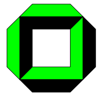 Universität Karlsruhe (TH)

Entwurf und Implementierung eines Prototyps zur skalierbaren, verteilten Messung und Analyse von Internetverkehr

Diplomarbeit am Institut für Telematik
Prof. Dr. Martina Zitterbart
Fakultät für Informatik
Universität Karlsruhe (TH)

von

cand. inform.
Björn Zülch

Betreuer:
Prof. Dr. Martina Zitterbart
Dipl.-Ing. Kendy Kutzner
Dr. Thomas Fuhrmann

Tag der Anmeldung: 15. Juni 2004
Tag der Abgabe: 14. Dezember 2004

Institut für Telematik TeleMatics

Gemäß der Prüfungsordnung der Fakultät für Informatik an der Universität Karlsruhe (TH) erkläre ich hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit hat weder in gleicher oder ähnlicher Form einer anderen Prüfungsbehörde vorgelegen, noch wurde sie veröffentlicht.

Karlsruhe, den 14. Dezember 2004

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Zielsetzung der Arbeit 1

1.2 Gliederung der Arbeit 1

1.3 Danksagung 2

2 Grundlagen **3**

2.1 Overlay-Netzwerke 3

2.2 Peer-to-Peer Netzwerke 4

2.3 Verteilte Hashtabellen 5

2.4 PlanetLab 7

2.5 Verteilte Messungen 8

3 Analyse **11**

3.1 Verwandte Arbeiten 11

3.2 Analysemethoden von Internetströmen 12

3.2.1 Analyse von passiven Messquellen 13

3.2.2 Analyse von aktiven Messquellen 13

3.3 Messquellen 14

3.3.1 Lib-PCAP 15

3.3.2 PlanetLab RAW Sockets 16

3.3.3 PlanetLab-Sensors 17

3.3.4 FlexiNet 17

3.3.5 Aktive Quellen 18

3.3.6 Andere Quellen 19

3.4 Hochbitratige Messströme 19
3.5 Datenkompression 23
3.6 Aggregation und Analyse von Messungen 25
3.6.1 Ergebnisanalyse 25
3.6.1.1 Round-Trip-Time 27
3.6.1.2 IP-TTL 29
3.6.2 Aggregation- und Analysemethoden 30
3.6.3 Funktionsanalytische Analyse 31
3.6.4 Statistische Analyse und Aggregation 32
3.6.5 Sampling 38
3.6.6 Events 40
3.6.6.1 Hoher Paketverlust 41
3.6.6.2 SACK-Pakete 43
3.6.6.3 Starke RTT-Schwankungen 43
3.6.6.4 Starke RTT-Änderung 44
3.6.6.5 Viele Verbindungsanforderungen 45
3.6.7 Aspekte verteilter Messung und Analyse 45
3.7 Speicherung und Abfrage von Daten 47
3.7.1 Lokation der Daten 47
3.7.2 Art der Speicherung 48
3.7.3 Struktur der Daten 49
3.7.4 Abfrage der Daten 51
3.8 Ergebnisnutzung zum Netzmanagement 51
3.9 Sicherheitsaspekte 52
3.9.1 Authentifizierung 53
3.9.2 Verschlüsselte Datenübertragung 53
3.9.3 Anonymisierung von Daten 54
3.9.4 Fazit 56

4 Entwurf **57**

4.1 Design des Frameworks . . . 57
4.2 Platzierung der Module . . . 60
4.3 Knoten-Typen . . . 62
4.4 Aufbau eines Knotens . . . 63
4.4.1 Dispatcher . . . 64
4.4.2 Aufbau eines Moduls . . . 66
4.4.3 Modulschnittstelle . . . 67
4.4.4 Socket Multiplexer . . . 68
4.4.5 Loader . . . 71
4.4.6 Job-Scheduler . . . 72
4.4.7 Logger . . . 72
4.5 Adressierung . . . 73
4.5.1 Adressierungsarten . . . 74
4.5.2 Adress-Layout . . . 74
4.5.3 Modul-Anforderungen . . . 75
4.5.4 Adressauflösung . . . 77
4.5.5 Adressangaben . . . 78
4.6 Service Discovery . . . 79
4.6.1 Aufgaben . . . 79
4.6.2 Erweiterungen . . . 80
4.7 Bootstrapping . . . 80
4.7.1 Granularität der Konfiguration . . . 81
4.7.2 Interaktionsverhalten . . . 81
4.7.3 Ausgangspunkt des Startvorgangs . . . 82
4.7.4 Verhalten bei Exceptions und Fehlern . . . 83
4.8 Zustandsmodell von Modulen . . . 83
4.9 Konfiguration und Management . . . 86
4.9.1 Konfiguration . . . 86
4.9.2 Management . . . 87
4.10 Funktionsmodule . . . 90

4.10.1 Traffic-Inputs 90
4.10.2 Traffic-Analysatoren 90
4.10.3 Aggregation und Eventgenerierung 91
4.10.3.1 TCP-Sampler 91
4.10.3.2 Differenz-Erzeuger 91
4.10.3.3 Korrelator 92
4.10.3.4 Vergleicher 92
4.10.4 Präsentation der Daten 93

5 Implementierung 97

5.1 Knotenhauptprogramm 97
5.2 Datenaustausch und Behandlung von Nachrichten 98
5.3 Aufbau eines allgemeinen Moduls 99
5.4 Aufbau eines Funktionsmoduls 102
5.5 Start einer Konfiguration 104
5.6 Benutzerdefinierte Variablen 105
5.7 RPC-Behandlung 106
5.8 Logger 108
5.9 Beispiel-Modul 108
5.10 Exception-Generierung 112
5.11 Management 113

6 Ergebnisse 117

6.1 Framework 117
6.2 Analysebespiel 118
6.3 Verteilte Messung 122
6.4 Beeinflussung der Analyse 125

7 Zusammenfassung und Ausblick 131

A XML-Konfiguration 133

A.1 Aufbau einer Konfigurationsdatei 133
A.2 Allgemeine Festlegungen 138

B Nachrichten-Typen 141
B.1 Allgemeine Definitionen 141
B.2 Frameworknachrichten 146
B.2.1 Loader 146
B.2.1.1 MSG_LOADER_STARTMODULE 146
B.2.1.2 MSG_LOADER_STARTCONFIG 148
B.2.1.3 MSG_LOADER_LISTCONFIGS 148
B.2.1.4 MSG_LOADER_LISTCONFIGMODS 149
B.2.1.5 MSG_LOADER_STOPCONFIG 149
B.2.1.6 MSG_LOADER_LISTDISPATCHER 149
B.2.1.7 MSG_LOADER_LISTPROTOCOL 149
B.2.1.8 MSG_MODULE_READY 150
B.2.1.9 MSG_LOADER_MODULEFAILED 150
B.2.2 Logger 150
B.2.2.1 MSG_LOGGER_MSG 150
B.2.3 Kernmodul 151
B.2.3.1 MSG_MODCORE_SHUTDOWN 151
B.2.3.2 MSG_MODCORE_LISTHANDLER 151
B.2.3.3 MSG_MODCORE_LISTQUEUE 151
B.2.3.4 MSG_MODIFACE_LISTDEPMOD 152
B.2.4 Module-Management 152
B.2.4.1 MSG_MODINFO_GETVALUE 152
B.2.4.2 MSG_MODINFO_LISTVAR 153
B.2.4.3 MSG_MODINFO_SETVALUE 153
B.2.5 Socket-Multiplexer 153
B.2.5.1 MSG_SMUX_CONNECTSMUX 153
B.2.5.2 MSG_SMUX_SOCKETFAILED 154
B.2.5.3 MSG_SMUX_LISTSOCKETS 154
B.2.5.4 MSG_SMUX_DISCONNECT 155
B.3 Funktionsmodule 155
B.3.1 Pcap-Modul 155

B.3.1.1 MSG_PCAP_PACKET 155
B.3.1.2 MSG_PCAP_GETCOMMAND 155
B.3.2 TCP-Analyse 155
B.3.2.1 MSG_TCPANALYSE_RTT 155
B.3.2.2 MSG_TCPANALYSE_RAW 156
B.3.2.3 MSG_TCPANALYSE_INTERPKT 156
B.3.2.4 MSG_TCPANALYSE_DUPPKT 156
B.3.2.5 MSG_TCPANALYSE_SACK 157
B.3.3 Differenz-Modul 157
B.3.3.1 MSG_DIFF_RTT 157
B.3.3.2 MSG_DIFF_TTL 157
B.3.4 Korrelator 158
B.3.4.1 MSG_KORRELATOR_RTT 158
B.3.4.2 MSG_KORRELATOR_TTL 158
B.3.5 Vergleicher 158
B.3.6 FileWriter 158
B.3.6.1 TLV-Nachricht 158

C Konfiguration des Hauptprogramms **161**

Literatur **165**

Abbildungsverzeichnis

3.1 Bottleneck zwischen zwei Zwischensystemen 19

3.2 Round-Trip-Time am Beispiel einer TCP-RTT 26

3.3 Analyse- und Aggregations Pyramide 30

3.4 Nutzung des Differenzenquotienten zur Analyse der RTT-Werte 31

3.5 Korrelation zwischen zwei Verkehrströmen mit verschiedenen Quellen und Zielen (a) 33

3.6 Korrelation zwischen zwei Verkehrsströmen mit verschiedenen Quellen und Zielen (b) 33

3.7 Korrelation zwischen zwei Verkehrströmen mit gleicher Quelle und verschiedenen Zielen 34

3.8 Korrelation zwischen zwei Verkehrströmen mit verschiedener Quelle und gleichen Zielen 35

3.9 Verkehrstrom zu verschiedenen Zeiten 36

3.10 Sampling-Beispiel 39

4.1 Design des Frameworks 58

4.2 Aufbau eines Framework-Knoten 63

4.3 Struktur des Nachrichten Flusses 65

4.4 Dispatcher-Primitive 66

4.5 Kommunikationsinterface eines Moduls 67

4.6 Socket Multiplexer und Wrapper 69

4.7 Speicherstruktur eines Loggereintrages in einer DHT 73

4.8 Zustandsautomat eines Moduls 84

5.1 Funktionsweise des RPC-Managers 107

6.1 Analyseszenario für eine Analye 119

6.2 Messaufbau 120
6.3 Messergebnisse des Samplers und des Differenz-Erzeugers 121
6.4 Messergebnisse des Korrelators 122
6.5 Konfiguration für eine verteilte Messung und Analyse 123
6.6 beteiligte Rechner in der verteilten Messung 123
6.7 Überlagerung der Samplerergebnisse der RTT 124
6.8 Sampler Ergebnis einer Knotens 125
6.9 Überlagerung der Korrelationsergebnisse 126
6.10 Überlagerung der Samplerergebnisse router/pundit - usc.edu . . 126
6.11 Samplerergebnis router - usc.edu 127

B.1 Aufbau einer Nachricht 141

Tabellenverzeichnis

3.1 Paketheadergrößen von Protokollen 22

3.2 Messdatenraten in verschiedenen Szenarien 23

3.3 Korrelationsmöglichkeiten 35

3.4 Protokollparameter zur Anonymisierung bei TCP/IP 54

4.1 Typdefinitonen für die Modulinformationen 69

4.2 Zugriffsrechte auf eine Variable 69

4.3 Vordefinierte Variablen der Modulinformation 70

4.4 Festgelegte Modul-IDs 75

4.5 Vordefinierte Variablen des funktionales Modulinterfaces 89

4.6 Vordefinierte Parameter eines Knotens 89

4.7 Parameter zur Konfiguration des Pcap-Moduls 94

4.8 Parameter zur Konfiguration des TCPAnalyse-Moduls 95

4.9 Parameter zur Konfiguration des TCP Sampler-Moduls 95

4.10 Parameter zur Konfiguration des TCP Differenz-Moduls 95

4.11 Parameter zur Konfiguration des Korrelators 96

4.12 Parameter zur Konfiguration des Präsentations-Moduls 96

5.1 Programmparameter eines Frameworkknotens 98

5.2 Variablendefinitionen des Logger 109

B.2 Definierte Nachrichtentypen 142

B.1 Typdefinitionen 147

B.3 Typen und Längen für eine TLV-Nachricht 159

Quellcodeverzeichnis

5.1 Modulecore-Interface 99
5.2 ModuleInterface-Interface 102
5.3 ModuleInfo-Interface 105
5.4 Modulbeispiel anhand des TCP-Analyse Moduls (1) 109
5.5 Modulbeispiel anhand des TCP-Analyse Moduls (2) 110
5.6 Beispiel Exception an Hand der Modul-Exception 112
5.7 Management-Bibliothek Interface 114
5.8 TCL-Script des Management-Clients 114
A.1 Beispiel einer Konfigurationsdatei 133
B.1 Allgemeiner-Abschnitt der Modulstartnachricht 146
B.2 Dependency-Abschnitt der Modulstartnachricht 147
B.3 Parameter-Abschnitt der Modulstartnachricht 147
B.4 Aufbau der Startconfig-Nachricht 148
B.5 Aufbau der MSG_LOADER_LISTCONFIGS-Nachricht 148
B.6 Aufbau der MSG_LOADEr_LISTCONFIGMODS-Nachricht . . 149
B.7 Aufbauf der MSG_LOADER_LISTDISPATCHER-Nachricht . . 149
B.8 Aufbau der MSG_LOADER_LISTPROTOCOLS-Nachricht . . . 149
B.9 Aufbau der Modul-Ready-Nachricht 150
B.10 Aufbau der Modulefailed-Nachricht 150
B.11 Aufbau der Logger-Nachricht 151
B.12 Aufbau der Zeige registrierte Nachrichtenhandler-Nachricht . . . 151
B.13 Aufbau der Zeige Queueinhalt-Nachricht 151
B.14 Aufbau der abhängige Module-Nachricht 152
B.15 Aufbau der ModuleInfo-GetValue-Nachricht (Req.) 152

B.16 Aufbau der ModuleInfo–Nachricht (Resp.) 152
B.17 Aufbau der ModuleInfo-Listvariable-Nachricht (Req.) 153
B.18 Aufbau der ModuleInfo-ListVariable-Nachricht (Resp.) 153
B.19 Aufbau der SMux ConnectSMux-Nachricht 153
B.20 Aufbau der SMux Socketfailed-Nachricht 154
B.21 Parameter-Abschnitt der Modulstartnachricht 154
B.22 Parameter-Abschnitt der Modulstartnachricht 155
B.23 Aufbau der Pcap-Paket-Nachricht 155
B.24 Aufbau der Pcap-GetCommand-Nachricht 155
B.25 Aufbau der TCPAnalyse-RTT-Nachricht 156
B.26 Aufbau der TCPAnalyse-RAW-Nachricht 156
B.27 Aufbau der TCPAnalyse-INTERPKT-Nachricht 156
B.28 Aufbau der TCPAnalyse-DUPPKT-Nachricht 157
B.29 Aufbau der TCPAnalyse-SACK-Nachricht 157
B.30 Aufbau der Differenz-RTT-Nachricht 157
B.31 Aufbau der Differenz-TTL-Nachricht 157
B.32 Aufbau der Korrelator-RTT-Nachricht 158
B.33 Aufbau der Korrelator-TTL-Nachricht 158
B.34 Allgemeiner Aufbau einer TLV-Nachricht für den Filewriter . . . 159
C.1 Konfiguration des Startverhaltens 161
C.2 Liste der Module 161
C.3 Modulbeschreibung 162
C.4 Parametereintrag innerhalb eines Moduls 162
C.5 Konfiguration des Hauptprogramms 162

1. Einleitung

1.1 Zielsetzung der Arbeit

Ziel der Arbeit ist es eine prototypische Implementierung eines Frameworks zur verteilten Messung und Analyse von Internetverkehrsströmen vorzunehmen. In einem ersten Schritt wird ein Framework entwickelt, welches auf flexible Art und Weise verteilte Messungen durchführt. Dazu ist es notwendig verschiedene Aspekte und Design-Möglichkeiten zu untersuchen und diskutieren. Ein Gesichtspunkt des Frameworks liegt in der Diskussion wie Peer-to-Peer Netze und verteilte Hashtabellen zur Analyse und Speicherung der Messergebnisse genutzt werden können. Ein anderer Aspekt ist in der Diskussion, wie die Messdaten analysiert und aggregiert werden können, so dass aus diesen konkrete Ergebnisse gewonnen werden können.
In einem zweiten Schritt der Arbeit soll eine prototypische Implementierung des entworfenen Frameworks erfolgen und innerhalb des PlanetLab Testbeds erprobt werden. Dazu wird ein existierendes Programm zur passiven Messung von TCP-Verkehr in das Framework integriert und erweitert.
In einem dritten und letzten Schritt sollen die Ergebnisse der Implementierung und der Messungen vorgestellt und abschließend diskutiert werden.

1.2 Gliederung der Arbeit

Im folgenden Kapitel werden grundlegende Begriffe und Architekturen, die in der Arbeit Verwendung finden, vorgestellt und definiert. Das Kapitel "Analyse" enthält Punkte, die bei einer Definition des Frameworks zur verteilten Messung und Analyse, zu beachten sind. Ebenso werden Methoden zur Messung und Analyse vorgestellt. Diese Punkte werden hinsichtlich der Tauglichkeit für das Framework diskutiert und untersucht.
Die einzelnen untersuchten Aspekte werden anschließend im Kapitel "Entwurf" zu einem Framework zusammengefasst, und einzelne Bestandteile des resultierenden Frameworks werden vorgestellt. Des Weiteren werden die Aufgaben der

einzelnen Teile des Frameworks definiert.
Das Kapitel "Implementierung" erklärt praktische Implementierungsdetails, die bei der Entwicklung des Frameworks relevant sind, näher. Es werden ebenso Details geschildert, wie entworfene Konzepte aus dem Entwurf in der Implementierung gelöst werden und welche Strategien geändert werden müssen, damit eine Implementierung erfolgen kann.
Im Kapitel "Ergebnisse" werden aus exemplarischen Messungen und Analysen die gewonnenen Erkenntnisse und Ergebnisse vorgestellt und bewertet. Abschließend findet im Kapitel "Zusammenfassung und Ausblick" eine Betrachtung der Ergebnisse statt.

1.3 Danksagung

Ich möchte mich bei allen bedanken, die mich wärend meines Studiums unterstützt haben.
Besonderer Dank gilt dem Institut für Telematik unter der Leitung von Prof. M. Zitterbart sowie meinen Betreuern Kendy Kutzner und Dr. Thomas Fuhrmann für die viele nützlichen Hinweise und Ratschläge während der Diplomarbeit.
Ein sehr großen Dank geht an meine Eltern, die mir das Studium ermöglicht und wärend dessen immer unterstützt haben.

2. Grundlagen

In diesem Kapitel werden die grundlegenden Konzepte und Architekturen beschrieben, auf denen diese Arbeit aufbaut. Das Kapitel umfasst eine Beschreibung der PlanetLab-Architektur sowie die Definitionen von Overlay-Netzwerken, Peer-to-Peer Netzwerken und von verteilten Hashtabellen. Weiterhin wird eine kurze Einführung in die Thematik des Messens von Internetverkehrsströmen gegeben.

2.1 Overlay-Netzwerke

> **Definition:**
> *Ein Overlay-Netzwerk ist ein logisches Netzwerk, welches auf Basis eines oder mehrerer Netzwerke existiert. Im Allgemeinen ist ein Overlay-Netzwerk aus Rechnern aufgebaut, die untereinander kommunizieren. Diese Rechner sind in verschiedenen Netzwerken installiert. Die Rechner zeichnet weiterhin aus, dass sie ein gemeinsames Overlay-Protokoll zur Kommunikation nutzen. Dieses Protokoll implementiert eine eigene Adressierung, Wegewahl und Vermittlung von Paketen und Daten. Damit erweitern Overlay-Netze die Funktionalität eines bestehenden Netzwerkes. Die Topologie, die Adressierung, Wegewahl und Vermittlung innerhalb des Overlays sind unabhängig von der vorhandenen Netzinfrastruktur-Topologie.*

In einer gewissen Weise war das Internet selbst ein Overlay-Netzwerk, da es in vielen Teilen logisch über das Telefon-Netzwerk gelegt wurde. Unabhängig von der Netzwerk-Schicht (OSI-Schicht 3) können innerhalb eines Overlay-Netzes verschiedene Eigenschaften und Verhaltensweisen geändert werden. Dies sind z.B. Routing, Quality of Service, Adressierungsschemata oder Sicherheit. Overlay-Netzwerke werden auch als Netzwerke auf Applikationsebene oder als logische Netzwerke bezeichnet. Diese Art von Netzwerken kann von der physikalischen

Struktur eines Netzwerkes und dessen Nachteilen abstrahieren. Es wird mit diesem Ansatz versucht, die Performance von Applikationen zu steigern, indem die Kommunikation entlang von Overlay-Kanten stattfindet und somit Performancenachteile auf physikalischen Verbindungen umgangen werden können[1]. Nachteil eines Overlays ist, dass eine zusätzliche Ebene innerhalb der Kommunikationsschichten etabliert wird. Dies kann zusätzliche Fehler und Komplexität hinzufügen. Overlay-Netzwerke und ihre Eigenschaften sind aktuelle Forschungsthemen. Diese Themen erstrecken sich über die Gebiete Routing, Dienstlokalisierung, Verteilung von Daten, Verfügbarkeit von Daten und weitere Themen. Es lassen sich verschiedene Beispiele für Overlay-Netzwerke angeben. So handelt es sich bei dem E-Mail System um ein Overlay-Netz, da die Pfade, die eine E-Mail zurücklegt, unabhängig von den Transportpfaden im Internet sind. Ebenso wird eine eigene Adressstruktur implementiert.

2.2 Peer-to-Peer Netzwerke

Der Begriff der Peer-to Peer (P2P) Netzwerke ist unter historischer Sicht nicht neu. Die Dienste des Internets wurde als P2P-System konzipiert. Einige Programme wie *Internet News*[2] kommen einem Peer-to-Peer Netz nahe. Durch die Verbreitung des World-Wide-Web (WWW) wurde der Eindruck einer Client-Server Architektur des Internets erweckt. Heute wird der Begriff der Peer-to-Peer Netzwerke maßgeblich durch Filesharing-Programme wie Napster geprägt. Jedoch ist ein P2P-System mehr als nur Filesharing zwischen verschiedenen Rechnern. Eine Definition von Peer-to-Peer Netzwerken gibt [Shir00]:

> **Definition:**
> *Peer-to-Peer is a class of application that takes advantage of resources – storage, cycles, content, human presence – available at the edges of the Internet. Because accessing these decentralized resources means operating in an environment of unstable connectivity and unpredictible IP addresses, P2P nodes must operate outside the DNS system and have significant or total autonomy from central servers.*

Ebenso findet sich in [Shir00] eine Klassifikation von bekannten Netzwerken, die darüber Auskunft gibt, ob es sich bei dem gegebenen Netz um ein P2P-System handelt. Eine andere Definition für Peer-to-Peer Netzwerke kann in [Scho02] gefunden werden. Peer-to-Peer (P2P) Netzwerke werden oft auf Basis eines Overlay-Netzwerkes betrieben. Die Hauptcharakteristik von P2P-Netzen

[1] z.B. durch Erhöhung der Ausfallsicherheit mit Hilfe von mehreren alternativen Pfaden (Resilient Overlay Networks [ABKM01a] und [ABKM01b])

[2] Internet News ist ein System, in dem Nachrichten ausgetauscht werden. Die Nachrichten werden zwischen verschiedenen News Servern ausgetauscht. Nachrichten werden in hierarchischen themenbezogenen Gruppen; sog. Newsgroups eingeteilt. Eine Nachricht ist ähnlich einer E-Mail aufgebaut. Das Format einer News-Nachricht ist in [HoAd87] definiert. Mehr zu Internet News ist in [KiDa01] Kapitel 20 zu finden.

liegt darin, dass die einzelnen Knoten der P2P-Netze gleichzeitig sowohl Client als auch Server sein können. Als Grundlage zur Indizierung und Verwaltung von Daten werden zum Teil verteilte Hashtabellen (Distributed Hash Tables - DHT) genutzt. Zum anderen existieren auch zentrale Indizierungen, bei denen nur die Daten auf den einzelnen Peers liegen. Suchanfragen werden an eine zentrale Stelle geleitet und dort ausgewertet.
Ein Vorteil von Peer-to-Peer Netzwerken liegt darin, dass die gesamte Rechenkapazität und der gesamte Speicher aller P2P-Knoten genutzt werden kann. Es müssen keine teuren, leistungsfähigen und zentralen Systeme installiert werden. Jedoch stellt die Konfiguration und das Management eine Herausforderung dar, die effizient gelöst werden muss, da es sich um ein verteiltes System mit keiner zentralen Komponente zum Management handelt. Beispiele für P2P-Netzwerke sind Gnutella (www.gnuella.com), Napster (www.napster.com) oder Kazaa (www.kazaa.com).

2.3 Verteilte Hashtabellen

Verteilte Hashtabellen sind eine Erweiterung der Hashtabellen. Diese erstrecken sich über mehrere Knoten. Bei Hashtabellen werden *Schlüssel–Wert* Paare geschickt auf eine Tabelle abgebildet. Hashing ist ein Kompromiss zwischen Zeit- und Platzbedarf. Einen Überblick über Hashtabellen und ihre Eigenschaften ist in [Sedg97] zu finden.
Eine Hashtabelle besteht immer aus nur einer Tabelle. Bei verteilten Hashtabellen geht man einen Schritt weiter und verteilt den Schlüsselraum auf mehrere Rechner (Knoten). Dazu ist es notwendig zu wissen wie ein Teil des Schlüsselraums zu finden ist. Es wird dazu eine Routing-Tabelle aufgebaut, die Anfragen auf *Schlüssel–Wert* Paare an andere Knoten weiterleiten kann. Die Suche benötigt im Mittel $O(\log n)$[3] Knoten bis der Knoten gefunden wurde, der den Wert zu dem angegebenen Schlüssel gespeichert hat. Bei CAN (Content Area Network) dauert eine Suche im Mittel $O(\log^2 n)$. Jeder Knoten ist mindestens für einen Teil des Schlüsselraums zuständig. Durch Hinzunahme von mehreren Knoten, die für einen Teilschlüsselraum zuständig sind, wird die Redundanz der Daten erhöht. Die Zuständigkeiten für den jeweiligen Teilschlüsselraum ändern sich minimal mit dem Beitritt und dem Verlassen eines Knotens zur verteilten Hashtabelle. Damit wird sichergestellt, dass nur ein kleiner Teil des Schlüsselraums inkonsistent ist. Konsistent heißt in diesem Zusammenhang, es wird nur ein kleiner Teil des Schlüsselraums auf den einen neuen Knoten übertragen. Diese Verschiebung von Zuständigkeiten ist lokal begrenzt, so dass nicht alle Knoten in die Verschiebung von Schlüsseln involviert sind. In [KLLL+97] heißt es dazu:

> *...This property says that if items are initially assigned to a set of buckets $\mathcal{V}_1$ and then some new buckets are added to form $\mathcal{V}_2$ then an item may move from an old bucket to a new bucket, but not*

[3] Dies ist der Fall, wenn es sich um ein DHT auf Basis eines Chord-Netzes handelt.

> *from one old bucket to another. This reflects one intuition about consistency: when the set of usable buckets changes, items shold only move if neccessary to preserve an even distribution...*

Mit *Buckets* sind in diesem Abschnitt einzelne Knoten der DHT gemeint. Dies beschreibt, wie die Daten auf einzelne Knoten verteilt werden. Knoten bestimmen durch die Wahl[4] ihrer Knoten-ID ihre Position in dem System der verteilten Hashtabelle und für welchen Teil des Schlüsselraums sie zuständig sind. Die Knoten-ID bildet den Schlüsselraum auf die einzelnen Knoten ab und teilt einzelnen Knoten Zuständigkeiten zu. Ebenso sorgt die Hashfunktion dafür, dass

> *...they distribute items among buckets in a balanced fashion...*

Dies bedeutet, dass die Daten gleichmäßig auf alle Knoten verteilt werden. Aber gleichzeitig ist die Variation der Knoten gering, auf denen die Daten gespeichert werden sollen. Dazu wird in [KLLL+97] folgende Aussage getroffen:

> *... The property sasy that accross the entire group, there are at most $\sigma(i)$ different opinions about which bucket should contain the item. Clearly, a goog hashfunction should habe low spread over all items...*

Hinzu kommt die Aussage über den Füllgrad (Anzahl der Daten pro Knoten) eines Knoten.

> *... The property says that there are at most $\lambda(b)$ distinct items that at least on person thinks belongs in the bucket. A good consistent hashfunction should also have low load...*

Darin kommt zum Ausdruck, dass die Auslastung pro Knoten gering sein sollte. Im Zusammenhang mit der Gleichverteilung ergibt sich daraus, dass auf einem Knoten übermäßig viel mehr an Daten gespeichert werden, als auf einem anderen. Zusammenfassend sollte die Hashfunktion möglichst den folgenden Eigenschaften gehorchen:

- Balance
- Monotonicity
- low Spread
- low Load

[4] Bei Chord besteht die Knoten-ID aus einem Hash über die Rechner IP-Adresse. Bei anderen verteilten Hashtabellen wird die Knoten-ID aus anderen Parametern gewonnen.

Werden diese vier Eigenschaften erfüllt, so handelt es sich um *Consistent Hashing*. Gelten für eine Hashfunktion diese Eigenschaften, werden die Werte möglichst gleichmäßig auf alle Knoten einer Verteilten Hashtabelle verteilt. Es müssen aber auch entsprechende Schlüssel gewählt werden, die eine möglichst gleichmäßige Verteilung der Daten auf die einzelnen Knoten erzeugen. Nur über die Schlüssel kann wieder auf die Daten zugegriffen werden. Das bedeutet, die Schlüssel müssen genügend verschieden sein, aber gleichzeitig muss ein Algorithmus hinter den Schlüsseln sein. Denn sind die Schlüssel gleich, werden diese auf dem gleichen Knoten bzw. im gleichen Tabelleneintrag gespeichert und es kommt zu Kollisionen, die aufgelöst werden müssen. Für Hashtabellen existieren verschiedene Verfahren, wie diese Kollisionen aufgelöst werden können. Mehr zu diesen Verfahren findet sich in [Sedg97]. Beispiele für diese Hashfunktionen sind die kryptographischen Hashfunktionen MD-5 (Message Digest Nr. 5) und SHA-1 (Secure Hash Algorithmus Nr. 1).[5] SHA-1 wird in *Chord*[6] genutzt um die Schlüssel zu hashen. SHA-1 besitzt Eigenschaften des *Consistent Hashing*.
Mit den Eigenschaften der Hashfunktionen lassen sich Eigenschaften der verteilten Hashtabellen ableiten:

- Lastverteilung
- Skalierbarkeit
- Selbstorganisation
- Unabhängig von Anwendungen[7]

2.4 PlanetLab

PlanetLab ist eine geographisch verteilte Plattform. Sie dient der Entwicklung und Erprobung neuer Netzwerk-Dienste. Diese Dienste basieren meist auf der Nutzung von Overlay-Netzwerken. Die PlanetLab Maschinen befinden sich global verteilt. Bevorzugt befinden sich diese Maschinen an Universitäten, Forschungseinrichtungen und bei Internet-Service-Providern . PlanetLab dient einer Vielzahl von Forschungsprojekten aus dem Bereich Overlay-Netzwerke als Testbed zur Erprobung. Forschungsschwerpunkte von PlanetLab sind Verteilte Systeme (Distributed Systems), Datenbank-Forschung und Netzwerk- bzw. Telekommunikations-Forschung. Die Projekte erstrecken sich über Datenaustausch (Filesharing), Netzwerk-integrierter Speicher, Inhaltsverteilung (Content Distribution Network), Routing und Multicast in Overlays, Quality of Service (QoS, Dienstgüte) Lokalisations-Diensten und Netzwerkmessungen.

[5] Mehr Informationen über die kryptographische Hashfunktionen MD-5 und SHA-1 finden sich in [Schä03].

[6] Chord ist eine Implementierung einer verteilten Hashtabelle. Mehr Informationen zu Chord befinden sich in [SMKK+01]

[7] Da Schlüssel keine semantische Bedeutung besitzen

Dabei stellt PlanetLab eine neue Art von Testbed dar. Die Neuerung bei PlanetLab liegt darin, dass eine Entwicklung von neuen Netzwerkdiensten unter realen Bedingungen ermöglicht wird. Ziel ist es ein Service-orientiertes Netzwerk aufzubauen, welches über eine große Anzahl von Maschinen skaliert.
PlanetLab unterscheidet sich von anderen Testbeds wie Internet2, Globus, ABone und XBone dahingehend, dass die einzelnen Maschinen "lose" über das Internet gekoppelt sind und keine exklusiven hochbitratigen Netzverbindungen wie innerhalb eines Grids existieren. Ebenso unterscheidet sich PlanetLab durch die große Anzahl von Maschinen weltweit. Andere Testbeds umfassen oft nur wenige dutzend Maschinen.
Im Oktober 2004 umfasste PlanetLab ca. 430 Maschinen an ca. 200 verschiedenen Institutionen weltweit. Die Maschinen sind über das Internet[8], das amerikanische Forschungsnetz *Internet2* (www.internet2.edu) und weitere internationale Forschungsnetze erreichbar.
Die Architektur von PlanetLab ist so gestaltet, dass viele Projekte gleichzeitig auf den einzelnen Maschinen laufen können. Dazu wird die Maschine für jedes Projekt – auch Slice genannt – virtualisiert. Für jedes Projekt existiert eine Virtuelle Maschine. Jedes Projekt wird in einer eigenen Umgebung ausgeführt. Alle Projekte teilen sich auf einem PlanetLab-Knoten einen gemeinsamen Kernel. Es müssen jedoch, hinsichtlich der Nutzung der Maschinen durch mehrere Projekte gleichzeitig, Maßnahmen getroffen werden um die Ressourcen der Maschinen aufzuteilen und zu limitieren.
Weitere Informationen zu PlanetLab finden sich in [PACR02] und [ABCKM+04], sowie unter www.planet-lab.org.

2.5 Verteilte Messungen

Verteiltes Messen (Distributed Measurement) von Internetverkehrsströmen stellt eine Erweiterung vom einfachen auf eine Maschine beschränktes Messen dar. Bei der Ende-zu-Ende Messung werden Informationen über das Verhalten von Paketen und Verkehrsströmen gemessen und ausgewertet. Dabei werden die Messdaten in Endsystemen erhoben. Der Fokus bei Endsystemen, die die IP-Protokolle implementieren, liegt auf der Messung und Auswertung von Parametern, die Verbindungen zwischen den beteiligten Endsystem charakterisieren. Dazu gehören Parameter wie die Round-Trip-Time von TCP (TCP-RTT) oder Time-to-Live des IP-Protokolls (IP-TTL). Eine Messung der Parameter kann direkt in den Endsystemen durchgeführt werden, oder auch in Zwischensystemen[9]. Die Verteilung einer Messung kommt darin zum Ausdruck, dass mehrere Maschinen Verkehrsströme beobachten. Diese Maschinen sammeln getrennt und unabhängig voneinander Messdaten. Es besteht auch die Möglichkeit, Messdaten von entfernten Rechnern nicht selber zu messen, sondern einfach nur passiv abzufragen. Dies geschieht meist über ein Standard-Protokoll wie SNMP (Simple Network Management Protocol). Es können aber

[8] *... All of the machines are connected to the Internet...* (http://www.planet-lab.org/php/overview.php, 26.Jul. 2004)

[9] In [Zülc04] wird beschrieben wie eine Messung in Zwischensystemen durchgeführt werden kann und welche Parameter sich dabei Messen lassen.

auch eigenentwickelte Protokolle genutzt werden.
Der nächste Aspekt, der bei der verteilten Messung eine Rolle spielt, ist die Art und der Ort der Datenspeicherung. Es lassen sich zwei Arten unterscheiden. Zum einen die zentrale Datenhaltung und zum anderen die verteilte Datenhaltung. Bei der zentralen Datenhaltung ist zu berücksichtigen, dass alle Messknoten ihre Daten an eine oder nur sehr wenige Maschinen übermitteln, die dann die Speicherung der Daten vornehmen. Für die Messknoten bedeutet dies, dass die Lokalisierung der Speicherknoten mit sehr geringem Aufwand geschehen kann. Jedoch bedeutet dies für den Speicherknoten selber, dass er zu einem Engpunkt (Bottleneck) in der gesamten Architektur werden kann, wenn viele Messknoten Messdaten ablegen möchten. Ebenso stellt der zentralistische Ansatz im Bezug auf die Ausfallsicherheit ein Problem dar. Die zentrale Datenhaltung stellt einen Single-Point of Failure dar. Mit dem Ausfall des Speicherknotens kann das gesamte System nicht mehr arbeiten und muss beendet werden.
Der Ansatz der verteilten Speicherung von Messdaten ist im Bezug auf die Ausfallsicherheit eleganter. Hierbei werden die Messdaten entweder direkt auf den verteilten Messknoten gespeichert, oder auf vielen verteilten Speicherknoten. Wobei der benötigte Speicherplatz auf den verteilten Speicherknoten gleich oder sogar größer ist, gegenüber der zentralen Speicherung. Es muss mehr Aufwand in die Lokalisierung der Daten auf den einzelnen Knoten investiert werden, und Daten, die analysiert werden sollen, müssen erst auf einem Analyseknoten bereitgestellt werden. Dies bedeutet eine Erhöhung der Kommunikation zwischen den einzelnen Knoten. Wodurch die Anforderungen an das Kommunikationsnetz zwischen den Knoten durch mehr Verkehr steigen. Ebenso ist das Management der verteilten Knoten ein Nachteil, da unter Umständen keine zentralen Knoten vorhanden sind. Dies stellt neue Anforderungen an die Organisation des Managements und die Implementierung.
Ein weiterer Aspekt der verteilten Messung liegt darin, wie Daten erfasst werden. Dazu stehen zwei Möglichkeiten zur Verfügung. Dies ist zum einen die Aktive Messung, in dem aktiv Pakete erzeugt werden und von einer Gegenseite Antwortpakete generiert werden. Aus diesen beiden Paketen werden dann Messwerte generiert. Beispiele für die aktive Messung sind “Ping” und “traceroute”.
Dem gegenüber steht die passive Messung. Hierbei werden keine Pakete erzeugt, sondern es werden Paketströme mitgelauscht und auf ihre Eigenschaften hin analysiert. Ein Beispiel für das passive Messen von Internetpaketströmen ist in [Zülc04] zu finden. Weitere Erklärungen zu den beiden Messarten und die Fehler, die dabei gemacht werden können, sind ebenfalls in [Zülc04] zu finden. Weitere Arbeiten zur Messung und Analyse von Internetverkehr finden sich in [Paxs97], [padh01], [bala96], [paxs01], [mell02], [paxs98] und [bolo93].

3. Analyse

In diesem Abschnitt sollen Funktionen und einzelne Teilaspekte, die zur Entwicklung eines verteilten Messsystems nötig sind, auf ihre Möglichkeiten und Problematiken hin untersucht und beschrieben werden. Dabei sollen für die einzelnen Teile verschiedene mögliche Verfahren und Methoden erörtert und diskutiert werden.

3.1 Verwandte Arbeiten

In diesem Abschnitt sollen kurz einige Arbeiten vorgestellt werden, die ebenfalls auf dem Gebiet der Verteilten Messung und Analyse verfasst wurden. Es werden kurz die Verfahren und Ergebnisse geschildert. Diese Liste der vorgestellten Arbeiten ist nicht vollständig, da sehr viele Arbeiten zu diesem Thema existieren. Einen weiteren Überblick über Messungen und Analysen von Internetverkehr findet sich in [Zülc04] Kapitel 3.4.

