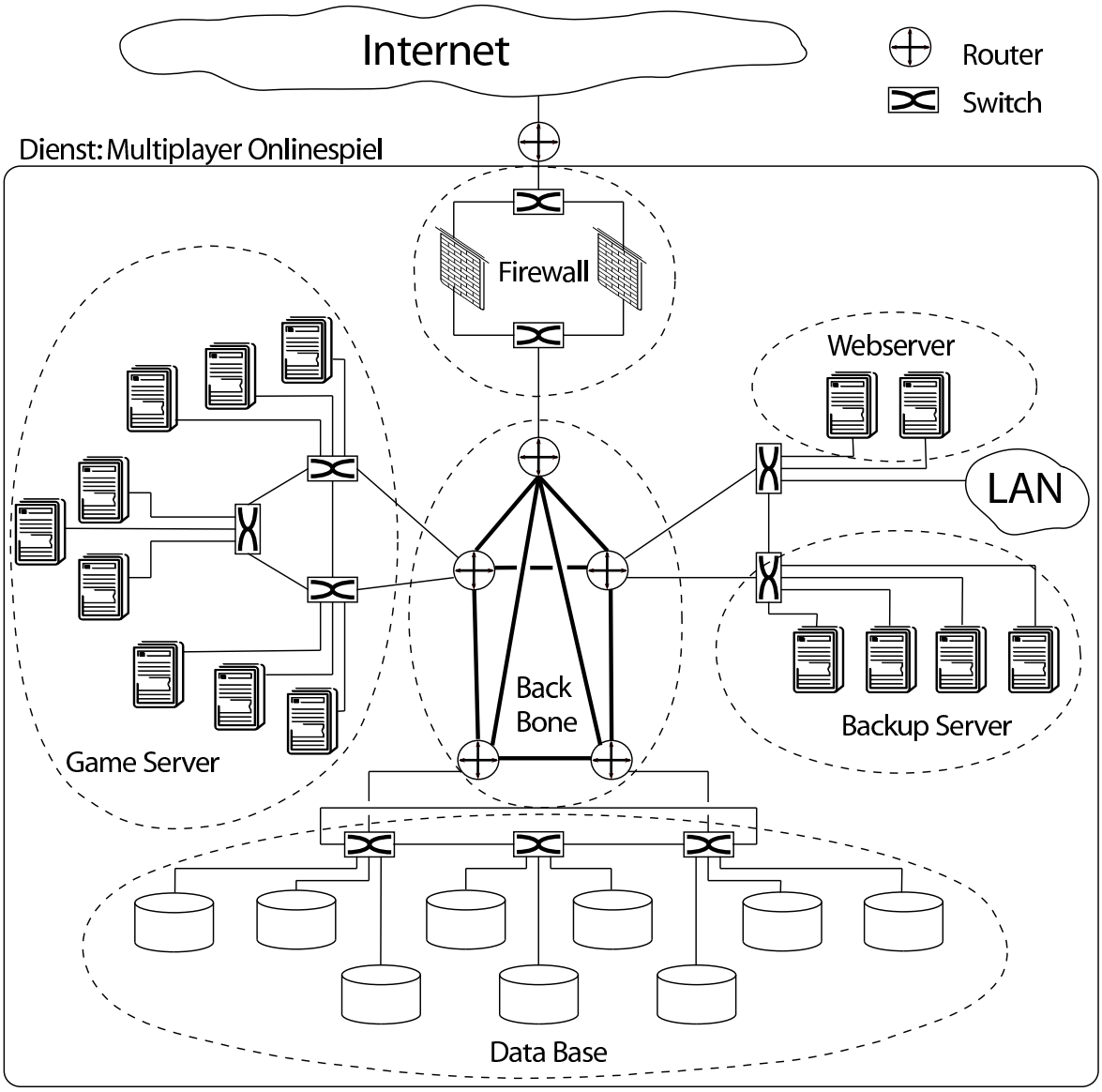


Abbildung 2.2: Szenario Detail

Intern ist der Spieleserver durch ein komplexes Netzwerk unterschiedlichster Komponenten realisiert, die alle als Ressourcen betrachtet werden können (z.B. Router, Switches, Server, ...). Das gesamte Rechnernetz ist in verschiedene Bereiche unterteilt, wovon jeder Bereich seinen eigenen zuständigen Administrator hat. So kennt z.B. der Verantwortliche der Datenbanken jedes Detail seines Teilabschnittes, ist jedoch in Fragen über Firewalls einfach überfordert. Je größer und verteilter das Netz, desto mehr einzelne Bereiche gibt es und desto weniger kann ein Einzelner alles überblicken.

Nun besteht die Aufgabe, Daten über die Ressourcen zu sammeln. Jeder Administrator hat seine Monitoring Anwendung laufen, die ihm über jede seiner Komponenten alle möglichen Attribute abfragt, sie in einer Tabelle auflistet und vielleicht noch einen Graphen über deren Verlauf anfertigt. Dabei geht der Zusammenhang zum Dienst völlig verloren, denn jede Ressource steht jetzt für sich alleine da. Außerdem besteht gar nicht die Möglichkeit, größere Zusammenhänge zu beobachten, weil ja nur der jeweilige eigene Bereich betrachtet wird. Eine solche Art der Datenbeobachtung nennt man **Ressourcen Monitoring**. Die Ressourcenwerte werden ohne jegliche Aggregation untereinander gesammelt und für sich ausgewertet. In Abbildung 2.3 ist diese Variante noch einmal graphisch dargestellt.

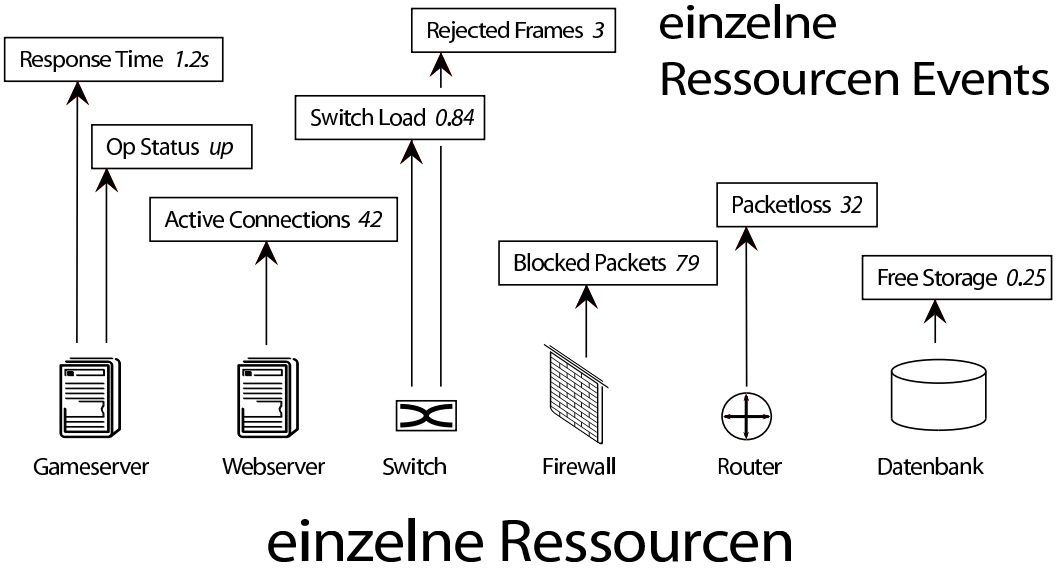


Abbildung 2.3: Ressourcen Monitoring

Mit den so gewonnenen Ressourceninformationen bezweckt man meistens eines von folgenden zwei Monitoringkonzepten. Einmal werden die Ressourcen beobachet, wenn ein Problem bei der Performance besteht. Läuft eine Komponente zu langsam, wird untersucht, woran es liegen könnte. Dazu werden die Tabellen der betreffenden Ressource ausgewertet, bis das Problem idendifiziert ist, und behoben werden kann. Diese Vorgehensweise wird im Fehlermanagement verwendet, und ist meistens iterativ, da ein Bottleneck auch nur das nächste verstecken kann. Die zweite Variante ist ein kontinuierliches Monitoring , um ein großes Gesamtbild einzelner Ressourcen zu bekommen. So wird versucht, frühzeitig Engpässe zu erkennen, und ein Problem im Vorhinein zu vermeinden.

Zusammenfassend kann man sagen, dass beim Ressourcen Monitoring Fehlermeldungen

generiert werden, wenn ein einzelner Zustand einer Ressource nicht mit dem erforderlichen Grenzwert übereinstimmt, bzw. ihn überschreitet. Es ist gut geeignet, um einzelne Geräteausfälle oder Engpässe zu erkennen und schnell zu beheben. Um allerdings Dienstgüte-Verletzungen zu erkennen ist diese Methodik fehl am Platz, da immer nur gesondert eine Ressource betrachtet wird. Es wird nun versucht, eine andere Art von Beobachtung, das sogenannte **Dienstmonitoring**, zu erreichen, welche der eingangs erwähnten Dienstsicht auf die Ressourcen entspricht.

2.1.2 Dienstmonitoring

Bisher war es so, dass der Benutzer zahlreiche einzelne Ressourcenwerte sieht. Er möchte aber erkennen, ob der Dienst zur Zufriedenheit läuft, oder nicht. Die Informationen zu den einzelnen Ressourcen müssen demnach zusammengefasst und aggregiert werden, um für ein bestimmtes Dienstmerkmal ein Gesamtbild zu formen.

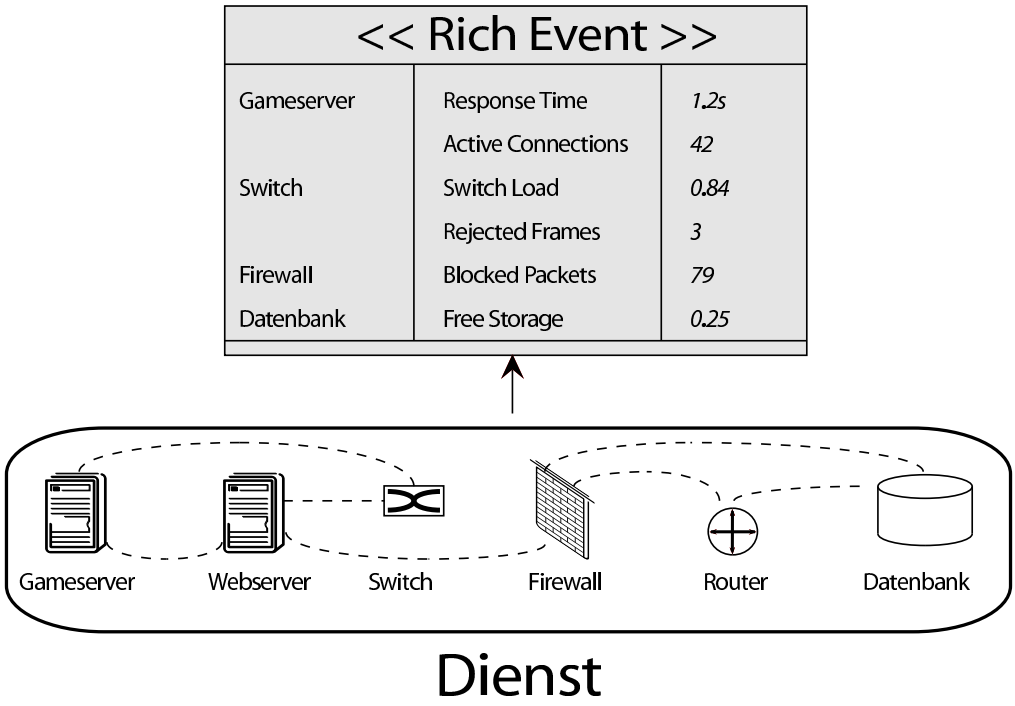


Abbildung 2.4: Dienst Monitoring

In Abbildung 2.4 wird so eine Aggregation gezeigt. Die Ressourcen, von denen die Werte kommen, werden jetzt nicht mehr als einzelne Entitäten gesehen, sondern zu einem Dienst zusammengefasst. Hierbei können durchaus Ressourcen aggregiert werden, die nicht in ein und demselben Bereich liegen. Die Dienstsicht wird also in den Vordergrund gerückt. Dementsprechend sind keine einzelnen Ressourcenereignisse vorhanden, sondern ein zusammengefasstes, „angereichertes" Ereignis, welches in der später vorgestellten Architektur als *„Rich Event"* bezeichnet wird. Dieses Ereignis beschreibt nicht mehr nur die Ressourcen allein, denn durch die Aggregation wird ihm der Sinn der bestimmten Dienstanforderung

übertragen.Zwei einfache Beispiele sollen den Unterschied zwischen einem Ressourcenereignis und einem Rich Event noch einmal herausstellen.

Beispiel 1
Gegeben seien fünf Server, die alle das Attribut Serverlast besitzen. In einem Dienst wird nun jeder dieser fünf Server gleichermaßen beansprucht. Nun zeigt einer dieser Server eine ungemein hohe Auslastung, die schon weit über dem Grenzwert liegt. Ein einzelnes Ressourcenereignis würde nun Alarm auslösen, da es den betroffenen Server ohne Zusammenhang sieht. Ein Rich Event dagegen, welches die Auslastung des Dienstes beschreibt, wird davon nicht gestört, da u.U. die Durchschnittslast aller fünf Server betrachtet wird. Ist einer davon temporär über einem Wert, ist das für den Dienst noch nicht kritisch.

Beispiel 2
Dieses Beispiel ist nun so konstruiert, dass erst bei zwei verknüpften Werten ein Alarmfall eintritt. Auf einem Server können sich mehrere Clients anmelden. Fährt der Server mit einer Auslastung von 90%, es sind aber immerhin 30 Benutzer angemeldet, ist kein Problem vorhanden. Sinkt allerdings die Benutzerzahl auf zwei, ist eine Auslastung von 90% schon als kritisch anzusehen, weil irgendwo zuviel Last generiert wird, was zwei einzelne Benutzer gar nicht schaffen sollten. Wenn die Ressourcen einzeln berachtet werden würden, wäre gar kein Grenzwert definierbar. Weder eine Last von 90% wäre schlimm (weil es auch viele Benutzer sein können), noch ist eine geringe Benutzeranzahl ein Alarmfall.

2.1.3 Allgemeine Anforderungen

An eine entsprechende Monitoring Anwendung werden natürlich gewisse Anforderungen gestellt. Im Besonderen sind dies:

- **Dienstmonitoring:** Ressourcen müssen untereinander aggregierbar sein, um zusammenhängende Aussagen über die Dienstgüte und den Dienst selbst machen zu können.
- **Flexibilität:** Bei gleichbleibendem Rechnernetz und Ressourcen muss schnell auf Änderungen in der Dienstsicht reagiert werden können. Neue Aggregationen und Bedingungen müssen einfach und schnell formuliert werden können.
- **Vielfältige Anwendung:** Die Architektur sollte eine Vielzahl von Anwendungsmöglichkeiten unterstützen, und nicht auf einzelne, wenige Strukturen beschränkt sein. Dienstmonitoring soll also möglich sein, egal, wie die Ressourcen verteilt sind.
- **Offenheit:** Der Benutzer kann seine persönliche Anwendung durch hinzufügen von neuen Bestandteilen schnell erweitern, und auf sein spezielles Problem maßschneidern.
- **Transparenz:** Es darf keine Rolle spielen, wo sich die Ressourcen befinden, die für die Dienstsicht aggregiert werden müssen. Sie können entweder alle im selben Netz oder irgendwo verteilt liegen. Der Benutzer muss alle gleichermaßen gut ansprechen können, und sollte, während er die Architektur benutzt, die lokale Verteilung der Ressourcen gar nicht bemerken.

- **Ausfallsicherheit:** Sollten Datenquellen plötzlich wegfallen, was bei einem verteilten System durchaus einmal passieren kann, muss die Anwendung dennoch stabil bleiben und warten können, bis die entsprechenden Werte wieder verfügbar sind.
- **Beschreibbarkeit:** Aggregationen und Aggregationsvorschriften müssen formuliert werden können. Es soll möglich sein, Attribute von Ressourcen zu beschreiben und mit Operatioen zu verknüpfen, um Bedingungen definieren zu können. Dabei muss auch der zeitliche Aspekt berücksichtigt werden können.

2.2 Zusammenfassung

Das hier dargestellte Szenario stellt die Vorzüge, aber auch die Schwächen von Ressourcen Monitoring dar. In der Zeit der Internet-Dienste und Webservices ist es wünschenswert, eine Monitoring Architektur zu haben, die nicht nur die Sicht auf einzelne Ressourcen ermöglicht, sondern auch einen Dienst als Ganzes sehen kann, um Aussagen über Dienstgüte (Quality of Service) und Dienstvereinbarungen (Service Level Agreements) treffen zu können.

Dazu ist es wichtig, Ressourcen unabhängig von der lokalen Verteilung aggregieren zu können. Es werden also anstelle von einzelnen Ressourcenereignissen mit Daten angereichterte Ereignisse, sogenannte Rich Events, betrachtet und ausgewertet. Außerdem wurden allgemeine Anforderungen erläutert, die solch eine Monitoring Architektur erfüllen muss.

Kapitel 3

State of the Art

Monitoring ist eine enorm wichtige Vorraussetzung für Management. Da in der heutigen Zeit Netz- und Systemmanagement so wichtig geworden ist, gibt es dementsprechend zahlreiche Anwendungen und Arbeiten, die sich mit dem Thema Monitoring auseinandersetzen. Dieses Kapitel soll die bisherigen Ansätze zu dienstorientiertem Monitoring herausarbeiten und bereits vorhandene Lösungen diskutieren.

Dazu wird zunächst ein kleiner Überblick über Grundlagen des Netzmanagements gegeben und wichtige Standards und Begriffe werden erläutert. Danach wird auf verschiedene existierende Sprachkonstrukte eingegangen, ob sie den Anforderungen der Dienstbeschreibung auf Monitoringebene gerecht werden, und verschiedene Anwendungen werden bezüglich Tauglichkeit für Dienstmonitoring bewertet. Zuletzt wird die, bereits in der Einleitung erwähnte Monitoring Architektur des Lehrstuhls von Prof. Hegering in ihrer Gesamtheit vorgestellt, wovon diese Arbeit nur einen Teil realisiert.

3.1 Begriffsbestimmung und Grundlagen zu Netzmanagement und Monitoring

Zunächst sollen die beiden Ausdrücke Management und Monitoring voneinander abgegrenzt werden. Sie werden häufig in einem Zug miteinander erwähnt, sind jedoch zwei unterschiedliche aber voneinander abhängige Begriffe.

Das **IT-management** umfasst die Gesamtheit aller Aktivitäten und Möglichkeiten, um den effektiven Einsatz eines verteilten Systems beziehungsweise seiner Dienste und Anwendungen sicher zu stellen (nach [HAN 99]). Man versteht darunter Planung, Verwaltung, Betrieb und Kontrolle bzw. Überwachung von IT-Netzen und Telekommunikationsnetzen. Die verschiedenen Aufgaben des Managements lassen sich zu Gruppen zusammenfassen, welche die Management-Funktionsbereiche definieren. Das Modell der funktionalen Bereiche (FCAPS) von OSI[1] versucht, diese in fünf Schlüsselkategorien einzuteilen. So können die

[1] Open Systems Interconnection

Aufgaben in Fehler-, Performance-, Konfigurations-, Abrechnungs- und Sicherheitsmanagement gegliedert werden.
Eine Vorraussetzung für das Management ist das Monitoring, denn ohne Komponenten und Anwendungen beobachten zu können, kann man auch keine Entscheidung über die Steuerung fällen.

Monitoring ist wichtig für jedes technische Management. Es bezeichnet das Beobachten und Überwachen von Systemkomponenten, wobei Informationen und Daten über die Zustände der jeweiligen Ressourcen gesammelt, und an die Managementanwendung zum Auswerten gegeben werden. Das Monitoring bemerkt einen möglichen Fehler, worauf das Management erst aktiv wird, und versucht, diesen wieder zu beheben. Monitoring ist daher ausschließlich für die Datensammlung und Datenpräsentation zuständig. Das Augenmerk dieser Arbeit liegt beim Dienstmonitoring, d.h. bei der Überwachung und Datenaggregation von Dienstinformationen.

Ressourcen-Informationen müssen modelliert werden, um eine Basis für einen möglichen Zugriff auf diese zu haben. Es werden Managementobjekte definiert, die jeweils eine Systemkomponente beschreiben, welche aus Managementsicht zu verwalten und zu überwachen ist. Diese *Managed Objects* abstrahieren die einzelnen Ressourcen eines Netzes und werden in einer *Management Information Base* (MIB) zusammengefasst. Eine MIB ist also eine Sammlung aller Daten und Informationen einer Ressource, die für Monitoringfragen interessant sein können. In dieser Arbeit wird daher versucht, die Daten aus einer Ressourcen-MIB zu aggregieren und so eine neue Beschreibungsweise für Dienste zu schaffen.

3.1.1 Informationsmodellierung

Es existieren verschiedene Informationsmodelle zur Beschreibung solcher MIB's. Die wichtigsten sind wohl die Internet-MIB der IETF[2] und das OSI-Informationsmodell. Diese zwei Modelle werden im Folgenden kurz vorgestellt, ebenso wie die zwei aktuelleren Ansätze *Common Information Model* (CIM) [Cim05] der DMTF[3] und *Shared Information/Data Model* (SID) [Sid06], eine Entwicklung des TeleManagement Forums.

Internet-Management

Das Modell für das Internet-Management wird in [RFC1155] festgelegt. Daraus geht hervor, dass jedes Managed Object eindeutig identifizierbar in einer hierarchischen Namensstruktur, dem sogenannten MIB-Baum, angeordnet ist und zur Beschreibung der Management Informationen die Sprache ASN.1[4] verwendet wird [HAN 99].

Der MIB-Baum ist einzigartig und wird weltweit durch die ISO[5] verwaltet. Der Beginn des MIB-Baums wird daher durch den Knoten des Baums „iso(1)" gebildet (vgl. auch Abbildung 3.1). Da der Namensbaum in der Praxis leicht handhabbar sein muss, können die ein-

[2] Internet Engineering Task Force
[3] Distributed Management Task Force
[4] Abstract Syntax Notation One
[5] International Standardization Organisation

zelnen Teilnamen durch Nummern dargestellt werden, die bei jedem einzelnen Knoten mit angegeben sind, z.B. „iso(1)" im Bild. So kann anhand der Nummern innerhalb des Baumes navigiert werden.

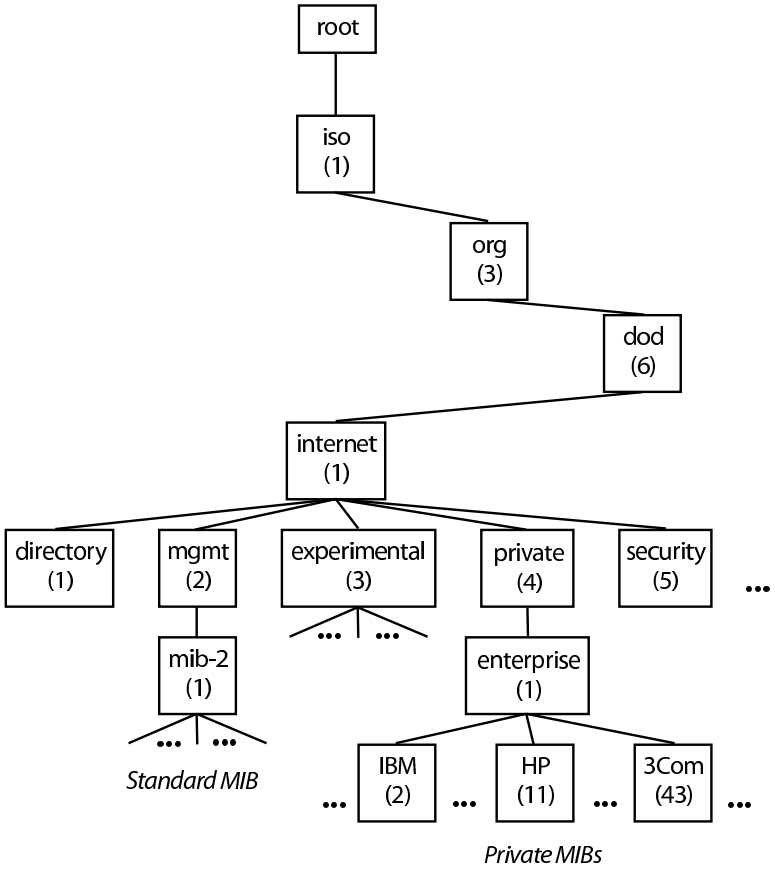


Abbildung 3.1: Baumstruktur einer Management Information Base [HAN 99]

Wie in Abbildung 3.1 zu sehen ist, beinhaltet die Internet-MIB alle für das Internet wichtigen Komponenten. Sie wurde in [RFC1155][6] bzw. [RFC1442] beschrieben, und hätte demnach folgende Kennung „iso(1).org(3).dod(6).internet(1)", wobei im technischen Einsatz nur die Ziffernfolgen „1.3.6.1" angegeben werden. Mehrere solcher untergeordneten MIB's sind in RFC's beschrieben, unter Anderem die MIB-2, die in [RFC1213] definiert wird und von allen Netzwerkkomponenten unterstützt wird. Unter dem Abschnitt „private(4).enterprise(1)" können auch verschiedene Unternehmen die privaten MIBs ihrer eigenen Produkte und Geräte definieren und bei der IANA[7] registrieren lassen. So werden die Schnittstellen für spätere Management Anwendungen genau spezifiziert, die dann auf diese Geräte zugreifen müssen.

Mit Hilfe einer MIB sind nun verschiedene Operationen möglich, die beim Netzmanagement benötigt werden. Das sind unter Anderem:

- Das Aktivieren oder Deaktivieren der Ressourcen bzw. Managementobjekte.

[6] Request For Comment
[7] Internet Assigned Numbers Authority

- Das Erfassen regelmäßiger Daten, welche den Netzwerkzustand beschreiben.
- Das Erfragen und Ändern von Netzwerkparametern, die für den Betrieb erforderlich sind.

RMON, WWW Service und Application MIB

Drei spezielle MIB's des Internet Managements sollen noch einmal gesondert erwähnt werden, weil hier versucht wird, einen Weg in die Richtung Dienstinformationsmodellierung einzuschlagen. *Remonte Network Monitoring* (RMON) ist ein Standard, um bei Netzwerkgeräten statistische Daten abzufragen, jedoch keine direkte Dienstmanagement-MIB. Details zur RMON MIB wurden von der IETF in [RFC2819] festgelegt. Hierbei werden eine Reihe von Managed Objects definiert, die nicht nur einfache Werte repräsentieren, sondern (komplexere) statistische Berechnungen auf diesen bereits ausgeführt haben. So wird versucht, Daten für Fehler- und Performance Management in verteilten Netzen zur Verfügung zu stellen, die mittels SNMP abgefragt werden können.

Die RMON MIB ist dabei noch einmal in verschiedene Gruppen unterteilt, wobei jede dieser Gruppen eine Menge bestimmter Attribute mitbringt, welche z.B. Aufzeichnungen zu Paketströmen oder statistische Daten zu den Hosts im Netz sind. Allgemeint scheint sich die RMON MIB sehr auf Netzverkehr zu beschränken. Mit RMON-2 [RFC2021] wird die bisherige RMON MIB mit der Möglichkeit, auch Analysen in der Anwendungsschicht durchzuführen, erweitert. Es existieren auch noch einige andere spezielle Erweiterungen zur RMON MIB. Eine IEFT Working Group ist damit beschäftigt, in einem RMON Framework alle diese „Untergruppen" zusammenzufassen [RmWo06]. Zu erwähnen sind auch noch die *Application Management MIB* [RFC2564] und die *WWW Service MIB* [RFC2594]. Während die Application MIB eine prozeßorientierte Sicht auf Anwendungen präsentiert, wird sie von der Webservice MIB um eine „Dienstsicht" erweitert. Allerdings beschränkt sich diese Dienstsicht auf Informationen zu Aufrufstatistiken des jeweiligen Webservices. Die Ansätze beider MIB's gehen in die richtige Richtung, allerdings sind sie noch zu unflexibel.

Wenn diese MIB's nun im Kontext der hier diskutierten Dienstsicht betrachtet werden, kann man sagen, dass einzelne (statistische) Informationen bestimmt nützlich sind. Dennoch sind es nur einzelne, mittels SNMP abfragbare Werte, die nicht zu einem bestimmten Dienst aggregiert sind (vgl. Abschnitt 3.1.2). Können sie auch nicht, da die MIB's allgemein formuliert sein müssen, und nicht auf spezielle Dienste zugeschnitten werden können. Das ist auch der Schwachpunkt der ganzen MIB's. Wird für eine spezielle Dienstsicht ein statistischer Wert benötigt, der von keiner MIB geliefert werden kann, besteht bei diesem Ansatz keine Möglichkeit, diesen zu erhalten. Allgemein gesagt sind die MIB's zu statisch und zu unflexibel, um sie alleinstehend als Dienstsicht anzubieten. Um Datenwerte zur weiteren Aggregation bereitzustellen, sind diese Schnittstellen für das in dieser Arbeit behandelte Thema jedoch gut geeignet.

OSI-Informationsmodell

Das OSI-Informationsmodell wird im Dokument *Structure of Management Information* (SMI) in [ISO 10165] beschrieben. Hierbei wird ein objektorientierter Ansatz gewählt. Die Managed Objects (MO), als Abstraktion realer Ressourcen, bieten an ihrer Schnittstelle Methodenaufrufe an, um die im Objekt enthaltenen Attribute zu manipulieren. Das innere Verhalten eines Managed Objects ist dabei nur an einer wohldefinierten Schnittstelle, der MO-Boundary sichtbar. Mehrere Objekte mit gleichen Eigenschaften können auch zu Managed object Classes (MOC) zusammengefasst werden. Die Managementobjekte eines einzelnen OSI-Systems bilden somit die Management Information Base (MIB).

Charakteristisch für einen objektorientierten Ansatz ist die Vererbungseigenschaft. So kann eine MOC als Unterklasse von einer oder mehreren Oberklassen definiert werden. Dadurch erbt sie alle Eigenschaften der Oberklasse und kann diese noch erweitern. Außerdem können Enthaltenseinshierarchien aufgebaut werden, um zusammengesetzte Managed Objects zu modellieren. Diese Hierarchie legt einen Objektbaum fest, dessen Wurzel das System-MO ist, und in den beliebig weitere Managed Objects eingehängt werden können.

Die Relevanz für diese Arbeit ist genau die selbe wie beim Internet-Management, denn das Modell lässt sich theoretisch in das OSI-Modell überführen [Kell 98]. Insgesamt ist es ein gutes Modell, aber da es nicht implementiert wurde, in der Praxis unbrauchbar. Einzelne Ressourcen können gut modelliert werden, allerdings besteht weiterhin das Problem, wie Dienstinformationen flexibel genug in einem Managed Object zusammengefasst werden können.

Common Information Model

Das Common Information Model (CIM) wurde von der DMTF entwickelt. Es basiert auf UML und schafft ein objektorientiertes Modell zur Beschreibung eines Rechnersytems. Die verschiedenen Komponenten werden mit Klassendiagrammen und einer Menge von Attributen beschrieben. Für die automatisierte Verarbeitung und den Austausch von Klasseninformationen wurde das *Managed Object Format* (MOF) spezifiziert, das man wegen der objektorientierten Ausrichtung auch als eine *Interface Description Language* bezeichnen kann.

CIM ist in verschiedene Schemata aufgeteilt, welche alle eine unterschiedliche Anzahl von Klassen definieren. So enthält z.B. das Core Schema Objekte, die in allen Arten des System-Managements benötigt werden und ist relativ klein und unveränderlich. Dieses wird erweitert vom Common Schema, welches mehrere Teile, die jeweils für einen bestimmten Bereich des Managements zugeschnitten sind, enthält. So wird eine leicht erweiterbare Klassenhierarchie aufgebaut. Das Thema Dienstmanagement wird nur von einer Arbeitsgruppe bearbeitet, und umfasst nur diejenigen Klassen und Attribute, welche aus den verschiedenen Komponenten direkt abgefragt werden können.

Allerdings befindet sich CIM noch immer in der Entwicklung. Abgesehen vom Core Schema werden viele Unterschemata teilweise mit jeder neuen Version gravierend geändert, so dass sich nicht darauf verlassen werden kann, die jetzt benutzten Klassen und Attribute im

folgenden Release noch vorzufinden. Außerdem findet sich hier das gleiche Problem wie bei den anderen Informationsmodellen. Es können zwar einzelne Ressourcen modelliert werden, aber eine Darstellung von Diensten bzw. Dienstinformationen ist nur unzureichend möglich und würde sehr komplexe Assoziationen erfordern. Weiterführende Informationen zu CIM kann man in [Cim05] nachlesen. Auch [Jähn 03] bietet eine umfangreiche Übersicht.

Shared Information/Data Model

Das *Shared Information/Data Modell* (SID) [Sid06] ist eine Entwicklung des TeleManagement Forums und Teil des Next Generation Operations Support Systems (NGOSS)- Frameworks. Es stellt ebenfalls ein objektorientiertes Modell dar, und weist Ähnlichkeiten zu CIM auf. Es wird ein Top-Down Ansatz verfolgt, der das Modell in verschiedene Level aufteilt, ähnlich zu eTOM [eTOM06]. Die oberen beiden Level sind schon relativ vollständig entwickelt, doch der Bereich der Attribute ist noch immer in der Diskussion. So gibt es eine Reihe von Klassendiagrammen, die ohne Methoden nicht wirklich anwendbar sind.

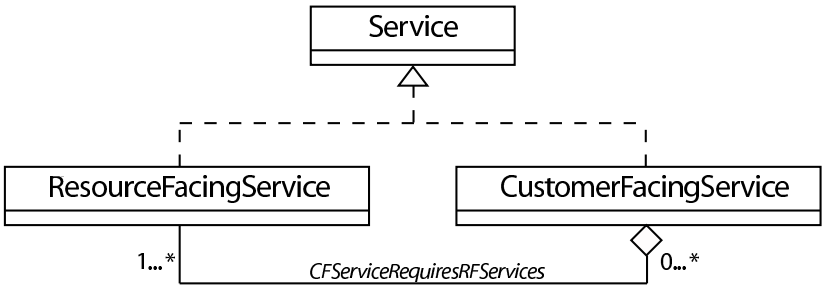


Abbildung 3.2: Service mit zwei Sichtweisen bei SID [Bitt 05]

Ein Punkt ist allerdings sehr interessant für diese Arbeit. Es ist die Unterteilung von Services in zwei verschiedene Sichten, die auch unterschiedlich modelliert werden. Einmal ist ein Service demnach ein *RessourceFacingService*, der sich mit der technischen Seite beschäftigen soll. Das andere ist die dem Benutzer zugewandte Seite *CustomerFacingService*. In Abbildung 3.2 ist diese Definition grafisch dargestellt. Diese Trennung macht das Modell zwar komplexer, da man beide Sichten gegenseitig abbilden muss, doch es könnte genau solch eine Dienstsicht erreicht werden, wie hier in dieser Arbeit gefordert ist. Allerdings ist auch dieses Modell noch in der Entwicklung, also noch nicht verfügbar.

3.1.2 Kommunikationsmodelle

Um mit den Managed Objects der Informationsmodelle zu interagieren, wurden verschiedene Protokolle entwickelt. Da dies für die vorliegende Arbeit in Bezug auf Datensammlung und Aggregation von Bedeutung ist, wird hier noch einmal genauer darauf eingegangen.

Für das Internet-Management ist dies das *Simple Network Management Protocol* (SNMP). OSI hat ebenfalls ein eigenes Protokoll für den Nachrichtenaustausch zwischen Managementeinrichtungen definiert, das *Common Management Information Protocol* (CMIP) bzw.

CMIP over TCP (CMOT), siehe [RFC1095] und [RFC1189]. Aufgrund einfacherer Handhabung, größerer Akzeptanz und Verbreitung als bei den beiden OSI-Protokollen wurde 1990 von der IAB[8] das SNMP als Standard Protokoll mit Empfohlen Status festgesetzt [RFC1157]. Deswegen soll hier die Funktionsweise von SNMP noch einmal genauer beschrieben werden.