Die erste Arbeit, die vorgestellt werden soll, von C. Simpson und G. Riley nennt sich *Neti@Home* [SiRi04]. Dieser Name wurde in Anlehnung an *Seti@Home*[1] gewählt. Jedoch bestehen Änderungen gegenüber Seti@Home. Zum einen handelt es sich bei Seti@Home nicht um die Messung und Analyse von Internetverkehr, sondern um die Datenanalyse eines Radioteleskopes. Ebenso findet bei Seti@Home die Datensammlung zentral statt und alle Endsysteme analysieren die Daten und senden die Ergebnisse zurück an Seti@Home. Bei Neti@Home findet dagegen die Datenerfassung verteilt in den Endsystemen statt. Es handelt sich bei diesen Daten um Messwerte von Internetverkehr. Diese werden dann zur Analyse an einen zentralen Server gesendet. Die Ergebnisse können von diesem Server zentral abgerufen und angezeigt werden. Neti@Home unterstützt bei den Endsystemen die Anonymisierung der Messdaten. Der Grad der

[1] Mehr Informationen zu Seti@Home finden sich unter http://setiathome.ssl.berkeley.edu. Bei Sei@Home handelt es sich nicht um ein System zur Analyse vom Verkehrströmen, sondern zur Analyse von Signalen eines Radioteleskopes. Bei Neti@Home soll der Name nur den Aspekt der Verteilung hervorheben.

Anonymisierung kann der Benutzer selber einstellen. Der Umfang der Parameter beinhaltet neben Parameter aus TCP auch Parameter aus den Protokollen UDP und ICMP.

In der Arbeit von U. Hofmann und I. Milouchewa [HoMi01] wird ein System vorgestellt, in dem, verteilt in einem Netz, verschiedene Messagenten platziert sind. Diese messen verschiedene Parameter des Verkehrs. Überwiegend findet die Messung aktiv statt, es wird jedoch auch passiv gemessen. Der Fokus dieser Arbeit liegt auf der Messung von QoS-Werten, bzw. gemessene Werte werden auf QoS-Parameter abgebildet. Die Auswertung der Daten findet jedoch zentral statt. Von dieser zentralen Stelle fragen die Netzelemente die benötigten Parameter für das Management von QoS-Verbindungen ab.

Eine andere Arbeit von Connie Logg, Jiri Navratil und Les Cottrell [CLCo04] befasst sich hauptsächlich mit der Auswertung von verteilten Messungen und versucht mit Hilfe von Korrelationen neue Erkenntnisse zu gewinnen. Das Tool setzt auf einige weltweit verteilte Knoten. Die Messungen erfolgen im *Full Mesh*[2]. Es werden die Tools IPerf, Traceroute und Ping zur Erfassung der Messdaten eingesetzt. Die gemessenen Daten werden dann an einen zentralen Knoten gesendet und dort zentral ausgewertet.

In der Arbeit von Deb Agarwal, José María González, Goujun Jin und Brian Tierney [DATi03] wird ebenfalls ein System zur verteilten Messung von Internetverkehr vorgestellt. Dabei werden die Daten über passive optische Empfänger gesammelt und an dem jeweiligen Knoten ausgewertet. Eine Messung von Internetverkehr muss von Client-Maschinen aus aktiviert werden. Erst dann beginnen die Messknoten mit der Erfassung von Paketen. Die Pakete werden verlustfrei komprimiert und an den Client gesendet, der die Messung aktiviert hat. Dort findet dann eine Auswertung der Messung statt.

Im Bereich der Anwendung von Ergebnissen von Messungen und Analysen von Internetverkehr muss die Arbeit von David Andersen, Hari Balakrishnan, M. Frans Kaashoek und Robert Morris [ABKM01b] zum Thema *Resilient Overlay Networks* (RON) genannt werden. Hier wird ein Modell vorgestellt, wie Messungen genutzt werden können um die Performance von Verbindungen zu verbessern. Diese Arbeit setzt auf die Messung von Verbindungen durch Pings. Die Verbindungen werden über ein Overlay geroutet, wenn sich dadurch Verbesserungen der Performance ergeben. Dazu werden die Verbindungen in dem Overlay gekapselt und bis zum Zieloverlayknoten transportiert.

3.2 Analysemethoden von Internetströmen

Zur Messung und Analyse von Internetverkehrsströmen gibt es mehrere Varianten, wie dies geschehen kann. Zum einen gibt es verschiedene Orte an denen Analysen durchgeführt werden können. Zum anderen existiert eine Unterscheidung der Analysearten nach passiver und aktiver Messung. Eine Klassifikation

[2] Jeder Knoten vermisst jeden anderen.

der beiden Messmethoden wird in [Zülc04] Kapitel 3.1.1 und Kapitel 3.1.2 geschildert. In den beiden folgenden Abschnitten sollen kurz die beiden Verfahren vorgestellt und diskutiert werden.

3.2.1 Analyse von passiven Messquellen

Bei der passiven Messung werden die Parameter passiv gesammelt. Dies bedeutet, dass kein Verkehr erzeugt wird, sondern die Pakete, die von dem Messsystem empfangen und gesendet werden, analysiert werden. Die Pakete können zum einen kopiert und anschließend analysiert werden. Hierbei findet die Analyse unabhängig von dem Empfang und dem Versenden von Paketen statt. Zum anderen können die Pakete direkt abgefangen und *on-the-fly* analysiert werden. Jedoch erfahren die Pakete hierbei eine Verzögerung, da die Pakete erst weitergereicht werden, wenn die Analyse beendet wurde. In dieser Form werden Pakete in FlexiNet ([FHSZ02]) und in [Zülc04] bearbeitet. Wenn mehr Pakete empfangen werden als von einem FlexiNet-Modul verarbeitet werden können, laufen evtl. Puffer über. Damit kommt es zu Paketverlusten.
Diese Form der Analyse und Paketsammlung kann an verschiedenen Orten stattfinden. Diese Analyse kann in Endsystemen oder auch in Zwischenknoten (sog. Aktiven Knoten) stattfinden. In Endsystemen bekommt man den kompletten Verkehr mit, den das System erzeugt und empfängt. Um in den anderen genannten Systemen an Verkehr zu kommen, müssen diese Router sein. Ebenfalls findet hier immer eine Ende-zu-Ende Analyse statt. Im Gegensatz dazu beschränkt sich die Analyse inden Aktiven Knoten auf eine Teilstreckenanalyse zwischen Aktiven Knoten und Endsystem. Befindet sich der Aktive Knoten nicht am Rand des Netzwerkes, so bekommt dieses System nicht den kompletten Verkehr mit, den das System erzeugt, sondern nur den Teil, der über diesen Aktiven Knoten gesendet wird. Eine Ende-zu-Ende Analyse kann nur aus zwei Teilanalysen zusammen gesetzt werden, in dem die beiden Analysen zum einen zwischen Client und Aktiven Knoten und zum anderen zwischen Aktiven Knoten und Server kombiniert werden. Aus der Analyse in einem Aktiven Knoten ergeben sich einige neue und veränderte Möglichkeiten, wie Parameter gemessen und berechnet werden. Ein Verfahren, wie diese Parameter gemessen werden können, wird in [Zülc04] Kapitel 4 beschrieben. Es muss ebenso beachtet werden, dass in einem Zwischensystem nicht alle Pakete einer Verbindung erfasst werden können. Dies liegt daran, dass die Routen oft asymmetrisch sind und nur eine Richtung einer Verbindung über einen Router gesendet werden.

3.2.2 Analyse von aktiven Messquellen

Bei der aktiven Analyse von Internetverkehr erzeugt die Analyseinstanz selbständig Pakete. Damit wird eine Reaktion von einem Endsystem provoziert. Diese Reaktion auf ein versendetes Paket wird aufgezeichnet und analysiert. Es handelt sich bei der aktiven Analyse immer um eine Ende-zu-Ende Analyse, da die Pakete in einem Endsystem erzeugt werden und im gleichen Endsystem die Antwort der Gegenseite analysiert wird. Damit ergibt sich für den Ort an dem die Messungen durchgeführt werden, dass dies immer in einem *Endsystem*

durchgeführt wird.[3] Es existiert noch eine Variante der Aktiven Messung, bei der das System, welches die Pakete erzeugt, nicht das System ist, auf dem die Analyse durchgeführt wird. Hierbei werden Pakete auf einem Endsystem erzeugt und diese Pakete auf einem anderen Endsystem mitgelauscht. Mit dieser Methode werden Zwischensysteme analysiert, und es kann beobachtet werden, welche Veränderungen diese evtl. an den Paketen vornehmen.

3.3 Messquellen

Die verschiedenen Messquellen stellen einen sehr wichtigen Bestandteil eines zu entwerfenden Frameworks dar. Von diesen Quellen aus werden die ermittelten Daten an verschiedene Analysatoren verteilt und dort analysiert. Da es sich um ein System zur verteilten Messung und Analyse von Verkehrsströmen handelt, kommt diesem Aspekt eine wichtige Bedeutung zu. Betrachtet man dies unter dem Augenmerk, dass evtl. verschiedene heterogene Systeme zum Einsatz kommen können, folgt daraus, dass verschiedene Quellen für Messdaten erörtert werden müssen, damit das resultierende System flexibel ist um in verschiedenen Umgebungen laufen zu können und dort eine Messwerterfassung stattfinden kann.
Es lassen sich prinzipiell zwei verschiedene Arten unterscheiden, wie die Messdaten erfasst und generiert werden können. Als Möglichkeiten stehen hier die Aktive- und Passive-Messung von Verkehrsströmen zur Verfügung. Bei der Aktiven-Messung werden Daten durch aktive Generierung von Paketen und die entsprechende Reaktion einer Gegenseite erzeugt. Zur Aktiven Messung zählen folgende Tools:

- traceroute ([Jacoc])
- ping
- etc...

Dem gegenüber existiert das Verfahren der Passiven Messung von Daten. Hierbei werden nur Paketströme, die durch ein Netz/Rechner übertragen werden, mitgelauscht. Aus den Paketen werden dann Messwerte abgeleitet. Es existieren folgende Quellen zur Passiven Messung:

- Lib-PCAP ([Jacoa])
- FlexiNet ([HSSW+02])
- PlanetLab-Raw/Admin Sockets
- PlanetLab-Sensors ([RoPe03])

[3] Auch wenn es sich um einen Router oder ein anderes Zwischensystem handelt, wird dieses System automatisch zu einem Endsystem.

- etc...

Diese aufgelisteten Quellen beziehen sich auf die Erfassung von Messdaten in internetbasierten Netzwerken und Systemen. Jedoch existieren neben dem Internet noch weitere Netzwerke und Transporttechniken, in denen Leistungsbewertungen durchgeführt werden können. Hierzu zählen ins besondere die Netze zur Telefonie[4] sowie paketvermittelte Netzwerke wie X.25. In den folgenden Unterpunkten werden die einzelnen Tools und Varianten der Messdatenerfassung vorgestellt und Problematiken bei ihrer Nutzung analysiert.

3.3.1 Lib-PCAP

Die Lib-PCAP (Packet Capturing Library) stellt eine häufig verwendete Methode zur Gewinnung von Internetverkehr dar. Sie abstrahiert von verschiedenen betriebssystemspezifischen Möglichkeiten Paketströme auf Netzwerkschnittstellen zu erfassen und speichern. Die Lib-PCAP stellt eine einheitliche Schnittstelle zur Erfassung von Paketdaten bereit. Zur Filterung der Pakete nutzt die Bibliothek die Syntax des Berkeley-Packet-Filters (BPF) um bestimmte Paketströme zu beobachten. Die PCAP-Bibliothek wird von den Programmen Ethereal ([Comb]) und tcpdump ([Jacob]) genutzt.
Die PCAP-Bibliothek unterscheidet beim Mitlauschen von Paketströmen zwischen zwei verschiedenen Modi. Es handelt sich dabei um die beiden Modi *Online-Capturing* und *Offline-Capturing*. Beim Online-Capturing werden die Pakete direkt von der Netzwerkkarte entnommen und der Analyse oder einer Datei zugeführt. Bei dem Offline-Capturing kommen die Pakete direkt aus einer Datei und nicht von einer Netzwerkkarte. Dies können gespeicherte Paket-Traces von tcpdump oder Ethereal sein, die nachträglich analysiert werden sollen.
Bei der PCAP-Bibliothek werden die Pakete schon innerhalb des jeweiligen Betriebssystemkernels kopiert. Dadurch wird die folgende Verarbeitung der Pakete in einem Analyse-Programm von der Weiterleitung der Pakete über das Netzwerk getrennt. Somit tritt keine Verzögerung bei der Weiterleitung auf und die Verarbeitung kann parallel zum Erfassen der Pakete erfolgen.

Die Lib-PCAP stellt somit ein mächtiges Werkzeug dar, mit dessen Hilfe Pakete zu Analysezwecken gewonnen werden können. In vielen Betriebssystemen ist die Bibliothek verfügbar. Mit dem *Berkeley-Packet-Filter* (BPF) verfügt die Lib-Pcap über ein mächtiges Filterwerkzeug um Paketdaten auf sehr flexible Art und Weise zu erfassen. Zum Beispiel filtert das folgende Beispiel

```
src host 10.15.0.2 dest net 141.3.0.0 mask 255.255.0.0 ip proto tcp
dst port 80 src port any
```

[4]Hiermit sind die klassischen leitungsvermittelten Netze der Telefonie gemeint. Heute befindet sich die Telefonie in einem Umbruch. Es existieren immer mehr Telefonie-Systeme, die auf Basis von Paketvermittlung arbeiten (Voice over IP)

alle TCP Verbindungen heraus, die von dem Endsystem *10.15.0.2* ausgehen und das Zielnetz *141.3.0.0/16* haben. Zusätzlich ist der Zielport (Anwendung) spezifiziert. Hier handelt es sich um HTTP-Pakete. Weitere Beispiele sind unter http://www.tcpdump.org/tcpdump_man.html zu finden. Aktuell existiert jedoch keine PCAP-Bibliothek für die PlanetLab-Umgebung. Dies kann verschiedene Gründe haben. Zum einen kann es sehr aufwendig sein, wenn der komplette Verkehr eines PlanetLab-Knoten für alle Slices (siehe auch Kapitel 2.4) kopiert und an alle Slices verteilt werden muss. Hinzu kommt, dass viele Slices nebeneinander ausgeführt werden können und die Ressourcen auf den Knoten beschränkt sind. Nimmt man an, dass die Knoten im allgemeinen mit 100 MBit/s an ein Ethernet angeschlossen sind und die Kommunikation in beide Richtungen gleichzeitig (Full-Duplex Betrieb) möglich ist, so müssten im günstigsten Fall ca. 10,27 MBit/s[5] auf N-Slices[6] verteilt und verarbeitet werden.
Zum anderen können es auch rechtliche Probleme sein, die es nicht zulassen, dass jeder Slice den Verkehr von allen anderen Slices auf dem Knoten mitlauschen darf.

3.3.2 PlanetLab RAW Sockets

Die PlanetLab RAW Sockets sind eine spezielle Variation der Socket-Schnittstelle. Es handelt sich dabei um einen Socket, der speziell auf PlanetLab angepasst und erweitert wurde. Der Socket wird mit speziellen Parametern über die bekannte Socket-Schnittstelle angesprochen. Er dient dem Mitlauschen von Verkehr, der über einen "echten" Socket transportiert wird. Dieser Socket ist nur lesbar. Aufgrund seiner Implementierung mit Hilfe der Socket-Schnittstelle, kann er nur für einen Port eingesetzt werden und bietet eine Erweiterung der Socket-Schnittstelle zum Mitlauschen des Verkehrs auf dem zugehörigen Socket. Es ist momentan nur möglich einen RAW-Socket pro Socket zu instantiieren.
Dies hat im Bezug auf die Flexibilität und Einsetzbarkeit Auswirkungen. Somit kann dieser Socket nur auf der Serverseite eingesetzt werden und damit verbundene Clients analysieren. Somit kann er nicht überall platziert werden. Außerdem muss vor dem Öffnen des Sockets bekannt sein, welcher Port analysiert werden soll, damit dieser geöffnet werden kann.
Dieser Socket-Typ lässt sich für "quasi"-Aktive Messungen einsetzen, in dem Sockets geöffnet werden und an diese der PlanetLab-RAW-Socket gebunden wird. Über die geöffneten Sockets wird dann Verkehr gesendet, der vom RAW-Socket gleichzeitig mitgelauscht werden kann. Statt den RAW-Socket an einen Socket zu binden, über den nur zufällig erzeugter Verkehr gesendet wird, kann

[5]Dieses Ergebnis errechnet sich unter der Annahme, dass 100 Byte pro Ethernetpaket kopiert werden und alle Pakete auf dem Ethernet 1500 Byte groß sind. Geht man von einer kleineren Paketgröße pro Ethernetpaket aus (z.B. 750 Byte), dann erhöht sich die Rate auf 19,56 MBit/s. Bei der Rechnung sind ebenfalls keine Kollisionen und Paketzwischenräume berücksichtigt. Durch Berücksichtigung dieser Tatsachen verringert sich das Ergebnis. Mehr Informationen zur Berechnung des Messdatenaufkommens finden sich in Kapitel 3.4

[6]N ist die Anzahl der aktuell laufenden Slices auf einem Knoten

der PlanetLab-RAW-Socket auch an existierende Sockets von Applikationen im eigenen PlanetLab-Slice (siehe auch Kapitel 2.4) oder anderen Slices gebunden werden. Ebenso kann der RAW-Socket an die Sockets gebunden werden, die vom verteilten Mess- und Analysesystem selbst genutzt werden.
Ein weiterer Socket-Typ, der auf PlanetLab angepasst wurde, ist der sog. Admin-Socket. Dieser Socket ist ebenfalls eine spezielle Variante eines RAW-Sockets. Er kann den gesamten Verkehr mitlauschen, der vom PlanetLab-Knoten empfangen wird. Jedoch ist es momentan nicht möglich die Daten zu erfassen, die vom PlanetLab-Knoten gesendet werden. Somit können mit diesem Sockettyp entscheidende Pakete, die zur Analyse von TCP-Verkehr oder anderen Protokollen benötigt werden, nicht mitgelauscht werden und dadurch ist der Admin-Socket zur Analyse von Verkehrsströmen nicht geeignet.

3.3.3 PlanetLab-Sensors

PlanetLab-Sensoren ([RoPe03]) können sehr verschiedenartige Daten über ein gemeinsames Interface bereitstellen. Sie bieten ein Interface, mit dem in Verteilten Systemen Informationen über lokale oder entfernte Daten sowie Ereignisse und Eigenschaften abgerufen werden können. Der Zugriff auf die Sensoren erfolgt durch eine Teilimplementierung des HTTP-Protokolls[7]. Die Sensor-Server unterstützen zwei Modi, wie Daten angeboten werden. Es handelt sich dabei um *Snapshot-Sensoren* und *Streaming-Sensoren*. Bei den Snapshot-Sensoren wird auf einen Request hin eine Antwort erzeugt, die eine definierte Länge hat. Ein Streaming-Sensor erzeugt bei einem Request eine Vielzahl aufeinanderfolgender Nachrichten. Diese werden asynchron übermittelt, bis einer der Kommunikationspartner die Verbindung beendet. Die Daten werden bei den Sensoren in Tupel übertragen. Handelt es sich um strukturierte Daten, kann XML oder eine andere Art der Kodierung erfolgen.
Auf den PlanetLab-Knoten existieren einige Sensoren, die Informationen über das Betriebssystem sowie über Netzwerkinformationen, wie Pfade zur entfernten Knoten, etc, verfügen. Beispiele für diese Sensoren finden sich ebenfalls in [RoPe03].

3.3.4 FlexiNet

Die Gewinnung von Messdaten mit Hilfe von FlexiNet (www.flexinet.de) stellt eine weitere Methode dar. Der Ansatz von FlexiNet beruht auf der Nutzung und Erweiterung von *Netfilter*[8] unter Linux. Dabei werden Pakete, aufgrund von Filterregeln, in *Netfilter* mitgelauscht und an sog. FlexiNet-Module weitergereicht. Die Module laufen im Benutzer-Modus des Rechners und bedürfen

[7] Es wird nur ein Teil der in HTTP spezifizierten Befehle genutzt. Dies sind die beiden Request-Methoden *GET* und *HEAD*. Siehe dazu auch [RoPe03].

[8] www.netfilter.org; Netfilter/IPTables ist ein Framework unter Linux, mit dem Paketfilterung sowie Netzwerk Adress Übersetzung (NAT) ermöglicht wird. Netfilter erlaubt das Setzen von Callback-Funktionen an verschiedenen Punkten im Netzwerk-Stack, die immer dann aufgerufen werden, wenn ein Paket dort verarbeitet werden soll. Dadurch wird es zu einem flexiblen Framework. Für Netfilter existieren verschiedene Plugins um Pakete zu filtern und zu bearbeiten.

keiner privilegierten Rechte innerhalb des Kernels. In den einzelnen Modulen kann dann eine beliebige Bearbeitung und Verarbeitung der Pakete erfolgen. Dies können z.B. Paketduplikaton oder andere Paketeveränderungen sein. Diese Techniken werden zum Beispiel in Stream-Reflektoren eingesetzt. Eine weitere Verwendungsmöglichkeit der FlexiNet-Module besteht darin, dass die gefilterten Pakete nicht bearbeitet sondern nur analysiert werden. Dazu werden Pakete mit FlexiNet gefiltert und einem Modul zur Messung von Verbindungseigenschaften zugeführt, welches passiv verschiedene Parameter auswertet und analysiert.[9] Jede Verarbeitung eines Paketes in einem Modul hat direkten Einfluss auf den Paketstrom, der durch FlexiNet gefiltert wird. Das gerade betrachtete Paket wird erst weitergereicht, wenn die Verarbeitung innerhalb des Moduls abgeschlossen ist. Weitere Informationen zu FlexiNet und dessen Architektur finden sich in [HSSW+02].

3.3.5 Aktive Quellen

Zu den aktiven Datenquellen zählen Tools und Methoden, die aktiv Messergebnisse sammeln, indem Pakete erzeugt und versendet werden. Daraufhin erzeugt das Zielsystem in der Regel eine Antwort. Diese Antwort wird , nach Erhalt, im Quellsystem ausgewertet. Die aktiven Quellen können in zwei Kategorien unterteilt werden. Zum einen sind das die Probing-Tools *ping* und *traceroute. Ping* versendet einzelne ICMP-Datagramme mit der aktuellen Zeit und wartet auf die Antwort von der Gegenseite.[10] Daraufhin wird die Round-Trip-Time (RTT) berechnet. Hiermit wird aktiv die RTT zum Zielsystem gemessen. Das *traceroute* Programm liefert eine Liste von Zwischensystemen bis zum Zielsystem. Dazu versendet es IP-Pakete mit steigendem IP-TTL Wert, beginnend bei dem Wert *Null*. Implizit erhält der Quellrechner eine Antwort von einem Zwischensystem, wenn ein Paket verworfen werden musste. Ein IP-Paket wird immer dann verworfen, wenn der aktuelle Wert im IP-TTL Feld auf Null steht und der empfangende Rechner nicht das Zielsystem ist. In jedem Zwischensystem wird der TTL-Wert um mindestens eins reduziert. Damit erhält der Quellrechner automatisch die Adresse des Zwischensystems, in dem das Paketverworfen wurde. Durch fortlaufende Erhöhung des TTL-Startwertes kann somit der Pfad von der Quelle bis zum Ziel bestimmt werden. Der TTL-Startwert wird so lang erhöht, bis das Endsystem erreicht wird. Somit erhält man die Liste der Zwischensysteme. Diese Liste kann nun ausgewertet werden und es können evtl. Veränderungen des Pfades zum Ziel beobachtet werden. Zum anderen sind dies aktive Protokoll-Probing-Tools, die eine Teilmenge der jeweiligen Protokollfunktionen implementieren und versuchen Erkenntnisse über die Gegenseite herauszufinden. In [padh01] wird ein Programm beschrieben, welches verschiedene Features und Informationen über TCP bei anderen Systemen durch das beschriebene Verfahren ermittelt. Es werden z.B. die unterstützen Features wie Selective Acknowledgements (SACK) oder Early Congestion Notification (ECN) sowie die unterschiedlichen Staukontrollmechanismen untersucht.

[9] Diese Parameter werden in [Zülc04] beschrieben.
[10] Die Gegenseite kopiert den Inhalt in ein neues Paket und sendet das Paket zurück.

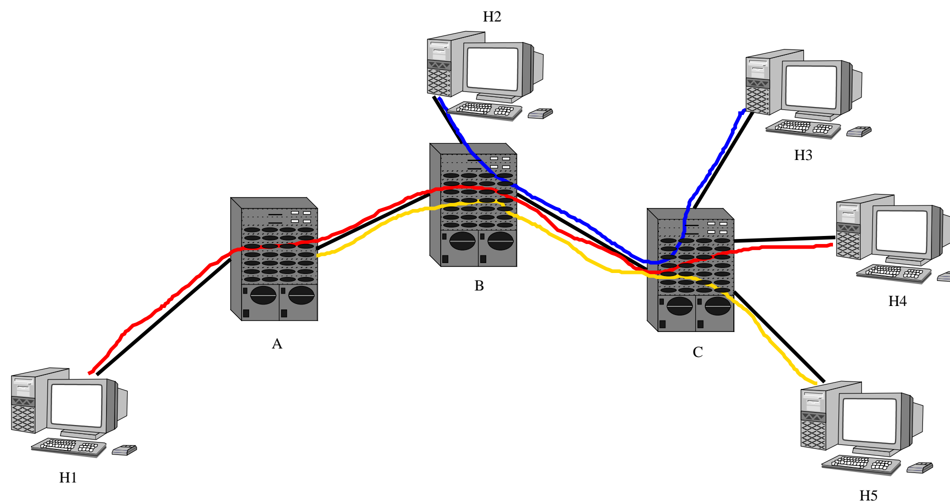

Abbildung 3.1: Dieses Szenario zeigt eine Überlastung zwischen den Zwischensystemen B und C. Alle Schnittstellen zwischen den einzelnen Systemen haben die gleiche Rate (z.B. 100 MBit/s). Die Verbindung (blau) zwischen H2 und H3 nutzt 25% der Bandbreite. Die Verbindung (rot) zwischen H1 und H4 nutzt 70% Bandbreite. Pakete dieser Verbindung werden analysiert. Die Verbindung (gelb) zwischen den Systemen A und H5 benötigt >5% der Bandbreite und enthält die Messdaten der zweiten Verbindung, die analysiert werden soll.

3.3.6 Andere Quellen

Neben den bisher geschilderten Quellen für Messdaten existieren noch weitere Quellen. Zu diesen Quellen zählt die Messwerterfassung durch Abfrage von Rechnersystemen mit Hilfe eines Management-Protokolls[11]. Art und Umfang der Parameter, die abgerufen werden können, hängen von dem jeweiligen System ab und werden von jedem System spezifiziert.
Eine weitere Quelle für Messdaten können auch direkt Routing-Informationen, die ein Router bereitstellt, sein. Dazu müssen das Routing-Protokoll interpretiert und Informationen daraus gewonnen werden. Diese Informationen können zur weiteren Analyse genutzt werden.

3.4 Hochbitratige Messströme

Im vorherigen Abschnitt wurden verschiedene Quellen geschildert, die Pakete und andere Messdaten erfassen können. Diese Quellen liefern ein breites Spektrum an Messdaten. Diese verschiedenen Messdaten, haben einen unterschiedlich großen Umfang in ihrer Menge. Die Menge der Daten hängt von der Art der Messdaten und der Häufigkeit, wie oft diese Daten erfasst werden,

[11] In IP-basierten Netzwerken wird das Simple-Network-Management-Protokoll (SNMP) eingesetzt. SNMP wird erstmals in [JFSJ90] erwähnt und in [JMRW93], [WiDP99] und [WiDP02] weiterentwickelt. Das Telecommunikation Management Network (TMN) wird hauptsächlich innerhalb der öffentlichen Telekommunikation (Telefonnetz) eingesetzt. Es basiert auf der ITU Empfehlung M.30

ab. Vorrangig beschreiben die Quellen aus dem vorhergehenden Kapitel, Messquellen, die Paketdaten erfassen. Heute werden die Netzwerkschnittstellen von Netzwerkkomponenten und Rechnern mit Netzwerkkarten immer schneller. Die Schnittstellenraten einer Netzwerkkarte eines Personal Computer (PC)) betrugen vor wenigen Jahren noch 10 MBit/s[12]. Heute sind in PCs Schnittstellen mit 100 MBit/s bis 1 GBit/s üblich. In den Netzwerken von Telekommunikationsanbietern sind die Schnittstellenraten nochmals um den Faktor 10 bis 40 größer. Dies bringt hinsichtlich der Erfassung von Paketdaten Probleme mit sich. Zum einen müssen Einrichtungen vorhanden sein, die die erfassten Pakete, die auf einer schnellen Schnittstelle erfasst wurden, auch in dieser Geschwindigkeit speichern können. Ist dies nicht möglich, so kann es dazu kommen, dass Pakete verworfen werden müssen, da sie nicht schnell genug gespeichert werden können. Zum anderen muss genügend Rechenleistung vorhanden sein, damit die Pakete analysiert werden können. Können die Pakete nicht schnell genug verarbeitet werden, kann es ebenfalls dazu kommen, dass Pakete verworfen werden müssen, da der Speicher, in dem die Pakete gespeichert werden, überläuft. Ein dritter Aspekt kommt hinzu, wenn man beachtet, dass Messung und Analyse nicht immer auf der gleichen Maschine stattfinden. Werden Messungen und Analysen getrennt auf verschiedenen Maschinen durchgeführt, ist es erforderlich, dass die Paketdaten zwischen den einzelnen Maschinen übertragen werden. Sind die Kommunikationswege dieser zusätzlichen Übertragung der Pakete von den Kommunikationswegen getrennt, auf denen die Pakete erfasst werden, stellt dies kein Problem dar. Müssen diese Daten jedoch über die gleiche Schnittstellenkarte transportiert werden, wie der eigentliche Messverkehr auch, so kann dies Probleme mit sich bringen. Diese Probleme sind darin zu sehen, dass durch den zusätzlichen Verkehr, der durch die Übertragung des Messdaten erzeugt wird, der eigentliche Messverkehr beeinflusst wird. Dadurch können die Ergebnisse verfälscht werden. Dies wird dadurch verursacht, dass durch diesen zusätzlichen Verkehr eine physikalische Verbindung überlastet wird und Pakete verworfen werden müssen. Ein solches Szenario ist in Abbildung 3.1 dargestellt.

In dieser Abbbildung wird die Verbindung zwischen den beiden Endsystemen H1 und H4 untersucht und deren Pakete in dem Zwischensystem A erfasst. Hinzu kommt, dass noch eine weitere Verbindung zwischen den Endsystemen H2 und H3 existiert, die als Querverkehr zwischen den Zwischensystemen B und C gewertet werden können. Beide zusammen benötigen auf der physikalischen Verbindung zwischen den beiden Zwischensystemen B und C ca. 95% der Bandbreite. Jetzt kommt aber noch der Verkehr der Verbindung zwischen dem Zwischensystem A und dem Endsystem H5 hinzu. Diese Verbindung beinhaltet die Messdaten, die das Zwischensystem A erfasst. Da dieses System jedoch nicht über die Rechenkapazität verfügt, um die Daten zu analysieren, schickt dieses die Daten an das Endsystem H5 weiter. Diese Verbindung benötigt nun aber mehr als die verbleibenden 5% der Bandbreite. Hierdurch kommt es zu einer Überlastsituation zwischen den Zwischensystemen B und C. Folglich werden Pakete verworfen. Zusätzlich hat diese Überlast eine Auswirkung auf die

[12]Megabit $\equiv 1024^2 Bit/s$

drei Verbindungen, wodurch sich diese an die neue Situation anpassen. Folglich hat der zusätzliche Messverkehr eine Auswirkung auf die eigentliche Messung selbst, wodurch eine mögliche Verfälschung der Messergebnisse erfolgt. Jedoch kann diese Situation aus zwei Perspektiven betrachtet werden. Zum einen kann diese Situation wie beschrieben als Fehler betrachtet werden, wodurch Messergebnisse künstlich verfälscht werden. Zum anderen kann dies auch so betrachtet werden, dass der Verkehr, der durch die Messdaten verursacht wird, als unabhängige Verbindung gesehen wird, die genauso gut wie Querverkehr behandelt werden kann. Dann ist der Verkehr durch die Messung kein Fehler und bringt nur zum Vorschein, dass die Verbindung zwischen den beiden Zwischensystemen B und C auch schon ohne den Verkehr fast vollständig ausgelastet ist. Dies macht die Betrachtung der Situation schwierig und eine Beurteilung, ob es sich um eine Verfälschung der Messergebnisse handelt, hängt von der Instanz ab, die die Ergebnisse interpretiert. Hat diese Instanz keine Sicht auf die Vorkommnisse vor dem Start der Messung, ist nicht erkennbar, durch wen diese Überlast erzeugt wird. Besteht jedoch eine Sicht auf die Gegebenheiten vor der Messung, so wird schnell klar, dass die Überlast durch die Messdaten verursacht wird. Jedoch dürfen sich die bestehenden Verbindungen nicht verändert haben.

Im vorherigen Absatz ist nur die Rede von einer prozentualen Auslastung einer Verbindung. Hier soll nun eine exemplarische Rechnung erfolgen, die einen Überblick darüber gibt,wie viele Daten eine Erfassung von Paketen ergibt und wie groß der Messdatenstrom zwischen einem Erfassungsknoten und einem Analyseknoten ist. Dies soll exemplarisch an einem 100 MBit/s Ethernet erfolgen, wie er oft in Netzen zu finden ist. Zuerst wird in Tabelle 3.1 die Größe der Paketköpfe einzelner Protokolle dargestellt. Wenn Ethernet-Paketköpfe übertragen werden, dann sind von diesen Paketköpfen nur 14 Oktett[13] relevant. Dies sind die Quell- und Zieladresse sowie der Protokolltyp, d. h. in die Berechnung fließen nur 14 Oktett mit ein. Bei IP wird von einer Paketkopfgröße von 20 Oktett ausgegangen und bei TCP von 40 Oktett.
Geht man nun von einem Ethernet mit 100 MBit/s aus und dieses ist voll ausgelastet, d. h. jeder Rahmen ist mit 1500 Oktett gefüllt, folgt daraus,

$$\begin{aligned} 100\frac{MBit}{s} &= 100 * 1024\frac{KBit}{s} \\ \frac{102400}{11.86}\frac{KBit * Rahmen}{KBit * s} &= 8634\frac{Rahmen}{s} \end{aligned} \tag{3.1}$$

dass 8634 Rahmen pro Sekunde übertragen werden können. In jedem Rahmen sind 1500 Oktett Daten enthalten. Diese Daten setzen sich aus Protokollköpfen und den eigentlichen Daten zusammen. Somit enthält ein exemplarischer Rahmen den Ethernet-Kopf (14 Oktett), einen IP-Kopf (20 Oktett) und einen

[13] 1 Byte = 1 Oktett

Tabelle 3.1: Paketheadergrößen von Protokollen

Protokoll	Kopfgröße		Datengröße		Bemerkung
	min	max	min	max	
Ethernet	26	26	46	1500	sind die Daten kürzer als 46 Oktett werden diese auf 46 Oktett aufgefüllt.
PPP	8	8	0	1500	kann noch weitere Sub-Header enthalten. Dadurch wird die Datenlänge kürzer.
ARP	24	24	0	0	
IPv4	20	40	0	65535	bei Ethernet typische Datenlänge von 1480 Oktett, kann jedoch geringer sein, wenn z.B: PPPoE genutzt wird
TCP	20	40	0	65535	Datengröße richtet sich nach maximaler MTU des Links
UDP	8	8	0	65535	Datengröße richtet sich nach maximaler MTU des Links
ICMP	8	8+X	0	Y	je nach ICMP-Typ

Alle Angaben wurden [Hein99] entnommen. Die Werte für die einzelnen Größen sind in Oktett (1 Oktett = 8 Bit) angegeben.

TCP-Kopf (40 Oktett), der Rest sind Daten. Folglich werden 74 Oktett Messdaten pro Paket (Rahmen) erfasst. Daraus folgt,

$$\begin{aligned} 8634\frac{Rahmen}{s} * 74 Oktett &= 638916\frac{Oktett * Rahmen}{s} \\ 638916\frac{Oktett}{s} &= 623.94\frac{KOktett}{s} \end{aligned} \tag{3.2}$$

dass 623.94 KByte/s an reinen Messdaten übertragen werden. Ist die untersuchte Verbindung bidirektional, so verdoppeln sich die Messdaten auf 1247.88 KByte/s. Diese Daten müssen nun noch zwischen dem Messknoten und dem Analyseknoten übertragen werden, daraus folgt, dass diese 1247.88 KByte/s noch in Pakete eingekapselt werden müssen. Legt man fest, dass diese Pakete immer vollständig gefüllt sein müssen, folgt daraus, dass bei einer genutzten TCP-Verbindung pro zu sendenden Rahmen 1440 Byte genutzt werden können. Daraus folgt, es müssen

$$\frac{1247.88}{1.40625}\frac{KByte * Rahmen}{KByte * s} = 887\frac{Rahmen}{s} \tag{3.3}$$

Rahmen zusätzlich zu dem Verkehr, der untersucht wird, übertragen werden. Das bedeutet ca. 10% mehr Pakete müssen übertragen werden. Geht man nun aber davon aus, dass die Rahmen nur zu 50% gefüllt sind, also 750 Oktett

Daten pro Rahmen übertragen werden, d. h. es sind in einem Rahmen 6.06 KBit enthalten. Bei einer Übertragungsrate von 100 MBit/s hat dies zur Folge, dass knapp doppelt so viele Pakete (1689 Rahmen/s) übertragen werden können. Folgt man nun den weiteren Berechnungen bedeutet dies, dass doppelt so viele Messdaten erfasst werden und zwischen Erfassungsknoten und Analyseknoten übertragen werden müssen. Bei dieser Berechnung wird jeweils davon ausgegangen, dass keine Kollisionen auf dem Medium stattfinden und in beide Richtungen gleichzeitig gesendet werden kann. Die Ergebnisse für verschiedene Übertragungsraten und verschiedene Datengrößen werden in Tabelle 3.2 dargestellt. Diese gibt einen Überblick über die zu erwartenden Datenmen-

Tabelle 3.2: Messdatenraten in verschiedenen Szenarien

Daten pro Rahmen Übertragungsrate	750 Byte	1500 Byte
10 MBit/s	1.95	1.02
100 MBit/s	19.56	10.27
1000 MBit/s	195.60	102.70

Die Angaben innerhalb der Matrix sind alle in MBit/s.

gen. Jedoch muss auch davon ausgegangen werden, dass nicht immer die volle Paketrate auf dem Medium anliegt. Die Werte für reduzierte Paketraten entsprechen einer verminderten Übertragungsrate.