Das Kommunikationsmodell des Internet Managements ist sehr einfach. Nach diesem Modell teilt sich ein Netzwerk in die zentrale Managementstation und die einzelnen, u.U. räumlich getrennten Netzwerkkomponenten auf. Auf der Station befindet sich eine Managementanwendung, deren Aufgabe es ist, die Netzwerkkomponenten zu überwachen und zu steuern. Die Komponenten, wie z.B. Server, Switches, Router oder Hosts haben Managementagenten, die die Managementfunktionen ausführen. Mittels SNMP findet zwischen der Managementstation und den Netzwerkkomponenten eine Kommunikation statt, wie es auch in Abbildung 3.3 zu sehen ist. Diese Kommunikation beinhaltet die Übertragung relevanter Managementdaten, die von den Netzwerkkomponenten zur Verfügung gestellt und zur Managementstation gesendet werden müssen, ebenso wie die Steuerungsdaten für die Netzwerkkomponenten.

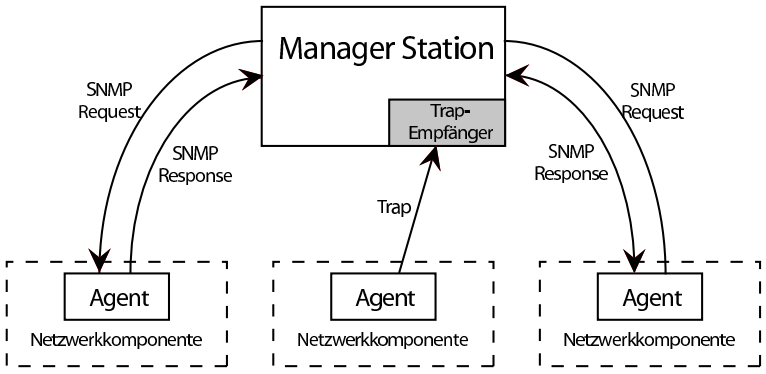


Abbildung 3.3: Das Prinzip der SNMP-Kommunikation

Welche Komponente welche Daten zur verfügung stellt, ist in ihrer MIB festgelegt, wo die Schnittstelle der jeweiligen Managed Objects definiert ist. Beispiele solcher Daten wären physikalische Attribute der jeweiligen Komponente, wie Netzwerkadressen oder Protokollparameter. Der Agent auf dem angesprochenen Gerät reagiert auf die SNMP Anfragen des Managers, wodurch entweder die erforderlichen Parameter neu gesetzt werden, oder dem Manager die gewünschten Informationen geschickt werden. Mit dem Neusetzen der Parameter wird die Netzwerkkomponente schließlich gesteuert. Ein Trap-Empfänger bedeutet, dass der Agent von sich aus auch Ereignisse (Traps) generieren und an die Managementstation senden kann. SNMP stellt nur die Regeln für die Kommunikation auf. Welche Art von Werten auf den jeweiligen Komponenten zur Verfügung stehen, ist egal. Wenn eine MIB definiert ist, kann mit SNMP darauf zugegriffen werden.

Es existieren einige Erweiterungen und neuere Versionen zum SNMP Protokoll. Hauptproblem des ursprünglichen Protokolls war der schlechte bzw. nicht vorhandene Sicherheitsa-

[8] Internet Activities Board

spekt [Wiki 06]. Dies wurde versucht mit SecureSNMP [RFC1352] und der darauffolgenden Version SNMPv2 (u.A. [RFC1441]) bzw. SNMPv3 zu ändern.

3.2 Untersuchung vorhandener Spezifikationskonzepte

In diesem Abschnitt werden verschiedene Spezifikationen, Festlegungen und Veröffentlichungen bezüglich ihrer Relevanz für Dienstaggregation und Dienstmonitoring diskutiert. Es werden drei Sprachen vorgestellt, welche vom Aufbau her alle mit dem Thema verwandt sind, aber dennoch nicht für die gewünschte Aufgabe benutzt werden können.

3.2.1 Resource Description Framework

Das *Resource Description Framework* (RDF) wurde vom W3C[9] entwickelt und steht jedem zur freien Verfügung [KlCa 04]. Es ist eine Sprache, die Informationen über Ressourcen im Internet darstellt, die nicht auf der eigentlichen Webseite enthalten sind. Sie ist besonders dafür gedeignet, Metadaten über Internet-Ressourcen darzustellen, so wie es im Umfeld des Semantic Web benötigt wird [SeWe04]. Solche Informationen sind z.B. Titel, Autor bzw. Erstellungsdatum einer Internetseite, oder Copyright Informationen zu einem Dokument, kurz, verschiedene Attribute der Internet-Objekte. Wenn man den Begriff einer Internet-Ressource verallgemeinert, kann man mit RDF Informationen über beliebige Objekte beschreiben, auch, wenn sie nicht direkt übers Internet zugänglich sind. Es muss nur eine Möglichkeit geben, sie eindeutig im Netz zu identifizieren.

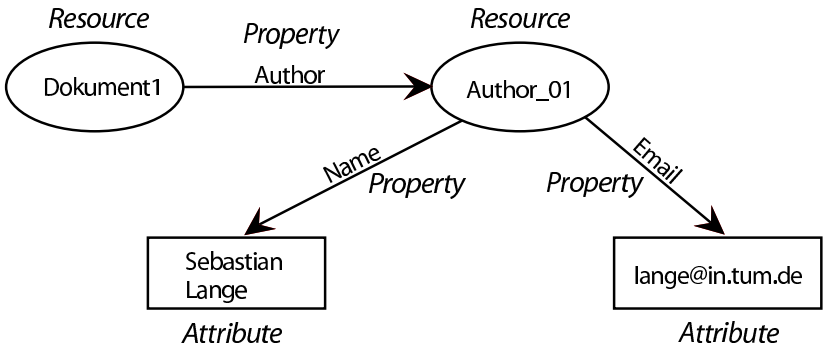


Abbildung 3.4: RDF-Graph einer einfachen Objektbeschreibung

RDF ist für Situationen entwickelt, wo diese Informationen von Andwendungen verarbeitet werden müssen, und nicht nur, um sie irgendwo anzuzeigen. Deswegen stellt RDF einen Rahmen zur Verfügung, um solche Informationen ohne Verluste zwischen Anwendungen transportieren zu können. RDF benutzt URI's[10] zum identifizieren der Objekte, und beschreibt Ressourcen mit einfachen Attributen, deren Werte gesetzt werden können. Es besteht sowohl aus einem grafischen Modell zur Repräsentation der erzeugten Metadaten,

[9] World Wide Web Consortium
[10] Uniform Resource Identifier

als auch aus einer XML-Syntax. Der RDF-Graph besitzt eine Baumartige Struktur, dessen Blätter die einzelnen zugewiesenen Attribute sind.

In Beispiel 3.4 ist ein solcher RDF-Graph zu sehen, während Listing 3.1 die zugehörige XML Notation zeigt. Das Beispiel beschreibt eine Person mit der Bezeichnung „Author_01", deren Name „Sebastian Lange" und Email „lange@in.tum.de" ist, und welche Autor von „Dokument1" ist.

Listing 3.1: XML Syntax zum Beispielgraph in Abbildung 3.4

```
<rdf:RDF>
  <rdf:Description about="Dokument1">
    <s:Author>
      <rdf:Description about="Author_01">
        <rdf:type resource="http://descr.org/schema/Person"/>
        <v:Name>Sebastian Lange</v:Name>
        <v:Email>lange@in.tum.de</v:Email>
      </rdf:Description>
    </s:Author>
  </rdf:Description>
</rdf:RDF>
```

Für diese Arbeit kann gesagt werden, dass mit RDF das Grundgerüst eines Dienstes beschrieben werden könnte. Man könnte definieren, das der Dienst aus diesen und jenen Ressourcen besteht, welche wiederum feste Attribute haben, die einen bestimmten Wert besitzen. Dazu müsste ein Dienst per URI identifizierbar sein, was aber nicht das Problem darstellt. Allerdings ist RDF nicht dafür geeignet, dynamische Werte abzufragen und zu aggregieren. Auch können auf den Attributswerten, die mit RDF-Anfragen gesetzt oder gelesen werden, keine Berechnungen oder Bedingungen formuliert werden. Insgesamt ist RDF zwar brauchbar, um den strukturellen Aufbau eines Dienstes zu beschreiben, allerdings ist es zu unflexibel hinsichtlich der Datenauswertung.

3.2.2 Object Constraint Language

Die *Object Constraint Language* (OCL) ist eine formale Sprache, um Bedingungen und Einsschränkungen an objektorientierten Modellen zu beschreiben. OCL wurde von IBM entwickelt, bevor es dann 1997 von der OMG[11] in die UML[12]1.1 Spezifikation mit aufgenommen wurde. Der neueste Stand der Sprache ist OCL 2.0, ebenfalls in UML 2 enthalten, was inzwischen als Standard für Beschreibung von Einschränkungen in objektorientierten Umgebungen zählt. Mehr dazu kann in [OMG-1] nachgelesen werden.

Mit OCL können durch aussagenlogische Ausdrücke Bedingungen an Objekte geknüpft werden, die dann eingehalten werden müssen. So können Invarianten auf Objekten, genauso wie Vor- und Nachbedingungen an Methoden gekoppelt sein. Da OCL eine reine Spezifikationssprache ist, sind die Auswertungen ihrer Bedingungen garantiert ohne Nebeneffekt für

[11] Object Management Group
[12] Unified Modeling Language

das beschriebene System. Wenn ein OCL-Ausdruck ausgewertet wird, wird nur ein Wahrheitswert zurück gegeben und es kann nichts im Objektmodell geändert werden. Da es keine Programmiersprache ist, können ebenfalls keine Methoden über OCL aufgerufen werden.

OCL wurde untersucht, weil auch in dieser Arbeit Bedingungen und Einschränkungen für eine Dienstbeschreibung benötigt werden. Ist eine Datenaggregation für einen Dienst spezifiziert, müssen die Bedingungen auf dieser Aggregation definiert werden, um Grenzwerte auszudrücken oder Vereinbarungen über die Dienstgüte zu beschreiben. Dies könnte OCL zuverlässig realisieren. Allerdings können mit OCL keine Ressourcen angesprochen, und somit auch nicht aggregiert werden, was das Erzeugen einer Dienstsicht von vorn herein ausschließt. OCL wäre demnach nur als integrierter Bestandteil denkbar, nicht aber allein.

3.2.3 Web Service Description Language

Die *Web Service Description Language* (WSDL) ist eine XML-basierte Sprache, um Webservices und den Zugriff auf diese zu beschreiben. Sie wurde 2001 vom W3C veröffentlicht und ist programmiersprachen- und protokollunabhängig. Es können damit die angebotenen Funktionen, Daten, Datentypen und Austauschprotokolle eines Webservice, also die Kommunikation mit ihm, strukturiert beschrieben werden. Meistens wird sie zusammen mit SOAP[13] verwendet, um einen Webservice im Internet anzubieten (siehe auch [CCMW 01]). Dabei sei darauf hingewiesen, dass ein Webservice nicht gleichbedeutend ist mit dem Diensbegriff, der hier in Kapitel 1 und 2 benutzt wird. Ein Webservice bietet meist vordefinierte, einzelne Funktionen im Internet an, während ein Dienst im Sinne dieser Arbeit ein Bereitstellen von mehreren Leistungen mit einer festgelegten Dienstgüte bezeichnet.

Möchte ein Client einen Webservice benutzen, kann er dessen WSDL-Beschreibung lesen, um zu bestimmen, welche Funktionen auf dem Server verfügbar sind, und welche Eingabedaten sie erhalten müssen. Sechs verschiedene XML Elemente bündeln dazu die Informationen des Webservices. Mit WSDL können also die Funktionalitäten, die ein Webservice anbietet, dargestellt werden. Es wird jedoch nicht gezeigt, aus welchen Komponenten der Webservice aufgebaut ist, um die Funktionen bereitstellen zu können. Es werden mit WDSL demnach keine Ressourcen des Dienstes beschrieben. Es sind auch keine Aggregationsvorschriften vorhanden, da keine Möglichkeit geboten wird, mit Daten des Dienstes zu arbeiten und sie gegebenenfalls miteinander zu verknüpfen. Daher ist WSDL für den Ansatz, der in dieser Arbeit verfolgt wird, nur unzureichend anwendbar.

3.3 Vergleich bestehender Monitoring Architekturen

In Abschnitt 3.2 wurden einzelne Konzepte und Bauteile untersucht, ob sie für Dienstbeschreibungen geeignet sind. Hier werden nun stellvertretend zwei reale Monitoringanwendungen vorgestellt und betrachtet, ob und wie diese Dienstmonitoring und Datenaggregation zu einer Dienstsicht realisieren. Da es unzählige Anbieter für solche Anwendungen gibt, werden hier exemplarisch zum einen das frei verfügbare Open Source Projekt Nagios und

[13] Simple Object Access Protocol

zum anderen das kommerzielle Produkt HP Openview angesprochen. Zuletzt wird detailliert die Idee zur Monitoring Architektur des Lehrstuhls von Prof. Hegering beschrieben, da diese genau die hier gestellten Anforderungen umsetzen soll.

3.3.1 Nagios

Nagios ist ein Monitoringprogramm und unter der GPL[14] frei verfügbar. Es überwacht Rechner, Netzwerkkomponenten und Dienste, ob alles nach den Wünschen des Administrators läuft. Die zu überwachenden Ressourcen werden über Konfigurationsdateien bei Nagios registriert. Danach steht eine Sammlung von Plugins bereit, welche verschiedenste Werte abfragen können. Über- bzw. unterschreitet ein Wert eine vorher vom Benutzer festgelegte Grenze, kann dieser über verschiedene Wege benachrichtigt werden. Außerdem steht eine graphische Oberfläche zum Beobachten der Daten bereit.

Auf der offiziellen Homepage[15] wird Nagios beschrieben als ein *„host, service and network monitoring program"*. Es können also Attribute von einzelnen Geräten abgefragt werden. Allerdings bezieht sich die Definition von *„service"* auf Dienste wie DNS[16], SMTP[17], http-daemons etc. Hier werden keine einzelnen Ressourcen zu einer Dienstsicht aggregiert, sondern nur separat aufgelistet. Nagios unterstützt eine Vater-Kind-Beziehung unter Ressourcen, so dass Fehlerfortpflanzung erkannt werden kann, sonst können Ressourcen nicht weiter miteinander verknüpft werden. Ein Pluspunkt ist allerdings, dass Nagios das Einbinden eigener Plugins unterstützt. So besteht die Möglichkeit, beliebig verschiedene Ressourcen zu beobachten. Genaueres zur Funktionsweise von Nagios ist in dessen Dokumentation [Gals 04] zu finden.

Zusammenfassend kann man sagen, dass Nagios sehr gut geeignet ist, einzelne Ressourcendaten zu beschaffen. Außerdem ist es flexibel, und man kann schnell auf Änderungen im überwachten Netz reagieren. Allerdings kann Nagios diese „Rohdaten" nicht weiter verarbeiten oder aggregieren, doch sie könnten weitergeleitet werden an eine übergeordnete Anwendung. Dazu sei auf Abbildung 3.5 in diesem Abschnitt verwiesen, wo Nagios als ein Beispiel für Adapter wieder auftaucht.

3.3.2 HP OpenView

OpenView ist eine von Hewlett-Packard angebotene Programmsuite für Netzwerkmanagement und Monitoring. Es besteht aus vielen Unterkomponenten, wovon der Network Node Manager für das Sammeln der Ressourcendaten zuständig ist. Openview bietet den gesamten Komfort einer kommerziellen Management Anwendung. Resourcendaten werden schnell und effektiv abgefragt, und in einer produktinternen Datenbank gespeichert. Dort können statistische Berechnungen an den Daten durchgeführt werden. Natürlich ist vom Benutzer konfigurierbar, wann eine Benachrichtigung im Fehlerfall gesendet werden soll. Außerdem

[14] General Public License
[15] Nagios Homepage: http://www.nagios.org/
[16] Domain Name System
[17] Simple Mail Transfer Protocol

sind alle Werte in einer grafischen Benutzeroberfläche zu sehen. Mehr zu den Merkmalen von OpenView und dem Node Manager kann in den Onlinemanuals von HP [HP 97] bzw. [HP 00] nachgelesen werden.

Der Network Node Manager benutzt primär SNMP, um die Ressourcenwerte abzufragen. Es ist also möglich, alle Komponenten zu erreichen, die eine MIB und somit SNMP Anfragen unterstützen. Wird von einer gewünschten Ressource das SNMP Protokoll nicht verstanden, kann der Node Manager an keine Werte gelangen. Darüberhinaus findet auch im Network Node Manager keine Datenaggregation statt. Die Daten werden nur zur weiteren Verarbeitung und Analyse in die Datenbank geschrieben. Somit ist er alleine, genauso wie Nagios, gut geeignet, um an die Primärdaten zu gelangen, aber nicht, um sie weiter zu Datensätzen einer Dienstsicht zusammenzusetzen. Es gibt Ansätze von HP, diese Datensätze zu korrelieren, und so Fragen zu Service Level Agreements zu beantworten. Allerdings außerhalb des Node Managers, was die Investition in ein weiteres Softwareprodukt voraus setzt. Auch muss erwähnt werden, dass OpenView kein Open Source Programm ist, und somit Veränderungen und Erweiterungen nur schwerlich bis gar nicht selbständig eingefügt werden können.

3.3.3 Management Architektur des Lehrstuhls

In diesem Abschnitt wird nun die Monitoring Architektur vorgestellt, die am Lehrstuhl Hegering der LMU München vorgeschlagen wurde (vgl. Abschnitt 1.1), und wovon diese Arbeit einen Teil realisiert. Abbildung 3.5 zeigt die Architektur in ihrem grafischen Modell, welches nun detailliert erklärt wird.

Mit dieser Architektur wird versucht, die im Szenario angesprochene Dienstsicht zu realisieren, d.h. Datenwerte von verteilten Ressourcen zu einer, der Dienstsicht entsprechenden Datenaggregation zusammenzufassen. Dabei ist die Architektur in verschiedene Schichten aufgeteilt, die jeweils den Zustand beschreiben, den die Ressourcendaten in diesem Moment besitzen. Diese Schichten werden im folgenden im Bottom-Up-Ansatz beschrieben.

- **Resource Layer:** Diese Schicht ist die Unterste der Architektur, und zeigt an, wo sich die Ressourcen mit ihren Rohdaten befinden. In einem verteilten Netz zählen verschiedenste Geräte oder Dienste als Ressource. Diese können räumlich verteilt sein und auf unterschiedlichen Quellen zur Verfügung stehen. Der Benutzer bzw. die Architektur hat somit einen großen Pool an möglichen, abfragbaren Ressourcenwerten. Die Aussage hier ist, dass es viele unterschiedlichste Ressourcen gibt, die möglicherweise für den Dienst von Belang sein können.

- **Platform Specific Layer:** Auf dieser Schicht werden lokale Lösungen implementiert bzw. verwendet, um die Ressourcendaten aus ihren Quellen zu extrahieren. Diese Zugriffsmethoden können je nach Betriebssystem oder Gerät unterschiedlich sein. Es können benutzerspezifische, spontane Lösugnen verwendet werden, genauso wie vorhandene Monitoring Werkzeuge, um die Datenwerte zu bekommen. Wie im Bild angedeutet, können das auf der spontanen Seite z.B. verschiedene Scripts sein und auf der Seite der integrierten Lösungen das zuvor erwähnte Nagios oder der Node Manager (vgl. Abschnitt 3.3.1 bzw. 3.3.2).

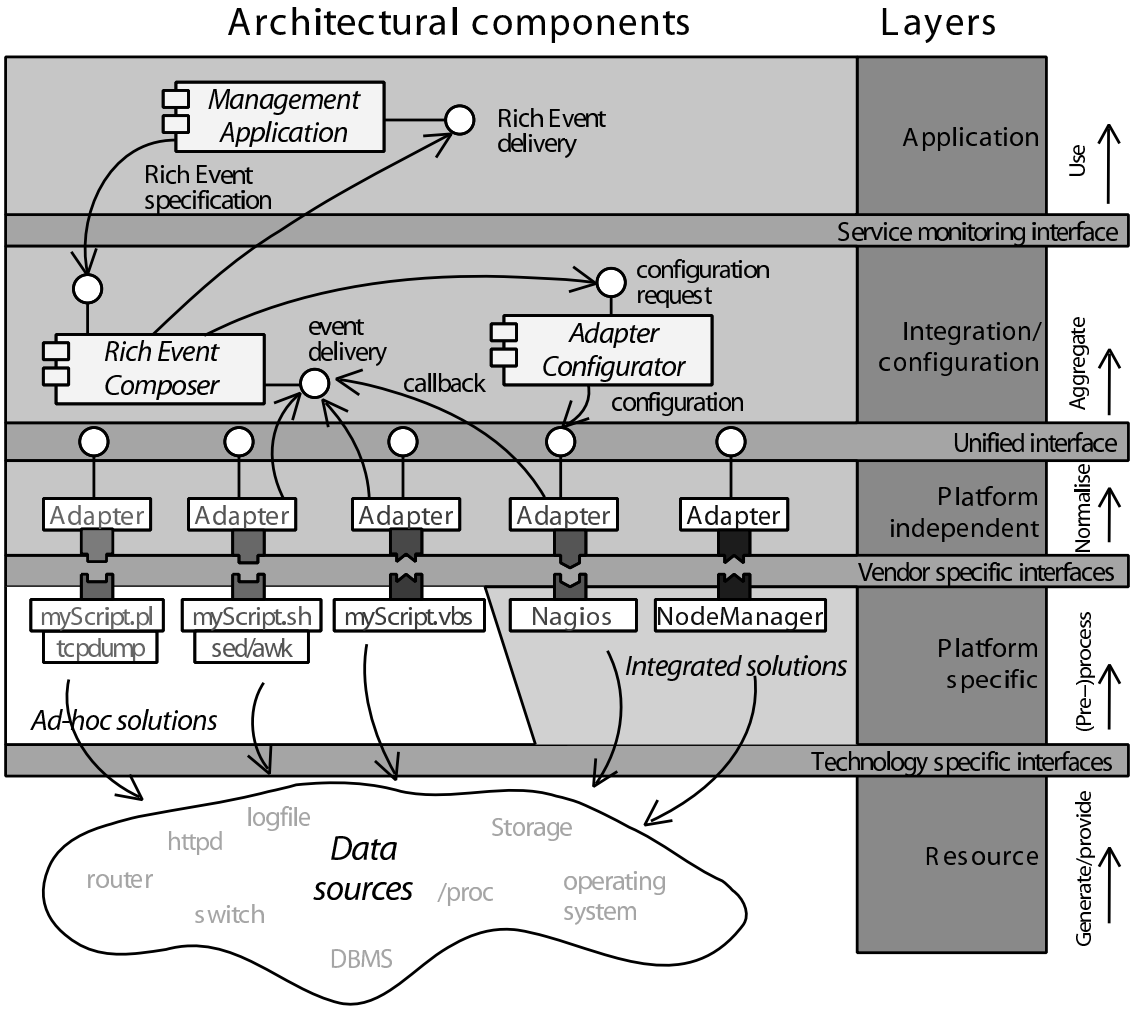


Abbildung 3.5: Die Monitoring Architektur des Lehrstuhls

- **Platform Independent Layer:** Hier werden die Daten in ein normalisiertes Format gepackt, denn auf dieser Schicht befinden sich die Adapterkomponenten. Sie kapseln den Zugriff auf die Ressourcen und stellen nach oben hin alle eine einheitliche Schnittstelle zur Verfügung. So muss eine übergeordnete Anwendung nur mit den Adaptern kommunizieren, dadurch nur eine einheitliche Schnittstelle implementieren und nicht verschiedenste Zugriffsprotokolle der darunterliegenden Schicht verstehen. Nach unten hin benutzen die Adapter, je nach Ressource, die unterschiedlichen lokalen Lösungen der vorhergehenden Schicht. Wird ein Adapter konfiguriert, um einen bestimmten Ressourcenwert abzufragen, so „übersetzt" er diese Anforderung für die darunter benutzten Werkzeuge. Diese liefern dann den entsprechenden Wert der Ressource zurück. Der Adapter wandelt diese spezifische Antwort wieder in die normalisierte um, und schickt den Wert weiter an die übergeordnete Anwendung.
- **Integration & Configuration Layer:** Auf dieser Schicht arbeiten der Rich Event Composer und der Adapter Configurator. Ein Rich Event ist ein zusammengesetztes Ereignis, was von einer zuvor spezifizierten Aggregation von Ressourcen erzeugt

wird. Diese Aggregation stellt eine, vom Benutzer festgelegte, Sicht auf einen bestimmten Dienst dar, und beinhaltet all die Ressourcen, die für diesen Dienst erforderlich sind. Der Composer ist dafür zuständig, die Anweisungen der Datenaggregation einzulesen, und die einzelnen Ressourcen zu identifizieren. Ist das geschehen, wird der Adapter Configurator aktiviert, um die Adapter nach diesen Werten zu fragen. Wie der Name schon sagt, konfiguriert er die Adapter, indem er auf ihre einheitliche Schnittstelle zugreift, und so die gewünschten Ressourcenwerte anfordert. Die Adapter liefern dann die gewonnenen Werte an eine zentrale Station des Composers zurück, wo sie weiteren Berechnungen unterzogen oder an bestimmte Bedingungen geknüpft sein können. Die verschiedenen Adapter sowie Composer und Configurator befinden sich höchstwahrscheinlich auf räumlich getrennten Systemen, so dass hier über eine geeigente Kommunikationsplattform nachgedacht werden muss.

- **Application Layer:** Am obersten Ende der Architektur befindet sich die Management Anwendung. Hier werden eingangs die Datenaggregationen in einer dafür vorgesehenen Sprache spezifiziert, und auch die ankommenden Rich Events weiter verarbeitet, die nun die zusammengefassten Ressourcenwerte enthalten.

Die vorliegende Arbeit kann man in das Integration & Configuration Layer einordnen. Sie beschäftigt sich einmal mit der Spezifikation der Aggregationsvorschriften, welche als Eingabe für den Rich Event Composer dienen. Dafür wird hier eine formale Spezifikationssprache entwickelt (siehe Kapitel 5). Zum Zweiten wird in der Arbeit gezeigt, wie die beiden Komponenten, Composer und Configurator, arbeiten und diese formalen Eingaben zu Rich Events umsetzen (siehe Kapitel 6).

3.4 Zusammenfassung

In diesem Kapitel wurden die grundlegenden Begriffe zu Management und Monitoring angesprochen. Danach wurden verschiedene vorhandene Ansätze hinsichtlich des Darstellens einer Dienstsicht untersucht. Dabei war festzustellen, dass es viele Sprachen gibt, die alle etwas ähnliches leisten, doch nicht dafür geeignet sind, Ressourcen zu einem Dienst zu aggregieren und zu verknüpfen. Dafür sind alle betrachteten Sprachen nicht mächtig genug. Teile dieser Arbeit sind jedoch mit gewisser Ähnlichkeit in den anderen Konzepten wiederzufinden.

Daraufhin wurden Komplettlösungen diskutiert, welche in der Form als Monitoring Anwendung auf dem Markt erhältlich sind. Keiner der zwei Stellvertreter hat die hier geforderte Datenaggregation für eine Dienstsicht implementiert oder etwas ähnliches umgesetzt. Damit bleibt zu sagen, dass bei den Anwendungen Ressourcenmonitoring noch zu stark im Vordergrund steht. Zuletzt wurde das Konzept einer Monitoring Architektur vorgestellt, das am Lehrstuhl extra für die zuvor beschriebene Dienstsicht erdacht worden ist, und wovon in folgenden Kapiteln ein Teil realisiert wird. Mit den Defiziten, die bei allen anderen Untersuchungen festgestellt wurden, können nun detaillierte und verfeinerte Anforderungen für die eigene Monitoringanwendung festgelegt werden.

Kapitel 4

Anforderungsanalyse

In Kapitel 3, „State of the Art", haben wir bereits gesehen, dass es einen Vorschlag zu einer Architektur gibt, welche die Probleme aus dem Szenario lösen kann. Wie sie funktionieren soll, und aus welchen Bausteinen sie aufgebaut ist, wird bereits dort detailliert beschrieben. Hier sollen nun die Anforderungen speziell für diese Arbeit hervorgehoben werden. Es stellen sich zwei Teilgebiete heraus, die betrachtet werden müssen. Zum einen ist das die Spezifikationssprache für die oben beschriebenen Rich Events (SISL - Service Information Specification Language) und zum anderen sind es die Anforderungen an die Architektur, bzw. die Integration/Configuration Schicht (siehe Abschnitt 3.3.3) selbst.

4.1 Anforderungen an die Spezifikationssprache

Ein Administrator soll mit Hilfe der Spezifikationssprache ein Rich Event genau beschreiben und zusammensetzen können. Die Sprache muss so allgemein gehalten werden, dass alle möglichen Zusammensetzungen machbar bleiben und flexible Konfigurationen möglich sind.

Bei dem Entwurf der Sprache müssen deswegen mehrere Dimensionen betrachtet werden. Diese fünf Teilabschnitte werden jetzt näher erläutert.

Ressourcenquellen Viele Geräte und Dienste sind über das Netzwerk verteilt und nicht zentral auf einem Hauptrechner abrufbar. Der Benutzer muss jede dieser einzelnen Ressourcenquellen in dem Rich Event spezifizieren können. Dabei hat es ihn nicht zu interessieren, auf welchem Weg er zu den betreffenden Informationen kommt. Noch nicht einmal, ob ein Adapter direkt auf die Quelle zugreift, oder nur als Proxy fungiert. Er spezifiziert nur, welchen Wert er in das Rich Event aufnehmen möchte, und die Architektur regelt den Rest.

Darüberhinaus soll es möglich sein, verschiedene mathematische Funktionen und Berechnungen auf den elementaren Werten auszuführen. Der Sinn dahinter ist, dass man unter Umständen auch solche berechneten Werte zu Vergleichen heranziehen will. Die Funktionen können von einer einfachen Addition zweier Werte bin hin zum Durchschnitt der letzten 100 aufgezeichneten Werte reichen. Es muss also eine Möglichkeit geben, eigene, gewünschte

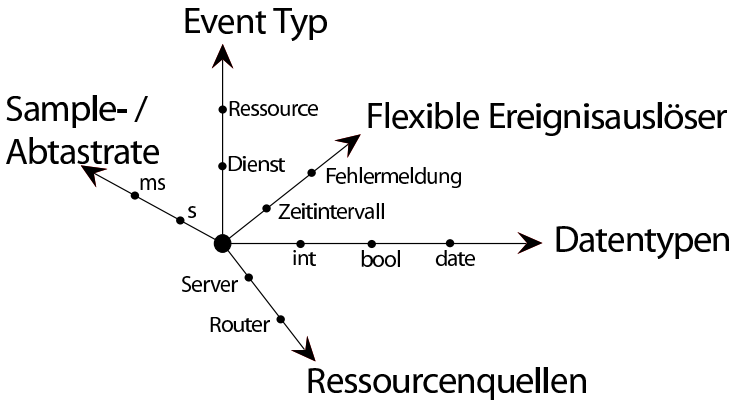


Abbildung 4.1: Anforderungen an die dienstorientierte Datenkomposition

Funktionen einzubinden und sie mit den passenden Ressourcenwerten zu „füttern". Dazu müssten unter anderem die Ressourcenwerte referenzierbar gemacht werden.

Datentypen Da theoretisch jeder abfragbare Wert eines Systems als Ressource bestimmt werden kann, muss die Sprache natürlich auch mit verschiedenen Datentypen arbeiten können. Es sollten alle Standard Datentypen, wie Integer, Boolean, String, etc. vorhanden sein, um Vergleiche möglich zu machen. Auch die mathematischen Funktionen müssen einen festen Rückgabe-Typ haben, anhand derer sie verglichen werden können. Da die Funktionen frei gewählt werden können, muss auch der Rückgabetyp frei einstellbar sein. (Dass dann die Funktion auch tatsächlich diesen Typ liefert, ist eine Sache der Implementierung der Anwendung.)

Samplerate Adapter haben kein festgesetztes Zeitintervall, in dem sie die Informationen, die sie von den verschiedenen Geräten sammeln, nach oben an die Architektur weiter schicken. Vielmehr muss der Benutzer bestimmen können, welche Werte in welchen Zeitabständen im Rich Event erscheinen. So müßte es z.B. möglich sein, ein Keepalive Signal jede Sekunde abzufragen, aber eine Prozeßliste, die womöglich vom selben Adapter geliefert wird, nur jede Minute. Die Sample- bzw. Abtastrate der einzelnen Ressourcen muss deshalb direkt in der Rich Event Beschreibung spezifiziert werden, damit die Architektur die Adapter mit den richtigen Zeitintervallen konfigurieren kann.

Ereignisauslöser Ein wichtiger Bestandteil der gesamten Architektur sind die Ereignisse, wann ein Rich Event zusammengestellt und verschickt werden soll. Die Daten werden ja die ganze Zeit über gesammelt, doch müssen bestimmte Bedingungen erfüllt sein, damit die aktuellsten Werte zusammengefasst und verschickt werden können. Es versteht sich von selbst, dass es keinen Sinn macht, starre, im Code verankerte Bedingungen zu verwenden. Der Benutzer muss seine Ereignisauslöser flexibel definieren können. Drei verschiedene Arten von Bedingungen können dabei noch unterschieden werden. Zum einen sind es die logischen Verknüpfungen von einzelnen Ressourcen. Wenn z.B. ein Wert über einem gewissen

Prozentsatz liegt, oder ein anderer eine Mindestgrenze nicht überschreitet, muss ein Rich Event zum Weiterverarbeiten geschickt werden. Weiter nützlich sind einfache Timeouts, um Statusmeldungen zu generieren. Es verstreicht eine gewisse Zeitspanne, und ohne, dass ein Fehler bei den Werten vorliegt, wird ein Rich Event verschickt. Genauso müssen Mengenangaben möglich sein, dass Bedingungen wie „Alle 500 Messwerte schicke ein Event!" realisiert werden können. Genau wie die Ressourcen selbst müssen auch die Bedingungen referenzierbar sein, um kompliziertere logische Ausdrücke bauen zu können.

Event Typ Außerdem muss der Benutzer das „Aussehen" des Rich Events definieren können. Bis jetzt steht nur fest, welche Werte alle gesammelt werden sollen, aber nicht welche dann auch wirklich im Rich Event auftauchen. So kann z.B. nur eine einzelne Ressource angegeben werden, womit dann im Prinzip ein Ressource Event (vgl. Abschnitt 2.1.1) erzeugt wird. Oder es werden mehrere Ressourcen miteinander kombiniert und man bekommt ein Service Event. Die Anzahl, wieviele der zuletzt gesammelten Werte im Event sichtbar sein sollen, kann ebenfalls vom Benutzer eingestellt werden.