Um nun diese Daten effektiv zu verarbeiten, ist es notwendig Optimierungen zu implementieren. Diese Optimierungen betreffen die Behandlung der erfassten Pakete und deren Verarbeitung innerhalb eines Rechners. Unter einer Optimierung wird hierbei verstanden, dass die gesammelten Paketdaten so wenig wie möglich innerhalb des Hauptspeichers kopiert werden sollten, da jede Kopieraktion Zeit kostet. Daraus folgt, dass diese Daten in einem gemeinsamen Speicher gehalten werden sollten, auf den unterschiedliche Teile der Erfassung und Analyse Zugriff haben. Ein Paket sollte möglichst nur zweimal auf einem Rechner kopiert werden. Zum ersten Mal, wenn es von der Netzwerkkarte in den Hauptspeicher kopiert wird. Zum zweiten Mal, wenn es wieder aus dem Hauptspeicher auf die Netzwerkkarte kopiert werden muss. Zwischendurch sollte das Kopieren vermieden werden.

3.5 Datenkompression

Die Datenkompression dient als Mittel um die Größe des Messdatenstroms zu reduzieren. Kompression beruht auf der Annahme, dass sich innerhalb eines Datenstroms viel Redundanz befindet, die durch geschickte Kodierung minimiert werden kann. Diese Minimierung hat zur Folge, dass weniger Daten übertragen werden müssen und die Übertragung effizienter gestaltet werden kann. Im Rahmen eines verteilten Systems soll die Datenkompression dazu eingesetzt

werden den Messdatenstrom zu verringern. Die Motivation für die Verringerung des Datenstroms liegt darin, dass es durch zusätzliche Datenströme bei Verkehrsmessungen zu Veränderungen des Messstroms kommen kann und somit die Ergebnisse verfälscht werden können. Dies tritt z.B. auf, wenn der Bottleneck einer Verbindung genau bei dem Zwischensystem oder Endsystem liegt, das den Messdatenstrom speichert und die Messdaten an einen anderen Knoten im verteilten System weitergeleitet werden sollen. Der Datenstrom, den es zu kompremieren gilt, besteht in diesem Fall aus Paket-Header Informationen und daraus abgeleiteten Messwerten. Unter Beachtung dieser Vorgaben bieten sich zwei verschiedene Möglichkeiten an, wie die Daten kompremiert werden können. Die erste ist eine allgemeine Variante zur Kompression von Daten. Die andere eine auf Paket-Header spezialisierte Kompressionsvariante.
Die erste Variante Daten zu kompremieren ist die Kompression durch Huffman-Kodierung [Trev03] oder LZW-Kompremierung (Lempel-Ziv-Welch) [Paul04]. Huffman-Kodierung nutzt zur Kompression die effektive Darstellung einzelner Zeichen. Dazu werden die absoluten Häufigkeiten der einzelnen Zeichen benötigt. Je häufiger ein Zeichen vorkommt, desto kürzer ist seine Bit-Darstellung im Huffman-Code. Es existieren zwei Modi zur Huffman-Kodierung *fix-length*-Code und *variable-length*-Code. Die Kompressionsrate ist abhängig von den Eingangsdaten. Die zweite Möglichkeit zur Kompression des Datenstroms berücksichtigt den speziellen Aufbau der Daten, d.h. Paket-Header. In [DeNP99, eal.01, Dege01] und [Jons04] sowie in [CeWF, DENP97] werden verschiedene Varianten zur Kompression von Paket-Headern geschildert. Sie machen sich die Tatsache zu Nutze, dass viele Informationen in den Paket-Headern in jedem Paket übertragen werden, sich aber nicht ändern. Diese Verfahren reduzieren die Headergrößen dadurch, dass komplette Paket-Header nur alle n-Pakete übertragen werden und dazwischen werden nur die Differenz-Informationen übermittelt. Diese Verfahren reduzieren die Daten von ca. 60 Byte auf ca. 5 Byte pro Paket. Diese Verfahren sind für die Kompression von Paketheadern bei Punkt-zu-Punkt Verbindungen ohne IP-Zwischensysteme in drahtgebundenen sowie drahtlosen Netzen entworfen worden. In den drahtlosen Netzen werden die Auswirkungen der Kompression auf eine veränderte Paketverlustwahrscheinlichkeit hin untersucht. Diese Verfahren sind schlecht für die Verwendung von IP-Paketen geeignet, die nicht in Punkt-zu-Punkt Verbindungen transportiert werden. Es wird in den Endsystemen ein Zustand aufgebaut, der auch in Zwischensystemen aufgebaut werden muss. Dies ist jedoch für große Router im Internet nicht möglich.
Die Open-Source Bibliothek libz implementiert das sogenannte *deflate* Kompressionsverfahren. Es nutzt zur Kompression der Daten die beiden Verfahren Huffman-Kodierung und LZ77 [LeZi97]. Beide Verfahren werden kombiniert, um eine höhere Kompressionsrate zu erzielen. Das Format von *deflate* wird in [Deut96] definiert. Die Kompression wird mit einem Parameter gesteuert. Dieser Parameter gibt die Kompressionsqualität an. Der Wertebereich liegt zwischen 0 und 9. Der Wert 0 gibt die schnellste Kompression und 9 die langsamste Kompression an. Bei der schnellsten Kompression wird am wenigsten CPU-Zeit verbraucht und die Datei wird nicht so stark kompremiert. Bei

der langsamsten Kompressionseinstellung benötigt der Algorithmus die meiste CPU-Zeit und die Kompressionsrate ist höher. Somit kann mit dem Parameter ein Kompromiss zwischen Kompressionsrate und verbrauchter CPU-Zeit eingestellt werden.
Fazit: Es ist im Rahmen dieses Frameworks unerheblich, welche der beiden Varianten eingesetzt wird. Es sollte jedoch die Variante gewählt werden, die einfacher zu implementieren ist und im praktischen Einsatz bessere Kompressionsraten erzielt, ohne dass der Overhead durch die Berechnung der Kompression zu groß wird. Die Kompression der Daten kann auch als Option behandelt werden, indem zu Beginn der Kommunikation zwischen Paketerfassung und Paketanalyse das Kompressionsverfahren ausgehandelt wird.

3.6 Aggregation und Analyse von Messungen

In diesem Kapitel geht es um die Fragestellung, wie Messdaten aggregiert und daraus weitere Ergebnisse abgeleitet werden können. Dabei ist ein Kompromiss bei dem Grad der Aggregation, d.h. der Granularität der Messdaten, zu erzielen. Es müssen aber noch weitere Analysen durchführbar sein, ohne zu großen Informationsverlust zu haben. Hierbei greifen die beiden Aspekte Aggregation und Analyse ineinander. Denn durch Analyse von mehreren Eingangsdaten und der Erzeugung eines Ausgangsdatums findet ebenfalls eine Aggregation der Messdaten statt. Werden die Messdaten nach der Analyse nicht gelöscht, so entstehen neue Daten und es findet keine Aggregation der Daten statt, sondern nur eine Analyse. Werden die Daten gelöscht, so handelt es sich um eine Aggregation, da nur noch aus einer Menge von Messdaten bleiben wenige oder sogar nur ein Datum erhalten. Es gestaltet sich schwierig zwischen der Zusammenfassung von Messdaten und der Analyse dieser zu unterscheiden. Mit der Aggregation der Messdaten ist hierbei nicht die Kompression der Daten wie in Kapitel 3.5 gemeint. Vielmehr versteht man unter Aggregation die Zusammenfassung von Daten durch Extraktion von Gemeinsamkeiten und der Verallgemeinerung von Informationen und Eigenschaften. Ebenso ist die Erzeugung von Regeln gemeint, denen die Messdaten folgen. Ziel ist es somit eine kompakte und gleichzeitig hinreichend detailliert beschreibende Darstellung der Daten zu finden. Dabei ist auf die Erhaltung des Informationsgehaltes zu achten.
Unter Beachtung dieser Anforderungen an die Aggregation werden in den folgenden Abschnitten mehrere Methoden beschrieben, mit deren Hilfe die Aggregation und Analyse der Messdaten erfolgen kann. Vorher ist zu klären, welche Parameter gemessen werden können und welche Aussagen über diese gemessenen Parameter sowie deren Veränderungen über die Zeit gemacht werden können.

3.6.1 Ergebnisanalyse

Die Art und der Umfang der Parameter, die gemessen und verarbeitet werden können, hängen stark von der Quelle der Messdaten ab. Im Kapitel 3.3 werden verschiedene Quellen beschrieben und im Hinblick auf ihre Verwendbarkeit

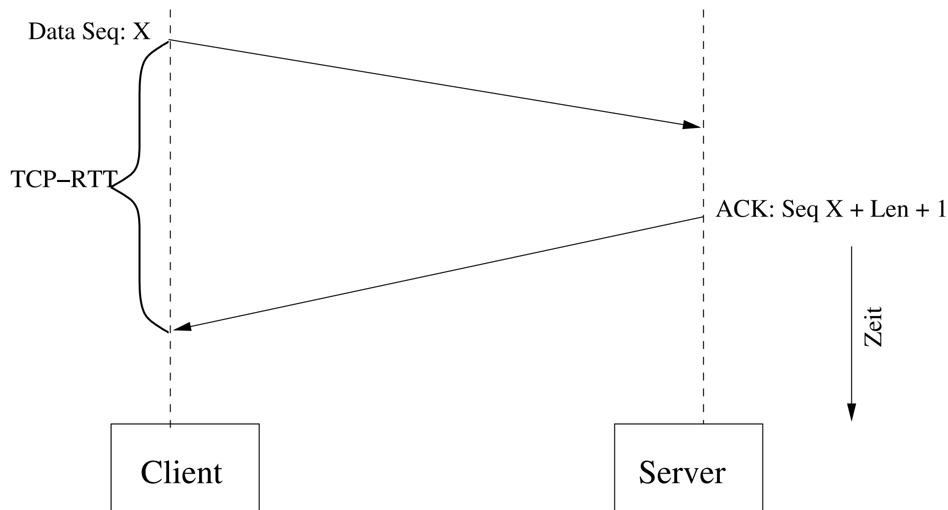

Abbildung 3.2: Round-Trip-Time am Beispiel einer TCP-RTT

untersucht. Des Weiteren wird in [Zülc04], Kapitel 3.5 beschrieben, welche Parameter bei passiven Messungen von Paketströmen ermittelt und analysiert werden können. Ebenso werden Hinweise und Möglichkeiten zur Analyse von Messparametern in Abschnitt 3.2 dieser Arbeit gegeben.
Nun soll aus den zur Verfügung stehenden Parametern extrahiert werden, welche Aussagen sich über den Zustand des Netzes, einzelner Teilbereiche eines Netzes oder einzelner Rechner treffen lassen.
Die Parameter lassen sich in verschiedene Klassen einteilen. Dies sind zum einen die Parameter, die direkt zur weiteren Analyse genutzt werden. Dazu gehört der IP-TTL Parameter. Zum anderen sind dies Parameter, die erst nach einer ersten Analysephase genutzt werden können. Hierzu zählt die Erfassung der Round-Trip-Time (RTT), die erst aus einer Folge von Paketen ermittelt werden kann. Dabei wird die Zeit gemessen, um ein Antwortpaket auf ein gesendetes Paket zu erhalten. Verdeutlicht wird die Definition der RTT am Beispiel einer TCP-RTT in Abbildung 3.2. Wie aus der Abbildung ersichtlich wird, setzt sich die RTT aus 3 Komponenten zusammen. Dies sind die Paketlaufzeit von der Quelle zum Ziel, die Verarbeitungszeit im Zielsystem und die Paketlaufzeit vom Ziel zur Quelle. Jede dieser drei Komponenten ist unabhängig von den anderen. So kann die Paketlaufzeit zwischen Quelle und Ziel unterschiedlich sein gegenüber der Paketlaufzeit zwischen Ziel und Quelle. Die Paketlaufzeit eines Paketes setzt sich aus der Paketisierungszeit, Übertragungszeit und der Wartezeit in Zwischensystemen zusammen.[14] Die RTT kann auf verschiedenen Schichten im OSI-Referenzmodell (z.B. in [Tane00] Kapitel 1.4) berechnet werden. Auf den OSI-Schichten: Sicherungs-, Vermittlungs-, Transport- und Anwendungsschicht ist die Berechnung von RTT möglich. Um die RTT berechnen zu können, muss es sich um ein *Request-Response* Protokoll handeln.

[14] Siehe dazu auch Vorlesung Experimentelle und Analytische Untersuchung von Kommunikationsprotokollen WS 2003/04, Kapitel 1, Folie 23 und 24; http://www.tm.uka.de/˜ fuhrmann/Protokollanalyse/Protokollanalyse-WS0304-1.pdf

Das Antwortpaket zu einem Request muss nicht vom gleichen Protokoll gesendet werden. Es kann ein anderes Protokoll sein, es muss jedoch festgestellt werden können, dass es sich bei dem Paket um eine Antwort/Reaktion auf ein vorhergehendes Paket handelt.[15] Dies erfordert, dass auf den jeweiligen Schichten die Antwort auf ein Paket identifiziert werden muss und somit die RTT berechnet werden kann. Beispiele sind HDLC mit Sliding-Window (Schicht 2) und TCP (Schicht 4).
Weitere Parameter die zur Messung und Aggregation herangezogen werden können, werden in [PAMM98] definiert. Neben der Round-Trip-Time und der Time-to-Live (IP-TTL) sind hier folgende Parameter definiert:

- *propagation delay of a link*
- *bandwidth of a link*
- *path at time t*

In RFC 2679, RFC 2680, RFC 2681, RFC 3148, RFC 3357, RFC 3393, RFC 3432 und RFC 3763 werden folgende Parameter definiert:

- *one-way delay*
- *one-way packet loss*
- *IP packet delay variation*
- *bulk transport capacy*
- *one-way loss pattern sample metrics*
- *...*

3.6.1.1 Round-Trip-Time

Begonnen soll hier mit der Untersuchung von Aussagen, die über die RTT getroffen werden können, d.h. welche Ereignisse sich aus diesem Parameter herausfiltern lassen. Nach der Definition der RTT trifft diese eine Aussage darüber, wie viel Zeit benötigt wird, bis eine Antwort auf ein versendetes Pakete empfangen wird. Betrachtet man diesen Wert über die Zeit hinweg, so lässt sich eine Aussage über die Auslastung des Pfades zwischen Sender und der Maschine, die die Paket mitlauscht, machen. Es wird nun versucht aufgrund des Verhaltens der RTT über die Zeit eine Analyse durchzuführen. Zunächst erwartet man, dass sich die RTT über die Zeit hinweg nicht verändert. Jedoch ist dies eine Fehlannahme. Bei TCP können schon aufgrund des Designs unterschiedliche RTTs gemessen werden. Diese lassen sich durch die Verwendung von sog. *Delayed*-Acknowledgements erklären, da TCP in der Lage ist,

[15] Beispiel hierfür ist das Senden eines Paketes an einen UDP und eine Antwort darauf mit einem ICMP Paket.

entweder auf jedes empfangene Paket eine Bestätigung zu verschicken, oder eine Bestätigung kommulativ für mehrere Pakete zu versenden, dann wird aber die RTT für das weitere empfangene Paket berechnet, da das Paket mit der höchsten Sequenznummer bestätigt wird. Wird kein weiteres Paket während der Wartezeit von ca. 500ms[16] empfangen, so wird die Antwort auf ein Paket erst nach dieser Zeit zurückgesendet. Dies führt zu einer höheren RTT, als tatsächlich gebraucht wird. Bei anderen Protokollen existieren keine kommulativen Bestätigungen, so dass immer das letzte korrekt empfangene Paket bestätigt wird. RTT Schwankungen treten auch auf dem Pfad zwischen den beiden Endsystemen auf. Diese sind bedingt durch unterschiedliche Auslastungen von Zwischensystemen (Routern), d.h. die Pakete erfahren unterschiedliche Wartezeiten in Warteschlangen in den Zwischensystemen. Dies ist z.B. bei TCP der Fall. Bei anderen Protokollen, wie z.B. auf OSI-Schicht 2, treten diese Schwankungen meist nicht auf. Diese Schwankungen treten auf Schicht 2 auf, wenn noch Zwischensysteme wie Switches in das Netz integriert sind, die ebenfalls Puffer enthalten. Dies führt zu sog. Jitter. Ebenso kann Jitter entstehen, wenn ein Paket nicht sofort auf das Übertragungsmedium gegeben werden kann. Dies tritt immer dann auf, wenn sich mehrere Systeme ein gemeinsames Medium teilen müssen (z.B. Ethernet). Ein weiterer Grund, der gegen gleiche RTTs über die Zeit spricht, ist, dass es aufgrund schlechter oder ausgelasteter Verbindungen zwischen je zwei Systemen zu Paketverlust kommen kann und das Pakete erneut gesendet werden müssen. Für TCP existieren verschiedene Verfahren, die den Paketverlust minimieren können. Diese werden in [BrJa88], [MMFR96] und [AlPS99] erörtert. Eine hohe Auslastung von Verbindungen wird durch die erhöhte Schwankung (Jitter) der RTT ersichtlich. Weiterhin kommt es zu Veränderungen der RTT, wenn sich der Pfad zwischen zwei Endsystemen während einer Übertragung ändert, d.h. das Routing-Protokoll wählt einen anderen Pfad zum Ziel. Diese Pfadänderung kann, muss aber nicht, durch einen Sprung der RTT angezeigt werden. In Kombination tritt eine Lücke in den RTT-Werten auf, d.h. für eine gewisse Zeit können keine RTT-Werte ermittelt werden. Diese Zeit benötigt das Routing um eine Routenänderung zu propagieren und die Pakete über diese veränderte Route zu senden. Eine Folge dieser Änderung kann sein, dass Pakete nicht rechtzeitig vor Ablauf des *Retransmission-Timers* empfangen werden oder aufgrund voller Routerwarteschlangen gelöscht und erneut gesendet werden. Wie schon angedeutet kann ein Routenwechsel mit einem Sprung in der RTT einhergehen. Dieser ist nur zu sehen, wenn der neue Weg kürzer oder länger gegenüber dem Pfad vor dem Wechsel ist. Existiert ein Parallelpfad, der in etwa der Länge des ursprünglichen Pfades entspricht, so ist unter Umständen kein Sprung in der RTT zu sehen. Ebenso ist es schwierig eine Lücke in der Reihe der RTT-Werte zu interpretieren. Es kann sich dabei um einen Routenwechsel handeln. Ebenso kann diese Lücke bedeuten, dass keine Daten übertragen wurden und somit keine Werte für die RTT berechnet werden können.

Alle geschilderten möglichen Schwankungen der RTT sind auf die Verände-

[16] Oft wird nur ca. 200ms gewartet bis das Antwortpaket gesendet wird.

rungen der drei Komponenten der Paketlaufzeit, sowie der Änderungen der Verarbeitungszeit im Zielsystem zurückzuführen.

3.6.1.2 IP-TTL

Der nächste Parameter, der auf seine Aussagekraft hin untersucht werden soll, ist der Time-To-Live (TTL) Parameter des IP-Protokolls. Aus diesem Wert lässt sich ablesen, wie viele Hops (Zwischensysteme) das IP-Paket noch gültig ist, und wie viele Hops das Paket schon passiert hat. Heute wird das TTL-Feld in IPv4 als Hop-Zähler genutzt. Jedoch gibt es laut [Post81] die Lebenszeit eines IP-Paketes in Sekunden an und muss pro Zwischensystem um mindestens eins dekrementiert werden. In IPv6 ([DeHi98] wird das TTL-Feld als Hop-Zähler spezifiziert, der pro Zwischensystem decrementiert werden muss. Aufgrund des TTL-Feldes zum Zeitpunkt des Erfassens, lassen sich Rückschlüsse auf die Anzahl der Zwischensysteme schließen, die von dem Paket durchlaufen wurden. Jedoch kann dies nur durch eine relative Angabe geschehen, da verschiedene Betriebssysteme unterschiedliche Startwerte verwenden. Typische Startwerte für den TTL-Wert sind: 64, 128 und 255. Man wählt zur Berechnung der Anzahl der durchlaufenen Zwischensysteme den nächst höheren Startwert und subtrahiert den aktuellen Wert von dieser Zahl. Der erhaltene Wert stellt mit großer Wahrscheinlichkeit die Anzahl der Zwischensysteme dar, die das Paket von der Quelle bis zu der Maschine, die das Paket mitgelauscht hat, durchlaufen hat. Die Maschine, die den Paketstrom mitlauscht, muss nicht das Ziel des IP-Paketes sein, sondern kann nur ein Zwischensystem auf dem Weg zum Ziel sein. Dies kann auftreten, wenn die Quelle der Pakete z.B. ein FlexiNet-PC ist, der zusätzlich als Router fungiert und sich nicht am Rand des Netzes befindet. Dies wird in [Zülc04] beschrieben. Problematisch wird die Berechnung der TTL, wenn einige IP-Verbindungen über MPLS (Multi Protocol Label Switching [RoVC01]) laufen. MPLS kapselt ein IP-Paket innerhalb eines eigenen Headers. Das TTL-Feld des IP-Pakete wird in den MPLS-Header übernommen und **sollte** pro Zwischensystem decrementiert werden, so dass die MPLS-Zwischensysteme sich im TTL-Wert wiederspiegeln.[17] Macht ein MPLS-Zwischensystem dies nicht, macht diese Tatsache die Interpretation des TTL-Wertes schwieriger. Wird jedoch von jedem MPLS-Zwischensystem der TTL-Wert korrekt verringert und beim Austritt aus dem LSP (Label Switched Path) der TTL-Wert in das gekapselte IP-Paket kopiert, macht dies die Interpretation des TTL-Wertes einfacher.
Welche Schlüsse lassen sich nun aus der Veränderung des TTL-Wertes ziehen? Die Änderung dieses Wertes ist schwer zu interpretieren. Ändert sich der TTL-Wert um eine kleine Zahl (+- 3 Hops), so kann nicht daraus geschlossen werden, dass es sich um einen Intra-Domain-Routenwechsel handelt. Vielmehr muss zusätzlich darauf geachtet werden, inwiefern sich bei dieser Änderung

[17] In RFC 3031 ([RoVC01]) wird dies in Kapitel 3.23 deutlich: *... When a packet travels along an LSP, it SHOULD emerge with the same TTL value that it would have had if it had traversed the same sequence of routers without having been label switched. If the packet travels along a hierarchy of LSPs, the total number of LSR- hops traversed SHOULD be reflected in its TTL value when it emerges from the hierarchy of LSPs. ...*

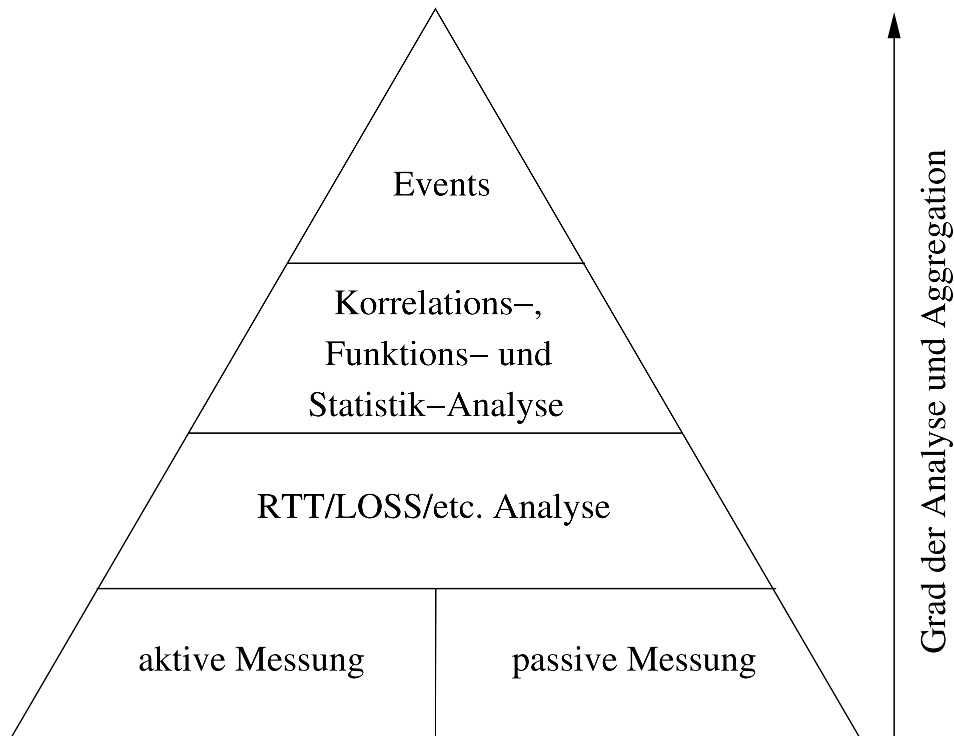

Abbildung 3.3: Analyse- und Aggregations Pyramide

auch der AS-Pfad (Autonomes System) ändert. Findet hier ebenfalls eine Änderung statt, so handelt es sich sogar um einen Inter-Domain-Routenwechsel. Bleibt der AS-Pfad jedoch gleich, so handelt es sich um einen Intra-Domain-Routenwechsel. Genauso sieht es aus, wenn sich der TTL-Wert um einen größeren Wert (+-10 Hops) ändert. Hier muss ebenfalls auf den AS-Pfad geschaut werden, bevor eine Aussage über die Art des Routenwechsels gegeben werden kann. Ebenso kann es passieren, dass sich der TTL-Wert nicht ändert, aber der AS-Pfad hat sich geändert. Dies bedeutet, dass ein anderer Pfad genommen wird, aber insgesamt die gleiche Anzahl an Zwischensystemen durchlaufen wird. Evtl. ist diese Änderung der Route bei der Beobachtung des RTT-Wertes von TCP zu erkennen, weil sich die physikalische Länge der Übertragung geändert hat.

3.6.2 Aggregation- und Analysemethoden

Im Folgenden sollen Methodiken zur automatischen Aggregation und Analyse der beschriebenen Merkmale vorgestellt werden. Die Methodiken lassen sich in mehrere Bereiche einteilen. Zum einen ist dies die Analyse durch Methoden der Analysis und zum zweiten durch Anwendung statistischer Methoden. Des Weiteren folgt die Aggregation durch das Sampling der Daten. Das letztes Glied in der Kette der Analyse und der Aggregation stellt die Gewinnung von Events (Ereignisse) aus den vorhergehenden Ergebnissen dar. Die einzelnen, im Folgenden geschilderten, Methoden bauen aufeinander auf. Dies wird aus

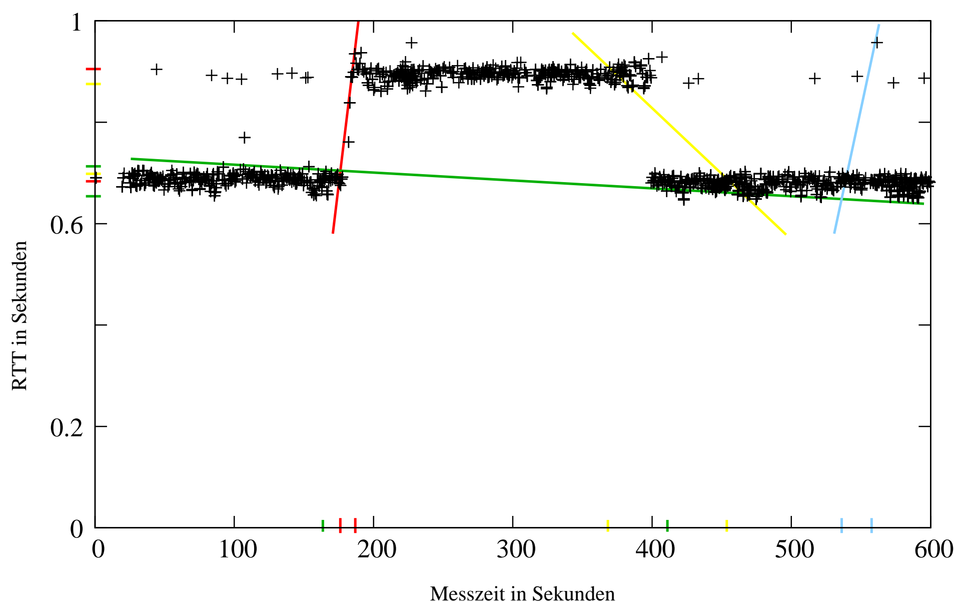

Abbildung 3.4: Nutzung des Differenzenquotienten zur Analyse der RTT-Werte

der Abbildung 3.3 ersichtlich. Zuerst werden aus einem Paketstrom durch die aktive und passive Messung Messdaten gewonnen. In einer ersten Stufe werden aus den Rohdaten erste Messergebnisse gewonnen. Diese können in einem weiteren Schritt durch die Korrelation, Statistik und Analysis weiter analysiert werden. Im letzten Schritt werden aus diesen Ergebnissen die Events generiert, die Ereignisse und Eigenschaften eines Netzes wiederspiegeln.

3.6.3 Funktionsanalytische Analyse

Dieser Abschnitt beschreibt die Aggregation und Analyse der Messergebnisse durch Erzeugung und Nutzung des Differenzenquotienten. Das Ergebnis der Bildung des Differenzenquotienten ist ein Maß für den Grad der Veränderung von Messwerten. Dazu werden die Messwerte von zwei Messzeitpunkten von einander subtrahiert und durch die Differenz der beiden Messzeitpunkte dividiert. Bei TCP-RTT Werten werden zwei RTT-Werte zu verschiedenen Zeitpunkten gewählt und subtrahiert diese beiden Messwerte sowie die Messzeitpunkte voneinander. Anschließend dividiert man die Messdifferenz durch die Zeitdifferenz. Das Ergebnis ist ein Maß für den Grad der RTT-Änderung pro gewähltem Zeitintervall. Um vergleichbare Ergebnisse zu erzielen, muss dabei das Ergebnis auf ein bestimmtes Zeitintervall genormt werden. Führt man dies für jeden RTT-Messwert durch, so erhält man den Grad der Veränderung der RTT gegenüber dem vorhergehenden Messwertpaar.

Jedoch wurde schon bei der Analyse der RTT-Werte angedeutet, dass sich die gemessenen RTT-Werte von Paket zu Paket aufgrund verschiedener Einflüsse verändern. In einer gröberen Auflösung (Analyse von Messwerten, die zeitlich weiter auseinander liegen) kann es vorkommen, dass keine Änderung stattfindet. Dies wird z.B. in Abbildung 3.4 durch die grüne Gerade ersichtlich. Dabei

wird ein zu großer Zeitabstand zwischen zwei Messzeitpunkten gewählt und der offensichtliche Sprung der RTT wird nicht dedektiert. Die blaue Gerade detektiert scheinbar einen Sprung der RTT, jedoch handelt es sich bei dieser Berechnung um einen Fehler, da es sich bei dem Messwert um einen Ausreißer handelt. Damit nicht Phantom-Sprünge erkannt werden, sollte hier z.B. mit statistischen Mitteln versucht werden, die Messergebnisse zu glätten um evtl. Fehler zu minimieren. Ebenso ist bei der gelben Geraden die Zeitdifferenz zwischen den Messwertpaaren zu groß gewählt, so dass darauf geschlossen werden kann, dass es sich nicht um einen Sprung, sondern um eine Stausituation handelt. Hieraus folgt, dass ein geeigneter Zeitraum gewählt werden muss, in dem die Berechnung des Differenzenquotienten stattfindet. Wird dieser Zeitraum zu groß gewählt, ist unter Umständen keine Aussage über den Verlauf der RTT-Werte zu treffen und ein zwischenzeitliches Ereignis geht verloren. Wählt man einen zu geringen Abstand zwischen den Messpunkten, so werden Ereignisse detektiert, die nicht vorhanden sind.
Aus diesen Beispielen wird ersichtlich, dass Schwellwerte festgelegt werden müssen, die die berechnete Steigung zwischen einem Messwertpaar klassifizieren. So handelt es sich bei einer geringen Steigung um kein Ereignis. Ist die Steigung in einem mittleren Bereich, so handelt es sich mit großer Wahrscheinlichkeit um eine Stausituation. Handelt es sich um einen großen Wert bei der Steigung, dann liegt mit Sicherheit ein Sprung der RTT vor. Da es keinen festen Steigungswert gibt, ab dem ein bestimmtes Ereignis vorliegt, bietet sich die Detektion von Ereignissen mit Hilfe von Fuzzy-Logik an. Die Steigung zwischen den RTT-Werten soll genutzt werden um Sprünge und starke Veränderungen in dem Verlauf der Werte über die Zeit zu detektieren. Diese Veränderungen sollen als ein Indikator von mehreren Indikatoren für die Detektion eines Routenwechsels genutzt werden.
Ebenso bietet diese Analysemöglichkeit die Chance Messdaten zu aggregieren, indem nur noch die Steigungen oder sogar nur erkannte Ereignisse gespeichert werden.

3.6.4 Statistische Analyse und Aggregation

Messdaten können mit Hilfe von Methoden aus der Statistik aggregiert und analysiert werden. Dabei können unterschiedliche Verfahren aus der Statistik angewandt werden. Zum einen besteht die Möglichkeit aus den Messdaten fortlaufend den Mittelwert, die Varianz sowie Spannweite, geometrische Mittel, Minima und Maxima zu berechnen. Eine mathematische Definition der Parameter findet sich in [Henz99]. Diese Werte können über den gesamten Messzeitraum berechnet werden oder die Berechnung findet für kleinere Zeitintervalle statt. Für die Größe der Zeitintervalle muss ein Kompromiss eingegangen werden, damit keine Informationen verloren gehen. Werden die Werte über zu große Intervalle berechnet, so können unter Umständen keine Veränderungen beobachtet werden, jedoch wird die Anzahl der Messdaten sehr deutlich reduziert. Dem gegenüber bedeutet die Wahl eines kleinen Zeitintervalls, dass die Zahl der Messdaten nicht so stark reduziert wird, aber die Wahrscheinlichkeit wird verringert, dass Veränderungen der Messwerte nicht detektiert werden.

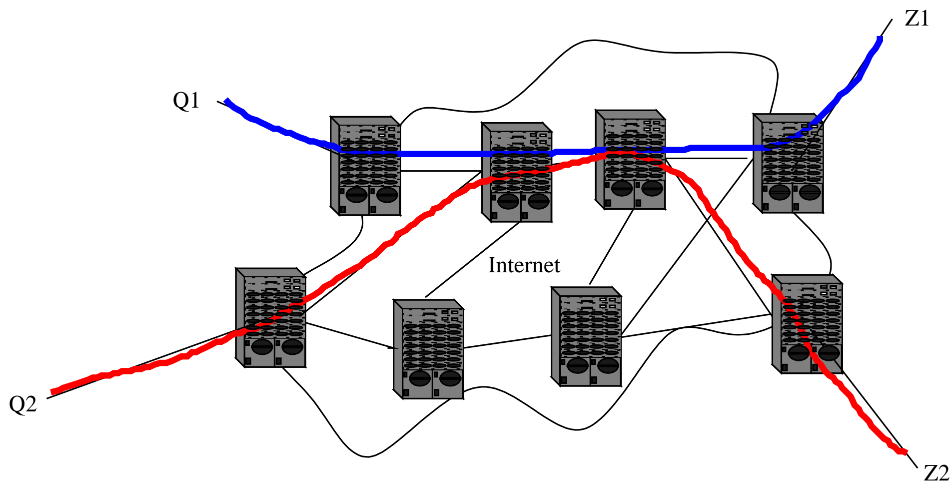

Abbildung 3.5: In der Abbildung haben die beiden Verkehrströme eine gemeinsame Teilstrecke. Momentaufnahme: Teilabbildung zum Zeitpunkt t_1

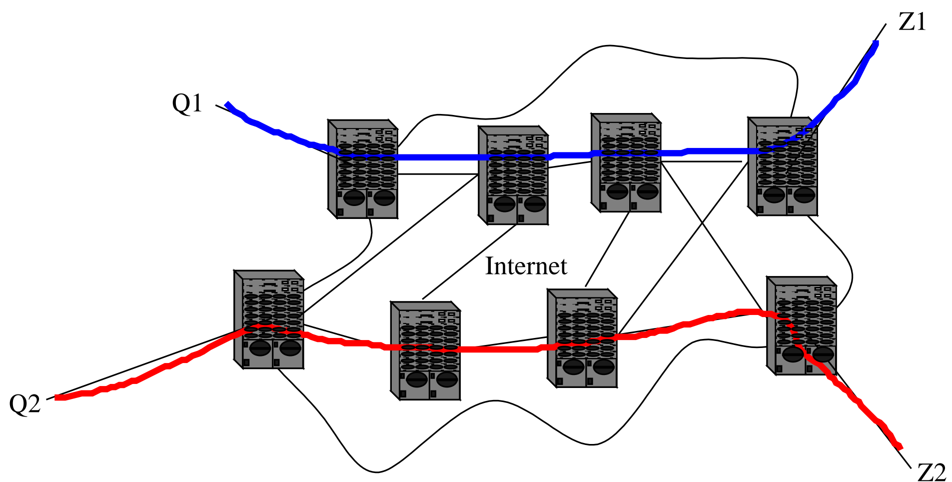

Abbildung 3.6: In der Abbildung sind die beiden Pfade der Verkehrsströme verschieden. Momentaufnahme: Teilabbildung zur Zeit t_2

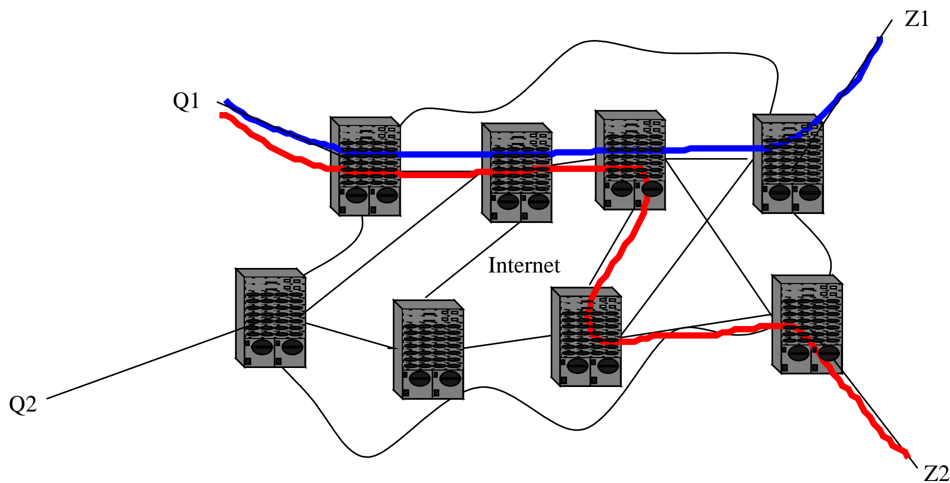

Abbildung 3.7: In der Abbildung haben die beiden Verkehrsströme eine gemeinsame Quelle und verschiedene Ziele. Beide Ströme haben einen gemeinsamen Teilpfad.

Zu den beschriebenen Werten, die berechnet werden, besteht die Möglichkeit die Häufigkeit der Messwerte über den Wertebereich zu bestimmen.[18] Mit Hilfe dieser Häufigkeitsverteilung lassen sich ebenfalls Veränderungen über die Zeit beobachten, wenn sich z.B. das Maxima verschiebt oder die Form der Verteilung sich in irgendeiner anderen Weise verändert (Varianz wird größer, etc).

Neben der Erstellung und Berechnung der genannten Werte und Verteilungen existiert eine weitere Möglichkeit die Daten zu aggregieren und analysieren. Sie bekommt im Hinblick auf die Entwicklung eines verteilten Systems zur Messung von Internetverkehr eine besondere Bedeutung. Es handelt sich dabei um die Korrelation von Messdaten. Mit Hilfe der Korrelation können Aussagen über die Ähnlichkeit von Messergebnissen getroffen werden. In den Abbildungen 3.5, 3.6, 3.7, 3.8 und 3.9 sind die prinzipiellen Möglichkeiten bei der Korrelation dargestellt. In den beiden Teilabbildungen 3.5 und 3.6 sind zwei Paketströme dargestellt, die jeweils in beiden Teilabbildungen die gleichen Quellen und Ziele haben. Diese Paketströme sind jedoch zu unterschiedlichen Zeitpunkten dargestellt. In der oberen Teilabbildung haben die Ströme einen gemeinsamen Teilpfad. Im unteren Teilbild haben die beiden Ströme keinen gemeinsamen Teilpfad und sind vollständig disjunkt. Bildet man nun über die beiden Paketströme aus der oberen Teilabbildung die Korrelation, können Übereinstimmungen in dem Verlauf von z. B. der RTT mit Hilfe der Korrelation gefunden werden. Diese Übereinstimmungen sind dann mit hoher

[18]Diese Häufigkeitsverteilung kann nur bei diskreten Werten einfach erstellt werden. Handelt es sich um kontinuierliche Werte, muss eine Einteilung in einzelne Abschnitte unternommen werden, in die anschließend die Werte einsortiert werden. Jedoch ist es schwierig die richtige Größe der Abschnitte zu wählen. Bei falscher Wahl der Abschnittsgröße können ebenfalls falsche Ergebnisse das Ziel sein.

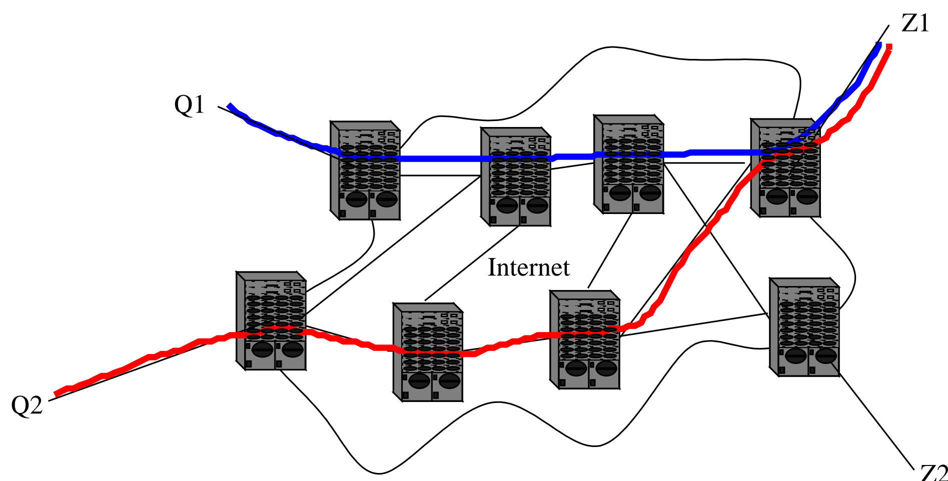

Abbildung 3.8: In der Abbildung haben beide Verkehrsströme das gleiche Ziel, jedoch unterschiedliche Quellen. Die Pfade sind zum Teil gleich.

Tabelle 3.3: Korrelationsmöglichkeiten

Nr.	$\mathcal{M}_1$	$\mathcal{M}_2$
1	$[\text{Q1Z1}]_{t_1}$	$[\text{Q2Z2}]_{t_1}$
2	$[\text{Q1Z1}]_{t_1}$	$[\text{Q1Z2}]_{t_1}$
3	$[\text{Q1Z1}]_{t_1}$	$[\text{Q2Z1}]_{t_1}$
4	$[\text{Q1Z1}]_{t_1}$	$[\text{Q1Z1}]_{t_2}$

Wahrscheinlichkeit auf den gleichen Teilpfad zurückzuführen. Im unteren Teilbild sollte keine Übereinstimmung im Verlauf der RTT zu finden sein. Wenn doch, dann kann dies höchstens daran liegen, dass sich verschiedene Pfade zu gleichen Zeiten gleich verhalten und ähnliches Verkehrsaufkommen haben. In den beiden Abbildungen 3.7 und 3.8 sollte in beiden Fällen eine Korrelation der beiden Verkehrsströme vorhanden sein, da in beiden Teilabbildungen die Ströme gleiche Teilpfade besitzen. In der oberen Teilabbildung haben sie eine gemeinsame Quelle und verschiedene Ziele. In der unteren Teilabbildung haben die Paketströme verschiedene Quellen, aber das selbe Ziel. In der dritten Abbildung 3.9 ist in den beiden Teilabbildungen jeweils nur ein Paketstrom dargestellt. Dieser ist jedoch zu verschiedenen Zeitpunkten (t_1, t_2) gemessen worden. Damit lässt sich mit Hilfe der Korrelation bestimmen, ob sich ein Paketstrom zu verschiedenen Zeiten gleich verhält, oder ob sich Unterschiede feststellen lassen. Dieser Unterschied zu verschiedenen Zeitpunkten ist hier durch einen anderen Pfad zum Ziel dargestellt. Es muss aber nicht immer ein anderer Pfad sein, der Unterschiede hervorruft, sondern auch eine veränderte Auslastung auf einem Pfad zwischen einer Quelle und einem Ziel.

Eine Zusammenfassung der Korrelationsmöglichkeiten ist in Tabelle 3.3 zu finden. Ein Verkehrsstrom wird durch jeweils eine öffnende und schließende

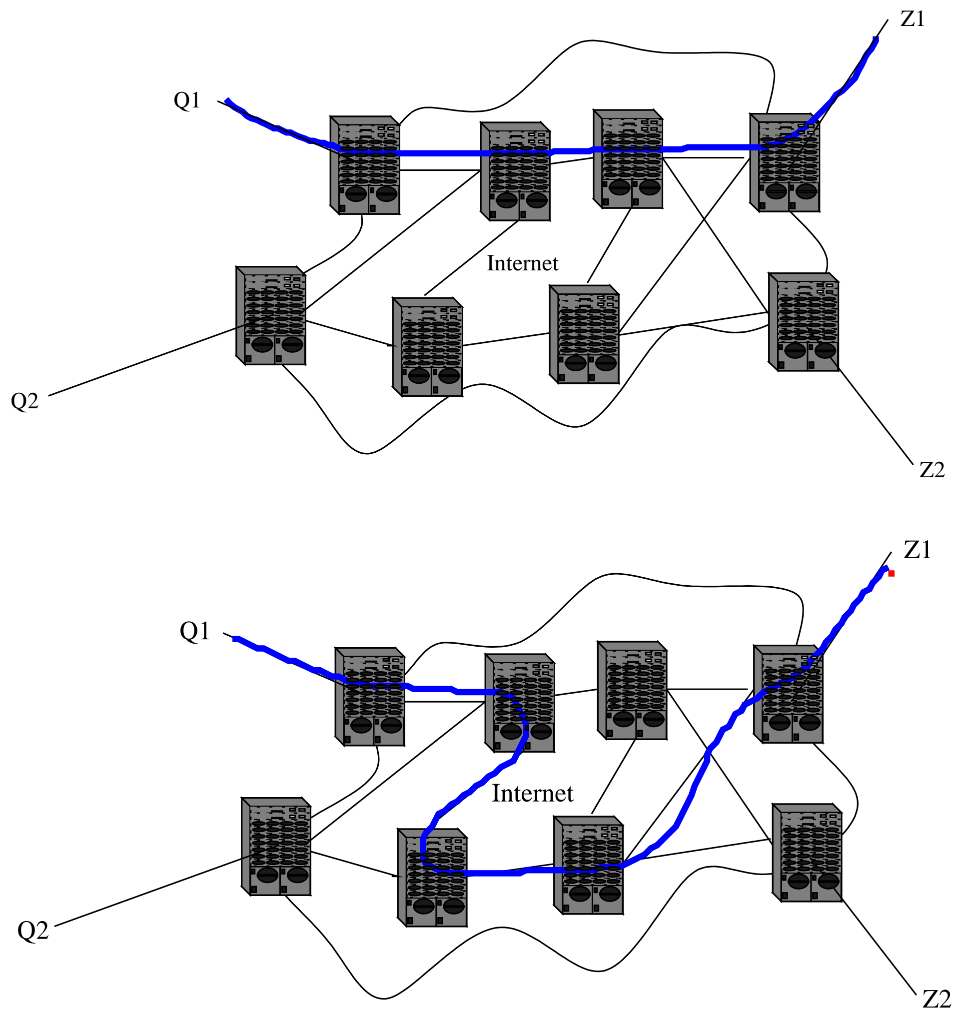

Abbildung 3.9: In der oberen Teilabbildung ist der Pfad eines Verkehrsstroms von Quelle (Q1) zu Ziel (Z1) zum Zeitpunkt t_1 dargestellt. Im unteren Teilbild ist der gleicher Verkehrsstrom zum Zeitpunkt t_2 dargestellt. Es gilt $t_1 < t_2$.

Klammer angegeben, die eine Quelle (Q) und ein Ziel (Z) hat. Mit angegeben ist jeweils der Zeitpunkt zu dem eine Messung gestartet wurde. Es gilt dabei, dass $t_1 < t_2$ ist. Die beiden Buchstaben $\mathcal{M}_1$ und $\mathcal{M}_2$ sind die beiden Messreihen, die miteinander verglichen werden sollen. Sie werden in Formel 3.4 genutzt.
Können Gemeinsamkeiten zwischen zwei Verkehrströmen verzeichnet werden, so können die Messreihen aggregiert und analysiert werden. So muss nur eine Messreihe gespeichert werden, wenn durch die Korrelation berechnet wurde, dass die Messreihen gleich, oder zumindest sehr ähnliche Verläufe haben und sich evtl. bis auf einen Offsetwert nicht unterscheiden.
Wie erwähnt ist die Korrelation ein Verfahren, mit dem der Grad der Ähnlichkeit zwischen zwei Messreihen berechnet werden kann. Um die Korrelation zu berechnen, wird ein Qualitätsmaß bestimmt, wie gut die beiden Messreihen übereinstimmen. Als Qualitätsmaß wird meist die Methode der kleinsten Quadrate nach *Carl F. GAUß* genutzt:

$$\mathcal{Q}(\triangle) = \sum_{i=1}^{n} (\mathcal{M}_1(x_i) - \mathcal{M}_2(x_{i-\triangle}))^2 \tag{3.4}$$

$$\mathcal{Q}(\triangle) \stackrel{!}{=} min. \tag{3.5}$$

$$\triangle \in \{-(n-1), \ldots, n\} \tag{3.6}$$

Aus dieser Definition folgt, dass die Messreihe $\mathcal{M}_2$ mindestens für die Zeiten x_{-n} bis x_{2n} definiert sein muss, bzw. Messdaten vorhanden sein müssen. Damit ist sichergestellt, dass die Korrelation der Messreihe $\mathcal{M}_1$ mit der Messreihe $\mathcal{M}_2$ von x_1 bis x_n berechnet werden kann. Bei der Messreihe $\mathcal{M}_2$ kann es sich auch um eine stetige Funktion im mathematischen Sinne handeln. Damit ist es möglich herauszufinden, ob die Messreihe $\mathcal{M}_1$ einer bestimmten gewählten Funktion folgt oder nicht. Es besteht weiterhin noch das Problem, dass es sich bei den beiden Messreihen um keine stetigen Funktionen handelt, sondern um diskrete Wertepaare. Hinzu kommt, dass die Messwerte nicht exakt zur gleichen Zeit gemessen werden können und die Messzeitpunkte keine äquitestanten Abstände haben. Aus diesem Grund müssen die Messzeitpunkte beider Messreihen äquitestant gemacht werden. Diese Neuberechnung der Messzeitpunkte hat zur Folge, dass sich dadurch auch die Messwerte verändern. Um die Messwerte den an den äquitestanten Messzeitpunkten zu berechnen, bietet sich hier die Interpolation der neuen Messwerte an. Dieses Problem kann auch mit Hilfe von Mittelwertbildung gelöst werden. Dazu wird über alle Messwerte innerhalb eines bestimmten Zeitintervalls der Mittelwert gebildet und dieses Zeitintervall hat immer die gleiche Größe. Liegen nun beide Messreihen korrigiert vor, so kann die Korrelation berechnet werden. Dazu wird die Summe der Residuen zu einer gegebenen zeitlichen Verschiebung $\triangle$ berechnet. Dies geht aus Formel 3.4 hervor. Anschließend wird das Minimum unter allen berechneten Summen bestimmt (siehe Formel 3.5). Damit ist die zeitliche Verschiebung beider Messreihen bestimmt, bei der die beiden Messreihen am ähnlichsten sind. In einem weiteren Schritt wird der Offset-Wert bestimmt, um den sich die beiden Messreihen im Mittel unterscheiden.

Wird die Korrelation nun auf den Verlauf von RTT-Messungen angewendet, so lassen sich evtl. Rückschlüsse auf Veränderungen der Verbindungsqualität ziehen. Die beiden Messreihen müssen einen gemeinsamen Teilpfad besitzen um eine Korrelation zu berechnen. Sind beide Pfade der Messreihen disjunkt, kann per Zufall eine Korrelation zwischen den beiden Messreihen bestehen. Ebenso lässt sich eine Aussage treffen, ob eine zeitliche Veränderung in einer Messreihe auch in der zweiten Messreihe beobachtet werden kann, und ob diese Veränderungen gleichzeitig aufgetreten sind, oder erst nach einer gewissen Zeit. Oder es handelt sich bei einer beobachteten Veränderung nur um eine lokale Änderung, die keine globalen Auswirkungen hat. Es lassen sich genauso Korrelationen zwischen der RTT und der IP-TTL bilden. Hieraus lässt sich evtl. auf die Ursache einer Änderung der RTT schließen. Denn ändert sich im gleichen Augenblick auch die TTL, so kann es möglich sein, dass ein Routenwechsel stattgefunden hat. Jedoch ist dies problematisch, da sich der TTL-Wert nur mit Schwierigkeiten interpretieren lässt.[19]
Ebenso ist die Korrelation zwischen zwei Varianz-Messreihen durchführbar. Hierbei ist ebenfalls erkennbar, ob sich die beiden Messreihen ähnlich verhalten und sich die Werte der Varianz ebenso ändern. Damit kann versucht werden Rückschlüsse zu ziehen, ob Teile des IP-Pfades gleich sind, oder ob die beiden beobachteten Paketströme verschiedene Wege zum Ziel nehmen. Ebenso ist es sinnvoll eine Korrelation der Varianz mit der Anzahl der Paketverluste durchzuführen. Damit kann zum Teil besser die Qualität einer Verbindung beurteilt werden. So kann dies Erkenntnisse liefern, ob sich die Paketverlustrate auf einen Stau zurückführen lassen, oder ob die Paketverluste auf eine physikalisch schlechte Verbindung zurückzuführen sind.
Hat man die Korrelation zwischen zwei Messreihen berechnet, so hilft dies, die Messdaten zu aggregieren. Da die Korrelation den Grad der Ähnlichkeit zweier Messreihen beschreibt, ist möglich eine der beiden Messreihen zu löschen. Dies sollte aber nur geschehen, wenn sich die beiden Messreihen in hohem Maß ähnlich sind. So brauchen nur eine Messreihe und noch ein paar weitere Parameter, wie die Korrelation und einige wenige Messwerte, sowie Mittelwerte und Varianz gespeichert zu werden.

3.6.5 Sampling

Unter dem Begriff Sampling wird eine Methode der Aggregation verstanden, mit der sich Messdaten gezielt aggregieren lassen. Dabei werden die Messdaten gezielt ausgedünnt. Das bedeutet, dass nicht mehr alle generierten und erfassten Messdaten gespeichert werden, sondern es werden nur noch einige wenige Daten einer Messung gespeichert. Es muss jedoch bei dem Sampling beachtet werden, dass die Daten nicht zu sehr aggregiert werden. Denn mit dem Sampling gehen automatisch auch Informationen verloren, die evtl. relevante Ergebnisse enthalten. Diese Informationen sind nach dieser Aggregation verloren. Es muss deshalb ein Kompromiss zwischen dem Grad der Aggregation, d. h. der Minimierung des Datenvolumens und dem Verlust an Informationen

[19]Diese Problematik wird ebenfalls in Kapitel 3.6.1.2 behandelt.

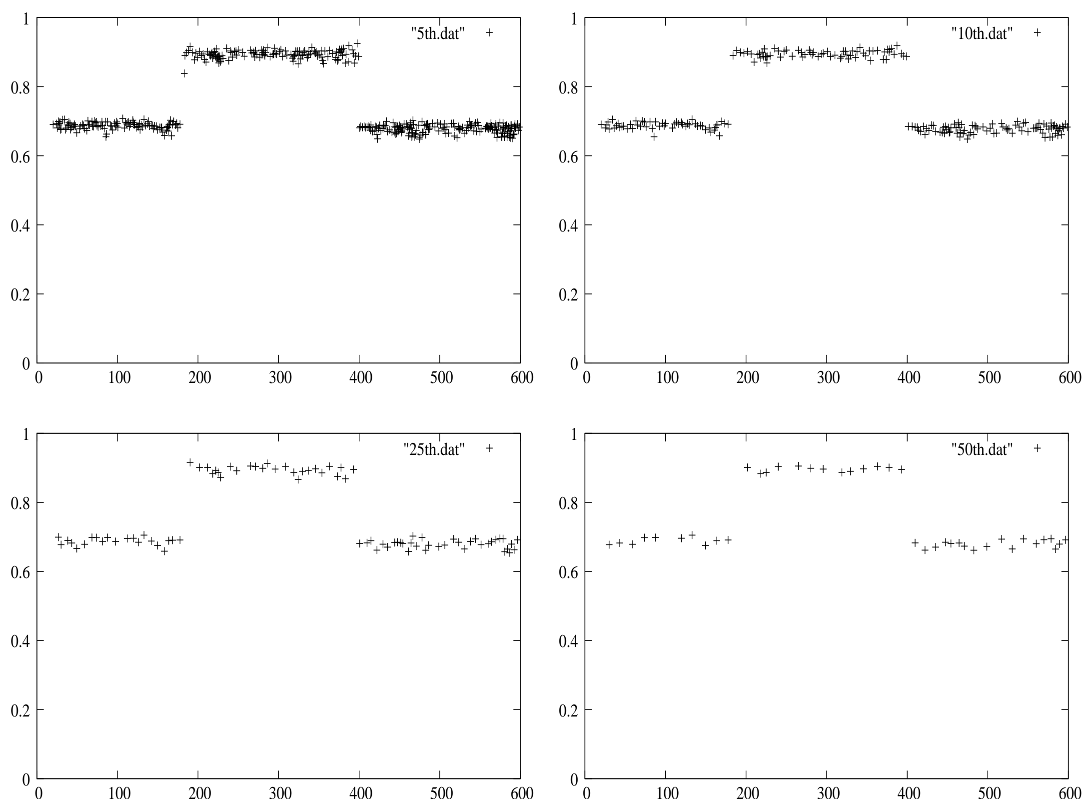

Abbildung 3.10: Das Original der Bilder ist in Abbbildung 3.4 zu sehen. Es zeigt den Verlauf einer gemessenen TCP-RTT über die Zeit. In diesen Bildern ist ein Sampling dieses Bildes in unterschiedlichen Graden dargestellt. Im linken oberen Bild wurde jeder 5. Punkt dargestellt. Im rechten oberen Bild jeder 10. Wert. Das linke untere Bild zeigt nur jeden 25. Wert und im letzten wird sogar nur jeder 50. Wert dargestellt.

gefunden werden.
Der Begriff Sampling sagt aus, dass Daten, meist kontinuierliche Daten, an vorher definierten Punkten neu erfasst werden. Diese Punkte können per Hand definiert werden, oder durch eine Funktion beschrieben werden. In Abbildung 3.10 ist ein Beispiel eines Samplings zu sehen. Dabei wird ein Messdatenverlauf[20] gezielt ausgedünnt. In diesem Beispiel wird je nach Teilbild nur noch jeder x-te Messwert dargestellt. Bei dieser Abbildung sind es von links oben nach rechts unten jeder 5., jeder 10., jeder 25. und jeder 50. Datenwert. Dies ist jedoch nur eine mögliche Metrik, die zur Aggregation genutzt werden kann. Metriken sind:

- Jeder x-te Wert
- alle x Millisekunden ein Wert
- Mittelwert über x Werte

[20]Hier ein Verlauf einer TCP-RTT über eine Messzeit von 600 Sekunden.

- gewichteter Mittelwert

Diese Aggregation muss nicht die einzige sein, die durchgeführt wird, sondern kann in regelmäßigen Abständen wiederholt werden. Hintergrund ist, dass Daten, je älter sie werden, immer weniger interessant für die Analyse werden, und somit nur Platz verbrauchen, der für neue Messdaten genutzt werden könnte. Das bedeutet, dass diese Aggregation iterativ wiederholt werden kann und damit die Daten nochmals aggregiert werden. Jedoch sollte ein Minimum an Informationen erhalten bleiben. Damit kann dann sichergestellt werden, dass neue Messdaten weiterhin mit den älteren Messdaten verglichen werden können und bestehende Erkenntnisse nicht verloren gehen.

3.6.6 Events

Wie aus der Abbildung 3.3 hervorgeht, steht die Gewinnung von Ereignissen (Events) und Eigenschaften an der Spitze der Analyse- und Aggregationshierarchie. Das bedeutet, das Ziel der kompletten Analyse und Aggregation ist es, aus den kontinuierlichen Messwerten, die erfasst wurden, letztendlich Events und Eigenschaften zu gewinnen, die in Kapitel 3.8 genutzt werden können. Die Generierung von Events ist keine leichte Aufgabe, da sich oft aus den Messdaten und der Analyse unterschiedliche Schlüsse ziehen lassen, und somit keine eindeutige Interpretation möglich ist. Diese Interpretationsschwierigkeiten werden z. B. in Kapitel 3.6.1.2 deutlich. Damit diese Interpretationsschwierigkeiten überwunden werden, muss überlegt werden, wie die einzelnen Ergebnisse besser interpretiert werden können und weniger Fehleinschätzungen erfolgen. Eine Möglichkeit Fehleinschätzungen zu minimieren, liegt darin, dass zur Erzeugung von einzelnen Events mehrere Ergebnisse herangezogen werden. So können z. B. Ergebnisse unterschiedlicher Analysen kombiniert werden. Dies kann eine Kombination aus einzelnen direkten[21] und indirekten[22] Messdaten, funktionsanalytischer Analyse und statistischer Analyse sein.
Die Erzeugung von Events und Eigenschaften soll es vereinfachen den Zustand eines Netzes besser einzuschätzen, da aus einfachen Messwerten meist keine Eigenschaften und Zustände abgelesen werden können. Jedoch sollen diese Events nicht nur Eigenschaften auf einer abstrakten Ebene widerspiegeln, sondern sollen diese Eigenschaften feingranular anbieten.
Es stellt sich nun die Frage, welche Ereignisse und Eigenschaften sollen erzeugt werden und machen im Zusammenhang mit der Messung und Analyse von Internetverkehr Sinn. In der folgenden Aufzählung sind einige Ereignisse und Eigenschaften aufgelistet:

- Routenwechsel
- starke RTT-Änderung

[21]Zu den direkten Messdaten gehört der IP-TTL Wert. Dieser wird direkt aus dem Wert im IP-Protokollkopf gewonnen.
[22]Die RTT ist ein indirektes Messdatum, da diese aus zwei Paketen berechnet wird.

- starke RTT-Schwankungen (hoher Jitter)
- gleiche RTT über x Sekunden
- hoher Paketverlust
- Vorhandensein von SACK-Pakete
- viele Verbindungsanforderungen
- Netz/Host nicht erreichbar
- viele fehlerhafte Pakete (falsche Checksummen)
- ausgelasteter Pfad
- ...

Es lassen sich weitere Ereignisse und Eigenschaften finden, mit denen ein Netz charakterisiert werden kann. Als Anhaltspunkt für weitere Eigenschaften können die Parameter in Kapitel 3.6.1 dienen. Im Folgenden sollen die einzelnen Events näher untersucht werden.

3.6.6.1 Hoher Paketverlust

Paketverlust bezeichnet den Verlust von Paketen durch verschiedene Ereignisse im Netz. Diese Verluste können durch das Überlaufen von Warteschlangen in Zwischensystemen hervorgerufen werden. Ebenso kann es zu Paketverlust kommen, wenn die Pakete eines unterliegenden Protokolls verloren gehen oder wenn die physikalische Verbindung eine schlechte Qualität aufweist und das Paket in einem Zwischensystem oder Endsystem, auf Grund vieler Bitfehler nicht mehr rekonstruiert werden kann. Geht ein Paket auf einer unterliegenden Verbindung (gegenüber der betrachteten Verbindung/Layer) verloren, so gehen auch implizit Pakete auf der betrachteten Ebene verloren.
Ein Paketverlust kann nicht direkt festgestellt werden, sondern kann nur indirekt ermittelt werden. Dies äußert sich darin, dass einige Pakete erneut übertragen werden. Dies setzt aber voraus, dass sich die einzelnen Pakete und ihre Reihenfolge identifizieren lassen. So kann ein Paketverlust nur mit Hilfe eines Protokolls ermittelt werden, dass eine zuverlässige Übertragung zwischen zwei Systemen implementiert. Zu dieser Kategorie von Protokollen zählt TCP. Daraus folgt, dass Paketverluste auf IP-Ebene nicht bemerkt werden können. Diese können nur in Verbindung mit der Erkennung von Paketverlust auf TCP-Ebene festgestellt werden. Jedoch muss ein Paketverlust auf TCP-Ebene nicht gleichzeitig bedeuten, dass auch ein Paket auf IP-Ebene verlorengegangen ist. Zum anderen muss beachtet werden, dass die Definition von TCP fest mit dem Verlust von Paketen rechnet und darauf seine Stau- und Flusskontrolle aufbaut. Somit ist bei einer TCP-Verbindung immer mit Paketverlusten zu rechnen. Paketverlust bei TCP heißt dem nach, dass mehr Pakete verloren gehen als bei einer ungestörten Verbindung. Diesen Paketverlust festzustellen erweist sich

als schwierig. Die Schwierigkeit liegt darin, dass der Paketverlust bei TCP von verschiedenen Faktoren abhängt, die über die oben genannten hinausgehen. Zum einen kommt es auf die Auslastung und Größe von Warteschlangen in verschiedenen Zwischensystemen an. Diese verschiedene Auslastung der Warteschlangen ist auf eine unterschiedliche Auslastung der einzelnen Teilstrecken einer TCP-Verbindung zurückzuführen[23]. Zum anderen hängt die Verlustrate von den parallel bestehenden TCP-Verbindungen und anderen Verbindungen (UDP, etc) ab. Diese verursachen sog. Querverkehr, der die beobachtete Verbindung beeinflusst. Diese parallelen Verbindungen verhalten sich auch nicht immer *fair* im Sinne von TCP und beanspruchen die Verbindungskapazität *greedy*.
Ein anderer Aspekt, der sich nicht auf den Paketverlust bei TCP beschränkt, ist die Frage nach der Verteilung der Paketverluste. Paketverluste können in Bündeln auftreten oder sind über die Zeit verteilen. Dabei ist aber darauf zu achten, wie der Zeitrahmen gewählt wird, in dem die Paketverlustrate ermittelt werden soll. Wird die Zeitspanne zu kurz gewählt, dann wird schon bei einem einzelnen Verlust eines Pakets erkannt, dass es die Paketverlustrate innerhalb des Zeitraumes hoch ist, da von den wenigen Paketen einige verlorengegangen sind. Wird ebenso die Zeitspanne zu groß gewählt, wird evtl. nur erkannt, dass in einem Zeitraum einige Paketverluste stattgefunden haben. Aber es wird nicht erkannt, dass die Paketverluste z.B. in Bündeln gehäuft aufgetreten sind, jedoch auf Grund der großen Zeitspanne auf einige wenige verteilte Paketverluste geschlossen wird. Diese Pakeverluste können dann fälschlicherweise als protokollabhängige Verluste (wie sie bei TCP vorkommen) gedeutet werden. Um nun Paketverluste deuten zu können. muss nicht nur eine einzelne Verbindung analysiert werden, sondern mehrere parallel. So können durch die Beobachtung paralleler TCP-Verbindungen Paketverluste erkannt werden, die durch zusätzliche Verbindungen / Verkehr erzeugt werden. Konkret bedeutet dies; es kann ein Zusammenhang zwischen dem Beginn einer neuen Verbindung und dem Paketverlust einer bestehenden Verbindung beobachtet werden. Dann lässt sich daraus schließen, dass ein Teilstück einer TCP-Verbindung voll ausgelastet ist und durch die Hinzunahme einer neuen Verbindung es zu zusätzlichen Paketverlusten kommt. Dieser Verlust ist dann bedingt durch die dynamische Verteilung der Bandbreite auf alle TCP-Verbindungen.
Bei der Untersuchung von Bündelfehlern ist, wie oben schon erwähnt, auf die richtige Granularität zu achten. Dies bedeutet, dass hierbei ein kleinerer Zeitrahmen gewählt werden sollte, in dem nur wenige Pakete liegen. Um nun zu ermitteln, ob es sich um einen Bündelfehler handelt, sollten fast alle Pakete in diesem Zeitrahmen verlorene Pakete sein und die verlorenen Pakete sollten aufeinander folgen und nicht auseinander liegen. Dies sollte eine untere Grenze haben, d.h. es sollte ein Minimum x von Paketen hintereinander verloren gegangen sein und es sollte ein Maximum y an Paketen zwischen diesen verlorenen Paketen sein, die korrekt angekommen sind. Werden diese Grenzen nicht eingehalten, so handelt es sich nicht um Bündelfehler. Dies könnte dann evtl. darauf hindeuten, dass es sich um einen Überlauf einer Warteschlange

[23]Eine Teilstrecke ist eine Verbindung zwischen zwei Zwischensystemen.

handelt und einzelne Pakete oder auch mehrere Pakete (aber Anzahl $< x$) hintereinander verloren gehen. Jedoch kann dies auch ein Irrtum sein, denn bei der Betrachtung von TCP stellt man fest, dass TCP dazu neigt in *Bursts* Pakete zu übertragen. Somit kann ein Überlauf einer fast vollen Warteschlange ebenfalls zu Bündelfehlern führen.

3.6.6.2 SACK-Pakete

Lassen sich SACK-Pakete beobachten, so ist dies ein sicheres Anzeichen für das Vorhandensein von Paketverlusten. Mit den SACK-Paketen beschreibt TCP genau die Pakete, die es nicht empfangen hat. Die fehlenden Pakete werden als Sequenznummernbereich innerhalb eines SACK-Paketes angegeben. Nimmt man weiterhin die in einer Verbindung ausgehandelte MTU (Maximum Transfer UNIT) an, so kann auf die Anzahl der verlorenen Pakete geschlossen werden. Bei älteren TCP Versionen, die keine SACK implementieren, tritt der in TCP implementierte Go-Back-N Algorithmus auf. Dieser überträgt alle Pakete ab dem verlorengegangenen Paket neu. Es ist dabei egal, ob nur ein Paket verloren gegangen ist oder alle bis zu dem Zeitpunkt der Wiederholung gesendeten Pakete. Dies hat zur Folge, dass mehr Pakete wiederholt werden, als nötig sind. Ebenso ist nicht die genaue Anzahl der verlorengegangenen Pakete ermittelbar. Somit läßt sich mit TCP ohne SACK nur feststellen, dass irgendein Paket verlorengegangen ist, jeodch nicht welches. Dies lässt sich nur feststellen, wenn sich die paketerfassende Instanz im Zielsystem befindet und auf IP-Schicht gelauscht wird. Dazu muss eine Analyseinstanz für TCP vorhanden sein, die unabhängig von der TCP-Implementierung des Zielsystems das Verhalten von TCP analysiert und so die Anzahl der verlorenen Pakete ermitteln kann. Wird TCP-Verkehr in einem Zwischensystem erfasst, muss ebenfalls eine Analyseinstanz für TCP vorhanden sein, die die erfassten TCP-Pakete analysieren kann und so auf die verlorenen Pakete schließen kann. In *Performance Enhanced Proxies* (PEP) wird diese Analyse in Kombination mit einem Paketcache zur Verbesserung von zuverlässigen Protokollen eingesetzt. Somit lässt sich mit einer parallelen[24] Analyse der TCP-Pakete eine recht genaue Aussage über die verlorengegangenen TCP-Pakete treffen. Diese Aussagen können für die Analyse über Paketverluste mit einfließen. Eine solche Analyse wurde im vorherigen Unterabschnitt beschrieben.

3.6.6.3 Starke RTT-Schwankungen

Diese Aussage, dass die RTT starken Schwankungen unterworfen ist, folgt der Annahme, dass ein Verbindungsabschnitt z.B. einer TCP-Verbindung stark ausgelastet ist. Diese Beobachtung sollte Folge einer hohen und schwankenden Auslastung von Warteschlangen in Zwischensystemen sein. Diese Schwankung kann z.B. durch viele kurze burstartige Verbindungen bedingt sein. Diese können z.B. durch Anfragen an Web-Server erfolgen, wodurch es zu vielen kleinen Verbindungen mit geringem Datenaufkommen kommen kann, welche zudem

[24] parallel zu dem in einem System implementierten TCP

noch in *bursts* übertragen werden. Die Folge ist, dass sich laufend die dynamische Verteilung der Verbindungskapazität ändert. Hinzu können auch Paketverluste kommen, die auf Grund der sich laufend dynamisch anpassenden Übertragungsrate einer Verbindung, hervorgerufen werden.
Die RTT-Schwankungen zeichnen ebenfalls aus, dass die RTTs innerhalb eines Zeitraumes eine große Streuung besitzen gegenüber einem Zeitraum, der keine RTT-Schwankung enthält. Es ist ebenfalls offensichtlich, dass sich die RTT zwischen je zwei Paketen ändert. Dies ist auf die hohe Auslastung der Verbindung bzw. des Verbindungsabschnittes zurückzuführen. Diese Änderung der RTT findet zwischen fast jeder RTT statt und stabilisiert sich nicht auf einem Niveau. Eine Stabilisierung der RTT auf einem Niveau würde eine andere Interpretation nach sich ziehen. die im folgenden Abschnitt erläutert wird. Die hier beobachtete Änderung der RTT geht über das Rauschen in dem Verlauf der RTT-Messwerte hinaus. Das Rauschen der RTT-Werte ist zum Teil aus Protokollmechanismen zurückzuführen (bei TCP kommt dies zum Teil durch kummulative Bestätigungen). Zu einem anderen Teil ist dies auch auf verschiedene andere Uraschen zurückzuführen (z.B. verschiedene Bearbeitungszeit im Endsystem).

3.6.6.4 Starke RTT-Änderung

Mit der starken Änderung einer RTT ist in diesem Zusammenhang eine Veränderung des Niveaus einer RTT gemeint. Darunter ist zu verstehen, dass sich bis zu einem bestimmten Zeitpunkt die RTT-Werte um einen relativ stabilen Mittelpunkt angeordnet haben, aber ab diesem Zeitpunkt ändert sich das Niveau der RTTs und stabilisieren sich dort. Die bedeutet die RTT-Werte liegen um einen neuen Mittelwert verteilt. Mit dem Niveau ist nicht gemeint, dass alle aufeinander folgende RTTs gleich sind, sondern innerhalb einer geringen Streuung liegen und nur einem Rauschen obliegen. Eine Änderung eines RTT-Niveaus kann auf verschiedene Art und Weise berechnet werden. Zum einen kann der Differenzenquotient zwischen den einzelnen Werten berechnet werden. Darauf folgt eine Messreihe von Werten, wie stark die RTT gestiegen bzw. gefallen ist. Die Messwerte bestehen aus positiven wie negativen Fließkommazahlen. Zu beachten ist, dass diese Differenzenquotienten genormt werden müssen, damit eine Vergleichbarkeit zwischen den einzelnen Werten der Messreihe gewährleistet ist. Die nächste Methode zur Berechnung der RTT-Änderung besteht darin, dass das Verhältnis zweier aufeinander folgender RTTs berechnet wird. Daraus folgt, dass eine Reihe von Ergebnissen entsteht, die sich um den Bereich um 1 bewegt. Die dritte Möglichkeit zur Kennzeichnung von starken RTT-Änderungen ist, die absolute Änderung zweier aufeinander folgender RTT-Werte. Eine Änderung sollte hierbei dadurch ersichtlich werden, dass eine Änderung der RTT durch einen Peak angezeigt wird und danach die Änderung zwischen aufeinander folgenden Werten wieder deutlich geringer ist.
Diese drei geschilderten Methoden können auch zur Erkennung von großen RTT-Schwankungen ohne Änderung des RTT-Niveaus genutzt werden. Dabei sind mehrere aufeinander folgende Peaks zu erkennen oder sogar fast alle Änderungen aufeinander folgender Peaks sind über einem gewissen Rauschpegel.

Jedoch ist dieser Rauschpegel zum einen von der absoluten RTT abhängig zum anderen durch andere geschilderte Faktoren bedingt. Somit kann kein fester Rauschfaktor angenommen werden und das Rauschen einfach heraus gerechnet werden. Vielmehr muss das Rauschen dynamisch herausgerechnet werden. Jedoch muss dabei darauf geachtet werden, dass das Rauschen nicht zu stark angenommen wird und somit reelle Änderungen der RTT herausgerechnet werden und somit Ereignisse nicht beobachtet werden können.