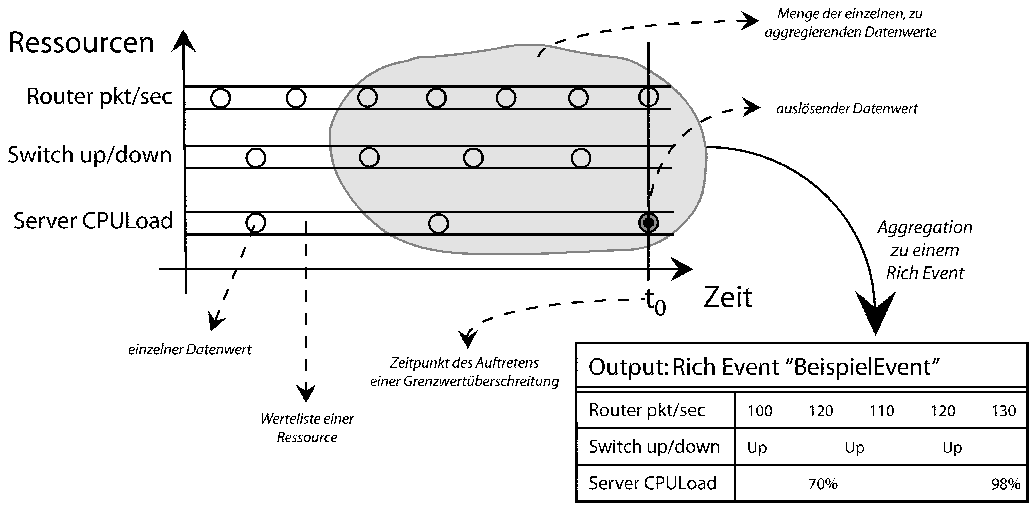


Abbildung 4.2: Generierung eines möglichen Rich Events

In Abbildung 4.2 ist das Beispiel eines Service Events noch einmal illustriert. Es sind verschiedene Ressourcen zu sehen, deren Wertelisten in voneinander unabhängigen Zeitabständen gefüllt werden. Überschreitet ein Wert eine bestimmte Schwelle, hier im Beispiel der Wert der *„Server CPULoad"*, ist die Problematik zu lösen, wieviele einzelne Ressourcenwerte aller beobachteten Ressourcen in den Datensatz aufgenommen werden sollen.

4.2 Anforderungen an die Architektur

Im vorhergehenden Abschnitt wurden die Anforderungen an die Sprache beschrieben, die eine Aggregation spezifiziert. Doch diese Theorie alleine ist unzureichend ohne die Anwendung, die dahinter steht. Die Sprache muss von der Architektur auch verstanden werden. In

Kapitel 3 sieht man, dass die Integration/Configuration Schicht aus zwei offensichtlichen Konstrukten besteht, nämlich dem Rich Event Composer und dem Adapter Configurator. Außerdem müssen die Adapter in irgendeiner Form verwaltet werden. Da dies im Hintergrund geschieht, ist die Idee dazu in dem Architekturmodell nicht mit berücksichtigt. An alle drei Teile werden ebenfalls genaue Anforderungen gestellt, die jetzt näher erläutert werden.

Composer Der Composer ist in erster Linie dafür zuständig, die Sprache, also die Eingabe des Benutzers zu verstehen. Dazu muss er geeignete Parser benutzen, um die Sprache einzulesen, und gegebenenfalls auf Fehler in der Event Spezifikation hinzuweisen. Da mehrere Events gleichzeitig angefordert werden können, muss es möglich sein, diese parallel zueinander abarbeiten zu können. Es darf also nicht die ganze Zeit ein einzelnes Event bearbeitet werden, und die anderen kommen erst an die Reihe, wenn das Erste fertig ist.

Desweiteren muss der Composer die Datenverwaltung der einzelnen Rich Events regeln. Da es eine Real Time Monitoring Anwendung ist, werden die Daten nicht für Ewigkeiten gespeichert. Sie können also nicht unendlich lang gehalten werden, weil keine Datenbank dafür vorgesehen ist. Dagegen müssen die Daten in einfachen Strukturen gespeichert werden, und zwar so lange, dass man mit den Funktionen und den Bedingungen zuverlässig arbeiten kann.

Zuletzt passiert die Funktionenberechnung und die Bedingungsauswertung ebenfalls im Composer, da hier ja die Werte der Ressourcen gespeichert werden. Ist eine Bedingung positiv, wird hier auch das Rich Event in einem vorher festgelegten Format zusammengesetzt, und an eine eventuelle Management Anwendung weiter geschickt.

Configurator Der Adapter Configurator muss, wie der Name schon sagt, mit den Adaptern sprechen. Das beinhaltet sowohl das Einstellen/Konfigurieren der Adapter, so dass sie die gewünschten Informationen in den richtigen Zeitintervallen schicken, als auch das Sammeln der einzelnen Werte. Dabei muss der Configurator die Kommunikation über ein verteiltes Netzwerk realisieren. Denn die Adapter sitzen direkt bei den Geräten und Diensten, die über das gesamte Netzwerk, welches beobachtet werden soll, verteilt sind.

Sind die Adapter einmal eingestellt, werden sie ihre Werte selbständig nach „oben" schicken. Allerdings kann es bedingt durch die Übertragung übers Netz zu Verzögerungen kommen. Der Configurator muss deshalb entscheiden können, bis zu welchem Grad noch eine vertretbare Verzögerung vorliegt, oder der Wert nicht geliefert werden kann, weil der Adapter unter Umständen ausgefallen ist.

Adapter Verwaltung Der Benutzer muss wissen, welche Werte von welchen Geräten/-Diensten abrufbar sind. Da er in der Sprache nur die Ressourcenwerte und die Quelle angibt, muss intern eine Zuordnung erfolgen, mit welchem Adapter sich diese Werte am besten abrufen lassen. Im sogenannten Repository sollen deshalb alle Adapter verzeichnet sein. Dabei ist zwischen gerade aktiven Adaptern und inaktiven zu unterscheiden. D.h. welche sind bereits installiert, um anderweitig Informationen zu sammeln, und welche stehen zur möglichen Installation (Deployment) bereit. Außerdem muss hier erkannt werden, wenn für ein anderes Rich Event bereits ein Wert geliefert wird, der nochmals angefordert wird. So können unnötige Redundanzen vermieden werden, was die Architektur und das Netz weniger belastet. Zuletzt müssen Adapterausfälle erkannt werden. Wenn Keepalive Signale nicht

mehr gesendet werden, müssen selbständig neue Adapter installiert, und Alte entfernt werden.

4.3 Anforderungskatalog

Hier sind noch einmal die Anforderungen an die Sprache und die Architektur übersichtlich und stichpunktartig zusammen gefasst.

Sprachanforderungen	
SPR 1:	Unterstützung verschiedener, verteilter Ressourcenquellen
SPR 2:	mathematische Berechnungen auf einzelnen Ressourcenwerten
SPR 3:	Unterstützung von verschiedenen Datentypen
SPR 4:	Realisierung unterschiedlicher Abtastraten bei Adaptern
SPR 5:	flexible Bedingungen als Ereignisauslöser
SPR 6:	Aussehen eines Rich Events selbst bestimmbar

Architekturanforderungen	
ARC 1:	Einlesen der Spezifikationssprache und Erkennen von Fehlern
ARC 2:	Planung der Verfügbarkeit von Ressourcenwerten
ARC 3:	Auswertung der Bedingungen
ARC 4:	Kommunikation über ein verteiltes Netzwerk (mit den Adaptern)
ARC 5:	anwendungsgesteuerte (De-)Installation von Adaptern
ARC 6:	Ausfallsicherheit & Fehlermanagement bei der Datenbeschaffung

Tabelle 4.1: Überblick der Anforderungen

Eine Bewertung der Arbeit nach den hier gestellten Anforderungen findet im letzten Kapitel in der Zusammenfassung statt, wo die verschiedenen Abschnitte noch einmal diskutiert werden. Nachdem hier nun alle detaillierten Anforderungen herausgearbeitet worden sind, wird in den nächsten beiden Kapiteln 5 und 6 beschrieben, wie die Spezifikationssprache definiert und der Prototyp entworfen wird. Dabei wird auch des öfteren auf die Vorgaben aus diesem Kapitel verwiesen.

Kapitel 5

Die Service Information Specification Language

Im vorherigen Kapitel haben wir gesehen, welche Anforderungen an die Spezifikationssprache gestellt werden. Hier wird nun erst auf einige grundlegende Sprachkonzepte eingegangen und danach die SISL-Grammatik in der erweiterten Backus-Naur-Form (EBNF) beschrieben. Anhand von einigen kleinen Beispielen werden einzelne Konzepte erklärt.

Ein abschließendes Anwendungsbeispiel der kompletten Sprache zeigt die einzelnen Komponenten im Zusammenhang an einem real möglichen Sachverhalt.

5.1 Grundlegende Sprachkonzepte

Das Ziel der Service Information Specification Language (SISL) ist es, eine Datenaggregation bzw. später das Rich Event genau und in allen Einzelheiten deklarativ zu beschreiben. Dazu müssen Ressourcen und Funktionen genau spezifiziert, und detaillierte Bedingungen eingestellt werden.

Folgende Übersicht (Abbildung 5.1) zeigt die allgemeine Struktur der SISL, ohne schon eine konkrete Grammatik vorzugeben. Zunächst folgt eine stichpunktartige Übersicht der verschiedenen Komponenten der Sprache:

- ***Aggregation*** ist das oberste Element der Sprache und die Wurzel der Baumstruktur. Sie beschreibt Ressourcen, Bedingungen und das Rich Event.
- ***Rich Event*** ist der Datensatz aus aggregierten Ressourcenwerten, der bei einer wahren Bedingung verschickt wird.
- ***Ressource*** ist die Oberklasse eines Wertes und einer Funktion. Sie beschreibt ein Element, welches von Bedingungen, Funktionen und dem Rich Event referenziert werden kann.
- ***Wert*** ist ein Attribut eines Objektes irgendwo im betrachteten Netzwerk. Er wird kontinuierlich in einem festgelegten Zeitintervall an die Anwendung geschickt.

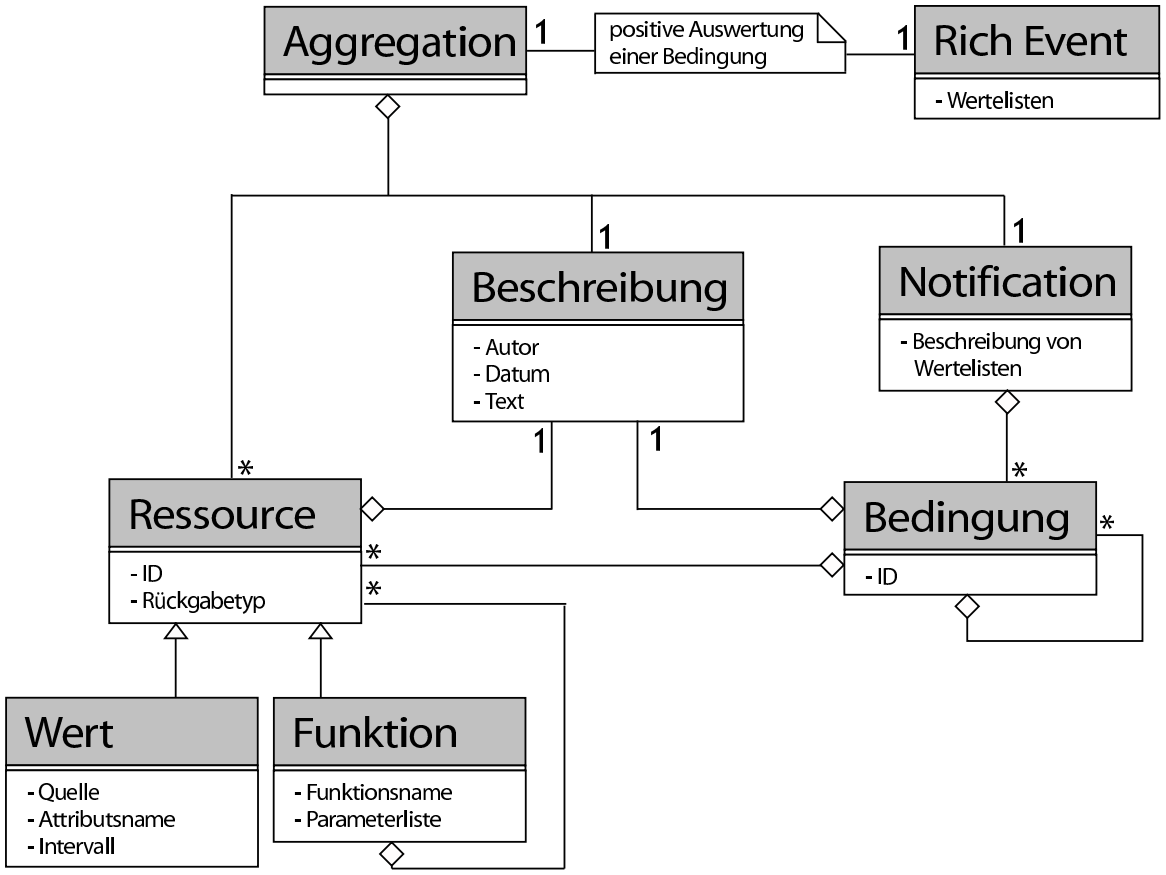


Abbildung 5.1: Aufbauskizze der Sprache

- ***Funktion*** ist eine mathematische Anweisung, die mehrere Ressourcen oder Konstanten miteinander verrechnen kann. Eine Funktion ist ihrerseits selbst eine Ressource.
- ***Beschreibung*** ist die Komponente zur Verwaltung und Erläuterung anderer Objekte der Sprache.
- ***Notification*** legt die Ressourcen fest, die in einem Rich Event erscheinen sollen. Außerdem beinhaltet sie die Bedingungen dafür.
- ***Bedingung*** ist ein logischer Ausdruck oder Vergleich von Ressourcen und Konstanten, um im Bedarfsfall ein Rich Event zu generieren. Eine Bedingung kann auf eine oder mehrere spezifizierte Ressourcen zugreifen.

Das Grundgerüst einer Aggregation ist immer gleich aufgebaut. Es besteht aus einem Ressourcenteil, wo all die Ressourcen aufgelistet werden, die innerhalb dieser Aggregation eine Verwendung finden sollen. Der Benutzer muss sich aus der Vielzahl der verfügbaren Werte genau diejenigen aussuchen, die er in der Ausgabe seines Rich Events braucht, oder die er in Bedingungen benutzen möchte. Später werden dann diese Informationen von der Anwendung benutzt, um die entsprechenden Adapter anzufragen. Eine Ressource, die hier in diesem Teil nicht aufgeführt ist, kann also nicht angefordert und später benutzt werden. Wie zu sehen ist, befinden sich die Funktionen auch mit im Ressourcenteil. Prinzipiell ist dies so, weil auch die Rückgabewerte der Funktionen in Rich Events auftauchen und später

den Bedingungen als Prädikate dienen können. Also ist die Verwendung der Rückgabewerte der Funktionen gleichwertig mit den einzelnen Werten einer Ressource. Genaueres zu den Funktionen findet sich weiter unten in diesem Kapitel.

Der zweite Teil (Notification) beinhaltet die Informationen für die Zusammensetzung des Rich Events und die gesamten Bedingungen, die der Benutzer stellen kann. Normalerweise stehen hier Abfragen, ob gewisse Werte eine vorgegebene Grenze überschreiten. Auch komplexere, untereinander kombinierte Bedingungen sind möglich. Würde keine Bedingung festgelegt werden, würden die angegebenen Ressourcen zwar gesammelt werden, allerdings nie ein „Alarmfall" auftreten, um tatsächlich ein Rich Event abzuschicken. Das heisst mindestens eine Bedingung für eine Statusmeldung sollte immer spezifiziert sein, denn andernfalls arbeitet die Anwendung für dieses Rich Event umsonst, da die Daten nie abgerufen werden können.

Der dritte Teil, der zu sehen ist, ist die Beschreibung. Darin stehen alle Informationen, die zur Verwaltung und näheren Erläuterung der betroffenen Komponente nötig sind. Nur die Aggregation selbst muss sicher eine Beschreibung haben, in welcher der Autor, das Erstellungsdatum und der Zweck der Aggregation verzeichnet ist. So kann zu späterem Zeitpunkt genau nachvollzogen werden, wer warum eine bestimmte Datenaggregation definiert hat. Der Beschreibungstext ist hilfreich dabei, den Sinn hinter der Aggregation zu verstehen, wenn man das Rich Event nicht selbst zusammen gesetzt, sondern nur von jemand anderem übernommen hat. Hier können z.B. auch Gründe stehen, warum bestimmte Werte in die Aggregation mit aufgenommen wurden, wie die Zusammenhänge sind und warum gerade die angewandten Bedingungen so sinnvoll sind. Für alle weiteren Komponenten ist die Beschreibung optional und oftmals hilfreich für Notizen. Dort hat sie mehr den Zweck für Kommentare zu einzelnen Ressourcen, Funktionen oder Bedingungen.

5.1.1 Mathematische Funktionen

Wie oben bereits erwähnt, sind neben elementaren Ressourcenwerten von den Adaptern auch mathematische Funktionen auf eben diesen möglich (**SPR 2**). Zuerst muss gesagt werden, dass es hier nur beabsichtigt ist, mathematische Funktionen zu unterstützen. Die hier angegebenen Funktionen werden intern in der Architektur Ressourcenwerte miteinander verrechnen, und einen Rückgabewert haben. Es ist nicht beabsichtigt, beliebige Methodenaufrufe in der Architektur durchführen zu können.

Solch eine Funktion ist immer aufgebaut aus folgenden 3 Teilen:

- dem Funktionsnamen
- den übergebenen Parametern
- dem Rückgabetyp

Der Funktionsname wird von dem Benutzer direkt in der Spezifikation des Rich Events angegeben. Dazu stehen ihm einige mathematische Funktionen zur Verfügung, die von vorn herein schon in der Anwendung implementiert sind, da man davon ausgehen kann, dass diese oft benutzt werden. So kann man zum Beispiel den Mittelwert der letzten 100 Einträge einer

Ressource ausrechnen, die Standardabweichung dazu, oder zwei verschiedene zusammengehörige Ressourcenwerte addieren. Möchte man eine spezielle Funktion benutzen, welche nicht standardmäßig implementiert ist, kann sie in der Anwendung entsprechend programmiert werden, und der Name der Funktion wird der Verfügbarkeitsliste hinzugefügt. So kann jeder Benutzer seine Funktionen persönlich auf sich abstimmen, und hat damit größtmögliche Flexibilität.