3.6.6.5 Viele Verbindungsanforderungen

Viele Verbindungsanforderungen für neue Verbindungen sind auf den ersten Blick nichts ungewöhnliches. Jedoch kann eine sehr hohe Anzahl an neuen Verbindungen negative Auswirkungen für das Zielsystem haben. In diesem Zusammenhang spricht man auch oft von *Denial of Service* (DoS) Angriffen. Diese sollen aber nicht Gegenstand dieser Analyse sein. Vielmehr soll aus den Verbindungsanforderungen heraus gefiltert werden, ob ein Zielsystem/-netz erreichbar ist, oder nicht. Dabei können Verbindungsanforderungen helfen, dies zu identifizieren. Bei einem Verbindungsaufbau z.B. bei TCP antwortet das Zielsystem mit einer Antwort an das Quellsystem. Bleibt diese Antwort aus, versucht das Quellsystem erneut eine Verbindungsanforderung an das Zielsystem zu senden. Der Gründe, warum das Zielsystem nicht antwortet, können vielfältig sein. Zum einen kann die Anfrage auf dem Hinweg verlorengegangen sein oder die Antwort kann auf dem Rückweg verlorengegangen[25] sein. Zum anderen wird der angeforderte Dienst auf dem Zielsystem nicht ausgeführt oder der Zugriff auf den Dienst wird aus administrativen Gründen verhindert. In diesen Fällen senden die Zielsysteme meist eine ICMP Antwort, warum der Verbindungsaufbau nicht erfolgt. Meist bedeutetdas, dass nicht immer eine Antwort generiert wird, sondern die Verbindungsanforderung einfach verworfen und keine Antwort versendet wird.
Damit lässt sich herausfiltern, dass ein Zielsystem/-netz oder Zieldienst nicht erreichbar ist, wenn wiederholt Verbindungsanforderungen gestellt werden, diese aber nicht beantwortet werden. Ebenso ist dies aus den ICMP-Nachrichten abzulesen, die generiert werden können. Diese Nachrichten können auch von Zwischensystemen wie Routern generiert werden, wenn keine Route zu einem Zielsystem existiert. Folglich lassen sich folgende Ereignisse generieren:
Verbindungsanforderung + Anwort –> reguläre Verbindung
Verbindungsanforderung (ohne Antwort) –> keine Aussage, da Antwort anderen Pfad nehmen kann und nicht erfasst wurde
Verbindungsanforderung + ICMP-Nachricht –> keine Verbindung, Grund ist in ICMP-Nachricht

3.6.7 Aspekte verteilter Messung und Analyse

Die im vorhergehenden Abschnitt 3.6.2 vorgestellten Methoden sind einfach zu handhaben, wenn die unterschiedlichen Messungen einer Verbindung immer

[25] Die Ursache für Paketverlust ist in einem vorherigen Abschnitt beschrieben und hat ebenfalls verschiedene Gründe.

auf dem gleichen Knoten durchgeführt werden. Dann gestaltet sich z. B. die Korrelation einfach, da die Messdaten immer von der gleichen Quelle[26] kommen. Damit werden viele mögliche Fehlerquellen[27] ausgeschlossen. Jedoch wie können die Methoden zur Analyse und Aggregation in einer verteilten Umgebung eingesetzt werden? In einer solchen Umgebung sind prinzipiell nicht nur eine Quelle von Messdaten vorhanden, sondern mehrere Quellen. Diese verschiedenen Quellen können jedoch ähnliche Messdaten mit gleichen Quell- und Adressdaten messen und empfangen. Dies kann durch Änderungen in Routen hervorgerufen werden. Wie ist mit diesen Daten umzugehen?
Betrachtet man den Parameter der Round-Trip-Time (RTT) aus dem Abschnitt 3.6.1.1, so spielt es keine Rolle wo dieser Parameter gemessen wird. Nach [Zülc04] ist es möglich die RTT in einem Zwischensystem zu messen. Will man jedoch vergleichende Analysen durchführen, so wird die RTT zwischen den beiden End-Systemen gebraucht. In [Zülc04] wird aber jeweils ein Teil der RTT zwischen je einem Endsystem und dem Zwischensystem gemessen. Um die Ende-zu-Ende RTT zu errechnen, sind die beiden Teile zu addieren. Es muss von diesem Wert evtl. noch eine Zeit, die sich aus der Analyse und daraus folgenden zusätzlichen Verzögerung der Pakete ergibt, abgezogen werden.
Diese Vorgehensweise gilt auch für andere Parameter. Wenn sich nur Teilparameter messen lassen, sind diese zu einem Gesamtwert zusammenzufassen. Jedoch müssen entsprechende mögliche Fehler herausgerechnet oder zu mindest minimiert werden.
Wenn die Analyse- und Aggregationsmethoden eine gleiche Datengrundlage erhalten, die zwar von verschiedenen Quellen (Knoten) stammen, aber von möglichen Fehlern befreit oder zumindest die Fehler minimiert wurden, sollten diese Methoden vergleichbare Ergebnisse liefern. Ein anderer Aspekt bei verteilten Messungen und Analysen besteht darin, dass Pfade zwischen Quellsystem und Zielsystem asymetrisch sind. Dies bedeutet, dass für den Weg vom Quellsystem zum Zielsystem ein anderer Pfad genommen wird gegenüber dem Rückweg. Dies hat zur Folge, dass bei verteilten Messpunkten[28] nicht immer alle Pakete, die zu einer Verbindung gehören, erfasst werden. Dieser Aussage wird zu Grunde gelegt, dass sich die Messpunkte in einem Zwischensystem (Router) befinden und nicht in einem Endsystem. In einem Endsystem, von dem eine Verbindung initiiert wird oder Ziel ist, werden offensichtlich alle Pakete einer Verbindung erfasst.
Damit wird die Aussagekraft von Ergebnissen herabgesetzt, da nicht mit Sicherheit entschieden werden kann, ob ein Paket nun einen anderen Pfad genommen hat, oder tatsächlich verlorengegangen ist. Somit treten Probleme bei der Analyse und Messung von von Internetverkehr in Zwischensystemen auf. Diese Probleme pflanzen sich in die Analyse und die Generierung von Ereignissen fort.

[26] im Sinne des gleichen Knoten
[27] Fehlerquellen und eine Klassifizierung von Fehlern findet sich in [Zülc04].
[28] Knoten/Endsysteme an denen Internetverkehr erfasst wird

3.7 Speicherung und Abfrage von Daten

Neben der Analyse und Aggregation der Messdaten sollte es auch eine Möglichkeit geben Messdaten zu speichern. Hinzu kommt, dass diese Daten nicht nur gespeichert, sondern auch wieder ausgelesen werden sollen. Um dies zu realisieren muss zunächst analysiert werden, wie die Messdaten gespeichert werden können. Dies muss im Hinblick auf die Speicherung und Abfrage von Paketdaten einer Internetverkehrsmessung sowie Ergebnissen der Analyse selbiger geschehen und sollte berücksichtigt werden. Zuerst sollte aber geklärt werden, welche Möglichkeiten bestehen Daten zu speichern. Neben der Art der Speicherung muss auch die Lokation der Daten betrachtet werden.

3.7.1 Lokation der Daten

Außer der Art der Speicherung ist die Lokation der Daten eine wichtige Eigenschaft, die die Zuverlässigkeit und die Skalierbarkeit der Speicherung wesentlich beeinflusst. Bei der Analyse der Lokation der Daten kommen zwei Möglichkeiten in Betracht. Zum einen ist dies die Speicherung der Daten an einer zentralen Stelle. Damit besteht aber Gefahr, dass diese zentrale Stelle schnell zum Engpass eines Systems werden kann, wenn viele Speicheranfragen und Leseanfragen bearbeitet werden müssen. Ebenso stellt diese zentrale Stelle einen *Single-Point-of-Failure* dar. Das bedeutet, fällt der zentrale Server aus, auf dem die Daten gespeichert sind, kann auf keine Daten mehr zugegriffen werden. Das System kann nicht mehr genutzt werden, bevor die Daten entweder mit einem Backup wiederhergestellt worden sind oder der Server wieder in Betrieb ist und Daten entgegen nehmen kann[29]. Diesem *Single-Point-of-Failure* kann in einem ersten Schritt entgegen gewirkt werden, in dem Lese-Repliken der Daten auf verschiedenen weiteren Servern angelegt werden. Das steigert die Flexibilität des Systems. Hierbei können nun parallele Datenabfragen gestartet werden, womit das System besser eine größere Anzahl an Leseanfragen bearbeiten kann. Schreiben ist weiterhin nur an einem zentralen Server möglich, der die Daten an die anderen Server verteilt. Jedoch muss immer auf die Konsistenz der Daten geachtet werden. In einem weiteren Schritt kann das System dann zu einem vollständigen verteilten Speichersystem erweitert werden, in dem an allen beteiligten Servern Leseoperationen und Schreiboperationen möglich sind. Hierbei ist noch zu klären, ob jeder beteiligte Server alle Daten vorhält oder nur Teile davon. Hält jeder Server alle Daten vor, entsteht wieder ein hoher Aufwand die Daten konsistent zu halten und er skaliert nicht mit einer großen Anzahl an beteiligten Systemen. Hält ein Server nur ein Teil der gesamten Daten vor, so muss aber jeder andere Server im Stande sein, die Daten auf anderen Servern zu lokalisieren und auf die Daten zuzugreifen. Dies erfordert aber einen großen Aufwand an Overhead zur Lokalisation und Synchronisation. Ebenso ist ein effizientes Management zur Replikation nötig. Eine neue Herangehensweise an die verteilte Speicherung der Daten ist mit Hilfe von verteilten Hashtabellen. Diese wird im nächsten Abschnitt geschildert.

[29]Hier liegt die Annahme zu Grunde, dass die Datenbasis nicht beschädigt wurde, sondern nur der Zugriff auf die Daten durch Fehler verhindert wurde. Z.B kann dies durch einen DDoS verursacht sein.

3.7.2 Art der Speicherung

Wie schon im vorherigen Abschnitt geschildert, muss bei der Speicherung der Daten analysiert werden, wie die Daten gespeichert werden können. Es bieten sich dabei drei Möglichkeiten an, wie die Daten gespeichert werden können. Dies sind Dateien, Datenbanken, und Hashtabellen. Dateien sind eine lang genutzte Art zur Speicherung von Daten und bieten eine sehr einfache Schnittstelle zum Speichern. Jedoch sind Dateien im Zusammenhang mit der verteilten Speicherung nicht einfach zu nutzen. Bei der Nutzung von Dateien muss unterschieden werden, wie die Daten innerhalb von Dateien abgelegt werden und wie diese organisiert sind. Zum einen lassen sich alle Daten in einer großen Datei ablegen, dann ist es notwendig innerhalb dieser Datei eine Struktur aufzubauen, mit der verschiedene Daten gefunden werden können. Jedoch sind große Dateien schwer zu verarbeiten und anfällig für Fehler. Eine andere Möglichkeit besteht darin, Dateien hierarchisch in einer Verzeichnisstruktur abzulegen und so der Datenspeicherung eine äußere Struktur zu geben. Zum anderen lassen sich dadurch mehrere Dateien parallel ansprechen und somit die Skalierbarkeit und Flexibilität des Zugriffs erhöhen.
Als nächstes bietet sich die Speicherung der Daten innerhalb einer Datenbank an. Diese kann die Daten innerhalb von mehreren Tabellen speichern. Jede Tabelle kann mehrere Spalten enthalten, die verschiedene Teile eines Datums aufnehmen können. Eine Datenbank enthält eine Abfragesprache, mit der die Daten mit Hilfe von relationaler Algebra (mengenorientierte Abfrage von Daten) aus einer Datenbank gelesen werden können. Dies macht eine Datenbank zu einem vorrangigen Kandidaten zur Speicherung und Abfrage von Messdaten. Ebenso ist es möglich durch Verweise zwischen einzelnen Tabellen einer Datenbank und über verschiedene Datenbanken hinweg Beziehungen zwischen Daten herzustellen. Zum anderen ermöglichen verteilte Datenbanken, indem einzelne Tabellen einer Datenbank auf verschiedenen Servern gespeichert werden. Somit kann auch hier die Last auf mehrere Server verteilt werden, wobei aber die Komplexität zur Synchronisation der Daten steigt.
Die dritte Möglichkeit zur Speicherung von Messdaten besteht in der Nutzung von Hashtabellen. Hashtabellen im Allgemeinen bieten nur die Abbildung von Schlüsseln auf entsprechende Werte. Es ist aber nicht möglich ohne Kenntnis der Schlüssel wieder auf die Daten zuzugreifen. Jedoch werden Hashtabellen oft eingesetzt. Ein Grund dafür ist auch der schnelle Zugriff auf die Daten (wenn der Schlüssel bekannt ist). Eine Weiterentwicklung der Hashtabellen zu verteilten Hashtabellen (siehe Kapitel 2.3) erlaubt es die Datenhaltung auf mehrere Rechner gleichmäßig zu verteilen. Damit wird dem Problem des *Single-Point-of-Failure* entgegen gewirkt. Zum anderen beherrschen einige Entwicklungen der verteilten Hashtabellen auch die Bildung von Repliken der Daten. Dazu werden die Dateninhalte auf mehrere Schlüssel abgebildet und somit auf verschiedene Rechner verteilt.[30] Diese verteilten Hashtabellen haben jedoch den

[30] Dies muss nicht unbedingt so sein. Deshalb sollten die Schlüssel so gewählt werden, dass sie den Schlüsselraum möglichst gleichmäßig aufteilen und somit eine gleichförmige Verteilung der Daten auf die einzelnen Knoten einer verteilten Hashtabelle erfolgt.

Nachteil, dass es unmöglich ist, Daten zu finden, wenn der Schlüssel unbekannt ist, unter denen die Daten gespeichert wurden. Ebenso ist es schwierig die Daten über eine mengenorientierte Abfragesprache, wie sie in Datenbanken verwendet wird, abzufragen. Bei verteilten Hashtabellen werden nur Daten zurückgeliefert, wenn der exakte Schlüssel bekannt ist. Dies gilt es zu überwinden, ohne jedoch die Vorteile, die verteilte Hashtabellen bieten einzubüßen. Ein anderes Problem der verteilten Hashtabellen ist, dass durch die Verteilung der Daten ein hoher Kommunikationsaufwand besteht, bis alle Daten einer Anfrage vorhanden sind. Auf Grund der Eigenschaften von verteilten Hashtabellen, dass diese nur Daten auf exakte Übereinstimmungen zu gegebenen Schlüsseln zurück liefern, gibt es einige Entwicklungen im Bereich der Datenbankforschung. Mehr dazu in Kapitel 3.7.4.

3.7.3 Struktur der Daten

Bei den Daten handelt es sich um Paketdaten von Internetverkehrsmessungen. Ebenso sind Daten der Analyse und der Aggregation zu speichern. Die Struktur der Pakete folgt aus der Definition der Protokolle, die erfasst werden. Zu den eigentlichen Messdaten wird zusätzlich die Zeit der Messung und der Ort der Messung erfasst. Nun muss erörtert werden, wie diese Daten, entweder in Datenbanken oder einer verteilten Hashtabelle, gespeichert werden können. Dies muss aber so geschehen, dass die Daten einfach wieder in dem Datenspeicher gefunden und gelesen werden können. Damit geht die Struktur, wie die Daten gespeichert werden mit der Frage nach der Abfrage der Daten einher. Denn eine Abfrage der Daten ist nur so einfach, wie die Daten strukturiert sind und entsprechend gespeichert sind. Es ist auch zu beachten, dass die Messdaten evtl. anonymisiert sind und somit nicht mehr alle Protokollfelder lesbar sind und damit nicht mehr nutzbar sind.[31] Zum anderen müssen die Besonderheiten der verschiedenen Datenspeicher, Datenbank und verteilte Hashtabelle, berücksichtigt werden. Eine Datenbank bietet eine strukturierte Speicherung der Daten in Tabellen an. Verteilte Hashtabellen erlauben zunächst nur eine flache Speicherung, da Hashtabellen nur eine Abbildung von Schlüsseln auf Daten bieten. Dies erlaubt zunächst keine Hierarchie in der Speicherung. Eine solche Hierarchie kann erzeugt werden, indem in einigen Schlüsseln die Daten eine Liste von Schlüsseln sind, die entweder auf weitere Schlüssel verweisen oder auf Daten. Damit lässt sich eine beliebig komplizierte und große Hierarchie von Daten erreichen. Dies hat aber im Bezug auf eine gleichmäßige Verteilung der Daten über alle beteiligten Knoten einer verteilten Hashtabelle Nachteile, da sich die Index-Schlüssel auf einigen wenigen Knoten befinden, die häufig abgefragt werden und somit zu einem Engpass des Systems werden können. Diesem Problem kann ein wenig damit begegnet werden, indem diese Schlüssel repliziert werden, bzw. unter einem anderen Schlüssel gespeichert werden, der ebenfalls bekannt ist und es somit zu einer Entlastung der Knoten kommt. Jedoch muss dabei die Konsistenz der Schlüssel-Wert Paare beachtet werden. Einher mit der Verteilung und damit mit der Skalierbarkeit

[31] Der Aspekt der Anonymisierung der Messdaten wird in Kapitel 3.9.3 diskutiert.

der Hashtabelle geht die Wahl der Schlüssel. Diese Wahl ist nicht einfach und muss für jede Anwendung überdacht werden. Für die Wahl eines Schlüssels stehen zum einen die Daten selber, die Variablennamen der Daten, sowie weitere Kontextinformationen wie Zeit, Knoten, Ursprung der Daten zur Verfügung. Die Daten selber als Teil des Schlüssels sin nur bedingt geeignet, da es nicht immer vorhersehbar ist, welche Werte diese annehmen. Als Beispiel für Daten, die genutzt werden können, sind z. B. IP-Adressen, TCP/UDP Portnummern. Werden die Schlüssel nicht sorgfältig gewählt, kommt es häufig zur Bildung von gleichen Schlüsseln, die dann zu Kollisionen führen und damit mehrere Daten unter dem gleichen Schlüssel auf einem Knoten gespeichert werden. Dies kann auf der einen Seite hinderlich sein, kann aber auf der anderen Seite dazu genutzt werden eine Hierarchie von Schlüssel-Wert Paaren aufzubauen. Wird ein zweiter Schlüssel generiert, der schon existiert, dann werden die neuen Daten zu den schon existierenden Daten hinzugefügt. Bestehen jedoch weitere Informationen, nach denen die Daten unterschieden werden können, wird daraus ein neuer Schlüssel gebildet und eine neue Hierarchiestufe damit gebildet. Aus der Bildung einer Hierarchie folgt, dass nur ein Einstiegsschlüssel bekannt sein muss, um auf alle weiteren Daten zuzugreifen. Zum anderen können durch die Wahl geeigneter Schlüssel die Daten gut verteilt werden, dann wird es aber schwieriger auf die Daten zuzugreifen, da der mögliche Schlüsselraum, in dem die Daten gespeichert sind, größer ist. Damit geht einher, dass viele Schlüssel nach möglichen Daten abgefragt werden müssen. Mit dieser Wahl wird eine flache Speicherstruktur erzeugt. Der Schlüssel wird hierbei aus einer Kombination der einzelnen zur Verfügung stehenden Variablen und Kontextinformationen generiert. Ein Beispiel für einen solchen Schlüssel ist der folgende:

{Source_IP}-{Dest_IP}-{Source_Port}-{Dest_Port}-{Protocol}-{Zeit}
(1.2.3.4-5.6.7.8-345-80-21:34:56.234642)

Hier ist dieser Schlüssel in seiner textuellen Darstellung. Genauso kann dieser Schlüssel auch in binärer Form dargestellt werden. Bei diesem Beispiel hätte der binäre Schlüssel eine Länge von 168Bit[32]. Ein Schlüssel für den Aufbau einer Hierarchie ist kürzer und besteht nur aus einem Teil der vorhandenen Variablen und evtl. einem Präfix/Postfix:

1. Hierarchiestufe: {Source_IP}_sip
2. Hierarchiestufe: {Dest_IP}_dip
3. Hierarchiestufe: ...

Es ist anzumerken, dass der Wert zu dem Schlüssel der 1. Hierarchiestufe aus Schlüsseln der 2. Hierarchiestufe besteht. Die Verteilung der Daten hängt somit direkt von der Wahl der Schlüssel ab.

[32] 32Bit je IP-Adresse, 16Bit je Port, 8Bit Protokoll und 64Bit Zeitstempel

3.7.4 Abfrage der Daten

Bei der Abfrage der Daten ist zu klären, wie wieder auf Daten zugegriffen werden kann. Dies ergibt sich aus der Art und der Struktur der Daten, wie diese gespeichert wurden. Werden die Daten in einer Datenbank gespeichert, so ist die Abfrage der Daten mit Hilfe einer integrierten Abfragesprache möglich. Diese Sprache hat einen großen Grad an Flexibilität, so dass die Daten so abgefragt werden können, wie sie zur weiteren Analyse benötigt werden.
Bei verteilten Hashtabellen sieht dies ein wenig anders aus. Um an die Daten einer Hashtabelle zu gelangen, ist es notwendig die Schlüssel zu kennen, unter dem die Daten gespeichert wurden, da eine Hashtabelle nur die Abbildung von Schlüssel auf Wert kennt. Somit ist es nicht möglich die Daten in einer Hashtabelle zu finden, wenn die Schlüssel unbekannt sind. Die Suche nach den Daten hängt somit direkt mit dem Aufbau der Schlüssel zusammen. Im vorherigen Abschnitt wurden zwei Beispiele für den Aufbau von Schlüsseln genannt. Es gibt zwei verschiedene Möglichkeiten um an die Daten heran zu kommen. Zum einen durch die Abfrage aller in Frage kommenden Schlüssel. Unter diesen Schlüsseln sind jedoch nur einige Schlüssel enthalten, die die gesuchten Daten enthalten. Die restlichen Schlüssel der Abfrage enthalten Daten, die nicht benötigt werden, oder es sind Schlüssel, die nicht existieren, d. h. es wurde kein solches Schlüssel-Wert Paar abgelegt. Um dies zu optimieren, können Metadaten angelegt werden. Diese Metadaten enthalten Listen von Schlüsseln die existieren und in der verteilten Hashtabelle abgelegt wurden. Somit müssen jetzt nur noch die Metadaten abgefragt werden und in der Folge nur noch die in den Metadaten enthaltenen Schlüssel. Eine Form der Metadaten ist die Bildung einer Hierarchie, d.h. die einzelnen Hierarchiestufen enthalten nur Metadaten, die die Menge der möglichen Schlüssel, die gesucht werden, mit steigender Hierarchiestufe verkleinern. In der vorletzten Hierarchiestufe sind dann die Schlüssel hinterlegt, die auf die gesuchten Daten verweisen. Eine andere Art die Daten aus einer Hashtabelle abzufragen ist die, dass eine Abfragesprache integriert wird. Diese Abfragesprache orientiert sich an der Abfragesprache, wie sie in relationalen Datenbanksystemen verwendet wird. Ein System, wie eine Abfragesprache in verteilte Hashtabellen integriert werden kann, wird in [MHHLS+02] und [RHLLS+03] beschrieben. Dieses System nimmt eine Abfrage entgegen und verteilt diese auf die einzelnen Knoten einer verteilten Hashtabelle. Auf den einzelnen Knoten wird die Abfrage soweit ausgeführt, wie es möglich ist und das Ergebnis an den Initiator der Anfrage geschickt. Dieser führt dann die Abfrage nochmals auf den Daten durch, die von den einzelnen Knoten zurückgeliefert wurden. Mit der Ausführung von Teilen der Abfrage auf den einzelnen Knoten der verteilten Hashtabelle wird ein zu großer Overhead, der durch die Kommunikation entstehen würde, vermieden und somit wird die verteilte Abfrage optimiert.

3.8 Ergebnisnutzung zum Netzmanagement

Die aus den erfassten Messdaten gewonnenen Ergebnisse sollen nicht nur dazu genutzt werden den Zustand eines Netzes zu charakterisieren und zu beob-

achten. Vielmehr sollen die Ergebnisse auch dazu genutzt werden Einfluss auf die beobachteten Netzwerke zu nehmen und die Performance zu verbessern. Ähnlich geschieht dies in [ABKM01b], jedoch wird dort zur Vermessung der Netzwerke eine aktive Vermessung durch Pings genutzt. Ebenso werden die Daten dort über ein Overlay weitergeleitet. Dies soll hier nicht geschehen. Hier soll der Verkehr über die *normalen* Verbindungen weitergeleitet werden. Jedoch sollen sich auf Grund der Ergebnisse die Wege ändern, die eine Verbindung zur Kommunikation nutzt.
Um die Ergebnisse zu nutzen, stehen verschiedene Protokolle zur Verfügung. Dies sind Routing-Protokolle oder auch Protokolle zum Abfragen von Status- und Managementinformationen von einzelnen Knoten – wie SNMP. Es würde jedoch über den Rahmen der Arbeit hinaus gehen, alle Möglichkeiten zur Darstellung der Ergebnisse für andere Systeme vorzustellen.
Die einfachste Möglichkeit besteht darin, die Ergebnisse auf dem Bildschirm auszugeben oder in eine Datei zu speichern.

3.9 Sicherheitsaspekte

In diesem Abschnitt soll geklärt werden, welche Sicherheits- und Anonymisierungsverfahren für eine verteilte Umgebung existieren. Das Spezielle an dieser verteilten Umgebung ist die Nutzung von verteilten Hashtabellen (Kapitel 2.3) und eines Overlay-Netzwerkes. Das Problem bei einer solchen Umgebung ist, dass keine zentralen Systeme existieren, die die gesamte Umgebung sichern können. Stattdessen müssen die einzelnen Knoten zusammenarbeiten um die Sicherheit herzustellen. Wenn jedoch ein offenes System entstehen soll, impliziert dies, dass es Knoten gibt, die nicht korrekt arbeiten und so an private Daten gelangen können. Meistens werden die Korrektheit und die Authentizität der Daten mit kryptographischen Techniken durchgesetzt. Um die Sicherheit in einem verteilten System, speziell einer verteilten Hashtabelle, zu etablieren werden in [dhts02] folgende Design-Entscheidungen getroffen:

- Verifizierbare Systeminvarianten definieren und diese verifizieren
- Transparenz gegenüber dem Abfragenden bei einer Datenabfrage
- Schlüssel müssen einem Knoten in einem verifizierbaren Verfahren zugeteilt werden.
- Serverauswahl durch das Routing soll vermieden und missbilligt werden.
- Kreuztests der Routingtabelle durch zufällige Abfragen
- Einzelne und zentrale Zuständigkeitspunkte vermeiden

Der komplette Umfang der Sicherheitsuntersuchungen im Bereich der verteilten Systeme, speziell bei verteilten Hashtabellen und Peer-to-Peer Netzen, fällt nicht in den Bereich dieser Arbeit, sondern tangiert diese nur. Aus diesem

Grund wird dieser Bereich nur kurz behandelt. Es finden noch umfangreiche Forschungen in dem Bereich statt. Jedoch soll noch auf die folgenden drei Punkte eingegangen werden, auch wenn diese nicht vollständig dargestellt werden können:

- Authentifizierung
- Verschlüsselte Datenübertragung
- Anonymisierung von Daten

3.9.1 Authentifizierung

Bevor überhaupt eine Datenübertragung zwischen zwei Maschinen oder Personen stattfindet, muss überprüft werden, ob es sich auch wirklich um die Maschine/Person handelt, mit der eine Kommunikation erfolgen soll. Durch die Authentifizierung wird dies sichergestellt. In klassischen Systemen[33] kann dies einfach vollzogen werden, da man sich gegenüber einer zentralen Stelle ausweisen kann und dies nachprüfen kann. Bei einem System, das keine zentrale Komponente/Maschine besitzt, wird es schwierig Authentifizierung zu etablieren und durchzusetzen. Die teilnehmenden Maschinen ändern sich dynamisch. Dadurch ändert sich auch fortlaufend die Gestalt des Systems und Kommunikationspartner wechseln ebenfalls laufend. Das Problem in dieser dynamischen Umgebung ist, wie kann eine Authentifizierung stattfinden, ohne auf *externe* Systeme zurückzugreifen?
In einer klassischen Umgebung kann eine Authentifizierung z. B. durch Zertifikate[34] aufgebaut werden. Um die Authentifizierung nachzuvollziehen wird die Echtheit eines Zertifikats überprüft.[35] Diese Zertifikate sind meist mit kryptographischen Verfahren[36] gesichert und werden ebenfalls durch diese signiert.
Ein anderes in der Praxis verwendetes Authentifizierungsprotokoll ist Kerberos. Dies wird unter anderem in [Schä03] beschrieben.

3.9.2 Verschlüsselte Datenübertragung

Mit einer Authentifizierung wird nur erreicht, dass der Gegenüber bekannt ist, keine falschen Daten injiziert werden, falsche Daten erkannt und verworfen werden können. Mit der verschlüsselten Datenübertragung erreicht man, dass diese Daten zwischen zwei Maschinen/Personen nicht von Dritten mitgelesen werden können. Damit ist es möglich, dass Daten vertraulich übertragen werden können. Dazu setzt man auch kryptographische Verfahren ein, die Daten,

[33] Netzwerksysteme mit einer zentralen Komponente

[34] Diese Zertifikate beinhalten die Merkmale, die eine Maschine oder Person identifizieren.

[35] Dies geschieht indem ein Echtheitspfad bis zu einer Stelle(CA oder selbst) aufgebaut wird, der man selber vertraut. Dies ist möglich, da die Zertifikate von anderen signiert werden. Damit wird bestätigt, dass dies die angegebene Person/Maschine ist. Die Überprüfung kann hierarchisch oder über sog. Web-of-Trust erfolgen.

[36] Einen Überblick über kryptographische Verfahren liefert [Schn96].

Tabelle 3.4: Protokollparameter zur Anonymisierung bei TCP/IP

Bezeichner	Protokoll
Source-Adresse	IPv4/IPv6
Destination-Adresse	IPv4/IPv6
Protokollnummer	IPv4/IPv6
Destination-Port	TCP/UDP
Source-Port	TCP/UDP

bevor diese versendet werden, verschlüsseln. Auf der Gegenseite müssen diese Daten nach dem Empfang wieder entschlüsselt werden. Da in der Kryptographie Schlüssel eingesetzt werden, mit deren Hilfe die Daten verschlüsselt werden. sind zusätzliche Protokolle zum Austausch der Schlüssel erforderlich.[37] In einem klassischen System werden häufig zum Verschlüsseln einer Datenverbindung die beiden Verfahren SSL/TLS und IPSEC eingesetzt. Eine Beschreibung dieser Protokolle befindet sich in [Schä03].

3.9.3 Anonymisierung von Daten

Neben der Möglichkeit Daten zu verschlüsseln, besteht die Möglichkeit Daten zu anonymisieren, in dem diese verändert werden. Das bedeutet, dass Hinweise auf den Ursprung und das Ziel von Daten soweit wie möglich entfernt werden. Es muss jedoch darauf geachtet werden, dass die Daten noch verwendet werden können. Im Hinblick auf die Messung und Analyse von Internetverkehr kommen hier die IP-Adressen und die TCP/UDP-Portnummer in Betracht, die anonymisiert werden können. Diese können in Kombination genau die Anwendung identifizieren, von der diese Daten verschickt wurden. Hintergrund der Anonymisierung ist, dass mit diesen Informationen nicht erlaubte Analysen gemacht werden können, die im juristischen Sinne verboten sind. Denn mit dem Mitlauschen von Verkehr ist es möglich in die Privatsphäre von Person unerlaubt einzudringen. Um dies soweit wie möglich zu umgehen und keine rechtlichen Probleme zu bekommen, gibt es Verfahren, mit denen die Daten und Adressen anonymisiert werden können.

Zum einen können Adressen durch eindeutige und zufällige andere Repräsentanten ersetzt werden. Die Wahl des Repräsentanten anstatt der IP-Adresse und der Portnummern aus TCP/UDP muss eindeutig sein. Weiterhin muss bei einer späteren Verbindung, die das gleiche Protokoll und die gleichen Ports benutzt, die Wahl wieder identisch sein, damit eine vergleichende Analyse der Daten möglich ist. Ebenso muss aus dem Repräsentanten die ursprüngliche Adresse nicht oder nur sehr schwer rekonstruierbar sein. Dies erreicht man durch eine Funktion, die nur einwegberechenbar ist. Dazu zählen kryptographische Hashfunktionen wie SHA-1 und MD-5 (siehe auch Kapitel 2.3). Diese Funktionen garantieren auch in einem gewissen Umfang eine Gleichverteilung. Dazu muss geklärt werden, welche Parameter zum Aufbau eines Wertes,

[37] Verfahren zum Schlüsselaustausch finden sich in [Schä03].

der gehasht werden soll, vorhanden sind und welche einen Sinn haben, genutzt zu werden. Diese Parameter gehen aus der Tabelle 3.4 hervor. Diese sind auf Protokolle im Internet beschränkt. Prinzipiell kann eine Kombination aus diesen Parametern einen Hashwert bilden. Jedoch macht der Client-Port der TCP/UDP Protokolle keinen Sinn, da sich dieser für jede Verbindung ändert. Die Server-Ziel Ports ändern sich für einen Dienst nicht. Diese Portnummern sind als Well-Known Ports bekannt. Eine Liste dieser Portnummern findet sich in [Hein99]. Ebenfalls findet sich dort eine Auflistung der Protokollnummern innerhalb des IP-Protokolls. Aus diesem Grund sollte ein Hashwert nur über eine Kombination der ersten vier Zeilen der Tabelle 3.4 erfolgen. Um nun die Anonymisierung durchzuführen, werden die in Frage kommenden Parameter genommen und in ein Byte-Array geschrieben.[38] Danach wird dieses Byte-Array mit Hilfe einer der obengeannten Hashfunktionen verarbeitet. Die zur Erstellung dieses Hashwertes benutzten Parameter eines Paketkopfes werden nun durch den resultierenden Wert der Hashfunktion ersetzt. Damit ist eine Anonymisierung des Paketkopfes erfolgt, da das Berechnen der Ursprungswerte aus einem kryptographischen Hashwert sehr schwierig ist. Jedoch ist dieses Verfahren deterministisch. Es werden immer die gleichen Parameterwerte auf den gleichen Hashwert abgebildet. Somit eignet sich dieses Verfahren zum Anonymisieren von Paketdaten. Jedoch gehen mit der Bildung eines Hashwertes auf diese Art (d. h. durch Aneinanderreihung von Paketparametern) Informationen verloren. Z.B. lassen sich nur noch Aussagen über Verbindungen mit gleichen Hashwerten herausarbeiten. Es gehen mit dieser Anonymisierung auch Informationen über gleiche Quell-/Zielnetze oder Quell-/Zielsysteme verloren, da jede strukturierte Information auf eine flache Identifikation abgebildet wird. Aus diesem Grund ist es sinnvoll, eine Anonymisierung zu entwickeln, die nicht mehr die Erkennung konkreter Quellen und Ziele zulässt, aber trozdem eine strukturierte Auswertung der Ergebnisse liefern kann. Um nicht alle Informationen, die eine detailierte Analyse verhindern, durch die Aneinanderreihung der Protokollparameter zu verlieren, können auch nur Teile eines Parameters benutzt werden um daraus einen Schlüssel für die Anonymisierung zu erstellen. Z.B. werden nur die ersten 16bit einer IPv4 Adresse genommen. Oder es werden mehrere Schlüssel generiert, die jeweils aus einem Teil der vorhandenen Parameter zusammengestellt werden. Werden die Parameter für die Anonymisierung schlecht gewählt, handelt man sich folgende Nachteile ein, die in [Peuh01] geschildert werden:

- *topology information is lost unless a seperate index for each network is maintained.*
- *there is no mapping between subsequent traces unless the table of mappings is stored securly.*

[38] Ein ähnliches Verfahren wird bei der Bildung von MPLS-Labeln genutzt. Bei dem Forwarding Equivalence Class (FEC) genannten Verfahren werden Quell- und Zieladresse sowie das ToS-Feld des IP-Protokolls und der Quell- und Zielport des TCP/UDP Protkolls genommen und auf ein 20bit langes Label gemappt.)

- *it is not possible to correlate traces collected from different points of network and,*
- *the table may grow large and thus difficult to store, especiallay in a capture device*

Zum andern sind für die Analyse des Verkehrs nur die Protokollköpfe der einzelnen Protokolle innerhalb eines Paketes von Interesse. Daraus folgt, dass zur Analyse nur die Protokollköpfe aufgezeichnet werden und nicht die Daten,die innerhalb eines Protokolls übertragen werden. Dies kann durch Abschneiden der Daten von den Protokollköpfen in einem Paket geschehen, d. h. es werden nur die Paketköpfe gespeichert und zur Analyse weitergeleitet. Bei einer Analyse von z. B. HTTP werden nur maximal 74 Oktett[39] benötigt. Damit ist ein Ausspähen der Nutzerdaten unmöglich, da nur die Protokollköpfe erfasst und betrachtet werden. Ebenso steigt die Zahl der Verbindungen im Internet, die ein Protokoll zur Verschlüsselung der Daten einsetzen. Zu diesen Protokollen zählen IPSec, TLS und SecSH.

3.9.4 Fazit

Im Bezug auf diese Arbeit spielt die Authentifizierung in dem Sinn eine Rolle, da mit dieser Hilfe festgestellt werden kann, ob die Daten auch von einer angegebenen Quelle stammen. Damit kann auch sichergestellt werden, dass nur authentifizierte Module und Personen Messungen und Analysen starten können und die Ergebnisse betrachten können. Es wird damit sichergestellt, dass die Daten nur einem bestimmten Kreis an Personen und Modulen ersichtlich werden und nur diese darauf zugreifen dürfen. Denn die Messdaten können sensible Daten enthalten, die vertraulich behandelt werden müssen. So könnten E-Mail-Inhalte mitgelesen werden. Werden die Inhalte aber zwischen Mail-Server und Client verschlüsselt, können keine Inhalte mitgelesen werden. Ein weiteres Mittel die Vertraulichkeit der Daten zu gewährleisten, ist, dass die Messdaten anonymisiert werden. Dies wird im vorhergehenden Abschnitt beschrieben. Ebenso sollten Dateninhalte ausgeblendet werden. Das bedeutet, es sollten nur die zur Messung und Analyse relevanten Teile eines Paketes verarbeitet und gespeichert werden. Die relevanten Teile eines Paketes sind die Paketköpfe.

[39] 14 Oktett Ethernet, 20 Oktett IP-Protokoll und 40 Oktett TCP Protokoll

4. Entwurf

In diesem Abschnitt soll ein Framework zur verteilten skalierbaren Messung und Analyse von Internetverkehrsströmen entworfen werden. Um eine verteilte Messung und Analyse durchzuführen, sind verschiedene Komponenten notwendig. Ebenso werden verschiedene Verfahren zur Interaktion der einzelnen Komponenten benötigt. Die einzelnen notwendigen Teile des Frameworks sollen ebenso verteilt angeordnet werden können. Dies erhöht die Flexibilität und Skalierbarkeit des ganzen Frameworks. Es ermöglicht auch die Messung und Analyse mehrerer Verkehrsströme gleichzeitig und unabhängig voneinander. Die einzelnen Teile des Frameworks sollen als einzelne Module realisiert werden. Die Kommunikation zwischen den einzelnen Modulen gehorcht einer gemeinsamen Schnittstelle, die jedes Modul implementiert. Mit dem Einsatz von Modulen wird es möglich, das Framework sehr leicht zu erweitern und es an neue Gegebenheiten anzupassen. Damit kann es mit der Größe des zu vermessenden Netzwerkes skaliert werden. Die Kommunikation der einzelnen Module untereinander wird durch die Nutzung eines Peer-to-Peer Netzes erfolgen. Mit der Nutzung dieser Netzwerkform ist es möglich, die Kommunikationsstrukturen flexibel zu gestalten und die Kommunikation robust gegenüber Störungen einzelner Verbindungen zu machen. Das Peer-to-Peer Netz wird ebenso zur Speicherung verschiedener Parameter und der Messdaten genutzt. Es hilft auch bei der Lokation einzelner Funktionsmodule und bei dem Management des kompletten Frameworks.
In den folgenden Abschnitten werden die einzelnen Aspekte des Frameworks definiert und entworfen.