In der Übersicht der unterstützten Funktionen kann man sehen, wieviele Parameter der entsprechenden Funktion übergeben werden müssen. Mindestens ein Parameter muss es immer sein, da ja nur mathematische Funktionen unterstützt werden sollen, und mit irgendeinem Wert gerechnet werden muss. Dabei prüft die Architektur beim Einlesen der Spezifikation, ob die angegebene Parameteranzahl mit der geforderten Anzahl der Funktion übereinstimmt, und wird im negativen Fall gleich zu Beginn darauf hinweisen. Zum einen können die Parameter einfache Konstanten sein. Diese bestehen dann aus einem Typ und einem Wert und werden in der Spezifikation des Events direkt angegeben. Zum anderen, und das ist der eigentliche Sinn, ist ein Parameter eine Liste von Ressourcenwerten, die bereits gesammelt wurden. Ob diese Liste jetzt ein oder hundert Elemente hat, ist irrelevant. Hauptsache so wird eine Ressource spezifiziert, die mit etwas Anderem verrechnet werden soll. Deswegen muss jeder Funktion mindestens eine solche Liste übergeben werden. Das genaue Aussehen solch einer Liste kann man im anschließenden Abschnitt bei der zugehörigen Grammatik nachlesen.

Jeder Funktionenaufruf liefert einen Rückgabewert, der seinerseits selbst intern in einer Liste abgespeichert wird. Deswegen sind als Rückgabewerte auch nur einzelne Werte zulässig, und keine Listen. Dieser Wert kann nun zur weiteren Verarbeitung in einer zweiten Funktion oder in einer Bedingung genutzt werden. Da diese Werte verrechnet oder verglichen werden sollen, muss natürlich ein Rückgabetyp festgelegt werden, da z.B. bei einer Bedingungsauswertung der Typ beider Attribute gleich sein muss. Der in der Spezifikation festgelegte Rückgabetyp muss natürlich mit der Implementierung der Funktion überein stimmen. Deswegen ist er in der Funktionenübersicht auch mit angegeben. Sollte ein anderer Typ in der Eventbeschreibung auftauchen, den die entsprechende Funktion nicht liefern kann, wird gleich beim Parsen der Spezifikation darauf hingewiesen.

5.1.2 Bedingungen

Ob ein Rich Event generiert wird oder nicht, das entscheiden die Bedingungen. Bedingungen werden vom Benutzer selbst in der Spezifikation formuliert. Meistens handelt es sich um den Vergleich zweier oder mehrerer Resourcenwerte mit einem bestimmten Grenzwert, allerdings sind auch zeitliche Bedingungen machbar. Folgende drei Bedingungstypen sind in der Sprache vorgesehen:

- Vergleich mindestens zweier Werte
- zeitliche Beschränkungen (Timeouts)
- Mengenbeschränkungen, bzw. das Überschreiten einer gewissen Anzahl von gesammelten Werten

Die letzten beiden Varianten sind eigentlich nur sinnvoll für Statusmeldungen zu gebrauchen, da man ja das Intervall sowieso selbst bestimmen kann, in dem die Werte gesammelt werden. So kann nach Ablauf einer Frist von z.B. zehn Minuten oder nachdem 500 Werte von Ressource X gesammelt wurden, ein Rich Event verschickt werden. Die Syntax und die Verarbeitung innerhalb der Architektur für solche Bedingungen unterscheidet sich etwas von den Vergleichen, und wird im folgenden Kapitel bei der Grammatik genau Beschrieben.

Die Struktur der Bedingungen, die Werte miteinander vergleichen, ist angelehnt an die Arbeit von V. Danciu [Danc 03]. Die dortige Sprache stellt Bedingungen nach demselben Muster bereit, wie sie hier benötigt werden. Deswegen beschränkt sich die Angabe von Bedingungen auch hier auf entweder konjunktive oder disjunktive Normalform und benutzt keine Ausdrücke im allgemeinen Prädikatenkalkül. Das Aussehen solch einer Vergleichsbedingung ist demnach entweder

`((A oder B) und (C oder D))` (konjunktive Normalform)

oder entsprechend genau andersrum

`((A und B) oder (C und D))` (disjunktive Normalform)

Dabei sind A, B, C und D Prädikate, die den Vergleich der Ressourcenwerte und Konstanten enthalten. So werden die Forderungen nach flexiblen Bedingungen (**SPR 5**) erfüllt.

5.1.3 Referenzierbarkeit

In den vorherigen beiden Unterkapiteln wurde bereits erwähnt, dass auf Ressourcen zugegriffen werden muss. Hier wird nun erläutert, wie so etwas funktioniert.

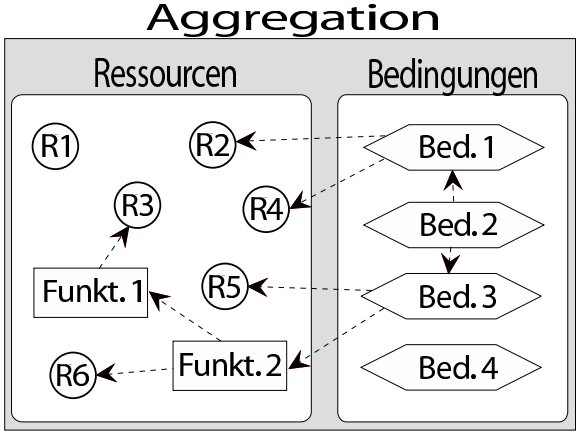


Abbildung 5.2: Referenzen einzelner Elemente

Wie man in Abbildung 5.2 sehen kann, werden zunächst alle Ressourcen (R1 - R6) in einen Container geworfen. Damit sich die Funktionen und die Bedingungen daran „bedienen" können, müssen diese einzelnen Ressourcen referenzierbar gemacht werden, so dass spezielle Ressourcen für bestimmte Berechnungen oder Vergleiche herrausgesucht werden können. Es muss also eine Möglichkeit geboten werden, eine Ressource einzeln anzusprechen.

Da die Funktionen ebenfalls Ressourcen sind, müssen diese genau wie elementare Ressourcen ansprechbar sein. So bietet sich z.B. die Möglichkeit, Funktionen von Funktionswerten zu berechnen. Das erlaubt es dem Benutzer, auch kompliziertere Funktionen nach dem Baukastenprinzip durch mehrere Einfache zusammen zu setzen (im Bild Funktion 2). Der Nutzen davon ist weiter oben schon beschrieben.

Auch Bedingungen sind referenzierbar. Die Vorteile davon sind, dass man so komplexere Bedingungen aus übersichtlicheren kleinen Teilen modular zusammensetzen kann. Außerdem können so schlank gehaltene Bausteine wiederverwendet werden, und müssen nicht jedes Mal neu in einer umfassenden Bedingung formuliert werden. Das erspart dem Benutzer große Sprachkonstrukte bei der Spezifikation des Rich Events und erhöht die Übersichtlichkeit erheblich.

Um diese Referenzierbarkeit in der Sprache zu erlauben, muss eine eindeutige Identifikationsmöglichkeit vorhanden sein. Es ist hier durch die Angabe einer ID der jeweiligen referenzierbaren Komponente (sprich Ressource, Funktion oder Bedingung) gelöst. Jedes Element der Sprache, was theoretisch von anderen Elementen benutzt werden kann, bekommt als Attribut eine ID zugewiesen, welche in der Aggregation als eindeutig vorrausgesetzt wird. Soll nun auf ein Element verwiesen werden, ist nur seine jeweilige ID zusammen mit dem entsprechenden Konstrukt (siehe Kapitel 5.2.2) anzugeben. Beim Parsen der Spezifikation wird dann diese ID von der Architektur dereferenziert, das passende Element gefunden, und in der entsprechenden Funktion oder Bedingung weiter verwendet.

Es ist anzumerken, dass Ressourcen, Funktionen und Bedingungen nicht zwangsläufig referenziert werden müssen. In Abbildung 5.2 kann man solche Fälle auch erkennen. Zum Beispiel wird eine Ressource mit in ein Rich Event aufgenommen, weil allgemein der Wert für den gegebenen Zusammenhang interessant ist, auch ohne, dass man eine Bedingung daran knüpfen muss (R1). Im Falle eines Timeouts müssen auch Bedingungen nicht zwangsläufig auf eine Ressource verweisen, sondern kommen ohne einen weiteren Verweis aus (Bedingung 4).

5.2 Grammatik der Sprache

Nachdem einige grundlegene Konzepte der Sprache im vorherigen Kapitel erläutert wurden, folgt hier nun die genaue Definition von SISL in der erweiterten Backus-Naur-Form. Zunächst werden Hilfsproduktionen der Grammatik erklärt, danach alle Elemente vorgestellt und anhand von Beispielen erläutert.

5.2.1 Anmerkung zur Verwendung der Sprache

SISL ist zum größten Teil so geschrieben, dass es nicht allein auf die zugehörige Architektur ausgelegt ist. Normalerweise kann sie auch in anderen Anwendungsbereichen eingesetzt werden. Lediglich das Element für Wertelisten (siehe Kapitel 5.2.2) ist an die Anwendung angepasst, und setzt eine bestimmte Form der Datensicherung der Ressourcenwerte voraus.

Unter Umständen müsste man noch eine Beschreibung für eine allgemeine Liste (array) hinzufügen.

Außerdem benötigt SISL eine objektorientierte Laufzeitumgebung. Die mathematischen Funktionen werden mit Methodenaufrufen realisiert. Außerdem setzt die Ressourcenspezifikation Objekte vorraus, auf denen die Werte zu finden sind. Außerhalb einer solchen Umgebung kann man die Sprache nur mit einigen Modifikatoren weiterhin brauchbar einsetzen.

Bei der Definition der Sprache wurde bewusst darauf geachtet, dass eine fertig formulierte Aggregation ein wohl geklammerter Ausdruck ist. Damit lässt sie sich leichter in ein XML Format überführen. XML wird benötigt, damit später die Architektur den Ausdruck leicht einlesen und wegen der Baumstruktur leicht weiter verarbeiten kann. Die Datentypen, die verwendet werden, sind ebenfalls in XML-Schema festgelegt.

5.2.2 Vereinbarungen und Hilfsproduktionen

Die Erweiterte Backus-Naur-Form beschreibt die Syntax und den Aufbau der Service Information Specification Language. Sie wird so in einer kontextfreien Grammatik dargestellt. Nichtterminalsymbole tauchen in der Spezifikation nicht mehr auf, denn sie werden durch die Produktionen so lange ersetzt, bis nur noch Terminalsymbole vorhanden sind. Diese Terminalsymbole sind dem Parser bekannte Schlüsselwörter, anhand derer die Sprache in die Anwendung eingelesen werden kann. Folgende Vereinbarungen wurden bei der Definition von SISL getroffen:

Nichtterminalsymbole	sind in eckige Klammern gesetzt und am Anfang einer Produktion fett gedruckt. Beispiel: `<Nichtterminalsymbol>`
Terminalsymbole	sind kursiv und in Anführungszeichen gesetzt. Beispiel: „*Terminalsymbol*"
Mehrfach	Ein Ausdruck, der keinmal oder mehrfach vorkommen darf, ist in geschweifte Klammern gesetzt. Beispiel: {mehrfach}
Optional	Ein Ausdruck, der keinmal oder genau einmal vorkommen darf, ist in eckige Klammern gesetzt. Beispiel: [optional]
Alternativ	Solch ein Ausdruck wird mit einem senkrechten Strich gekennzeichnet. Beispiel: `<exp1>` | `<exp2>`

Gruppierung	Ausdrücke werden mit runden Klammern zu einer Gruppe zusammengefasst. Beispiel: (<exp1> <exp2>) | <exp3>
xsd:	wird den Datentypen vorangestellt, die in [XMLS-2] festgelegt sind. Beispiel: <xsd:dateTime>

Da in der Sprache viele Hilfsproduktionen benutzt werden, werden diese schon vorab einmal erläutert.

<**identifier**> =	<alphachar> {<alphachar> | <digit>}
<**digit**> =	”*0*” | ”*1*” | ... | ”*9*”
<**alphachar**> =	”_” | ”*a*” | ... | ”*z*” | ”*A*” | ... | ”*Z*”

Das Element <identifier> wird gebraucht, um externe Namen anzugeben, die aus der Anwendungsumgebung stammen, und die sich der Benutzer nur bedingt selbst aussuchen kann. Es bezeichnet normalerweise Objekt- und Methodennamen und muß in der Regel einzigartig sein. Deswegen tauchen die <identifier> bei den Ressourcendefinitionen wieder auf. Da z.B. ein Methodenname nie mit einer Zahl beginnt, ist auch hier festgelegt, dass ein <identifier> immer mit einem <alphachar> zu beginnen hat.

<**identity**> =	„*id*=” <xsd:ID>
<**ressourceRef**> =	”*ressourceRef*{” <ref> ”}”
<**conditionRef**> =	”*conditionRef*{” <ref> ”}”
<**ref**> =	<xsd:IDREF>

Wie bereits in Abschnitt 5.1.3 beschrieben wurde, müssen Ressourcen und Bedingungen referenzierbar gemacht werden. Dazu wird jedem referenzierbaren Element eine ID zugewiesen, um es zu kennzeichnen. Die Einzigartigkeit dieser ID muss die Implementierung der Anwendung bereitstellen. Anhand dieser ID können dann andere Konstrukte der Sprache auf einzelne Elemente verweisen. Mit dem Nichtterminal <identity> wird so eine ID angegeben.

Die Konstrukte <ressourceRef> und <conditionRef> sind genau die Gegenstücke zur ID. Hiermit wird auf eine Ressource oder eine Bedingung verwiesen. Wegen des unterschiedlichen Anwendungsbereiches wird noch einmal unterschieden, und nicht gleich mit <ref> gearbeitet.

<**amount**> =	”*amount*{” <ressourceRef> ”,” <xsd:integer> ”}”
<**value**> =	(<type> ”{” <literal> ”}”) | <ressourceRef>
<**type**> =	”*float*” | ”*integer*” | ”*boolean*” | ”*date*” | ”*string*”
<**literal**> =	<xsd:float> | <xsd:integer> | <xsd:boolean> | <xsd:dateTime> | <xsd:string>

Die Grammatik der SISL unterstützt zwei verschiedene Arten von Werten, nämlich Listen und einzelne Konstanten.

Listen stellen einen Ausschnitt der gesammelten Werte einer bestimmten Ressource dar. Der Wert einer Ressource wird in regelmäßigen Zeitabständen von den Adaptern an die Anwendung gepusht, und sammelt sich dort in einer Warteschlange (vgl. Abbildung 5.3).

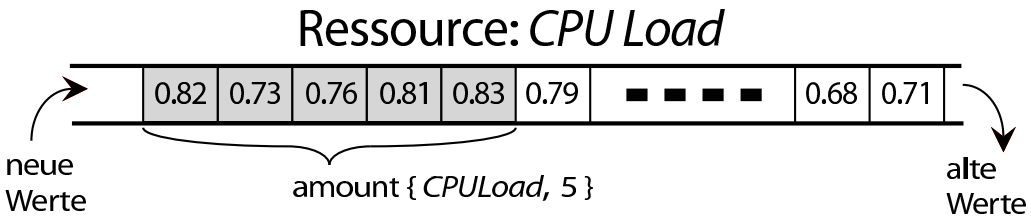


Abbildung 5.3: Bedeutung von <amount> am Beispiel einer Werteliste einer Ressource

Da diese Warteschlange systembedingt nur begrenzt lang sein kann, werden ab einem gewissen Zeitpunkt die ältesten Werte raus fallen, und von oben rutschen die aktuelleren Werte nach. (Mehr dazu findet sich in der Anwendungsbeschreibung in Kapitel 6). Das Element <amount> beschreibt solch eine Liste, die <xsd:integer> Werte ab dem aktuellsten Wert repräsentiert. Natürlich muss ein Verweis <ressourceRef> darin enthalten sein, damit die Anwendung weiss, welche Ressource gemeint ist. Es können natürlich nur maximal so viele Werte angesprochen werden, wie in der Liste enthalten sind. Ist die Zahl größer als die Anzahl verfügbarer Werte, werden alle Werte dieser Ressource referenziert. Eine einelementige Liste ist natürlich auch möglich. Nicht zu verwechseln ist diese Art Liste mit einem Array, wo unterschiedliche Konstanten hintereinander eingetragen werden können.

Einzelne Konstanten bestehen aus einem Typen und dem dazugehörigen Wert. Konstanten (<value>) kann sich der Benutzer selbst definieren. Es werden fünf verschiedene Typen unterstützt:

1. Zeichenketten = string
2. Wahrheitswerte (true / false) = boolean
3. Zeitangaben der Form yyyyddmmhhmmss = dateTime
4. Ganze Zahlen = integer
5. Gleitkommazahlen = float

Unter <value> kann man auch eine einzelne Ressourcenreferenz angeben. Dies wurde nur aus Gründen der Vereinfachung gemacht, wenn man, wie z.B. in Bedingungen, nur auf den aktuellsten Wert einer Ressource zugreifen möchte. Eine einzelne Ressourcenreferenz als Konstante bedeutet nichts anderes als eine einelementige Liste mit dem <amount>-Ausdruck.

Die folgenden zwei Beispiele zeigen die Wertedefinitionen:

- Liste der letzten 25 Eintragungen der Paketdrops eines Routers
  amount{pctdrp, 25}
- konstante Grenzwerte, anhand denen z.B. in Bedingungen verglichen wird
  float{0,85} oder boolean{false}

5.2.3 Die Aggregation

<**aggregation**> =	"*aggregation*{" <description> <ressources> <notifications> "}"
<**description**> =	"*description*{" [<author>] [<date>] <text> "}"
<**author**> =	"*author*{" <xsd:string> "}"
<**date**> =	"*date*{" <xsd:dateTime> "}"
<**text**> =	"*text*{" <xsd:string> "}"
<**ressources**> =	"*ressources*{" <ressource> {<ressource>} {<function>} "}"
<**notifications**> =	"*notifications*{" <condition> {<condition>} <declaration> "}"
<**declaration**> =	"*declaration*{" <amount> {<amount>} "}"

Eine Aggregation hat den Zweck, Ressourcen festzulegen, und Bedingungen an diese Ressourcen zu knüpfen. Sollte sich mindestens eine Bedingung als wahr erweisen, werden alle oder ein Teil der gesammelten Ressourcenwerte in das Rich Event verpackt, welches dann wiederum weitergeschickt und weiterverarbeitet wird. Wie zu sehen ist, besteht eine Aggregation aus drei Teilbereichen. Wie bereits in Absatz 5.1 erwähnt, muss die Aggregation eine Beschreibung haben. Die <description> dient hauptsächlich zur Dokumentation der Aggregation. Sie beinhaltet das Erstellungsdatum der Arggregation, den Autor und vor

allem den Grund bzw. Hintergrundinformationen, warum diese Aggregation zusammengesetzt wurde. Spätere Elemente können ebenfalls eine <description> haben, es ist aber keine zwingende Vorraussetzung.