4.1 Design des Frameworks

Die funktionale Struktur des Frameworks geht aus der Abbildung 4.1 hervor. Die einzelnen Teile des Frameworks decken die Bereiche zur Sammlung, Analyse, Aggregation und Speicherung der Messdaten ab. Die Messdaten zur verteilten Analyse werden in den Modulen der aktiven und passiven Messung gesammelt. Hinzu kommt eine PlanetLab spezifische Erfassung von Messdaten durch

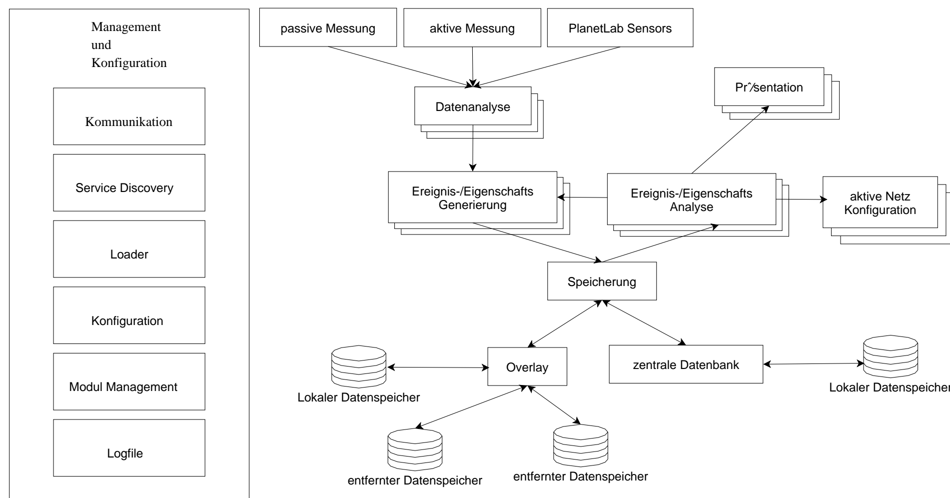

Abbildung 4.1: Design des Frameworks

sogenannte "PlanetLab-Sensors".[1] Die auf diese Weise gewonnenen Messdaten, die Paket- und Protokolldaten enthalten, werden anschließend in verschiedenen Analysatoren auf unterschiedliche Art und Weise analysiert. Diese Analysatoren sind auf verschiedene Protokolle und Paketformate spezialisiert. Es können auch mehrere Analysatoren des gleichen Typs gleichzeitig ausgeführt werden. Dann werden die Messdaten durch geeignete Filter getrennt und möglichst gleichmäßig auf die Analysatoren vom gleichen Typ verteilt. Diese Verteilung kann entweder auf Grund von Benutzereinstellungen und Konfigurationsvorgaben oder durch dynamische Verteilung, durch Analyse der Systemlast auf den einzelnen Analysatoren-Knoten geschehen. Die von den Analysatoren generierten Ergebnisse werden anschließend an Ereignis- und Eigenschaftsgeneratoren weitergereicht. die aus diesen Werten dann Ereignisse – wie *Routenwechsel* – oder ähnliche Eigenschaften generieren. Mit diesen Modulen wird der Messdatenstrom aggregiert und intensiver analysiert. Andere Generatoren fassen z.B. Ereignisse und Eigenschaften von mehreren Messquellen mit einem gemeinsamen Zielnetz zusammen und generieren daraus Ereignisse, die Aussagen über den globalen oder lokalen Zustand eines Netzes beschreiben.

Diese Eigenschaften und Ereignisse werden nun an allgemeine Kommunikations- und Speichermodule weitergereicht, die die Daten transparent speichern können. Diese Schicht verbirgt gegenüber dem restlichen Framework, wo und wie die Daten gespeichert werden. Dies kann, wie aus der Abbildung 4.1 ersichtlich wird, auf unterschiedliche Art und Weise vorgenommen werden. Ebenso wird die Lokation der einzelnen Module transparent gestaltet.

Die Messdaten werden einfach weitergeleitet, ohne dass auf die Lokation der Module geachtet wird. Nur im Inneren eines Framework-Knotens wird beachtet, wohin Messdaten transportiert werden. Die funktionale Struktur der einzelnen Module wird durch Konfiguration vorgegeben oder dynamisch festgelegt, an welche Module die Messdaten gesendet werden. In der Konfiguration

[1] Die Messquellen werden in Kapitel 3.3 vorgestellt.

werden optional Capabilities angegeben, die die Lokation des zu startenden Moduls beschreiben. Je weniger Optionen angegeben werden, desto freier kann das Modul innerhalb des Frameworks platziert werden. Diese Optionen legen z.B. die IP-Adresse oder das Netz fest, indem das Modul gestartet werden soll. Das Framework soll flexibel und skalierbar bleiben, in dem keine festen Aussagen über den Mechanismus, wie die Daten gespeichert werden, getroffen werden. Als Möglichkeiten bieten sich zentrale Datenbanken, wie auch Verteilte Hashtabellen, wie sie in Peer-to-Peer Netzwerken eingesetzt werden, an. Die Daten werden zur Speicherung an ein Modul übergeben, welches ein Interface zur Speicherung von Daten implementiert. Wie diese Daten gespeichert werden, bleibt dem Modul überlassen und wird nicht festgelegt.
Da es sich bei dem Framework um eine verteilte Architektur zur Messung und Analyse von Internetverkehr handelt, sollte auch eine Möglichkeit geschaffen werden, wie die gesammelten Messdaten von unterschiedlichen Quellen abgefragt werden können und gemeinsam (z.B. durch Korrelation) analysiert werden können. Zum einen wird dies durch allgemeine Kommunikations- und Speichermodule realisiert. Zum anderen existiert die Ereignis- und Eigenschaftsanalyse, die Daten mit Gemeinsamkeiten, wie gleiche Ziel- oder Quellnetze oder gleichen Protokollen abfragt und eine Analyse dieser Daten vornimmt. Wenn sich aus diesen Daten Ereignisse oder Eigenschaten ableiten lassen, so werden diese durch spezielle Ereignis-/Eigenschaftsgeneratoren erzeugt und in der verteilten Speicherstruktur abgelegt. Ebenso können diese Ergebnisse gleich an ein Präsentations- oder Visualisierungsmodul weitergereicht werden, welches die Daten auf unterschiedliche Art und Weise darstellt.
Das Präsentationsmodul kann nun verschiedene Schnittstellen implementieren, welche diese Daten auf unterschiedliche Weise darstellen. Eine Schnittstelle könnte die Daten per HTML darstellen oder eine andere könnte die Daten in einem Monitorprogramm anzeigen. Ebenso kann von dort aus eine Abfrage gestartet werden, ob Messdaten oder bestimmte Ereignisse existieren. Sind die darzustellenden Ereignisse oder Messdaten nicht gespeichert, bestehen zwei Möglichkeiten wie vom Präsentations-Modul aus entsprechende Analysen gestartet werden können. Dies geht aus dem Kapitel 4.7 hervor.
Ein spezielles Modul zur Konfiguration von Netzkomponenten implementiert Protokolle, z.B. die auf Grund der Ergebnisse Routenänderungen oder andere Ereignisse propagieren sowie per SNMP, DiffServ oder per FlexiNet-Modul die Ereignisse und Eigenschaften anderen Komponenten im Netz in einem geeigneten Format bereitstellen.
Neben diesen funktionalen Modulen gibt es noch weitere Module, die zur Verwaltung und Administration des Frameworks benötigt werden. Es sind Module zum Starten von neuen Modulen, zur Suche nach vorhandenen Modulen und zur Adressauflösung der Moduladressen vorhanden. Ebenso implementieren die Module ein verteiltes *Logfile*[2] und sorgen für die Etablierung von Sicherheitsmechanismen. Die Implementierung von Sicherheitsmechanismen wird in dieser Arbeit nicht behandelt. Eine kurze Analyse von Sicherheitsmechanismen findet sich in Kapitel 3.9.

[2]Das Logfile wird in Kapitel 4.4.7

4.2 Platzierung der Module

Da es sich bei der Struktur des Frameworks in Abbildung 4.1 nur um die funktionale Sicht auf das Framework handelt, ist zu klären, wie die einzelnen Module des Frameworks interagieren und wo diese platziert werden. In der Abbildung 4.2 und Abbildung 4.3 wird der nähere Aufbau eines Knotens dargestellt. Aus der Sicht der Platzierung und Kommunikation lassen sich drei mögliche praktische Umsetzungen für die einzelnen Module finden. Dabei ist die Kommunikation und Koppelung der Module abhängig von der Platzierung des Moduls auf verschiedenen Knoten. Für die Platzierung der Module ergeben sich zwei verschiedene Varianten. Zum einen können alle Module auf einem Knoten lokalisiert sein. Zum anderen sind die einzelnen Module über verschiedene Framework-Knoten verteilt. Daraus ergeben sich drei Möglichkeiten für die Kommunikation der einzelnen Module untereinander. Dies ist zum einen die *enge-Koppelung* und zum anderen die *lose-Koppelung* sowie die *hybride-Koppelung*.

"enge-Koppelung" Die erste Möglichkeit besteht darin, dass alle Module "eng" bzw. "fest" miteinander gekoppelt sind. "Eng" heißt in diesem Zusammenhang, dass die Module auf einem Knoten ausgeführt werden und direkt miteinander kommunizieren. Dies impliziert, dass alle Module auf einem Knoten ausgeführt werden und keine weiteren Knoten zur Analyse benötigt werden. Die Kommunikation kann durch zwei verschiedene Verfahren erfolgen. Zum einen durch Sockets[3] und zum anderen durch synchronisierte Warteschlangen. Wenn die einzelnen Module auf einem Knoten per Socket kommunizieren, bedeutet dies, dass die Module in verschiedenen eigenen Prozessen laufen. Zum anderen hat dies zur Folge, dass die einzelnen Module keine eigenen Warteschlangen benötigen, da der Socket als Warteschlange fungiert. Die andere Möglichkeit, wie die Module miteinander kommunizieren, ist die Kommunikation mittels gemeinsamen Speichers (Shared-Memory) und Threads. Dabei benötigen die einzelnen Module Eingangswarteschlangen, in denen die Nachrichten und Daten gepuffert werden, bis sie verarbeitet werden können. Um eine Nachricht weiterzuleiten, muss ein Modul nur die Warteschlange lokalisieren und die Nachricht in diese einstellen. Dies geschieht durch Aufruf einer Funktion. Der Zugriff auf die Warteschlangen wird gegen gleichzeitige Zugriffe geschützt.
Es besteht jedoch ein Problem bei beiden Verbindungsarten. Kann keine Nachricht mehr in der Warteschlange angehängt werden. gehen die Daten und Nachrichten verloren. Findet die Kommunikation oder Socket statt und der Socket wurde geschlossen, werden die Nachrichten ebenfalls gelöscht.

[3] Bei diesen Sockets handelt es sich um TCP-Sockets. Diese bieten die Möglichkeit, dass Module auch auf entfernten Maschinen ausgeführt werden können, ohne dass ein Loader und ein Dispatcher vorhanden sind. Diese Module können aber nicht durch den Loader gestartet werden, sondern müssen von Hand gestartet werden.

“lose-Koppelung” Die zweite Möglichkeit umfasst die “lose”-Koppelung der einzelnen Module. Alle Module laufen dabei auf verschiedenen Knoten. Dies impliziert, dass eine Kommunikation, über Knoten hinweg, zwischen einzelnen Modulen stattfindet. Hinzu kommt, dass eine Möglichkeit zur Adressierung der Module auf entfernten Knoten existiert, die es ermöglicht mit diesen Modulen Nachrichten auszutauschen. In diesem Szenario werden die Module entweder dynamisch im Netz platziert oder durch Angabe von Eigenschaften werden die Lokationen der einzelnen Module näher spezifiziert, bishin zur festen Beschreibung, auf welchem Knoten ein Modul ausgeführt werden soll. Diese Beschreibung des Moduls ist in der Konfiguration (Kapitel 4.9) angegeben. Bei der dynamischen Platzierung der Module im Framework werden Parameter wie die momentane Systemlast der einzelnen Knoten, wie auch die Verfügbarkeit einzelner Module auf den Knoten beachtet. Die Suche nach einem geeignetem Knoten, auf dem ein neues Modul ausgeführt werden soll, übernimmt das Service Discovery. Gestartet werden die Module anschließend durch einen Loader.
Um mit den Modulen über Knotengrenzen hinweg zu kommunizieren, muss es spezielle lokale Module geben, die die einzelnen Framework-Knoten miteinander verbinden. Dies geschieht transparent gegenüber einem Modul, dass das lokale Kommunikationsmodul als Proxy für das entfernte Module anspricht. Dieses Modul implementiert ein Protokoll, wie die einzelnen Knoten miteinander verbunden sind. Dies kann z.B. der Aufbau eines Peer-to-Peer Netzes sein, oder direkt Kommunikation der einzelnen Knoten per Sockets.

“hybride-Koppelung” Die dritte Möglichkeit eintspricht einer “hybrid”-Lösung. Diese ist eine Kombination aus der “engen”- und der “losen”-Koppelung der einzelnen Module. Bei dieser Lösung laufen nicht alle Module auf einem Knoten, sondern nur Teile der Framework-Module. Mit dieser Lösung wird das Ziel verfolgt, mehrere Module auf einem Knoten auszuführen, und damit diesen Knoten besser auszulasten. Die Möglichkeit alle Module, die zu einer Messung und Analyse gehören, auf einem Knoten auszuführen ist immer noch gegeben. Jedoch ist es bei der “engen”-Koppelung schwierig, eine verteilte Analyse durchzuführen, wenn alle Module lokal vorhanden sind und keine Kommunikation mit anderen Knoten besteht. Die Möglichkeit jedes Modul auf einen anderen Knoten zu platzieren, hat zur Folge, dass ein großer Overhead zur Verwaltung der einzelnen Knoten existiert. Jedoch wird pro Knoten nur ein Funktionsmodul ausgeführt.
Damit stellt diese Möglichkeit der “hybriden”Koppelung ein sehr hohes Maß an Flexibilität und Skalierbarkeit zur Verfügung. In Kombination mit der Definition unterschiedlicher Knoten-Typen in Kapitel 4.3 lassen sich die einzelnen Knoten klassifizieren. Dies hat zur Folge, dass die Module intelligent verteilt werden können und sich Spezialknoten etablieren können, die z.B. Analyse-Knoten dienen, wobei andere leistungsschwä-

chere Knoten nur die Messung von Verkehr vornehmen. Die Messdaten werden dann von diesen Messknoten zu den Analyseknoten weitergeleitet.

4.3 Knoten-Typen

Auf Grund der Einteilung der Koppelungen der einzelnen Module untereinander, lassen sich fünf Grundtypen an Knoten definieren. Dies sind *Full Node, Presentation Node, Analyse and Configure Node, Embedded Node (Router und Switches)* und *Proxy Node.*

Ein **Full Node** beinhaltet alle Funktions-Module des Frameworks. Diese können als einzelne Prozesse (Daemons) nebeneinander auf dem Knoten laufen oder innerhalb eines Prozesses als Threads. Dies bedeutet jedoch nicht, dass alle Module gestartet sein müssen. Es besteht nur die Möglichkeit alle Module zu starten, je nachdem, ob sie für eine Messung und Analyse benötigt werden. Dieser Knotentyp enthält neben den Funktionsmodulen auch alle Module, die zur Verwaltung und zum Management eines Knoten benötigt werden.

Der **Presentation Node** enthält nur die Module des Frameworks, die zum einen die Ergebnisse der Messungen und Analysen anzeigen können und zum anderen die Konfigurationsparameter und Zustände der einzelnen Framework-Knoten ausgeben können. Dazu muss er ebenfalls über Module verfügen, mit deren Hilfe alle Module und laufenden Messungen und Analysen abgefragt und gesteuert werden können.

Analyse und Configure Nodes nutzen die Module zur Analyse und Aggregation, sowie zur Erzeugung von Eigenschaften und Ereignissen und die zur Konfiguration von externen Knoten (Routern, Switches).

Im **Embedded Node** finden sich nur die Module, die Internetverkehr mitlauschen und sammeln. Eventuell sind die Module zur ersten Analyse der Messdaten, zur Speicherung und Weiterverarbeitung an andere Knoten enthalten. Die vorhandenen und genutzten Module hängen von der Leistungsfähigkeit der Embedded Knoten ab. Bei einem solchen Knoten muss es sich nicht um einen vollständigen Framework Knoten handeln, sondern die Module zur Erfassung von Messdaten sind an einen Proxy Knoten angebunden und besitzen keine eigenen Module zum Management der Module.

Der **Proxy Node** führt keine eigenden Module aus. Er stellt nur Informationen über andere Module (z.B. auf Embedded Knoten) auf anderen Knoten bereit. Bevorzugt sollte er als Stellvertreter für Embedded Knoten agieren und diese im Framework vertreten. Diese Embedded Knoten werden nur über den Proxy von außen angesprochen . Er implementiert die komplexeren Mechanismen zur Interaktion mit anderen Modulen im

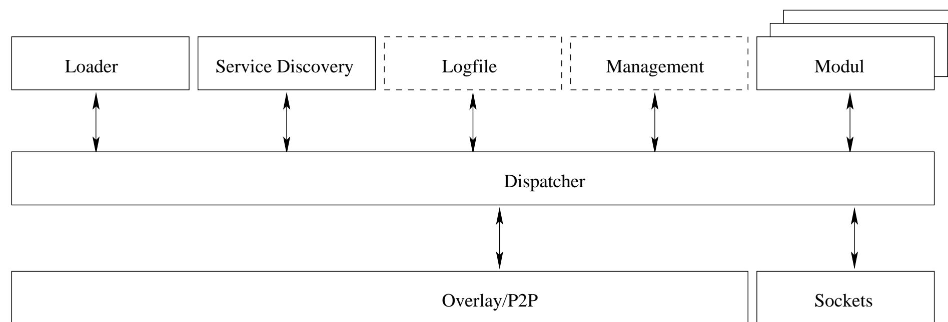

Abbildung 4.2: Aufbau eines Framework-Knoten

Framework[4]. Die Embedded Nodes können diese Mechanismen zum Teil aufgrund ihrer geringen Leistungsfähigkeit nicht erbringen.

4.4 Aufbau eines Knotens

In dem vorherigen Abschnitt wurde das funktionale Design eines verteilten Systems zur skalierbaren Messung und Analyse von Internetverkehrsströmen definiert. In diesem Abschnitt soll nun der reale Aufbau des Frameworks beschrieben werden. Begonnen wird mit dem Design eines Framework-Knotens. Dieser Aufbau eines Knotens ist in der Abbildung 4.2 dargestellt. Daraus geht hervor, dass ein Knoten selbst wieder aus einer Reihe von Modulen besteht. Damit setzt sich hier die Modularität fort, die schon im vorherigen Abschnitt erwähnt wurde.
Die Zentrale Komponente eines Knotens ist der Nachrichten-Dispatcher. Dieser sorgt dafür. dass die einzelnen Module untereinander kommunizieren können. Ebenso sorgt er für die richtige Weiterleitung der Nachrichten zu den entsprechenden lokalen Modulen. Um den Dispatcher sind verschiedene weitere Module gruppiert. Als erstes sind dies spezielle Module, die die Kommunikation mit anderen Knoten des Frameworks bereitstellen. Dazu zählt das Modul zur Nutzung des Overlays/DHT (z.B.Chord). Hinzu kommt das Socket-Modul. Dieses Modul stellt sicher, dass die Messdaten direkt zwischen zwei Knoten ausgetauscht werden können und die Kommunikation nicht über mehrere Overlay-Knoten geführt werden muss, denn jeder Zwischenknoten fügt der Kommunikation zusätzliche Verzögerung hinzu. Diese Socket-Schnittstelle kann ebenfalls dazu genutzt werden einzelne Module auf anderen Systemen auszulagern und sie somit an das Framework anzugliedern. Damit ist es möglich weitere Module in getrennten Adressräumen (Prozessen) auszuführen. Genauer geht der Aufbau eines Knotens aus Abbildung 4.3 hervor. Zu diesen Kommunikations-Modulen kommen weitere hinzu. Dies ist der sog. *Loader*. Dieser sorgt dafür, dass einzelne Module gestartet werden. Ebenso kann dieser *Loader* eine komplette Messung und Analyse zur Ausführung bringen, indem er für eine Messung und Analyse notwendige Module anhand einer Konfiguration lädt und

[4]unter der Annahme, dass die Module in einer “hybriden”-Struktur eingesetzt werden.

miteinander verbindet. Der *Loader* benötigt zur Erfüllung seiner Aufgaben ein Modul zum Service Discovery. Des Weiteren kommen Module zur Konfiguration und zum Management sowie Module zum Speichern von Daten und das Logfile hinzu.
Damit ein Knoten korrekt arbeiten kann und als Knoten für das Framework genutzt werden kann, müssen die, mit einer durchgezogenen Linie umrandeten, Module auf einem Knoten vorhanden sein. Die gestrichelten Module können auf einem Knoten ausgeführt werden, müssen aber nicht. Es muss nicht auf jedem Knoten ein Modul für das globale Management vorhanden sein. Jedoch muss ein Modul für das lokale Management vorhanden sein, worin Informationen und Daten über den lokalen Knoten gespeichert werden. Aus den vorhandenen Modulen auf einem Knoten lassen sich verschiedene Knoten-Typen generieren. Diese werden in Abschnitt 4.3 erklärt.
In den folgenden Unterabschnitten werden einzelne Aspekte und Komponenten beschrieben, die für den Betrieb eines Knotens wichtig sind. Ebenso werden Protokolle entworfen, wie verschiedene Abläufe durchgeführt werden.

Definition Framework Knoten:
Es handelt sich bei einem Knoten um einen vollständigen Framework Knoten, wenn auf diesem Knoten ein *Dispatcher*, *Loader* und ein *Kommunikationsmodul* sowie mind. ein Funktionsmodul ausgeführt wird. Wird nur ein einzelnes Modul auf einem Knoten ausgeführt, dann handelt es sich dabei um ein abgesetztes Modul auf einem Rechnersystem, welches kein Framework Knoten ist.

4.4.1 Dispatcher

In Abbildung 4.2 wird der Dispatcher als zentrale Komponente eines Knotens dargestellt, der alle Module miteinander verbindet. Jedoch besteht der Dispatcher aus mehreren einzelnen Teilen. Zum einen gehören zum Dispatcher die zentralen Tabellen, in denen die Weiterleitungsinformationen für die einzelnen Nachtrichten stehen. Dies geht aus der Abbildung 4.3 hervor. Hinzu kommt noch die Tabelle, in der die registrierten Kommunikationsprotokolle hinterlegt sind. Zum anderen besteht der Dispatcher aus einer Komponente, die in jedem Modul vorhanden ist. Genauer gesagt ist dieser Teil in der Modulschnittstelle verborgen, die in Abschnitt 4.4.3 beschrieben und in Abbildung 4.5 gezeigt wird.
Mit diesem Design des Dispatchers ist es möglich, eine Vielzahl von Nachrichten zum gleichen Zeitpunkt zu übertragen. Soll eine Nachricht von einem Modul übertragen werden, übergibt das Modul die Nachricht an die Kommunikationsschnittstelle. Innerhalb dieser wird der Dispatcher aufgerufen. Dieser schaut in den zentralen Tabellen nach, wohin die Nachricht lokal geschickt werden soll. Danach wird die Nachricht direkt an das lokale Modul weitergeleitet. Handelt es sich um ein entferntes Modul, so wird es entweder der Socket-

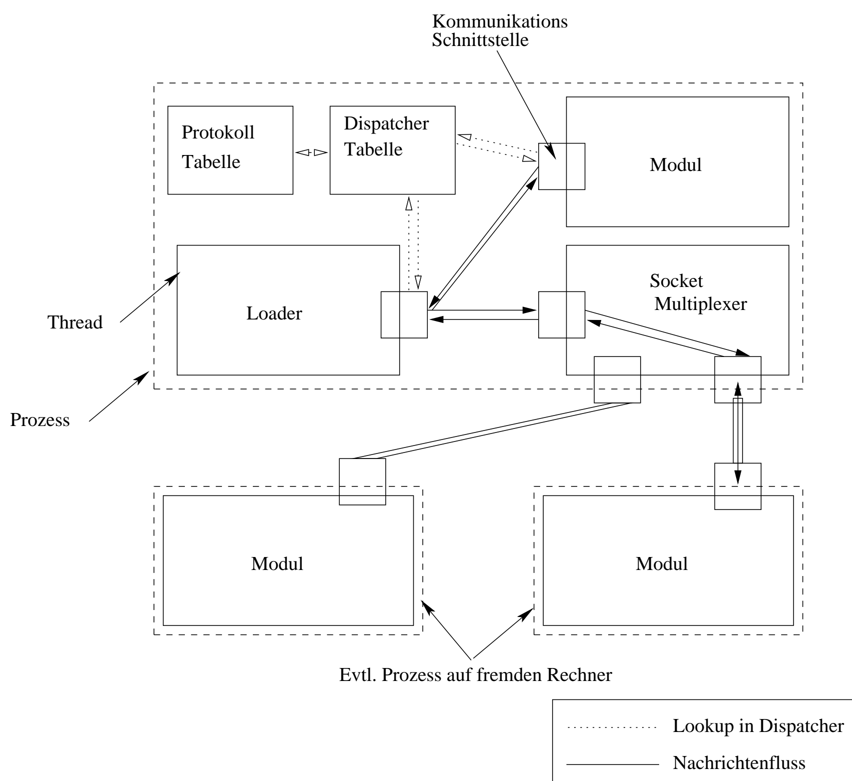

Abbildung 4.3: Struktur des Nachrichten Flusses

Schnittstelle, oder an ein Kommunikationsmodul[5] übergeben. Diese fungieren als Proxies für die entfernten Module und senden die Nachrichten zum entfernten Framework-Knoten. Dort wird diese Nachricht an das lokale Funktionsmodul übergeben.
Innerhalb eines Knoten werden Nachrichten über einen gemeinsamen Speicher ausgetauscht. Jedes Modul kann in einem gemeinsamen Adressraum oder in einem getrennten Adressraum laufen. Bei getrennten Adressräumen oder abgesetzten Modulen kommuniziert ein Modul mit den zentralen Komponenten über eine Socket-Schnittstelle. Innerhalb des zentralen Knoten-Programms werden die Nachrichten, die per Socket empfangen wurden in einem Socket-Multiplexer in Shared-Memory-Aufrufe umgewandelt, die die zentrale Wegewahltabelle abfragen. Danach wird die Nachricht über den gemeinsamen Speicher ausgetauscht.
Die zentrale Wegewahltabelle wird durch den *Loader* aufgebaut und nach Beendigung eines Moduls werden die Einträge wieder gelöscht. In der Wegewahltabelle sind die Modul-ID und eine Referenz auf das Modul hinterlegt. Die Modul-ID besitzt nur lokale Gültigkeit und wird beim Start des Moduls festgelegt. Die Modul-ID ist Teil der Adresse des Moduls. Die Adressierung wird in Kapitel 4.5 beschrieben. In jedem Modul sind Informationen für das lokale

[5]Bei einem Kommunikationsmodul handelt es sich um ein Modul, welches die Konnektivität zwischen verschiedenen Knoten herstellt. Dies kann z. B. ein Overlay, ein P2P Netz oder eine direkte Kommunikation per Socket sein.

Management der Module hinterlegt. Diese können ebenfalls über die Modulschnittstelle von dem Modul selber und anderen Modulen abgefragt werden. Weitere Informationen hierzu finden sich in den beiden folgenden Abschnitten.

*Dispatch(addr *to, unsigned int ModID) = local_module*

Abbildung 4.4: Dispatcher-Primitive

Dem Dispatcher die Zieladresse und die ModulID übergeben. Als Ergebnis liefert der Aufruf das lokale Objekt (Modul) zurück, zu welchem die Nachricht geschickt werden soll. Für das lokale Objekt wird die *Receive*-Funktion aufgerufen. Der Aufruf des Dispatchers ist in Abbildung 4.3 definiert.

4.4.2 Aufbau eines Moduls

Ein Modul besteht aus zwei Teilen. Zum einen aus dem funktionalen Teil, in dem die Funktionen des Moduls implementiert werden. Zum anderen besteht ein Modul aus dem Kommunikationsinterface, welches in Abschnitt 4.4.3 beschrieben wird. Diese beiden Teilkomponenten bilden ein Modul. Mit der Kommunikationsschnittstelle wird die Interkonnektivität und die Lokalisation anderer Module dem Modul verborgen. Ein Modul kommuniziert über die einheitliche Schnittstelle mit der Außenwelt und empfängt durch diese die Nachrichten, die es verarbeiten soll. Damit wird eine hohe Flexibilität der Module erzeugt.
Ein Modul läuft entweder in einem eigenen Prozess, mit eigenem Adressraum, oder es läuft als Thread innerhalb des Knoten-Hauptprogramms. Dies ist in der Abbildung 4.3 durch die gestrichelten Umrandungen verdeutlicht. Wobei ein durchgezogenerRahmen um ein Modul bedeutet, dass dieser in einem eigenen Thread oder Prozess läuft. Die Kommunikationsschnittstelle kann je nach Startform des Moduls mit den Kernkomponenten des Frameworks kommunizieren. Dies sind entweder die Kommunikation per Socket oder per Warteschlange. Beim Start eines Moduls wird die Kommunikationsschnittstelle geladen. Danach werden die Informationen über das lokale Modul an die Kommunikationsschnittstelle übergeben. Nun registriert das Modul seine Nachrichtentypen, die es implementiert, bei dem Zentralen Modulmanagement. Damit ist klar, welche Nachrichten das Modul verarbeiten kann und somit ihm zur Verarbeitung übergeben werden können. Diese Nachrichtentypen werden global veröffentlicht.
Alle Module sehen nach Außen hin gleich aus. Sie unterscheiden sich nur darin, dass sie unterschiedliche Nachrichtentypen implementieren und evtl. unterschiedlich mit einem Knoten kommunizieren. Ein Funktionsmodul teilt dem

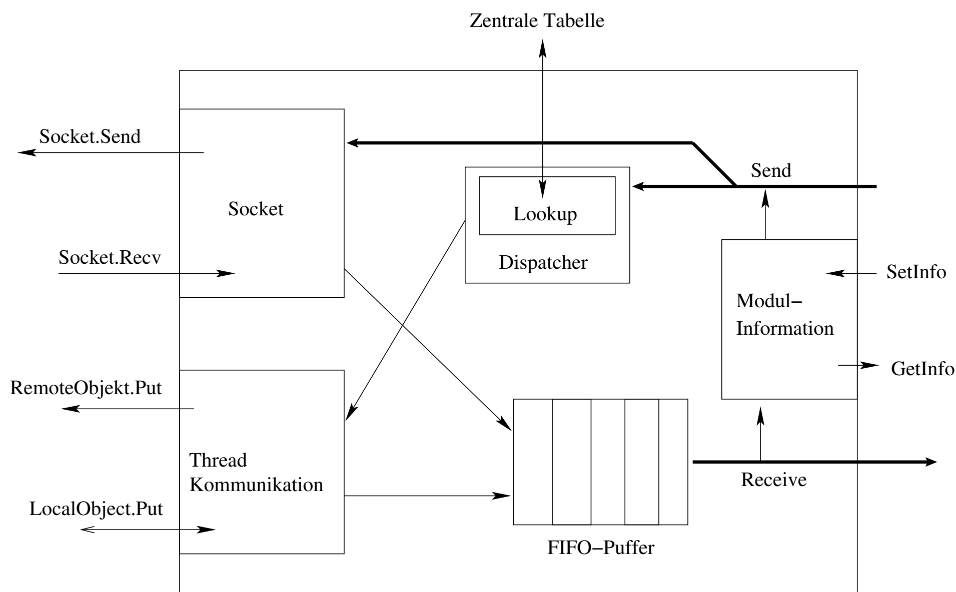

Abbildung 4.5: Kommunikationsinterface eines Moduls

Frameworkknoten und den Verwaltungsmodulen Informationen über sich mit. Dies bedeutet, dass ein Funktionsmodul sich automatisch beim Service Discovery registriert und sich sicher sein kann, dass die Verwaltungsmodule gestartet sind und diese Nachrichten empfangen können. Zu diesen Modulen zählt der Loader, das Service Discovery Modul und der Logger. Falls die beiden letztgenannten Module nicht gestartet sein sollten, bekommt dies der Loader mit und kann diese wieder neu starten.

4.4.3 Modulschnittstelle

Ein Modul kommuniziert mit den anderen Modulen ausschließlich über die Kommunikationsschnittstelle. Es werden zum Senden und Empfangen zwei Methoden genutzt. Dies sind *Send* und *Receive*. Das vollständige Interface für allgemeine Module befindet sich in Kapitel 5.3. Es wird die lokale Zieladresse und der Nachrichtentyp in der Funktion *Send* angegeben, da die Wegewahl und die Lokalisation des Zielmoduls transparent durch das Framework erfolgt. Die Zieladresse kann jedoch per Konfiguration beim Start eines Moduls über Capabilities angegeben werden. Die Zieladresse kann aber auch dynamisch festgelegt werden, indem keine oder nur wenige Capabilities angegeben werden. Die Zieladresse an die die Daten gesendet werden, werden dann erst beim Start des Moduls durch den Loader festgelegt. Innerhalb des Modulinterfaces wird zuerst entschieden, ob das Modul per Socket mit dem Framework kommuniziert oder ob es direkt per Warteschlange auf die Module zugreifen kann. Findet die Kommunikation über Warteschlangen statt, so wird nun der Dispatcher aufgerufen, der die Wegewahl vornimmt und als Ergebnis an das lokale Modul zurückgibt. Dies ist entweder ein Funktionsmodul oder Kommunikationsmodul, welches die Nachricht transparent auf einen anderen Framework-Knoten transportiert. Danach wird die Nachricht an das entfernte Modul gesendet, indem

die Receive-Funktion des Moduls aufgerufen wird. Diese stellt die Nachricht in eine Warteschlange. Ist das Modul über einen Socket mit der zentralen Knotenverwaltung verbunden, wird die Nachricht durch den Socket getunnelt und auf der anderen Seite im Socket-Multiplexer entpackt. Es findet bei der Anbindung eines Moduls per Socket kein Marshaling der Daten statt. Diese müssen in einer Form vorliegen, so dass diese ohne Probleme über einen Socket gesendet werden können. Bei dieser Anbindungsvariante wird das Nachrichten-Dispatching im Socket-Multiplexer durchgeführt und nicht in der Kommunikationsschnittstelle selbst. Der Socket-Multiplexer ist in Abbildung 4.6 dargestellt. Bei der Anbindung eines Moduls über einen Socket an den Framework-Knoten, handelt die Modulschnittstelle nur als Umsetzer zwischen den Kommunikationsformen. Werden die Nachrichten per Warteschlange weitergereicht, so werden nur die Zeiger auf die Datenstrukturen weitergegeben. Die Daten werden nicht kopiert. Dies geschieht nur, wenn ein Wechsel der Kommunikationsform (z.B. Nutzung eines Sockets) stattfindet. Damit werden die Daten nur kopiert, wenn es notwendig ist. Für die Löschung der Daten ist das Modul zuständig, welches die Datenstruktur empfangen hat.
Des Weiteren enthält die Kommunikationsschnittstelle einen FIFO-Puffer. Damit ist es möglich, Nachrichten zwischenzuspeichern, bis das Modul die Nachricht verarbeiten kann. Der Puffer hat keine feste Größe, sondern kann dynamisch wachsen und schrumpfen. Die maximale Größe des Puffers ist nur durch die Beschränkung der Ressourcen auf einem Knoten begrenzt. Dieser Puffer ist bei dem Empfang einer Nachricht über einen Socket nicht notwendig, aber er bleibt erhalten. Da ein Socket wie ein Puffer behandelt werden kann, ist im Grunde kein zusätzlicher Puffer notwendig. Jedoch wird der Datenempfang über einen Socket in dem Puffer gespeichert. Damit wird dem funktionalen Teil des Moduls erlaubt über einen einheitlichen Aufruf auf die Daten zuzugreifen und die Daten zu verarbeiten. Der Puffer hat zusätzlich Mechanismen für den exklusiven Zugriff auf die Puffer-Einträge implementiert, da jedes Modul als Thread läuft und die Entkoppelung der Threads untereinander durch Warteschlangen vollzogen wird.
Die Modulschnittstelle enthält Funktionen zur Identifizierung des Moduls. Diese Funktionen lauschen auf festgelegte Nachrichtentypen. Diese sind für alle Module gleich und beschreiben die Eigenschaften eines Moduls. Diese Eigenschaften werden beim Start des Moduls festgelegt und können zusätzlich zur Laufzeit des Moduls geändert werden. Ebenso sind diese Informationen global abrufbar. Das Interface für den Zugriff auf die Modulinformation/Modulvariablen wird in Kapitel 5.6 definiert.

Zusätzlich zu den Variablen, die in Tabelle 4.3 definiert werden, werden für ein funktionales Modul folgende Variablen definiert. Diese Variablen sind in Tabelle 4.5 definiert.

4.4.4 Socket Multiplexer

Der Socket Multiplexer (SMux) arbeitet als Umsetzer zwischen der Socket Kommunikation und der Warteschlangen Kommunikation bei der Anbindung

Tabelle 4.1: Typdefinitonen für die Modulinformationen

Bezeichner	Wert	C-Typ	Größe (Oktett)
VAR_NOTDEF	0		
VAR_INT	1	unsigned int	4
VAR_SINT	2	short int	2
VAR_BYTE	3	char[1]	1
VAR_STRING	4	char[*]	nullterminiert
VAR_STRUCT	5	struct{unsigned int size, void *ptr}	8
VAR_USER	6	void*	nicht spez.

Tabelle 4.2: Zugriffsrechte auf eine Variable

Bezeichner	Werte	Beschreibung
ACCESS_NOTDEF	0	Dieser Wert wird angegeben, wenn keine Callback Funktion angegeben wird
ACCESS_READ	1	Der Zugriff auf diese Variable ist nur lesend erlaubt
ACCESS_READWRITE	2	Lesender und schreibender Zugriff auf die Variable

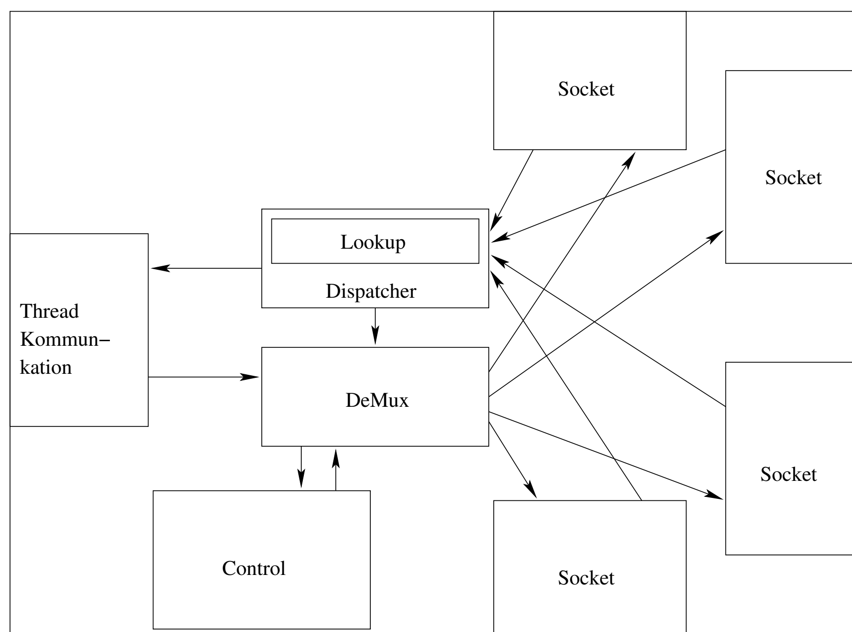

Abbildung 4.6: Socket Multiplexer und Wrapper

Tabelle 4.3: Vordefinierte Variablen der Modulinformation

Variablenname	Typ	Beschreibung
modulename	VAR_STRING	Name des Moduls
moduleID	VAR_INT	ModulID, die dem Modul beim Start durch den Loader mitgeteilt wird
starttime	VAR_USER	Uhrzeit zu der das Modul gestartet wurde. Nimmt als Struktur das C-Struct *timeval* auf.
modulestatus	VAR_STRING	Zustand, in dem sich ein Modul befindet. Die Zustände sind in Abschnitt 4.8 erklärt.
description	VAR_STRING	Beschreibung des Moduls.
startmethod	VAR_INT	Startmethode des Modules (thread, fork oder binary)
filename	VAR_STRING	Dateiname der Modullibrary
fr_host	VAR_STRING	Hostname oder IP-Adresse des SMux
fr_port	VAR_SINT	Portnummer des Smux
fr_protocol	VAR_STRING	Protokoll über den der SMux erreichbar ist. (TCP oder UDP)

von Modulen, die in einem anderen Prozess ausgeführt oder auf einem anderen Rechner laufen. In Abbildung 4.6 wird der Aufbau eines Socket Multiplexers gezeigt. Hinzu kommt, dass der Socket Multiplexer für mehrere Module gleichzeitig die Umsetzung vornehmen kann. Dazu ist es notwendig, dass er die Modul-IDs in Socket-Adressen umsetzen kann und ein Demultiplexing der empfangenen Nachrichten vornehmen kann. Diese Informationen über die Zugehörigkeit der Socketadressen zu den Modul-IDs speichert der Socket Multiplexer in einer Tabelle. Diese ist mit dem Namen *Control* in Bild 4.6 bezeichnet. Damit steuert er das Multiplexing und Demultiplexing der Nachrichten. Bevor die Nachrichten an die Kommunikationsschnittstelle weitergereicht werden, wird überprüft, ob die Nachricht für ein Modul bestimmt ist, welches beim SMux registriert ist. Die Nachrichten, die er über die Sockets von den einzelnen Modulen empfängt, gibt er danach an seine Kommunikationsschnittstelle weiter. Dort findet die Zielbestimmung durch den Dispatcher statt. Selber muss der SMux seine interne Tabelle aktualisieren, wenn ein neues Modul hinzukommt oder wenn ein Modul beendet wird. Dann werden die entsprechenden Abbildungseinträge hinzugefügt oder gelöscht. Läuft einer der Sockets in einen Timeout, so kann von dem SMux aus kein Neuaufbau der Verbindung erfolgen. Dies muss immer von dem Modul ausgehen. Die Nachrichten gehen in diesem Fall verloren. Es findet keine Zwischenspeicherung der Nachrichten in dem SMux statt. Der SMux kann auch als Kommunikationsschnittstelle zwischen zwei Frameworkknoten agieren. Dazu wird die Funktionalität des SMux

um einige Nachrichten erweitert. Diese Nachrichten sorgen dafür, dass beim Dispatcher das Protokoll *socket* registriert wird. Ebenso wird der SMux um die Umsetzung der Adressen von *local* nach *socket* erweitert. Ein zusätzlicher Frameworkknoten wird dann wie ein weiteres Modul behandelt. Jedoch hat dieser Socket zusätzliche Funktionalitäten. Die erweiterten Nachrichten des SMux werden zum Teil nur in der Kommunikation zwischen zwei SMux verwendet, die für die Kontrolle der Verbindung notwendig sind. Wenn der SMux als Kommunikationsschnittstelle zu einem anderen Frameworkknoten agiert, wird dies im Service Discovery bekannt gemacht.