Die beiden Hauptbestandteile sind <ressources> und <notifications>. Im Element <ressources> sind alle Ressourcen, die später in den Bedingungen oder den Rich Events Verwendung finden sollen, spezifiziert. Dabei muss mindestens eine Ressource angegeben werden, da eine Aggregation mindestens einen Wert sammeln muss. Alle weiteren Ressourcen oder mathematische Funktionen sind nicht zwingend notwendig. Die genaue Syntax von <ressource> und <function> ist komplexer, deswegen wird sie weiter unten in einem eigenen Unterabschnitt genauer beschrieben.

Der zweite große Teil sind die <notifications>. Hierin befinden sich eine oder mehrere Bedingungen <condition>. Bei den Bedingungen muss, genau wie bei den Ressourcen, mindestens eine vorhanden sein. Der Grund wurde schon einmal genannt. Wäre keine Bedingung spezifiziert, könnte es nie einen Grund geben, ein Rich Event abzuschicken. Da die Bedingungen indirekt bestimmen, wann ein Rich Event erzeugt wird, ist hier auch seine Beschreibung <declaration> definiert. Bis jetzt weiß die Anwendung, welche Ressourcen sie zu sammeln hat, und welche Bedingungen sie auszuwerten hat, aber nicht, wie ein Rich Event aussehen soll. Aussehen bedeutet hier, wieviel von welchen Ressourcenwerten in das Rich Event Format geschrieben werden (vgl. Anforderung **SPR 6**). Wie in der Grammatik zu sehen ist, werden die Listen der gewünschten Werte mit dem <amount>-Konstrukt angegeben. So kann der Benutzer ganz genau spezifizieren, von welchen Werten er die aktuellsten haben möchte. Die Spanne der Werte reicht von nur dem letzten Eintrag bis zu allen Einträgen, die die Liste zu dieser Ressource noch gespeichert hat.

Ein kleines Beispiel soll die <declaration> noch einmal verdeutlichen. In Listing 5.1 sieht man das Grundgerüst einer Aggregation, wo alle anderen Teile außer der Event Beschreibung außen vor gelassen werden. Die fiktiven Ressourcen res1, res2 und res3 werden zu einem Rich Event zusammengestellt. Dabei soll von Ressource res1 nur der letzte Wert auftauchen, bei der Zweiten sind die letzten 200 und bei der Dritten die letzten 50 Werte von Interesse.

Listing 5.1: Das Grundgerüst und eine Beispiel-Declaration

```
aggregation{
        description{ ... }
        ressources{ ... }
        notifications {
                ...
                declaration {
                        amount{ res1, 1}
                        amount{ res2, 200}
                        amount{ res3, 50}
                }
        }
}
```

5.2.4 Das Element <ressource>

Eine Ressource ist im weitesten Sinne ein Attribut eines Objektes, welches sich irgendwo im überwachten Netzwerk befinden kann. Meistens wird als Ressource ein technischer Zustand oder Wert eines Gerätes bezeichnet. Dieser Wert kann sich mit fortlaufender Zeit verändern, und es wird versucht, diese Veränderung zu protokollieren und zu untersuchen. Wie bereits erwähnt müssen mit dem <ressource>-Element alle Ressourcen beschrieben werden, die in der Aggregation eine Verwendung finden sollen. Hier wird nun der Aufbau einer Ressourcendefinition genauer erklärt.

<**ressource**> =	"*ressource*{" <identity> [<description>] <source> <sourceAttrib> <interval> <return> "}"
<**source**> =	"*source*{" <identifier> "}"
<**sourceAttrib**> =	"*sourceAttrib*{" <identifier> "}"
<**interval**> =	"*inverval*{" <xsd:float> "}"
<**return**> =	"*type*{" <type> "}"

Wie man sehen kann, muss eine Ressource eine Identität haben, eine Beschreibung ist optional. Der Grund dafür wurde bereits in früheren Kapiteln erläutert.

Quellen und Attribute

Essentiell wichtig für die Festlegung einer Ressource sind die beiden Elemente <source> und <sourceAttrib>. Wie zu Anfang dieses Abschnittes schon geschrieben, muss festgelegt werden, von welchem Objekt welches Attribut überwacht werden soll. Dieses Objekt wird mit dem <source>-Element beschrieben. Der Benutzer muss sich an dieser Stelle nicht darum kümmern, wie und über welche Wege das Objekt angesprochen werden kann. Der Zugriff ist zu diesem Zeitpunkt völlig transparent. Später muss anwendungsintern geklärt werden, wie am effizientesten die gewünschte Quelle zu erreichen ist. Der Benutzer kann innerhalb der Anwendung eine Auflistung aller netzwerkweiten Objekte einsehen, um dort den richtigen Namen dafür zu finden. Möchte er z.B. von einem Datenbankserver eine Ressource anfordern, sucht er sein Label in der dazugehörigen Tabelle, z.B. *dbServ1*, und trägt dieses bei <source> in die Spezifikation ein.

Genauso wird bei <sourceAttrib> verfahren. In der selben Tabelle sind alle Werte verzeichnet, die das gewünschte Objekt liefern kann. Der korrekte Name eines Attributes wird also ausgesucht, und ebenfalls in die Spezifikation eingetragen. Es kann allerdings immer nur ein Attribut auf einmal angegeben werden, da es sonst bei der Dereferenzierung der Ressource zu Mehrdeutigkeiten kommt. Selbst dann, wenn beide Attribute von der selben Quelle stammen würden. Da für jede Ressource eine Liste angelegt wird, auf die später

zugegriffen wird (siehe Abbildung 5.3), wäre es bei einem Verweis auf eine „doppelte" Ressource nicht entscheidbar, welches Attribut nun mit der Referenz gemeint wäre. Aus diesem Grund muss für jedes Attribut, egal ob es von der gleichen Quelle geliefert wird oder nicht, ein neues Ressourcenelement mit eigener ID gebaut werden.

Mit der oben genannten Struktur wird das Anforderungsziel, nämlich die Unterstützung verteilter Ressourcenquellen (**SPR 1**), erfüllt. Denn die Sprache kann theoretisch jede Ressource irgendwo im Netz spezifizieren. Es könnte die Frage auftauchen, warum der Benutzer nicht direkt den Adapter in der Aggregation angibt. Aus drei wichtigen Gründen wurde von dieser Möglichkeit abgesehen. Erstens verletzt es das Gebot der Transparenz, wenn man selbst schauen muss, welcher Adapter die richtigen Werte liefern kann. Es ist unnötige Arbeit, mit der der Benutzer nichts zu tun haben will und auch nicht muss. Zweitens wäre solch eine Aggregation nicht fehlerresistent. Angenommen ein Adapter fiele aus, aber ein anderer könnte den selben Wert dennoch bereit stellen. Wenn ein bestimmter Adapter hier festgelegt wäre, hätte der Ersatzadapter einen anderen Namen als der Angegebene, und die Ressource könnte nicht erreicht werden. Mit der vorhandenen Lösung könnte das nicht passieren, da die Anwendung den zweiten Adapter erkennt und diesen stattdessen benutzt. Der dritte Grund ist, dass man mit der sprachlichen Fixierung auf das Adapterkonzept die Sprache schwerer in anderen Anwendungsbereichen einsetzen kann, da sie in der entsprechenden Architektur etwas voraussetzen würde. So bleibt es der jeweiligen Anwendung überlassen, wie der Zugriff auf Ressourcenquellen realisiert wird.

Das Pollingintervall

Unter `<interval>` sind die Zeitabstände angegeben, in welchen die Ressourcenwerte immer wieder abgefragt werden sollen (siehe **SPR 4**). Um umständliche Konstrukte zu vermeiden, wird innerhalb der Anwendung als Standard mit Sekunden gerechnet. Deswegen wird auch hier das Pollingintervall immer in Sekunden angegeben. In der Sprache spart man sich dadurch Elementdefinitionen für Minuten, Millisekunden, etc. Da durchaus auch kleinere Zeiteinheiten als Sekunden gewünscht sein können, kann hier im Intervall eine Gleitkommazahl angegeben werden. Liegt das kleinste Pollingintervall des zugehörigen Adapters über dem angegebenen Wert, wird stattdessen das kleinste verfügbare Intervall für diese Ressource ausgewählt und der Benutzer darauf aufmerksam gemacht. Wieder sind hier die Grenzen in der Anwendung zu sehen, denn theoretisch wäre jedes Intervall möglich. Sollte der Fall eintreten, dass in der Aggregation zwei unterschiedliche Attribute von der selben Quelle benutzt werden, ist es natürlich zulässig, diesen Beiden ein unterschiedliches Intervall zu geben. (Sie müssen sowieso einzeln spezifiziert werden.) Wie der Adapter unterschiedliche Zeitintervalle auf der gleichen Quelle realisiert ist abhängig von der Implementation der Anwendung.

Datentypen

Jede Ressource hat ihren eigenen Datentyp, der mit dem Element `<return>` angegeben wird. So wird der Anforderung **SPR 3** Folge geleistet. Der Name mutet im ersten Moment etwas seltsam an. Er wurde wegen den Funktionen so gewählt, da deren Rückgabewert auch ein Datentyp zugewiesen werden muss. Als Datentypen stehen die Gleichen zur

Verfügung, die bereits in Abschnitt 5.2.2 aufgelistet worden sind. Die Angabe der Datentypen ist verpflichtend, da Vergleiche in den Bedingungen nur zwischen Werten ausgeführt werden können, die den selben Typ haben.

Beispiel einer Ressourcendefinition

Im folgenden Beispiel 5.2 werden drei Ressourcen spezifiziert. Für jede Ressource muss ein eigenes Element definiert werden, auch wenn sie wie Ressource 1 und 2 von der selben Quelle stammen. Der Datenbankserver *dbServ1* wird einmal nach seinem Status gefragt, ob er läuft oder nicht. Dies soll jede Sekunde passieren, denn falls er wirklich down ist, muss schnell etwas unternommen werden. Der Typ dieser Ressource ist natürlich boolean. Entweder der Server läuft, oder nicht. Die zweite Ressource zeigt seine Speicherkapazität an. Da das kein kritischer Wert ist, hat der Benutzer beschlossen, diesen nur jede Minute zu erneuern, um die Last auf dem internen Netz gering zu halten. Die letzte Ressource beschreibt die aktiven Anfragen an den zugehörigen Applicationserver *appServ1*. Alle drei Ressourcen müssen natürlich eine unterschiedliche ID haben. Außerdem wurde aus Gründen der Übersichtlichkeit eine Beschreibung weg gelassen.

Listing 5.2: Spezifikation von drei unterschiedlichen Ressourcen

```
ressources{
        ressource{
                id= r00001
                source{ dbServ1 }
                sourceAttrib{ up_down }
                interval{ 1 }
                return{ boolean }
        }
        ressource{
                id= r00002
                source{ dbServ1 }
                sourceAttrib{ storage_capacity }
                interval{ 60 }
                return{ float }
        }
        ressource{
                id= r00003
                source{ appServ1 }
                sourceAttrib{ number_of_active_requests }
                interval{ 5 }
                return{ integer }
        }
}
```

<**function**> =	"*function*{" <identity> [<description>] <method> <parameters> <return> "}"
<**method**> =	"*method*{" <identifier> "}"
<**parameters**> =	"*parameters*{" <amount> {<amount> | <value> } "}"

5.2.5 Das Element <function>

Eine mathematische Funktion wird hier in der Sprache auch als eine Ressource betrachtet. In periodischen Zeitabständen (nämlich genau dann, wenn der Ressourcenwert, welcher Grundlage der Berechnung ist, aktualisiert wird) wird ein neuer Funktionswert berechnet und gespeichert. Analog zu den Ressourcen, wo die Adapter im gleichmäßigen Zeitintervall Werte liefern. Deswegen werden sie mit unter dem Konstrukt <ressources> definiert. Die Rückgabewerte einer Funktion werden genauso wie die elementaren Ressourcen in Listen abgelegt, und können von Bedingungen und anderen Funktionen referenziert werden. Deswegen haben die Funktionen, genau wie die Ressourcen einen <return>-Typ, der festlegt, welches Datenformat der berechnete Wert hat. Hier muss gesagt werden, dass die Funktionen immer nur einen einzelnen Rückgabewert haben, keine Arrays mit mehreren Ergebnissen. Denn genauso wie bei den Ressourcen führen mehrere Rückgabewerte zur Unentscheidbarkeit bei Referenzen der Bedingungen.

Methodenaufrufe

Der Unterschied liegt also nicht in der Verwendung der Werte, sondern nur in deren Herkunft. Zuerst wird eine Methode angegeben, die den Funktionswert berechnen soll. Wie bereits in Kapitel 5.1.1 beschrieben, sind alle verfügbaren Funktionen in einer Liste vermerkt. In den <method>-Bereich wird also nur der entsprechende Name von der Funktion eingetragen, die benutzt werden soll. Es können natürlich nur solche Funktionen berechnet werden, die entweder von vorn herein zur Verfügung gestellt werden, oder sich der Benutzer selbst implementiert hat.

Parameter der Funktion

Jede Funktion bekommt eine gewisse Anzahl an Parametern zur Berechnung übergeben. Wieviele es sein müssen, kann man in der jeweiligen Tabelle nachlesen, wo die verfügbaren Funktionen aufgelistet sind. Diese Parameter werden in <parameters> spezifiziert. Wie in der Grammatik zu sehen ist, muss mindestens ein Parameter ein Verweis auf eine Ressource sein. <amount> legt eine Ressource und ihre Anzahl der einzelnen Werte fest (vgl. Kapitel 5.2.2). Mit der so übergebenen Werteliste kann nun gearbeitet werden. Sollen diese Werte mit anderen Ressourcen oder Konstanten verrechnet werden, müssen gegebenenfalls noch weitere Parameter übergeben werden.

Der Anwendungsbereich der mathematischen Funktionen teilt sich im Großen und Ganzen in zwei Bereiche auf. Der erste Teil ist, dass mehrere Werte einer Ressource in irgedneiner

Form zusammengerechnet werden können. In dem Fall hat die übergebene Liste mehrere Einträge, ist aber meistens der alleinige Parameter. Der zweite Fall ist die Verechnung zweier unterschiedlicher Ressourcen miteinander oder einer Ressource mit einer Konstante. Charakteristisch hierfür ist, dass die übergebene Liste der Ressourcenwerte nur einen einzelnen Wert angibt. In den beiden folgenden Beispielen sollen genau diese Fälle noch einmal anschaulich beschrieben werden.

Beispiele zweier Funktionen

Listing 5.3 zeigt eine Funktion, die das arithmetische Mittel der letzten 25 Werte der Ressource *CPUload* berechnet. Zum besseren Verständnis wurde der Referenz genau dieser Name gegeben und nicht irgendeine Zahlen-Buchstaben-Kombination. Die Funktion *arithMittel* ist intern definiert und bekommt genau eine Liste als Parameter. Darin sind im Beispiel die 25 Ressourcenwerte enthalten. Als Rückgabetyp liefert *arithMittel* eine Gleitkommazahl.

Listing 5.3: Funktion auf einer Liste mit mehreren Werten

```
function{
        id= f00001
        method{ arithMittel }
        parameters{
                amount{ CPUload, 25 }
        }
        return{ float }
}
```

In Listing 5.4 sieht man, wie die jeweils aktuellsten Paketwerte, die über Port A und B eines Routers empfangen wurden, addiert werden. Vielleicht interessiert den Benutzer nur die Gesamtanzahl der angekommenen Pakete, die so aber kein Adapter liefern kann. Deswegen muss er sie sich selbst errechnen. Als Parameter werden der Funktion *addTwoValues* hier zwei einelementige Listen übergeben, die immer den aktuellsten Wert beinhalten. Als Rückgabetyp liefert *addTwoValues* einen ganzzahligen Wert.

Listing 5.4: Funktion auf zwei ein-elementigen Listen

```
function{
        id= f00002
        method{ addTwoValues }
        parameters{
                amount{ pktPortA, 1}
                amount{ pktPortB, 1}
        }
        return{ integer }
}
```

Wie auch bei den Ressourcen müssen die Funktionen zwecks Referenzierbarkeit eine ID besitzen, und können zum besseren Verständnis eine Beschreibung enthalten.

5.2.6 Das Element <condition>

Die wichtigsten Elemente neben den Ressourcendefinitionen in der Sprache sind die Bedigungen. Denn ohne eine Bedingung, die als wahr ausgewertet wird, kann kein Rich Event verschickt werden.

<**condition**> =	”*condition*{” <identity> [<description>] (<cnf> | <dnf> | <constraint>) ”}”
<**cnf**> =	”*and*{” <binaryLogOpOr> {”,” <binaryLogOpOr>} ”}”
<**dnf**> =	”*or*{” <binaryLogOpAnd> {”,” <binaryLogOpAnd>} ”}”
<**constraint**> =	((”*equal*” | ”*smaller*” | ”*greater*” | ”*greaterEqual*” | ”*smallerEqual*”) ”{” <binaryPredicate> ”}”) | (["!"] <value>) | <conditionRef> | (”*timeout*{” <xsd:float> ”}”) | (”*counter*{” <ressourceRef> ”,” <xsd:integer> ”}”)
<**binaryLogOpOr**> =	<constraint> {”||” <constraint>}
<**binaryLogOpAnd**> =	<constraint> {”&&” <constraint>}
<**binaryPredicate**> =	(<value> | <triplePredicate>) ”,” <value>
<**triplePredicate**> =	(”*min*{” | ”*max*{” | ”*exact*{”) (<xsd:integer> | ”*all*”) ”,” <ressourceRef> ”,” (<xsd:integer> | <time>) ”}”
<**time**> =	”*time*{” <xsd:float> ”}”

Wie bereits in Abschnitt 5.1.2 zu sehen ist, werden die Bedingungen in entweder disjunktiver oder konjunktiver Normalform beschrieben. Dazu dienen die Elemente <dnf> und <cnf>. Im einfachsten Fall, wenn keine weiteren Verknüpfungen gebraucht werden, kann eine Bedinung auch nur ein einzelnes Prädikat sein, worin der logische Ausdruck enthalten ist. Hierfür kann ein <constraint> direkt unter der Bedingung angegeben werden.