4.4.5 Loader

Der Loader ist neben dem Dispatcher die wichtigste Komponente eines Framework-Knotens. Dieser verrichtet viele Aufgaben, die das Starten von Modulen und das Weiterleiten[6] von Nachrichten betreffen. Seine Aufgabe ist es *Start-Requests* entgegenzunehmen und daraufhin einzelne Module und komplette Konfigurationen zu starten. Auf Grund der vielen Möglichkeiten, die eine Konfiguration offen lässt und die beim Starten eines Moduls beachtet werden müssen, handelt es sich bei dem Loader um ein sehr komplexes Modul. Er kommuniziert mit dem Service Discovery um Module zu lokalisieren. Das Service Discovery wird benötigt, wenn in einer Konfiguration keine Angaben über die Lokation eines Moduls gemacht werden und ein Knoten, auf dem ein Modul ausgeführt werden soll, gesucht werden muss. Die modulspezifischen Variablen werden ebenfalls gesetzt. Diese können aber auch von dem Modul selber, aus einer eigenen Konfigurationsdatei, geladen werden und in der Modulinformation gespeichert werden.
Weiterhin kommuniziert der Loader auch mit anderen Loadern auf anderen Knoten. Dies ist nötig, wenn ein Loader eine komplette Konfiguration starten soll. Nach dem das Service Discovery durchgeführt wurde und der Knoten auf dem ein Modul ausgeführt werden soll, feststeht, wird der eigentliche Start des Moduls an den Loader auf dem entfernten Knoten delegiert. Dieser nimmt den Start-Request entgegen und startet das Modul. Konnte das Modul nicht gestartet werden, so wird dies dem Loader, der den Start-Request generiert hat, mitgeteilt. Ebenso wird der Erfolg eines Modulstarts dem Requester mitgeteilt. Der Requester kann entweder ein anderer Loader, ein Job-Scheduler oder ein Präsentationsmodul sein. Diese teilen den Erfolg bzw. Misserfolg eines Modulstarts entweder direkt einem Benutzer mit oder generieren einen Logeintrag. Zum anderen bearbeitet der Loader die Dispatchertabellen, die beim Versenden einer Nachricht zwischen zwei Modulen notwendig sind. Die Weiterleitungstabelle wird vom Loader bearbeitet. Er fügt neue Einträge hinzu und löscht diese auch wieder. Hinzu kommt die Kommunikationsprotokoll-Tabelle. Diese wird ebenfalls vom Loader verwaltet. Für jedes Kommunikationsprotokoll wird in dieser Tabelle ein Eintrag geführt. Obwohl die Tabellen zum Dispatcher gehören und dessen Arbeit steuern, werden diese vom Loader verwaltet und bearbeitet. Des Weiteren teilt der Loader den Modulen mit, mit welchen anderen Modulen diese kommunizieren. Diese speichern diese Einträge in ihren

[6] Er ändert die Weiterleitungstabellen des Dispatchers.

Modulen und tragen die Adressen beim Versenden einer Nachricht als Zieladresse ein. Es werden auch Ready-Meldungen an die Module weitergeleitet. Beim Start eines Moduls teilt der Loader dem Modul dessen Modul-ID mit. Ebenso fängt der Loader Nachrichten ab, die nicht zu gestellt werden können und löst daraufhin den Start eines Ersatzmoduls aus. Dazu registriert sich der Loader zusätzlich mit der Modul-ID 0 und nimmt dort alle Nachrichten entgegen, die vom Dispatcher nicht aufgelöst werden können. Im Dispatcher ist per Konvention festgelegt, dass Nachrichten an Modul-IDs, die nicht existieren an das Modul mit der ID 0 weitergeleitet werden.
Die benötigten Tabellen und verwendeten Algorithmen werden im Kapitel 5 vorgestellt. Nachrichten, auf die der Loader reagiert, werden in Anhang B.2.1 definiert.

4.4.6 Job-Scheduler

Der Job-Scheduler ist ein optionales Modul, mit dem zeitliche Messungen und Analysen gestartet werden können. Dieser nimmt von einer Managementumgebung den Befehl zum Starten einer Messung und Analyse für eine bestimmte Uhrzeit entgegen. Diese Anfrage speichert der Scheduler und führt zur gegebenen Zeit diesen Befehl aus. Der Job-Scheduler nimmt komplette Start-Requests für komplette Konfigurationen entgegen. Diese Start-Requests haben das gleiche Aussehen wie ein normaler Start-Request an einen Loader. Hinzu kommt lediglich die Uhrzeit zu der dieser Request ausgeführt werden soll. Die Zeitangabe wird in der Unix-Timestamp gespeichert. Des Weiteren kommen noch Angaben hinzu, ob der Plan einmalig ausgeführt werden soll oder täglich/wöchentlich.
Neben dem Hinzufügen von Jobs, können auch Jobs wieder gelöscht werden und eine Liste der wartenden Jobs und momentan laufenden Jobs zurückgegeben werden.

4.4.7 Logger

Mit Hilfe eines Loggers ist es möglich die Funktionsweise eines Knotens und der einzelnen Module besser zu überwachen. Diese können Nachrichten an den Logger schicken. Dieser speichert diese Nachrichten dann. Die Speicherung der Einträge kann auf verschiedene Arten geschehen. Zum einen kann der Logger einfach nur das Interface des Linux-Syslog implementieren und die ankommenden Nachrichten dem Syslog übergeben. Zum anderen kann der Logger auch in einem eigenen Format die Einträge speichern. Er kann diese Einträge auch in einer verteilten Hashtabelle speichern. In Kapitel 3.7 werden Formen diskutiert, wie die Daten gespeichert werden können. Zur Speicherung eines Loggereintrages in einer verteilten Hashtabellen wird ein hierarchisches Konzept genutzt. Das oberste Schlüssel-Wert Paar besteht aus Schlüssel = Logger und der Wert dieses Eintrags repräsentiert eine Liste von Schlüsseln der Knoten des Frameworks, die einen Logeintrag gespeichert haben. Diese Liste der Knotennamen ist je Eintrag wieder ein Schlüssel, der auf eine Liste von Einträgen zeigt, die die Loggereinträge enthält. Die Hierarchie, wie die Daten gespeichert

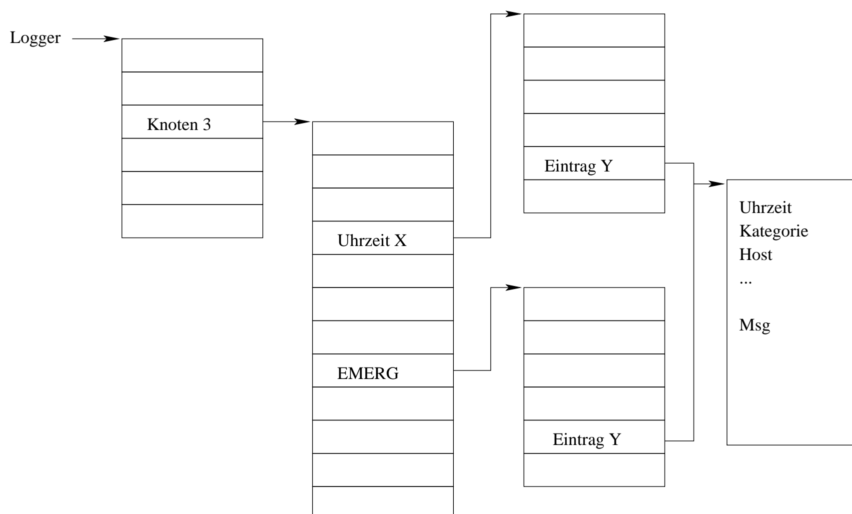

Abbildung 4.7: Speicherstruktur eines Loggereintrages in einer DHT

werden sollen, ist in Abbildung 4.7 dargestellt. Damit ist es möglich, dass auf verschiedenen Knoten ein Logger installiert ist. Die Einträge können damit auch von verschiedenen Knoten aus abgerufen werden und in einem Präsentationsmodul angezeigt werden. Auch wenn die Einträge des Loggers in einer verteilten Hashtabelle gespeichert werden, sollten sie ein ähnliches Format wie das Syslog unter Linux haben. Ebenso sollte die Einteilung der Nachrichten nach den gleichen oder ähnlichen Kategorien erfolgen. Dies sind die folgenden Kategorien: *DEBUG, INFO, NOTICE, WARNING, ERROR, CRIT, ALERT und EMERG*. Diese Kategorien sind in der Nachricht an den Logger angegeben. Hinzu kommt der eigentliche Text, der im Logger gespeichert werden soll. Der Logger fügt jedem Eintrag die Uhrzeit in UTC hinzu. Ebenfalls wird der Knotenname hinzugefügt. Dieser wird aus der Absender-Adresse eines Moduls gewonnen. Wenn die Nachricht von einem lokalen Modul kommt, wird die Adresse aus dem lokalen Management des Knotens genommen.

4.5 Adressierung

In diesem Abschnitt soll geklärt werden, wie eine Adressierung aussehen kann, mit der es möglich ist jedes Modul innerhalb des Frameworks anzusprechen. Dazu muss im ersten Schritt eine Adressierungsform gewählt werden. Anschließend wird in einem zweiten Schritt geklärt, welche Teile einer Adresse ein Modul angeben muss, damit eine Nachricht versendet werden kann und welche Teile der Adresse durch den Dispatcher und die Kommunikationsmodule aufgelöst und vervollständigt werden. Dazu ist eine Anforderungsanalyse der einzelnen Module eines Framework Knotens erforderlich, da es auf einem Knoten zwei unterschiedliche Modul-Klassen gibt. Dies sind zum einen die Verwaltungs- und Managementmodule und zum anderen die Funktionsmodule. Jedoch soll-

ten beide Modul-Klassen das gleiche Interface haben und keine Unterschiede bestehen.

4.5.1 Adressierungsarten

Es muss bei der Adressierung zwischen einer *internen Adressierung* und einer *externen Adressierung* unterschieden werden. Die *externen Adressierung* beinhaltet die Adressierungsform, wie sie in der Konfiguration und im Management verwendet wird. Diese sollte für einen menschlichen Eingriff lesbar und nutzbar sein. Ebenso ist es für einen Benutzer unerheblich wie ein laufendes Modul auf einem Knoten angesprochen wird, sondern nur auf welchem Knoten ein Modul läuft. Bei der *internen Adressierung* müssen andere Parameter angegeben werden, damit ein Modul angesprochen werden kann. Zu diesen Parametern gehört die Modul-ID.
Folglich muss eine Umsetzung zwischen den intern verwendeten Adressen und den extern verwendeten Adressen erfolgen. Diese muss in beide Richtungen möglich sein.

4.5.2 Adress-Layout

Für die beiden Adressierungsarten aus Abschnitt 4.5.1 sind nun geeignete Darstellungsformen zu definieren. Die *externe Adressierung* wird im Kapitel "Konfiguration und Management" (Kap. 4.9) beschrieben. Dort wird die Adressierung eines Moduls durch Capabilities und durch Abhängigkeiten beschrieben. Diese Angaben spezifizieren die Lokation eines Moduls. Wird die Lokation des Moduls dort nicht angegeben, muss sie durch das Service Discovery gefunden werden. Für die *interne Adressierung* wird eine erweiterte Darstellung benötigt. Diese enthält ebenfalls den Knoten und die Angabe des Moduls, jedoch in einer anderen Darstellungsform. Die interne Adresse hat eine Ähnlichkeit mit einer URL:

Protokoll://Knoten/Modul

Jedoch werden die einzelnen Bestandteile in einer anderen Form dargestellt. Der Teil *Protokoll* gibt an, über welches Kommunikationsmodul mit einem Knoten kommuniziert wird. Ein Modul, welches ein Protokoll implementiert, darf pro Knoten nur einmal gestartet werden. Die Angabe des Bestandteils *Knoten* gibt dann den Knoten in der benötigten Form des Kommunikationsmoduls an. Handelt es sich bei der Kommunikationsform werden der Nachrichtentyp sowie die Quell- undz. B. um ein Chord-Netz, so wird *Protokoll* durch *Chord* ersetzt und bei der folgenden Knotenangabe handelt es sich um den Hash-Wert des Knotens z.B. *AC5935AFB2BB4...*. Bei der Angabe des Modul handelt es sich um eine lokale Angabe der Modul-ID. Diese Modul-ID kann z.B: durch die Prozess-ID gebildet werden, da jedes Modul als eigenständiger Thread oder Prozess ausgeführt wird. Die Modul-ID wird vom Loader bereitgestellt und beim Start mitgeteilt. Der Loader erzeugt die Modul-ID aus einem Zähler, der pro gestartetem Modul erhöht wird. Damit die Adressierung von

Tabelle 4.4: Festgelegte Modul-IDs

Modulename	Modul-ID
Loader	0, 1
SMux	2
Logger	3
Service Discovery	4
Job-Scheduler	5
Reserviert	6, . . . , 10
Funktionsmodule	11, . . .

Modulen zur Verwaltung erleichtert wird und diese nicht kompliziert aufgelöst werden müssen, werden in Tabelle 4.4 feste Modul-IDs für diese Module vergeben. Die Modul-IDs für Funktionsmodule beginnen mit der Modul-ID 11. Vollständig sieht die Adresse folgend aus:

chord://AC5935AFB2BB44F529CBA4DDE354/23853

Damit Adressierungen an lokale Module nicht fälschlicherweise an das angegebene Kommunikationsmodul übergeben werden, muss vorher geprüft werden, ob es sich bei der Knotenangabe um den lokalen Knoten handelt oder nicht. Bei einer Kommunikation zwischen lokalen Modulen handelt es sich bei der Absenderadresse immer um das Protokoll *local*. Wird eine Nachricht an ein entferntes Modul versendet, wird in der Absenderadresse von Protokoll *local* auf das verwendete Kommunikationsprotokoll umgesetzt, wenn die Nachricht in einem Kommunikationsmodul angekommen ist. Dies wird von dem entsprechenden Kommunikationsmodul übernommen. Es findet keine weitere Adressumsetzung statt. Dann ist die Modul-ID aufzulösen. Jedes implementierte Protokoll muss einen eindeutigen Namen wählen. Ebenso ist es jedem Protokoll überlassen eine eigene Form für die Bezeichnung eines Knotens einzuführen. Es findet kein Tunneln von Nachrichten und Kommunikationsprotokollen statt. Alle Absender und Empfänger müssen mit dem angegeben Kommunikationsprotokoll erreichbar sein. Ebenso findet keine Adressumsetzung zwischen zwei Kommunikationsprotokollen statt.[7] Zu Beginn existiert nur das Protokoll *local*. Bei diesem ist auch nur die Angabe eines Knotennamens möglich. Dieser Name lautet *localhost*, gefolgt von einem Slash und der Modul-ID. Bei der Implementierung werden die Adressen innerhalb einer Struktur dargestellt. Es existieren aber Funktionen, die die Darstellungen transformieren können.

4.5.3 Modul-Anforderungen

Auf Seiten der Module bestehen unterschiedliche Anforderungen, wie Nachrichten an andere Module adressiert werden. Dies lässt sich weiter in die Unterscheidung nach Adressierung von Modulen bei einer Anfrage und bei einer

[7] Die Umsetzung von Protokollen und das Tunneln der Protokolle kann Gegenstand späterer Betrachtungen sein. Dies geht über den Rahmen der Arbeit hinaus.

Antwort auf eine Anfrage differenzieren. Dies hat zur Folge, ob und welche Adressinformationen angegeben werden müssen, wenn eine Nachricht gesendet werden soll. Des Weiteren besteht die Anforderung an ein Modul, dass es so wenig wie möglich von der Lokation des Moduls wissen sollte. Damit geht einher, dass beim Versenden einer Nachricht so wenig Adressinformationen wie möglich angegeben werden sollen. Verbunden damit ist zu entscheiden, ob Absenderinformationen notwendig sind oder nicht. Oder wird einem Modul beim Startvorgang mitgeteilt, an welches Modul es seine Nachrichten schicken muss? Ebenso ist dann zu klären: Wer und wo findet die komplette Adressauflösung und Adressierung statt? Dies wird im folgenden Abschnitt 4.5.4 beschrieben. Die Frage nach der Angabe von Adressinformationen und wie viel ist für jedes Modul eines Framework Knoten zu stellen.

Loader: Der Loader benötigt zur Ausführung seiner Aufgaben die direkte Adressierung eines Loaders auf einem entfernten Knoten. Ebenso muss ein Loader wissen, von wem er eine Nachricht bekommen hat, damit er eine Nachricht über Erfolg oder Misserfolg eines Ladevorgangs melden kann. Damit folgt, dass der Loader die Angabe von Absender- und Zieladresse benötigt.

Service Discovery: Das Service Discovery benötigt eine Angabe der Absenderadresse, da das Service Discovery seine Ergebnisse an das Modul zurücksenden muss, von dem die Anfrage kam. Folglich muss auch hier die Absender- und Zieladresse beim Empfang bzw. Senden von Nachrichten erfolgen.

Funktionsmodul: Ein Funktionsmodul kann in zwei verschiedenen Modi arbeiten. Zum einen im *Push-Modus*, dabei steht per Konfiguration fest, an welches Modul Nachrichten gesendet werden. Dazu werden die Zielmodule für definierte Nachrichtentypen angegeben und diese werden automatisch genutzt, wenn eine solche Nachricht versendet wird. Damit muss in diesem Modus keine Zieladresse beim Versenden angegeben werden. Eine Angabe der Absenderadresse entfällt hier ebenfalls. Beim *Poll-Modus* muss hingegen eine Absenderadresse angegeben werden, da die Anfragen von verschiedenen Modulen kommen können und somit das Ziel noch nicht feststeht. Beim Versenden der Nachricht muss daher ebenfalls die Zieladresse angegeben werden.

Konfiguration u. Management: Das Management muss einzelne Module adressieren können, wenn es Informationen über die Module abfragen will. Ebenso benötigt es die Adresse des Moduls, von dem es eine Aufforderung zum Übermitteln von Management-Informationen erhalten hat. Daraus folgt, dieses Modul benötigt die Angabe von Absender- und Zieladresse beim Empfang und beim Senden von Nachrichten.

Logger: Der Logger kann eine Angabe der Absenderadressen enthalten. Wenn diese enthalten ist, kann die Information für den Ursprung einer Lognach-

richt aus der Absenderadresse gewonnen werden. Handelt es sich bei einer Adressangabe um das Protokoll local, so muss der Knotenname aus dem Management gewonnen werden. Ist die Absenderadresse nicht angegeben, muss die Information aus der Loggernachricht selbst extrahiert werden. Daraus folgt, es muss keine Absenderadresse angegeben sein, vielmehr muss der Knotenname einer Lognachricht in dieser selbst angegeben werden.

Socket Multiplexer: Der SMux muss zum Erkennen, ob eine Nachricht für ein, bei ihm, registriertes Modul bestimmt ist, die Zieladresse der Nachricht kennen. Die Absenderadresse benötigt dieser nur, wenn er als Kommunikationsmodul zu einem anderen Frameworkknoten agiert.

Schließlich stellt sich noch die Frage nach der Kenntnis eines Moduls über seine globale Adresse: Ein Modul kennt seine globalen Adressen nur implizit durch den Empfang von Nachrichten, die als Zieladresse eine globale Adresse des Moduls enthalten. Die lokale Adresse kennt das Modul, da diese sich einfach aus der Modul-ID und der Angabe *local://localhost/* konstruieren lässt. Die globalen Adressen sind unbekannt, da verschiedene Protokolle zur Kommunikation mit anderen Frameworkknoten existieren können. Somit kann ein Modul mehrere globale Adressen haben. Diese unterscheiden sich aber nur in der Angabe des Protokolls und der Knotenadresse.

4.5.4 Adressauflösung

Nun muss noch die Frage nach dem Ort der Adressauflösung geklärt werden. Unabhängig von der Angabe einer Zieladresse in einer Nachricht ist zu klären, ob die komplette Adressauflösung vom lokalen Dispatcher gelöst werden soll oder ob dieser nur einen Teil der Adresse auflöst. Der Dispatcher muss von der Adresse genau so viel auflösen, bis er die Nachricht an ein anderes Modul auf dem Knoten weiterleiten kann. Dies bedeutet, wenn das angegebene Protokoll nicht *local* ist, muss er in einer Tabelle nachschauen, ob ein Modul bekannt ist, welches das angegebene Protokoll implementiert.
Neben der Auflösung des Transport-Protokolls kann der Dispatcher die lokalen Modul-IDs auflösen und anschließend zustellen. Zusammenfassend werden folgende Teile einer internen Adresse von folgenden Komponenten und Modulen aufgelöst:

Protokoll wird im Dispatcher aufgelöst

Knoten wird im entsprechenden Kommunikationsmodul aufgelöst

Modul-ID wird im Dispatcher aufgelöst (wenn das verwendete Protokoll *local* ist)

local://localhost/XXXX wird im Dispatcher aufgelöst. Dies stellt die Langform einer Modul-ID dar.

Die Adressauflösung innerhalb des *Dispatchers* wird von einer Adresse in dem oben definierten Format in eine Objektadresse für das lokale Zielmodul umgewandelt.

4.5.5 Adressangaben

Dieser Abschnitt soll klären, ob Adressen bei einer Nachricht angegeben werden müssen, wenn diese Nachricht versendet werden soll. Zum anderen muss klar sein, welche Adressen beim Empfang einer Nachricht innerhalb eines Moduls angegeben sein müssen, damit diese verarbeitet werden können.
Für den Empfang einer Nachricht benötigt man prinzipiell keine Angabe einer Absenderadresse. Dies erfolgt aber unter der Annahme, dass die Nachrichten keine Antworten hervorrufen und somit immer nur das Versenden von Nachrichten erfolgt. Jedoch ist es in diesem Framework erforderlich, dass auf Nachrichten (Requests) Antwortnachrichten generiert werden. Somit folgt daraus, dass beim Empfang einer Nachricht ein Absender angegeben sein muss, da es sonst schwierig wäre eine Antwort auf einen Request an die Quelladresse zu senden. Der Dispatcher müsste aus diesem Grund für jede Nachricht einen Zustand merken, von wo die Nachricht kommt und wohin diese gesendet wurde. Da aber nicht auf jede Nachricht eine Antwortnachricht erfolgt, müsste dies ein Soft-State[8] Zustand sein. Bei jedem Dispatcheraufruf müsste dann geschaut werden, ob ein solcher Eintrag existiert und ebenso müsste diese Liste mit Einträgen auf ungültige Einträge hin überprüft werden. Ebenso ist für den Dispatcher nicht ersichtlich, ob es sich bei einer Nachricht um eine Antwort-Nachricht oder eine eingenständige Nachricht handelt, wenn keine eindeutige Kennzeichnung einer Nachricht erfolgt. Daraus folgt, dass es dem Modul überlassen wird, einen Zustand für eine Nachricht aufzubauen, damit eine Antwort an das entfernte Modul gesendet werden kann. Eine genauere Analyse zur Angabe von Absender- und Zieladresse findet sich in Abschnitt 4.5.3. Der Aufbau der Empfangsprimitive ist in Kapitel 5.3 gezeigt.
Es folgt, dass beim Versenden einer Nachricht aus einem Modul heraus die Zieladresse angegeben werden muss, an die die Nachricht gesendet werden soll. Die Absenderadresse wird automatisch durch die Sendeprimitive eingetragen und vervollständigt. Es handelt sich bei dieser Adresse um die lokale Adresse (*local://localhost/ModulID*). Neben der Angabe einer Zieladresse in dem Nachrichtenkopf wird der Nachrichtentyp sowie die Größe der Nachricht und der Zeiger auf diese angegeben.
Damit ist die Sendeprimitive definiert. Ihr Format ist in Kapitel 5.3 beschrieben. Aus diesen Betrachtungen folgt, dass beim Empfang einer Nachricht die Absender- und Zieladresse angegeben ist. Beim Versenden einer Nachricht wird nur die Zieladresse angegeben, die Absenderadresse wird innerhalb der Sendeprimitive aus den Modulinformationen eines Moduls aufgebaut. Für den Dispatcher ist es unerheblich, ob es sich bei einer Nachricht um eine Antwort handelt oder nicht. Der Dispatcher ist zustandslos. Das Merken eines Zustandes bei einer Requestnachricht liegt in der Verantwortung des Moduls welches die Nachricht sendet. Für die Implementierung von RPCs zwischen Modulen existiert ein RPC-Manager. Dieser wird in Kapitel 5.7 beschrieben.

[8] Soft-State bedeutet, dass der Status nur semipermanent ist. Nach dem Ablauf eines Timers erfolgt die Löschung des Zustandes. Eine Antwort-Nachricht, die nach der Löschung des Zustandes gesendet wird, kann damit evtl. nicht zugestellt werden.

4.6 Service Discovery

Zum einen müssen Module lokalisiert werden, da entweder durch die Konfiguration kein genauer Framework Knoten bestimmt wurde oder ein Modul muss durch ein neues Modul ersetzt werden, weil es abgestürzt ist. Zum anderen werden Informationen über laufende oder vorhandene Module auf Framework-Knoten benötigt um weitere Module zu starten.

4.6.1 Aufgaben

Das Service Discovery (SD) ist dazu bestimmt Module auf verschiedenen Knoten zu lokalisieren. Dies geschieht anhand von Bedingungen und Anforderungen. So kann z.B. ein Presentation Node eine Analyse anfordern, die Sensoren in bestimmten Netzen oder Autonomen Systemen paltziert hat. Dafür ist es notwendig, dass Knoten mit entsprechenden Eigenschaften lokalisiert werden können und Module auf diesen gestartet werden können. Um dies zu erreichen ist es notwendig, dass ein Service Discovery existiert. Dies impliziert, dass jeder Frameworkknoten ein Service Discovery Modul ausführt. Damit kann von jedem Knoten aus eine Suche nach einem Modul mit bestimmten Eigenschaften vorgenommen werden. Sobald ein neuer Knoten startet, fügt er Informationen zum Service Discovery hinzu, indem eine entsprechende Nachricht an das Service Discovery gesendet wird. Mit der Nutzung einer Verteilten Hashtabelle wird das Service Discovery unabhängig von einzelnen zentralen Knoten. Es werden Informationen über vorhandene Module auf dem Knoten veröffentlicht. Hinzu kommen allgemeine Informationen über lokale IP-Adressen, ein dazu gehörendes Autonomes System und weitere Informationen.
Benötigt nun ein Modul ein anderes Modul, damit es korrekt ausgeführt werden kann, so startet der *Loader* einen Lookup nach dem entsprechenden Modul. Dies kann näher spezifiziert werden, indem Eigenschaften für diese Abhängigkeit angegeben werden. Diese Eigenschaften werden der Abfrage nach der Lokation eines Moduls hinzugefügt. Das Service Discovery antwortet mit einer Liste von Knoten, die in Frage kommen. Nach der Auswahl eines geeigneten Knotens sendet das Modul einen Start-Request an den *Loader* auf diesem Knoten. Der *Loader* antwortet dem Requester, ob das Starten des Moduls erfolgreich war oder nicht. Danach informiert der Knoten, auf dem das neue Modul gestartet wurde, das Service Discovery mit der aktualisierten Liste von auf dem Knoten ausgeführten Module.
Damit das Framework skalierbar und flexibel bleibt, können von jedem Modul mehrere Instanzen gestartet werden. Diese müssen jedoch durch die Adressierung unterschieden werden, da beide Instanzen die gleichen Nachrichtentypen implementieren. Damit ist es möglich, das einzelne Analysatoren nur bestimmte Quell- oder Zieladressen verarbeiten oder neue Verbindungen werden auf solche Knoten zur Analyse verteilt, die aktuell eine niedrige Systemlast haben. Es sollen verschiedene Eigenschaften möglich sein um Module auszuwählen. Dazu zählen:

1. IP-Adresse(n) des Knotens

2. IP-Bereich(e) des Knotens
3. Autonomes System(e) des/der Knoten(s)
4. Systemlast des/der Knoten(s)
5. nächstgelegener Knoten[9]
6. bestimmte Implementierung eines Moduls
7. Eigenschaften der Analysedaten[10]

etc ...

4.6.2 Erweiterungen

Eine mögliche Erweiterung des Service Discovery besteht darin, dass es zu einem Repository für einzelne Module erweitert wird. Dazu ist es notwendig, dass auch die Heterogenität der einzelnen Systeme (Windows, Linux, Router-OS, etc) berücksichtigt werden. Soll auf einem Knoten ein neues Modul gestartet werden und es ist nicht auf dem lokalen Knoten vorhanden, stellt dieser Knoten einen Lookup an das Service Discovery, ob ein benötigtes Modul zum Download vorhanden ist. Nun stellt ein Loader einen Request an das Service Discovery und fordert es auf das Modul zu übertragen. War die Übertragung erfolgreich, versucht der Loader das neue Modul im Dateisystem zu speichern und zu starten. Gleichzeitig aktualisiert der Loader das Service Discovery um die Informationen über das neue Modul und die Liste der momentan ausgeführten Module und die lokal vorhandenen Module.

4.7 Bootstrapping

Das Bootstrapping beschreibt die möglichen Verfahren, wie einzelne Module oder sogar komplette Messungen und Analysen mit mehreren Modulen gestartet werden können. Hinzu kommt die Fragestellung nach der Granularität und Implementierung der Konfiguration der Module und Modulgruppen, sowie mögliche Parameter, die das Modul zum Starten benötigt. Als dritter Teilbereich kommt das Interaktionsverhalten der einzelnen Module untereinander hinzu. Insgesamt lassen sich folgende vier Bereiche des Bootstrappings mit ihren jeweiligen Unterbereichen definieren:

1. Granularität der Konfiguration

[9]Hier liegt die Annahme zugrunde, dass benachbarte Knoten meist mit einer hochbitratigen physikalischen Verbindung miteinander verbunden sind. Wobei hingegen weit entfernte Knoten oft über langsamere physikalische Verbindungen mit einander kommunizieren können. Für die Definition des nächstgelegenen Knotens muss eine Metrik bestimmt werden, nach der ein Knoten ausgewählt wird. Metriken für eine Wahl des nächstgelegenen Knotens können folgende sein: RTT, Hop Anzahl, Uptime des Knoten, ...

[10]Bei Paketdaten z.B. Quell-/Zieladressen, Quell-/Zielports, Protokolle

2. Interaktionsverhalten
3. Ausgangspunkt des Startvorgangs
4. Verhalten bei Exceptions und Fehlern

Diese Bereiche sollen in den folgenden Abschnitten beschrieben werden.

4.7.1 Granularität der Konfiguration

Das Framework soll flexibel gestaltet werden. Es soll möglich sein auf einem Knoten des Frameworks ein Modul oder mehrere Module zu starten (siehe auch Kapitel 4.4 und 4.2). Es stellt sich hier die Frage, wie die Module auf den Knoten konfiguriert werden und wo die Konfiguration gespeichert ist. Hinzu kommt die Granularität der Konfiguration. Diese entscheidet, ob es pro Modul eine eigene Konfigurationsdatei gibt oder ob für eine komplette Messung und Analyse eine Konfiguration existiert, die alle benötigten Konfigurationsparameter der einzelnen Module enthält. Diese wird in Kapitel 4.9.1 geklärt.
Für das Bootstrapping ist im Bezug auf die Konfiguration relevant, dass der *Loader* einen Start-Request empfängt, in dem eine Konfiguration angegeben ist. Diese Konfiguration öffnet der *Loader* und führt die einzelnen Schritte zum Starten der Module aus. Sind in der Konfiguration spezielle Parameter zur Konfiguration der einzelnen Module angegeben, so werden diese Parameter im Modul gesetzt.
Das Modul selbst lädt seine Standardparameter aus der Konfiguration für ein einzelnes Modul oder setzt diese Parameter in der Implementierung statisch.

4.7.2 Interaktionsverhalten

Ausgehend von diesem Aspekt der Granularität der Konfiguration ist zu untersuchen, wie die einzelnen Module nun tatsächlich gestartet werden und wie die Interaktion zwischen den einzelnen Modulen erfolgt. Die prinzipielle Kommunikation zwischen den einzelnen Modulen wird in Kapitel 4.4.1 beschrieben. Die Adressierung der einzelnen Module erfolgt im Abschnitt 4.5. Als prinzipielle Möglichkeiten der Interaktion zwischen den Modulen lässt sich die statische und dynamische Kommunikation erkennen.
Bei der statischen Kommunikation sind die Module fest miteinander verbunden. Im Kontext des Frameworks bedeutet dies, dass die Module per Konfiguration feststehen, an welche die Nachrichten gesendet werden müssen. Dies beinhaltet auch die Angabe von festen Lokationen, an denen das Zielmodul ausgeführt werden soll. Von dieser statischen Kommunikationsweise wird dann abgewichen, wenn ein Modul wegen eines Fehlers ausfällt. Dann wird per Service Discovery ein Ersatz-Modul gewählt, welches die Aufgabe erfüllt. Wie auf den Ausfall eines Moduls reagiert wird, wird in Abschnitt 4.7.4 beschrieben.
Dynamische Kommunikation zwischen den Modulen bedeutet, dass Module ihre Kommunikationspartner mit Hilfe des Service Discovery selbst suchen müssen. Im Kapitel 4.6 ist die dynamischen Suche nach einem Partner geschildert.

Die dynamische Kommunikation wird in der Konfiguration dadurch erreicht, dass keine Aussage über die Lokation eines Moduls gemacht wird, sondern dass das Zielmodul in der Konfiguration nur über seinen Namen angegeben wird oder nur der Nachrichtentyp. Daraufhin durchsucht das Service Discovery Modul seine Datenbank, nach passenden Modulen und schlägt diese dem Loader als mögliche Kandidaten vor. Diese komplett dynamische Suche nach einem Folgemodul kann bis hin zur festen Angabe eines Folgemoduls variieren. Z.B. ist es möglich verschiedene Module anzugeben. Aus diesen wählt der Loader das bestmögliche nach verschiedenen Kriterien aus. So kann zum einen eine Lastverteilung stattfinden, indem der Datenstrom des Moduls möglichst gleichmäßig auf die Zielmodule verteilt wird. Jedoch muss dabei darauf geachtet werden, dass zusammengehörende Messdaten nicht getrennt werden (z.B. Messdaten einer TCP-Verbindung dürfen nicht getrennt werden). Zum anderen kann mit diesem Verhalten auch eine gezielte Verteilung des Messdatenstroms auf verschiedene Folgemodule erfolgen, um ihn etwa einer speziellen Analyse zu unterziehen oder möglichst nah bei der Quelle zu analysieren.

4.7.3 Ausgangspunkt des Startvorgangs

Um eine Messung und Analyse zu starten sind verschiedene Funktionsmodule zu starten und miteinander zu verbinden. Es muss nun geklärt werden, wer das Starten von Modulen auslöst. Ebenso muss geklärt werden, was genau ausgelöst wird.
Die Frage nach dem wer oder was lässt sich einfach beantworten. Der Startvorgang für eine Messung und Analyse lässt sich durch zwei Arten auslösen. Zum einen ist dies ein Job-Scheduler, der zeitlich gesteuert Messungen und Analysen ausführt. Zum anderen kann diese Aufgabe von einem Präsentationsmodul sowie durch administrativen Eingriff von Außen erfolgen. Diese lösen einen Request zum Starten von Messungen aus und schicken eine Nachricht zum Starten einer Messung an einen *Loader*. Dieser übernimmt die Aufgabe die Module zu starten und miteinander zu verbinden.
Nun muss noch geklärt werden, was von diesem Start-Request gestartet wird. Dies kann prinzipiell jedes Modul sein, welches durch den *Start-Request* gestartet werden soll. Meist handelt es sich um eine komplette Analyse mit mehreren Modulen, die gestartet werden soll. Es kommt nur darauf an wo sich das Modul in der Kette der Verarbeitung einer Messung und Analyse befindet. Es kann sich dabei um ein Paket-Capture Modul handeln oder auch um ein reines Analyse Modul. Der Ausgangspunkt des Startvorgangs einer Messung und Analyse hängt davon ab, welcher Start-Request gestellt wird. Fordert ein Präsentationsmodul eine bestimmte Analyse an, so wird die Analyse gestartet. Jedoch benötigt diese Analyse bestimmte andere Module, die Messdaten liefern. Dies bedeutet, dass Module gestartet werden müssen, die dem Analysemodul bestimmte Messdaten liefern. Daraus folgt: Damit ein Startvorgang auch rückwärts von der Analyse aus gestartet werden kann, müssen in der Konfiguration auch die Abhängigkeiten in Rückwärtsrichtung angegeben werden. Fordert ein Präsentationsmodul das Capturen von bestimmten Paketen an, wird zuerst das Modul zum Capturen von Paketdaten gestartet. Damit dieses

seine Messdaten weiterleiten kann, müssen weitere Module gestartet werden. Bei diesen Modulen handelt es sich um Analysemodule. Welche Modultypen auf ein gegebenes Modul folgen können oder davor geschaltet werden können, geht aus der Abbildung 4.1 hervor.
Auf die Konfiguration hat dies die Auswirkungen, dass Abhängigkeiten sowohl in Vorwärtsrichtung, als auch in Rückwärtsrichtung angegeben werden. Der generelle Aufbau der Konfiguration ist davon nicht betroffen. Aus der Anforderung des Präsentationsmoduls muss auf ein Modul geschlossen werden, welches als erstes für die Messung und Analyse gestartet werden muss. Daraus leitet sich dann ab, welche Module als Folge von diesem Start ausgeführt werden müssen.