Referenzen auf Bedingungen

Das Element <conditionRef> wird benötigt, um verschiedene Bedingungen miteinander verknüpfen zu können. So ist es möglich, dass komplexe Bedingungen übersichtlich

durch modulare Einzelteile aufgebaut werden. Genau nach dem selben Schema wie bei Ressourcenreferenzen wir hier über eine ID auf eine andere Bedingung verwiesen. Solch eine Referenz kann z.B. an Stelle eines Vergleiches oder Timeouts für ein <constraint> eingesetzt werden. So lassen sich die logischen Operatoren UND und ODER darauf anwenden.

Beispiel 5.5 zeigt solch eine UND-Verknüpfung zweier Bedingungen, die einen beliebigen Inhalt haben können (und hier im Beispiel nicht genauer definiert sind).

Listing 5.5: Referenzen auf Bedingungen im Einsatz

```
notifications{
        condition{
                id= c00008
                ...
        }
        condition{
                id= c00009
                ...
        }
        condition{
                id= c00010
                and{
                        conditionRef{'c00008'},
                        conditionRef{'c00009'}
        }
}
```

Zeitüberschreitungen (Timeouts)

Das *timeout*{} kennzeichnet eine bestimmte Zeitspanne, nach der ein Rich Event getriggert wird. Meistens wird diese Art der Bedingung dafür genutzt, um nach x Minuten eine Statusmeldung zu schicken, obwohl eigentlich keine Probleme aufgetreten sind. Es hat nichts gemeinsam mit einem normalen Vergleich, deswegen wird ein Timeout hier gesondert in das <constraint> eingefügt. Der Benutzer muss keine Referenzen auf Ressourcen angeben, da die Auswertung an die Systemzeit gekoppelt ist. Deswegen ist nur die gewünschte Zeitspanne interessant, deren Ablauf intern berechnet wird. Ist die Zeit verstrichen, wird die Bedingung wahr und der Zähler fängt erneut an, runterzuzählen. Die Zeit wird, aus den selben Gründen wie bei dem Pollingintervall, ebenfalls in Sekunden eingetragen.

Im Beispiel 5.6 wird ein einfacher Timeout definiert. Die Bedingung lautet: „Schicke alle 10 Minuten (=600 Sekunden) ein Rich Event als Statusbericht!" Zwecks Übersichtlichkeit wurde der Rest der Aggregation durch Punkte verkürzt.

Listing 5.6: Timeout von 10min als Bedingung

```
aggregation{
        ...
        notifications{
                condition{
                        id= c00001
```

```
                        timeout{ 600 }
                }
                ...
        }
}
```

An dieser Stelle sei noch einmal der genaue Unterschied zwischen einem *timeout*{} und einem *interval*{} genannt (vgl. auch Abbildung 5.4). Ein Timeout wird, wie oben beschrieben, in Bedingungen verwendet. Er bezieht sich auf die Zeit, die verstreichen muss, bis ein Rich Event gesendet werden soll. Das Intervall legt dagegen die Zeitspanne fest, in der die Ressourcenwerte von den Adaptern an die Anwendung geschickt werden. Mit beiden Elementen wird eine Zeitspanne spezifiziert, jedoch auf unterschiedlichen Anwendungsebenen.

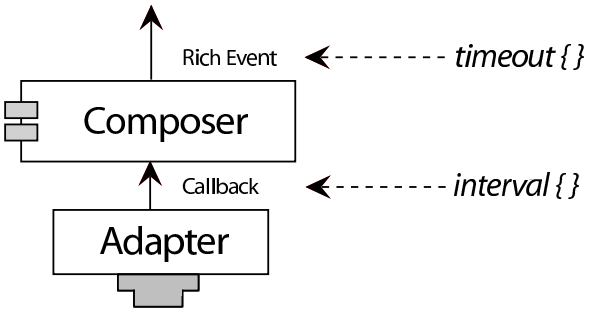


Abbildung 5.4: Unterschied zwischen *timeout*{} und *interval*{}

Wertezähler

Genauso wie beim *timeout*{} wird ein *counter*{} gerne für Statusmeldungen verwendet. Statt der vergangenen Zeit wird bei dieser Möglichkeit die Anzahl der gesammelten Werte gezählt. Wenn eine festgelegte Anzahl von Werten einer Ressource gesammelt wurde, wird die Bedingung wahr. So kann in regelmäßigen Intervallen nach x gesammelten Werten eine Meldung ausgelöst werden. Natürlich muss hier auf eine Ressource verwiesen werden, von welcher die Werte gezählt werden sollen. Das Beispiel 5.7 zeigt einen Zähler, der nach 300 neuen Einträgen der Ressource *CPUload* wahr wird.

Listing 5.7: Counter von 300 Einträgen als Bedingung

```
aggregation{
        ...
        notifications{
                condition{
                        id= c00002
                        counter{ CPUload, 300 }
                }
                ...
        }
}
```

Einfache Vergleiche

Eine weitere Option von <constraint> sind die Vergleiche zwischen zwei (oder mehreren) Werten. Alle möglichen Vergleichsoperatoren wurden in die Sprache übernommen. So ist es vorgesehen, zwei Werte auf Gleichheit, größer, größer/gleich, kleiner und kleiner/-gleich testen zu können. In der Grammatik stellt das Konstrukt <binaryPredicate> mit den zwei <value> Einträgen dieses Wertepaar bereit. Die Hilfsproduktion mit dem Element <triplePredicate> beschreibt Wertemengen, auf die später in der Arbeit eingegangen wird. Ein einfacher Vergleich ist in Beispiel 5.8 zu sehen. Sobald der aktuelle Wert der Ressource *free_storage_capacity* einmalig unter 5% fällt, soll ein Alarm ausgelöst werden. (Aus Gründen der Übersichtlichkeit wird nur noch die Bedingung angegeben, nicht mehr das Gerüst der Aggregation außen herum.)

Listing 5.8: Einfacher Vergleich eines Ressourcenwertes mit einer Konstante

```
condition{
        id= c00003
        smaller{ free_storage_capacity, float{0.05} }
}
```

Um solche Vergleiche mit UND und ODER zu verknüpfen, muss man in der Grammatik den „Umweg" über die Produktionen <cnf>, <dnf> sowie <binaryLogOpOr> und <binaryLogOpAnd> gehen. Wie dies in der ausformulierten Sprache aussieht, sieht man in Beispiel 5.9. Hier werden zwei Vergleiche mit ODER verknüpft. Die Bedingung lautet hier: „Entweder die freie Speicherkapazität fällt einmalig unter 5% oder der Server ist nicht mehr vergügbar." Auch hier sind *free_storage_capacity* und *up_down* als Ressourcenreferenzen zu sehen.

Listing 5.9: Zwei einfache Vergleiche mit logischem ODER verknüpft

```
condition{
        id= c00004
        or{
           smaller{ free_storage_capacity, float{0.05} },
           equals{ up_down, boolean{ false }
        }
}
```

Komplexe Vergleiche auf Wertemengen

Bei den bisher betrachteten Vergleichen war immer nur der neueste und aktuellste Ressourcenwert gefragt. Allerdings kann bei manchen Ressourcen ein Ausreißer nach oben oder unten schon einmal erlaubt sein, und man möchte wegen einer einmaligen Werteüberschreitung keinen Alarm auslösen. Wenn sich allerdings die Überschreitungen in einem gewissen Zeitraum häufen, ist dennoch ein Alarm von Nöten. Hier muss also eine Menge von Ressourcenwerten auf einen bestimmten Zustand hin getestet werden. Die neue Bedingung wäre derart formuliert: „Für mindestens X der letzten Y Werte der Ressource *ressource* soll gelten, dass ...".

Das Element <triplePredicate> beschreibt eine solche Wertemenge. Es sind folgende drei Mengenspezifizierungen erlaubt:

- mindestens X der letzten Y Ressourcenwerte = min{ X , *ressource*, Y}
- höchstens X der letzten Y Ressourcenwerte = max{ X , *ressource*, Y}
- genau X der letzten Y Ressourcenwerte = exact{ X , *ressource*, Y}

Die übergebenen Parameter in der Klammer bedeuten also immer {kritische Anzahl, Ressourcenreferenz, Gesamtanzahl}. Die Grammatik erlaubt nun, über <binaryPredicate> und <triplePredicate> diese Mengenangaben in die Bedingungen einzubauen. In Beispiel 5.10 ist eine solche Bedingung auf einer Menge von Ressourcenwerten formuliert. In deutscher Sprache lautet sie: „Wenn mindestens 10 der letzten 20 CPUload Werte über 90% liegen, dann..."

Listing 5.10: Vergleich auf einer Menge von Ressourcen

```
condition{
        id= c00005
        greater{ min{10, CPUload, 20}, float{0.9} }
}
```

Bis dahin kann man alle Fälle abdecken, die eine Bedingung auf einer Menge formulieren, welche durch die direkt angegebene Anzahl an Ressourcenwerten beschränkt ist. Eine zeitliche Beschränkung ist mit diesen Regeln noch nicht möglich. In den Produktionen der Grammatik kann man die Lösung des Problems erkennen. Anstelle der festen Integerzahl zur Beschränkung kann auch eine Zeitangabe erfolgen, die in <time> spezifiziert wird. Genau wie bei allen anderen Zeitangaben wieder nur in Sekunden. Da der Benutzer nicht genau wissen kann, wieviele Ressourcenwerte jetzt tatsächlich in der Menge sind, er aber unter Umständen alle ansprechen will, ist anstatt einer festen Zahlenangabe auch das Schlüsselwort *all* erlaubt. Das abgewandelte Beispiel von oben lautet nun folgendermaßen: „Wenn mindestens 10 der CPUload Werte in der letzten Minute über 90% liegen, dann ...", und ist in Listing 5.11 zu sehen.

Listing 5.11: Vergleich auf einer Menge von Ressourcen

```
condition{
        id= c00006
        greater{ min{10, CPUload, time{60}}, float{0.9} }
}
```

Beispiel 5.12 zeigt die selbe Bedingung, nur wenn exakt alle Werte der letzten Minute über 90% liegen sollen.

Listing 5.12: Vergleich auf einer Menge von Ressourcen mit zeitlicher Beschränkung

```
condition{
        id= c00007
        greater{ exact{all, CPUload, time{60}}, float{0.9} }
}
```

5.2.7 Anwendungsbeispiel

An dieser Stelle soll noch einmal ein komplettes Fallbeispiel gezeigt werden, wo nicht nur Sprachausschnitte zu sehen sind, sondern die komplette Aggregation im Zusammenhang. Anders als in den obigen Beispielen sind jetzt die Ressourcenreferenzen wirklich Verweise auf die vorher definierten Ressourcen. Um Zusammenhänge besser verfolgen zu können, wurden im Quelltext und in der Aggregation äquivalente Aussagen extra gekennzeichnet. Eine Aufgabenstellung könnte folgendermaßen aussehen:

*„Es soll eine Aggregation zur Überwachung eines Routers erstellt werden. Der Router besitzt die Ports **R1** und **R2**, die regelmäßig immer wieder Pakete verlieren. Nun soll der gesamte Paketeverlust jede Sekunde (**F1**) beobachtet werden. Ebenso die Auslastung der CPU des Routers (**R3**), um zu prüfen, ob bei höherer Auslastung eine höhere Verlustrate festzustellen ist.*
*Ein Rich Event soll in folgenden Fällen generiert werden. Erstens, wenn innerhalb der letzten Minute die Werte der CPU-Last den Grenzwert von 85% mindestens 5 Mal überschritten haben (**C1**), und zweitens, wenn die Zahl der insgesamt verworfenen Pakete innerhalb eines Intervalls 80 oder mehr ist (**C2**). Außerdem soll jede Minute ein Rich Event als Statusmeldung erzeugt werden (**C3**), falls keine Grenzwerte überschritten werden. Das Rich Event beinhaltet alle Messungen, die innerhalb einer Minute gesammelt wurden.(**D1**)”*

Listing 5.13: Vollständig definierte Aggregation

```
   aggregation{
      description{
         author{Sebastian Lange}
         date{20061004034932}
         text{/*Das ist die Beispiel-Aggregation.*/}
      }
      ressources{
/*R3*/   ressource{ id=r00001
            description{
               text{/*Die CPU Auslastung des Routers.*/}
            }
            source{TestRouter}
            sourceAttrib{cpu_load}
            interval{1}
            return{float}
         }
/*R1*/   ressource{ id=r00002
            description{
               text{/*Seine verlorenen Pakete auf Port R1.*/}
            }
            source{TestRouter}
            sourceAttrib{pkt_reject_portR1}
            interval{1}
            return{integer}
         }
/*R2*/   ressource{ id=r00003
```

```
            description{
               text{/*Seine verlorenen Pakete auf Port R2.*/}
            }
            source{TestRouter}
            sourceAttrib{pkt_reject_portR2}
            interval{1}
            return{integer}
         }
/*F1*/   function{ id=f00111
            description{
               text{/*Die aktuellsten Werte beider Ports werden
                      addiert.*/}
            }
            method{addTwoValues}
            parameters{
               amount{ressourceRef{'r00002'}, 1},
               amount{ressourceRef{'r00003'}, 1}
            }
            return{integer}
         }
      }
      notifications{
/*C1*/   condition{ id=c10000
            description{
               text{/*Mindestens 5 CPU Werte über Grenzwert in den
                      letzten 60sec.*/}
            }
            greater{
               min{5, ressourceRef{'r00001'}, time{60} },
               float{0.85}
            }
         }
/*C2*/   condition{ id=c20000
            description{
               text{/*Summe der Paketverluste (= Funktionswert)
                      größergleich 80.*/}
            }
            greaterEqual{
               ressourceRef{'f00111'},
               integer{80}
            }
         }
/*C3*/   condition{ id=c30000
            description{
               text{/*Nach 60 Sekunden eine Statusmeldung.*/}
            }
            timeout{60}
         }
/*D1*/   declaration{
```

```
            amount{ressourceRef{'r00001'}, 60}
            amount{ressourceRef{'r00002'}, 60}
            amount{ressourceRef{'r00003'}, 60}
            amount{ressourceRef{'f00111'}, 60}
        }
    }
}
```

5.3 Zusammenfassung

Dieses Kapitel behandelte die Service Information Specification Language, die in der Architektur verwendet wird, um Datenaggregationen zu definieren und Rich Events festzulegen. Nachdem der strukturelle Aufbau der Sprache beschrieben und grundlegende Konzepte erklärt wurden, folgte die Spezifikation der Grammatik in der EBNF. Einzelne Elemente der Grammatik wurden detailliert beschrieben und an einfachen Beispielen erläutert. Das Beispiel am Ende des Kapitels zeigt eine vollständige Aggregationsdefinition im Zusammenhang mit einem möglichen Anwendungsfall.

Die Sprache wurde so definiert, dass alle Anforderungen aus Abschnitt 4.3 erfüllt sind. Es ist möglich, verteilte Ressourcenquellen anzusprechen, und deren Werte möglicherweise miteinander verrechnen zu lassen. Ebenso sind Elemente vorhanden, um Bedingungen frei formulieren zu können. Darüberhinaus ist die Sprache im Bereich der Funktionen leicht erweiterbar und auch in anderen Bereichen flexibel einsetzbar, da sie weitestgehend anwendungsunabhängig ist.

Nachdem die Sprache nun definiert ist, werden im folgenden Kapitel die Teile der Monitoring Architektur entworfen, welche die Sprache verstehen und die beschriebenen Aktionen auch ausführen können.

Kapitel 6

Entwurf des Prototypen

In Kapitel 3 wurde die am Lehrstuhl Hegering vorgeschlagene Architektur einer Monitoring Anwendung schon detaillierter vorgestellt. Nachdem nun genaue Anforderungen heraus gearbeitet worden sind, und die Service Information Specification Language definiert wurde, kann in diesem Kapitel der obere Teil der Architektur entworfen werden. Zunächst wird hier der Aktions- und Ereignisfluß innerhalb der Architektur beschrieben. Daran lässt sich erkennen, welche Schritte nacheinander abgearbeitet werden müssen. Ausgehend von den Aufgaben, die zu bewältigen sind, werden die dafür vorgesehenen Komponenten für das System vorgestellt und deren Funktionsweise beschrieben. Außerdem wird gezeigt, wie aus dem in der Spezifikationssprache geschriebenen Dokument ein objektorientiertes Modell einer Datenaggregation generiert wird und dort dynamische Aspekte, wie Funktionsauswertung und Bedingungsauswertungen, realisiert werden.

6.1 Architektur des Prototypen

Wie man bereits in State of the Art in Abbildung 3.5 sehen konnte, besteht das gesamte Architekturmodell aus mehreren Schichten. Relevant für diese Arbeit ist nur die Integrations- und Konfigurationsschicht, sowie ein kleiner Teil der Applikationsschicht. Abbildung 6.1 zeigt noch einmal, welcher Teil des gesamten Modells hier bearbeitet wird. Komponenten, welche nicht Gegenstand dieser Arbeit sind, sind deshalb in der Abbildung leicht grau dargestellt.

Im vorrangegangenen Kapitel wurde die Spezifikationssprache bereits ausführlich erklärt und beschrieben, und wird deshalb nun als gegeben vorausgesetzt. Die Managementanwendung, welche die aggregierten Datensätze verarbeiten kann, ist hier nur als Möglichkeit zur Weiterverwendung der Rich Events zu sehen. Sie ist für das Monitoring und Aggregieren der Ressourcendaten nicht notwendig. Der Kern dieses Kapitels ist das Zusammenspiel von Composer und Configurator, bzw. der Vorgang, gewünschte Ressourcenwerte zu sammeln und passend zu aggregieren. Unter dem Deckmantel beider Komponenten sind noch diverse andere Hilfskonstrukte vorhanden, die einen reibungslosen Ablauf garantieren sollen. Diese werden in folgenden Unterabschnitten näher besprochen und ihre Funktionsweise wird