4.7.4 Verhalten bei Exceptions und Fehlern

Bei der Beschreibung des Verhaltens des Frameworks gegenüber dem Auftreten eines Fehlers oder einer Exception ist zu definieren, wie das Framework darauf reagiert.
Auf den Ausfall eines Moduls kann auf zwei Arten reagiert werden. Zum einen wird die gesamte Analyse abgebrochen, weil ein Modul in der Mess- und Analysereihe ausgefallen ist. Dann werden alle Module beendet, die zu der Messung und Analyse gehören. Zum anderen kann das Framework Nachrichten abfangen, die an ein Modul gerichtet sind, welches aber auf Grund eines Fehlers nicht mehr ausgeführt wird. Der Loader fängt alle Nachrichten an Module ab, die nicht mehr ausgeführt werden und startet die Ersatzmodule. Das Framework versucht daraufhin ein Modul zu starten, welches die abgefangenen Nachrichten verarbeiten kann. Damit dieses Modul seine Daten weiterleiten kann, benötigt dieses wieder andere Module, an die es seine Daten weiterreichen kann. Dies können entweder Module sein, die schon gestartet sind, oder die neu gestartet werden müssen. Um ein Ersatzmodul zu finden, wird das Service Discovery benötigt. Dort ist verzeichnet, wo welche Module ausgeführt werden.
Bekommen Module keine Eingangsdaten mehr, so können sich diese entweder beenden oder nicht. Beenden sich diese Module nicht, so warten diese darauf, dass sie wieder Daten erhalten, die verarbeitet werden können.
Es sollte jedoch darauf geachtet werden, dass die ausgeführten Aktionen nachvollziehbar bleiben. Dazu sollten, soweit es möglich ist. die aufgetretenen Fehler mitprotokolliert werden und im Logfile gespeichert werden. Dieser Logeintrag hat die Kategorie *EMERG*.

4.8 Zustandsmodell von Modulen

Die Ausführung eines Moduls wird in verschiedene Phasen unterteilt. Dies entspricht der Haltung eines Zustandes für jedes Modul. Dieser wird innerhalb jedes Moduls gespeichert. Er dient der besseren Strukturierung eines Moduls. Der Zustandsautomat ist in der folgendenen Abbildung 4.8 dargestellt. Mit dieser Einteilung lässt sich die Ausführung in drei Phasen einteilen, denen verschiedene Aufgaben zugeordnet werden. In der *Start-Phase* wird ein Modul

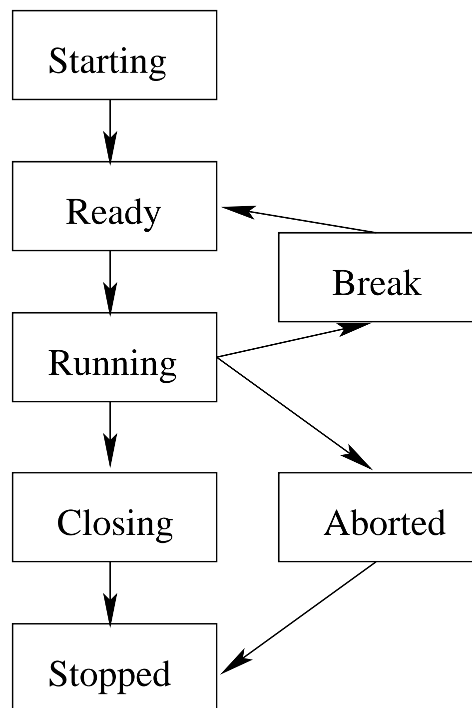

Abbildung 4.8: Zustandsautomat eines Moduls

gestartet und initialisiert alle nötigen Komponenten. Dazu zählen die Initialisierung und Registrierung seiner Funktionen im Framework sowie das Starten von abhängigen Modulen. Diese Phase wird durch die Zustände *Starting und Ready* repräsentiert. In der zweiten Phase (*Ausführungs-Phase*) übt das Modul seine Funktion aus. Es empfängt für das Modul bestimmte Nachrichten, verarbeitet diese und versendet neue Nachrichten an andere Module. Diese Phase wird durch die beiden Zustände *Running und Break* abgedeckt. In der *Stop-Phase* wird das Modul beendet oder es muss auf Grund eines Fehlers beendet werden. Hierbei teilt es dem Framework mit, dass seine Dienste nicht mehr vorhanden sind und gibt alle seine belegten Ressourcen wieder frei. Zu dieser Phase gehören die drei Zustände *Closing, Aborted und Stopped.* Zur Freigabe gehören Mitteilungen an das Service Discovery, den Dispatcher und den Loader, dass dieses Modul beendet wird.
Zu Beginn befindet sich jedes neu gestartete Modul im Zustand *Starting.* Dieser Zustand wird so lang beibehalten, bis alle Aufgaben, die zum Starten des Moduls gehören, durchgeführt worden sind. Zu diesen Aufgaben zählen die initiale Konfiguration des Moduls sowie die Herstellung der Verbindungen zu allen abhängigen Modulen. Erst wenn diese Module sich im *Ready*-Zustand befinden und eine *Ready-Meldung* an den Loader gesendet haben und dieser wiederum eine Nachricht an das eigene Modul gesendet hat, wechselt das Modul ebenfalls in den *Ready*-Zustand. Als abhängige Module zählen die Module, an die das Modul seine Daten weitergibt oder von denen es Daten empfängt[11].

[11]Es kommt auf die Startreihenfolge der Module an. Siehe dazu auch Kapitel 4.7

In einer folgenden Situation spielen die vorhergehenden Module eine Rolle. Befinden sich alle Module der Verarbeitungskette im Ready-Zustand, so wird mit der Verarbeitung begonnen. Dies bedeutet, es kann z.B. mit dem Capturen von Paketen begonnen werden. Empfängt ein Modul eine *Ready-Nachricht*, aber befindet sich schon im *Running*-Zustand, so antwortet es diesem Modul mit einer *Ready-Nachricht*. Jedes Modul wechselt in den *Running*-Zustand, so bald die ersten Daten zur Verarbeitung empfangen werden. In diesem Zustand verbleiben die Module so lang, bis sie entweder einen Request von Außen erhalten, in dem sie aufgefordert werden, beendet zu werden, oder es zu einem Fehler innerhalb eines Moduls kommt. Handelt es sich um einen Fehler, der behoben werden kann, wird der Zustand nicht gewechselt. Ist es ein kontrollierter Fehler (Exception), der abgefangen werden kann, aber nicht korrigiert werden kann, so wechselt ein Modul in den *Aborted*-Zustand. Handelt es sich um einen Fehler, der nicht behoben werden kann, aber die Kommunikationsverbindungen sind nicht von dem Fehler betroffen, so kann ebenfalls der *Aborted*-Zustand an die anderen Module (vorhergehende und nachfolgende Module) signalisiert werden. Damit wird den abhängigen Modulen die Chance gegeben alternative Module zu starten. Wie Module alternativ gestartet werden können, ist in Kapitel 4.7 geschildert. Darin ist ebenfalls geklärt, wie auf den Ausfall eines Moduls in einer Verarbeitungskette reagiert wird. Alle Module, die eine *Aborted-Nachricht* oder *Break-Nachricht* erhalten, wechseln in den *Break*-Zustand und unterbrechen die Verarbeitung der Daten, bis sie von allen ihren Kommunikationspartnern eine *Ready-Nachricht* erhalten. Wird ein Modul von mehreren anderen Modulen als vorhergehendes oder nachfolgendes Modul genutzt, so wird nur die Verarbeitung aller Daten von und zu diesem Modul unterbrochen. Während der *Break*-Zustand gesetzt ist, versucht das vorhergehende Modul seine Abhängigkeiten zu erfüllen und versucht ein Ersatz-Modul zu starten[12]. Erst wenn dieses Modul seinen *Ready*-Zustand signalisiert, wechseln auch alle anderen Module wieder in den *Ready*-Zustand und die Verarbeitung kann wieder aufgenommen werden.
Ein Modul kann auf zwei verschiedene Arten beendet werden. Zum einen durch einen äußere Signalisierung, dass es beendet werden soll. Zum anderen durch den Empfang einer *Closing-Nachricht* von einem anderen Modul. Das Modul wird jedoch erst beendet, wenn es keine anderen Module hat, von denen es Daten erhält. In der *Closing-Nachricht* ist ein Grund angegeben, warum das Modul beendet wird. Wird ein Modul aufrund eines *Reboots* eines Framework-Knotens beendet, so müssen die abhängigen Module informiert werden, damit das fehlende Modul durch ein Ersatzmodul ersetzt werden kann. In diesem Fall werden sowohl vorhergehende Module, als auch nachfolgende Module benachrichtigt. Es wird so gehandelt, als handele es sich um eine Exception. Handelt es sich um ein *normales*-Beenden eines Moduls, weil z.B: eine Analyse beendet wird, so wechselt das Modul ebenfalls in den *Closing*-Zustand (nachdem die letzten Daten verarbeitet worden sind) und versendet dann ebenfalls die *Closing-Nachricht* an seine Modulpartner. Während dieser Phase werden even-

[12]Der Startvorgang eines Moduls wird vom Loader auf einem Knoten wahrgenommen. Er ist auch derjenige, der den Ausfall eines Moduls bemerkt.

tuelle Konfigurationsänderungen gespeichert und genutzter Speicher freigegeben. Dem Framework wird mitgeteilt, dass die Dienste diese Moduls nicht mehr vorhanden sind, danach werden die Kommunikationsbeziehungen geschlossen. Nachdem alle Ausgaben beendet wurden, wechselt das Modul in den *Stopped*-Zustand und beendet sich.

4.9 Konfiguration und Management

Dieser Abschnitt soll beschreiben, wie das Framework konfiguriert und verwaltet werden kann. Diese Aufgaben übernehmen zwei Module, die auf den Framework-Knoten ausgeführt werden können. Diese müssen nicht auf jedem Knoten ausgeführt werden, sondern es werden nur einige wenige Module benötigt. Das Konfigurationsmodul speichert seine Daten z. B. in einer verteilten Hashtabelle. Diese muss nicht durch das Modul selbst implementiert werden, sondern kann von einem anderen Modul übernommen werden, welches die verteilte Hashtabelle implementiert und ein entsprechendes Interface zur Verwaltung der Daten bereitstellt.
In den Konfigurationen sollen Daten abgelegt werden, wie einzelne Messungen und Analysen ausgeführt werden sollen und welche Funktionsmodule zu diesen Messungen und Analysen benötigt werden, sowie wie die einzelnen Module miteinander verknüpft sind. Ebenso sollen Informationen zu einzelnen Modulen in der Konfiguration abgelegt werden. Damit kommt man dem Management näher, welches die Informationen zu einzelnen Modulen in sich speichert, die gerade ausgeführt werden. Diese Informationen der Module werden beim Start eines Moduls zum einen aus einer Konfiguration geladen und zum anderen kann ein Modul selbst die Informationen zu sich im Management ablegen. Das Management lässt sich in das lokale Management und das globale Management unterteilen.

4.9.1 Konfiguration

In diesem Abschnitt wird geklärt, welche Form die Konfiguration hat und welche Parameter in der Konfiguration gespeichert werden. Wie schon angedeutet, beherbergt die Konfiguration Informationen, wie einzelne Messungen und Analysen durchzuführen sind und welche Module dazu benötigt werden. Diese Konfigurationen werden durch einen administrativen Eingriff erstellt und gespeichert. Hinzu kommt, dass diese Konfigurationen mit dem Job-Scheduler für zeitliche Analysen und Messungen gestartet werden können. Ebenso beinhaltet die Konfiguration Informationen zu einzelnen Modulen, die beim Start eines Modul festgelegt werden können. Diese Informationen sind dynamisch änderbar und stellen nur die Startparameter des Moduls dar. Ein Modul kann für eine Messung und Analyse speziell konfiguriert werden. Neben den Informationen über komplette Analysen und einzelne Module enthält die Konfiguration die Startinformationen über einen Framework-Knoten. Dies impliziert, dass es zwei Konfigurationen gibt. Zum einen eine Konfiguration, mit der komplette Messungen und Analysen gestartet werden können. Zum anderen die Standardkonfiguration der einzelnen Module und Framework-Knoten.

Ebenfalls wurde schon angedeutet, dass die Konfiguration innerhalb einer verteilten Hashtabelle gespeichert werden soll. Dies erfordert einige Überlegungen, die die Adressierung und die Persistenz der Konfiguration innerhalb der verteilten Hashtabelle betreffen. Zum einen müssen die einzelnen Informationen eindeutig identifizierbar sein. Zum anderen muss die Konfiguration persistent gemacht werden, wenn ein Framework-Knoten beendet wird. Dies enthält zum einen die Forderung nach einer verteilten Hashtabelle, die redundant ausgelegt ist und zum anderen, dass Informationen der Hashtabelle beim Beenden in Dateien geschrieben werden können. Es müssen mindestens die Daten, die den lokalen Knoten und die lokalen Module betreffen, in Dateien oder persistente Datenbanken gespeichert werden.
Nun ist noch erforderlich festzulegen, welche Form die Konfiguration besitzt. Hierbei bieten sich zwei Formen an, die genutzt werden können und sinnvoll erscheinen. Zum einen ist dies die Beschreibung der Konfiguration durch die Form

Variablenname = Wert

und die Abschnitte für die einzelnen Module durch

[Abschnittsname]

getrennt werden können. Wobei ein Abschnitt beim Ende der Datei oder bei der Definition eines neuen Abschnitts endet.
Dem gegenüber steht die Darstellung der Konfiguration durch eine zu definierende XML-Struktur[13]. Die Syntax einer XML-Datei ist in [EaRS02] definiert. Eine XML-Datei ist streng hierarchisch gegliedert. Durch die Definition einer Schablone kann eine XML-Datei validiert werden und festgestellt werden, ob es sich um eine korrekte Form einer Konfiguration handelt.
Für beide Varianten existieren Parser, mit denen die Konfigurationen gelesen werden können und bereitgestellt werden können. In der Konfiguration müssen die Parameter gespeichert werden, die sich aus den Anforderungen zum Starten und Konfigurieren eines Moduls ergeben. Diese Anforderungen werden in Kapitel 4.7 ersichtlich. Hinzu kommt, dass die Konfiguration leicht lesbar sein sollte und manuell änderbar. Gleichfalls soll sie einfach verarbeitet werden können und die gespeicherten Variablen und Werte sollten entnehmbar sein. Mit dem streng hierarchischen Aufbau und gleichzeitiger Möglichkeit auf Validierung sowie der vorhandenen Flexibilität fällt hier die Entscheidung auf die Nutzung einer Konfiguration im XML-Format. Das Layout der XML-Datei und dessen Beschreibung findet sich im Anhang A.

4.9.2 Management

Das Management-Modul fasst alle Aufgaben zusammen, die mit der Verwaltung des Frameworks zusammenhängen. Ein Teil der Verwaltung wird schon

[13] eXtensible Markup Language (RFC 3275) [EaRS02]

von dem Service Discovery übernommen, da dies Informationen über laufende Module und Modullokationen speichert. Das Management lässt sich in zwei Teile untergliedern. Ein Teil des Managements besteht aus der Verwaltung eines Knotens als Ganzes. Zum anderen besteht das Management aus der Verwaltung der einzelnen Module auf einem Knoten. Die beiden Teile gehen fließend ineinander über. Dabei spielt auch das Service Discovery eine Rolle. In der Verwaltung eines Knotens werden verschiedene Informationen über den Knoten selber gespeichert. Diese Informationen können dem Service Discovery entnommen werden, da dort diese Informationen vorhanden sind. Des Weiteren werden Informationen über vorhandene Netzwerkschnitstellen, Speicher, Prozessorleistung, etc. dem Management entnommen. Ebenso werden Informationen über die einzelnen Module auf einem Knoten über das Knotenmanagement zugänglich gemacht. Das Knotenmanagement soll als einheitliche Schnittstelle zur Verwaltung eines Knotens dienen. Die Informationen kommen zum Teil aus dem Service Discovery, zum Teil aus dem Betriebssystem des Knotens und der letzte Teil wird den einzelnen Modulen auf dem Knoten entnommen. In den Informationen der einzelnen Module werden statistische Daten über ein Modul zusammengefasst und zugänglich gemacht. Zur Speicherung der Daten des Managements kommt evtl. eine verteilte Hashtabelle zum Einsatz, die dem Knoten verfügbar ist oder es kann intern geregelt sein. Die Speicherung in einer verteilten Hashtabelle oder einer vergleichbaren globalen Speicherstruktur hat den Vorteil, dass auf diese Informationen von jedem Knoten des Frameworks aus zugegriffen werden kann und nicht jede Information von einem Knoten selber besorgt werden muss. Zum anderen muss aber ein gesicherter Zugriff auf Managementparameter gegeben sein, damit es zu keinen inkonsistenten Daten innerhalb des Frameworks kommt. Dies ist besonders wichtig bei Informationen, die von mehreren Knoten bearbeitet werden können. Bei Daten, die ein einzelnes Modul betreffen, werden schreibende Zugriffe an das Modul selber weitergeleitet, welches dann den Zugriff auf die Parameter koordinieren kann. Sind die Daten verteilt gespeichert und verschiedene Knoten und Module können auf diesen Daten schreiben, muss eine Koordinierung des Zugriffs auf diese Daten erfolgen. Mechanismen zur verteilten Koordinierung von schreibenden Zugriffen auf Daten werden in [Tane00] beschrieben.
Managementinformationen der Module sind in die Module selber integriert und können über eine einheitliche Schnittstelle ausgelesen werden und zum Teil verändert werden. Die Knoteninformationen, welche ebenfalls über die gleiche Schnittstelle abgefragt werden können wie die Modulinformationen, sind in einem eigenen Modul implementiert. Es handelt sich bei den Knotenparametern jedoch um andere Informationen, als die bei einem Modul. Die per default in jedem Modul vorhandenen Informationsdaten werden in der Tabelle 4.3 aufgelistet. Für Funktionsmodule lassen sich noch weitere Parameter definieren. In Tabelle 4.5 werden diese Parameter beschrieben.

Für das Knoten-Management lassen sich folgende Parameter definieren. Diese werden in Tabelle 4.6 beschrieben. Diese Werte sind ein Vorschlag und sind in der implementiertn Version nicht enthalten. Sie sollen einen Eindruck davon geben, welche Parameter in dem Modul abgelegt werden sollen.

Tabelle 4.5: Vordefinierte Variablen des funktionales Modulinterfaces

Variablenname	Typ	Beschreibung
uuid	VAR_INT	eindeutige ID des Moduls. Diese ID wird in der Konfigurationsdatei festgelegt.
moduletype	VAR_STRING	Legt den Modultyp fest.
fqdn	VAR_STRING	vollständiger DNS-Name des Knotens auf dem das Modul ausgeführt wird.

Tabelle 4.6: Vordefinierte Parameter eines Knotens

Variablenname	Typ	Beschreibung
node_name	VAR_STRING	Knotenname des Knotens als DNS-Hostnamen
node_ifaces	VAR_USER	Liste von Namen der Netzwerk-Interfaces des Knotens
node_iface_xxx_type	VAR_STRING	Typ des Interfaces
node_iface_xxx_addrl2	VAR_STRING	Layer 2 Adresse des Interfaces
node_iface_xxx_addrl3	VAR_STRING	Layer 3 Adresse des Interfaces
node_iface_xxx_recv	VAR_INT	Empfange Oktett
node_iface_xxx_send	VAR_INT	Gesendete Oktett
node_modules	VAR_USER	Liste von Modulen auf dem Knoten
node_running_mods	VAR_USER	Liste von ausgeführten Modulen auf dem Knoten
node_os	VAR_STRING	Name des Betriebssystems des Knotens
node_cpu	VAR_STRING	Name der CPU des Systems
node_uptime	VAR_USER	Uhrzeit, zu der der Knoten gestartet wurde. Es handelt sich um ein *timeval*-struct der Unix-Zeit.
...	...	...

4.10 Funktionsmodule

In diesem Abschnitt soll der Entwurf verschiedener Funktionsmodule geschildert werden. Diese Funktionsmodule benötigen zu ihrer Funktionsweise andere Module, auf denen diese aufbauen. Diese grundlegenden Module werden in den vorhergehenden Abschnitten geschildert. Es handelt sich dabei um die Module, die die Kommunikation sowie das Management zwischen einzelnen Funktionsmodulen herstellen. Die Funktionsmodule dienen der Messung und Analyse von Internetverkehr. Module dieser Klasse lassen sich aus der Definition in Kapitel 4.1 entnehmen.

4.10.1 Traffic-Inputs

Mit den Funktionsmodulen zum Traffic-Input sind Möglichkeiten gemeint, wie Internetverkehr auf unterschiedliche Weise erfasst werden kann. Die verschiedenen Möglichkeiten zur Erfassung von Paketdaten werden in Kapitel 3.3 vorgestellt. Exemplarisch für die prototypische Implementierung soll hier der Entwurf zur Erfassung von Paketen mit Hilfe des Lib-Pcap Interfaces geschildert werden.

In Kapitel 3.3.1 wird das Pcap-Interface und der *Berkeley-Packet Filter* (BPF) vorgestellt. Diese Bibliothek soll als Grundlage zur Erfassung von Paketdaten dienen. Dieses Funktionsmodul erfasst Paketdaten und versendet diese mit Hilfe des Frameworks zu weiteren Modulen, die die erfassten Paketdaten analysieren. Zur Spezifikation der Paketdaten, die erfasst werden sollen, müssen verschiedener Parameter in einer Konfigurationsdatei angegeben werden. Diese Parameter werden in der Tabelle 4.10.1 spezifiziert. Es muss darauf geachtet werden, dass ein Knoten mit *root*-Rechten ausgeführt werden muss, da sonst das Pcap-Modul nicht initialisiert werden kann. Diese Parameter werden in der Tabelle 4.10.1 definiert. Mit Hilfe dieser Parameter kann das Pcap-Modul initialisiert werden. Nachdem sich das Modul im *Ready*-Zustand befindet, wird ein eigener Thread zur Erfassung der Pakete gestartet. Diese Pakete werden dann so lang erfasst, bis die Anzahl der angegebenen Pakete erreicht wird, das Modul in den *Break*-Zustand wechselt oder eine Nachricht zum Beenden des Moduls erreicht wird. Danach werden keine neuen Paket mehr erfasst. Es werden nur noch die restlichen gepufferten Pakete an die Analysemodule weitergeleitet. Danach beendet sich das Modul.

4.10.2 Traffic-Analysatoren

Dieser Abschnitt soll den Entwurf eines Verkehrsanaylsators beschreiben. Beispielhaft soll in diesem Framework ein Analysator für TCP-Verkehr erstellt werden. Dieser wird zu großen Teilen aus der Arbeit [Zülc04] übernommen. Jedoch wird dieser Analysator hinsichtlich des Frameworks und den darin verwendeten Schnittstellen angepasst. Zum einen wird der Empfang der Pakete so angepasst, dass dieser Analysator nur die Pakete verarbeiten kann, so wie

das Capturing-Module (Abschnitt 4.10.1) die Pakete versendet. Der Analysator war bisher darauf ausgerichtet die Pakete über ein FlexiNet-Interface zu bekommen und zu verarbeiten. Dies wird nun in den Empfang von Nachrichten und das demultiplexen der einzelnen Pakete aus dieser Nachricht geändert. Zum anderen wird die Ausgabe der analysierten Daten so verändert, dass die Daten nicht mehr in einer Datei abgelegt werden, sondern mit Hilfe von Nachrichten versendet werden können und in anderen Modulen weiterverwendet werden können. In der Tabelle 4.10.2 werden die Parameter des Moduls beschrieben.

4.10.3 Aggregation und Eventgenerierung

In diesem Unterabschnitt sind einige Beispielmodule beschrieben, die die Funktionsweise des Frameworks demonstieren sollen.

4.10.3.1 TCP-Sampler

Der TCP-Sampler greift den Aspekt der Aggregation von Messergebnissen aus Abschnitt 3.6.5 auf. Dieser Sampler ist auf das Sampling von TCP-Paketen spezialisiert. Er regagiert auf Pakete, die mit Hilfe des TCP-Analysators generiert wurden. Das Sampling findet auf zwei verschiedene Arten statt. Zum einen wird nur jedes x-te Paket einer Verbindung durchgelassen. Zum anderen kann der Sampler einer Mittelwert über x-Pakete berechnen und diesen Mittelwert weiterleiten. Diese beiden Modi werden durch eine Konfigurationsvariable gesteuert. Diese Variable ist bei Nichtangabe auf den ersten Modi eingestellt. Das Sampling findet für jede TCP-Verbindung getrennt statt. Das bedeutet, dass für jede Verbindung ein eigener Kontext erstellt wird und darin festgehalten wird. ob das nächste Paket der Verbindung verworfen wird oder nicht. Ebenfalls wird die Richtung bei TCP-RTTs berücksichtigt.[14] Die Konfigurationsvariablen sind in Tabelle 4.10.3.1 definiert.

4.10.3.2 Differenz-Erzeuger

Mit der Differenz-Erzeuger existiert ein Modul, welches RTT-Werte aus einem TCP-Sampler oder direkt aus der TCP-Analyse entgegennimmt. Dieser Differenz-Erzeuger berechnet die Differenz zwischen zwei RTT-Werten. Dazu wird dem Modul mitgeteilt, welche RTT-Werte für die Berechnung herangezogen werden sollen. Die Berechnung geht immer von dem aktuell empfangenen RTT-Wert aus. Um die Differenz zu berechnen wird per Konfiguration ein Zahlenwert angegeben, der den RTT-Wert vorgibt, der zur Berechnung herangezogen wird. Dieser Wert gibt die Anzahl der RTTWerte vor dem aktuell empfangenen RTT-Wert an. Dieser Wert muss eine ganzzahlige Zahl sein. Wird z.B. ein Wert von 4 angegeben, bedeutet dies, dass zur Berechnung der RTT-Wert herangezogen wird, der vor 4 Paketen berechnet wurde. Nun berechnet dieses Modul die Differenz zwischen dem angegebenen Wert und dem

[14] Aus der Arbeit[Zülc04] geht hervor, dass sich die TCP-RTT für zwei verschiedene Abschnitte einer Verbindung berechnen lässt.

aktuellen Wert. Danach reicht das Modul das Ergebnis an ein anderes Modul weiter, welches mit Hilfe der RTT und der Differenz zu einem Vorgänger weitere Analysen durchführen kann. Die gleiche Rechnung führt dieses Modul auch für TTL-Werte durch und reicht diese ebenfalls an ein anderes Modul zur weiteren Verarbeitung weiter. Das Modul ist auf den Empfang von TCP-RTT Werten spezialisiert und kann die Differenz nach Richtungen getrennt, berechnen. Ebenso wird auf den Verbindungskontext (Quell- und Zieladresse sowie Quell- und Zielport) geachtet. Für dieses Modul sind in Tabelle 4.10.3.2 die Konfigurationsparameter dokumentiert.

4.10.3.3 Korrelator

Der sogenannte Korrelator nimmt die Daten von beliebig vielen Differenz-Erzeugern entgegen. Dabei müssen die Differenz-Erzeuger jeweils die Differenz zu einem anderen alten RTT- oder TTL-Wert berechnen. Der Korrelator vergleicht im Anschluss an die Differenzberechnung die Ergebnisse der Differenz-Erzeuger. Über zwei Konfigurationsvariablen wird der Korrelator gesteuert. Der erste Parameter (*rttpercent*) gibt an, dass sich bei einem Differenz-Erzeuger die RTT um mindestens *rttpercent* ändern muss, damit die RTT-Änderung als solche erkannt wird. Der zweite Parameter (*minindi*) legt fest, bei wie vielen Differenz-Erzeugern eine Änderung um die angegebenen Prozent detektiert werden muss. Wird diese Mindestanzahl erreicht, wird daraus eine Nachricht erzeugt, die den Zeitpunkt, die RTT, die Veränderung der RTT und die Richtung der Änderung beinhaltet. In der Tabelle 4.10.3.3 werden die Parameter dieses Moduls beschrieben.

4.10.3.4 Vergleicher

Der Vergleicher ist eine Mischung aus einem Differenz-Erzeuger und einem Korrelator. Zum einen bildet dieses Modul die mittlere Differenz zwischen verschiedenen RTT-/TTL-Werten, wobei die einzelnen Messwertquellen von unterschiedlichen Messknoten stammen. Es wird immer die Differenz zwischen zwei RTT-Werten zweier Messdatenströme berechnet, die am dichtesten in der Zeit beieinander liegen. Ursahe hierfür ist, dass auf verschiedenen Rechnern die Ankunft von Paketen zu unterschiedlichen Zeiten stattfindet.[15] Aus dieser Differenz werden Nachrichten generiert, die denen der TCP-Analyse gleichen. Damit wird sichergestellt, dass die erzeugten Ergebnisse in nachgeschalteten Differenz-Erzeugern weiterverwendet werden können. Jedoch handelt es sich bei diesen Daten dann nicht um direkte RTT-Werte, sondern um die Differenz derer.
Zum anderen versucht dieses Modul die Ergebnisse des Korrelators weiterzuverwenden. Dazu sucht es nach gleichen Ereignissen, die von verschiedenen Korrelatoren erzeugt wurden. Mit der Berechnung wird versucht eine Verbindung zwischen mehreren Verkehrsströmen aufzuzeigen. Damit wird der Versuch unternommen zu schauen, ob bei Ereignisse nicht nur lokal vorhanden

[15] Es wird angenommen, dass die beiden Rechner jedoch ihre Systemuhren miteinander synchronisieren.

sind, sondern von globaler Natur sind und ebenfalls von anderen Korrelatoren erkannt wurden.

4.10.4 Präsentation der Daten

Für die Präsentation der erzeugten Daten lassen sich verschiedene Interfaces definieren, über die die Ergebnisse angezeigt werden können. Hier soll ein einfaches Präsentationsmodul entworfen werden, welches in der Lage ist verschiedene Daten zu speichern. Zum einen soll über einen Parameter angegeben werden, ob die Daten auf dem Bildschirm ausgegeben werden sollen, oder ob diese in eine Datei geschrieben werden sollen. Die Daten sollen im CSV-Format ausgegeben werden. Dazu spezifiziert ein Parameter das Trennzeichen der einzelnen Spalten. Ein weiterer Parameter gibt an, wie viele Spalten eine Zeile hat. Die Daten müssen für dieses Modul in einem TLV (Type-Length-Value) Format zugeführt werden. Der Typ wird in einem 2 Oktett großen Integer-Wert angegeben. Ebeno wird die Länge des Wertes in einem 2 Integer-Wert angegeben. Als Ausgabe Typen werden Integer (Typ 1), String (Typ 2), Double (Type 3) akzeptiert. Andere Typen werden als String ausgegeben.

Tabelle 4.7: Parameter zur Konfiguration des Pcap-Moduls

Parameter	Typ	Beschreibung
device	VAR_STRING	Spezifiziert das Device (Netzwerkinterface, auf dem der Verkehr erfasst werden soll.)
promicous	VAR_INT	Gibt an, ob alle Pakete, die auf einem Netzwerkinterface empfangen werden, erfasst werden sollen oder nur die für das lauschende System betimmten Pakete.
snaplen	VAR_INT	Gibt die Länge in Oktett an, wie viele Oktett eines Paketes gespeichert werden sollen.
protocol	VAR_STRING	Spezifiziert das Protokoll, welches erfasst werden soll. das können TCP, UDP oder ICMP sein.
timeout	VAR_INT	Gibt die Wartezeit in Millisekunden an, wie lang auf den Empfang eines Paketes gewartet werden soll. Mit diesem Parameter kann die Kommunikation zwischen Kernel und Modul optimiert werden, damit gleichzeitig mehrere erfasste Pakete verarbeitet werden können.
numpkt	VAR_INT	Anzahl der Pakete, die erfasst werden sollen. Dieser Parameter ist optional. Wird er nicht angegeben, so werden solang Pakete erfasst, bis das Modul gestoppt wird.
command	VAR_STRING	Beschreibt den Filter in BPF-Syntax zur genauen Spezifikation des Paketstroms, der erfasst werden soll.
capfile	VAR_STRING	Gibt den Dateinamen an, aus dem ein Paketdatenstrom entnommen werden soll. Ein solcher Paketdatenstrom kann vorher mit *tcpdump* erfasst worden sein und soll nun zur weiteren Analyse verarbeitet werden.

Tabelle 4.8: Parameter zur Konfiguration des TCPAnalyse-Moduls

Parameter	Typ	Beschreibung
debuglevel	VAR_INT	Debug Level der TCP-Analyse, je niedriger der Wert ist, desto mehr Informationen werden angezeigt. Bei Nichtangabe dieser Variable werden alle Debugausgaben automatisch unterdrückt. 0 ist der kleinste Wert.

Tabelle 4.9: Parameter zur Konfiguration des TCP Sampler-Moduls

Parameter	Typ	Beschreibung
mode	VAR_STRING	Modus des Samplers, drop == jedes X-te Paket wird durchgelassen, avg == Mittelwert über X-Werte wird berechnet und nur jedes x-te Paket durchgelassen.

Tabelle 4.10: Parameter zur Konfiguration des TCP Differenz-Moduls

Parameter	Typ	Beschreibung
numback	VAR_INT	Anzahl der RTT-Werte, die in die Vergangenheit gegangen werden soll um die Differenz der RTT-Werte zu berechnen.
rate	VAR_INT	Gibt die Häufigkeit an, wie oft die Differenz berechnet werden soll. 1 entspricht die Berechnung bei jedem ankommenden Paket. $1 < X < MAXINT$ entspricht der Berechnung bei jedem X-ten RTT-Wert.

Tabelle 4.11: Parameter zur Konfiguration des Korrelators

Parameter	Typ	Beschreibung
rttpercent	VAR_INT	RTT-Änderung um die angegebenen Prozent
ttlpercent	VAR_INT	TTL-Änderung um die angegebenen Prozent
rttoffset	VAR_INT	Änderung der RTT um diesen Offsetwert
ttloffset	VAR_INT	Änderung der TTL um diesen Offsetwert
minindi	VAR_INT	Anzahl der Differenz-Erzeuger, die eine Änderung erkennen müssen
command	VAR_STRING	Modus der Änderung; entweder *Prozent* oder *Offset*

Tabelle 4.12: Parameter zur Konfiguration des Präsentations-Moduls

Parameter	Typ	Beschreibung
ausgabe	VAR_STRING	Dieser Parameter dient der Steuerung der Ausgabe. Er kann die beiden Werte *bildschirm* oder *datei* annehmen. Hat er den Wert *bildschirm*, so werden alle Daten auf dem Bildschirm ausgegeben. Bei der Ausgabe in eine Datei müssen zusätzlich die beiden folgenden Parameter angegeben werden.
filepath	VAR_STRING	Gibt das Verzeichnis an, in dem die Daten gespeichert werden sollen.
filename	VAR_STRING	Dateipräfix der Dateien vorangestellt wird.
csvsep	VAR_STRING	Seperator für eine CSV-Datei. Wenn dieser Parameter nicht angegeben ist, so ist er per default auf das Leerzeichen (ASCII #32) gesetzt.
numcol	VAR_INT	Anzahl der Spalten, die in der Datei vorhanden sein sollen. Nach *numcol* Spalten wird ein Zeilenumbruch durchgeführt.
msgid	VAR_INT	Nachrichten-ID auf die der FileWriter hören soll.

5. Implementierung

In diesem Kapitel wird geschildert, wie das entworfene Framework und die einzelnen Funktionsmodule in der Implementierung umgesetzt wurden. Aus dem Entwurf des Frameworks geht hervor, dass es modular aufgebaut ist. Es ist möglich, mehrere verschiedene Module auf einem Knoten zu starten und auszuführen. In der Implementierung wird diese Möglichkeit durch Laden von zusätzlichen Bibliotheken, die die Funktionalitäten enthalten unterstützt. Das Framework ist vollständig in C/C++ implementiert, da sich auf Grund der Modularität eine objektorientierte Programmierung anbietet. In weiten Teilen des Programms wird die STL[1] verwendet. Die Implementierung des Frameworks fand ausschließlich unter Linux statt.

5.1 Knotenhauptprogramm

Ein Frameworkknoten wird in der Implementierung durch ein Hauptprogramm repräsentiert. Dieses wird zuerst gestartet. In diesem Hauptprogramm werden evtl. vorhandene Kommandozeilenparameter ausgewertet und für die weitere Ausführung des Programms bereit gehalten. Folgende Parameter sind definiert: Nachdem die Kommandozeilenparameter analysiert wurden, wird der Nachrichtendispatcher initialisiert und zur Ausführung gebracht. Es wird bei der Ausführung des Dispatchers darauf geachtet, dass dieser nur einmal pro Frameworkknoten ausgeführt wird. Nun wird die Konfiguration des Hauptprogramms geladen und analysiert. Ebenso werden die darin konfigurierten Module geladen (siehe dazu Anhang C). Bei diesen Modulen handelt es sich um keine Funktionskomponenten, sondern um die folgenden Funktionalitäten:

- Loader
- Socket Multiplexer

[1] Standard Template Library. Diese Library umfasst standardisierte Algorithmen und Funktionen. Diese sind in einem ISO-Standard festgeschrieben.

Tabelle 5.1: Programmparameter eines Frameworkknotens

Parameter	Beschreibung
-h, –help	Auflistung aller Kommandozeilenparameter auf dem Bildschirm
-f FILE, –cfgfile FILE	alternative Konfigurationsdatei des Hauptprogramms
-d, –daemon	Programm im Hintergrund oder Vordergrund ausführen
-a FILE, –anafile FILE	Analyse Datei, die zusätzlich geladen werden soll und ausgeführt werden soll. Diese Datei wird an den Loader übergeben, welche diese dann parsed und ausführt.
-D, –debug	Debugging Modus. Es werden mehr Ausgaben auf der Konsole gemacht.

- Logger
- Service Discovery
- Job-Scheduler
- Kommunikationsmodule (Chord, etc...)

Sind diese Module geladen, ist ein Knoten komplett und funktionsbereit. Die einzelnen genannten Module liegen als Bibliotheken (Shared-Objects) vor und werden mit Hilfe eines dynamischen Loaders gestartet. Jedes Modul wird in einem eigenen Thread[2] innerhalb des Hauptprogramms oder als eigenständiger Prozess ausgeführt. Die Kommunikation der Module untereinander wird durch Nachrichten realisiert. Der Nachrichtenverlaufes ist im Abschnitt 5.2 geschildert. Der Aufbau von Nachrichten wird in Anhang B beschrieben. Dort werden auch festgelegte Nachrichtentypen und deren Aufbau geschildert.

5.2 Datenaustausch und Behandlung von Nachrichten

Im Kapitel "Entwurf" wurde definiert, dass der Daten- bzw. Nachrichtenaustausch zwischen den Modulen per Warteschlange realisiert wird. Dies ist in der Implementierung so beibehalten worden. Wenn eine Nachricht versendet werden soll, ruft eine Funktion des Moduls seine "*Send*"-Methode auf und übergibt dieser die Daten der Nachricht und die Adresse, des Moduls an welche die Nachricht gesendet werden soll. Zusätzlich enthält der Aufruf noch die Größe der Nachricht, den Zeiger auf die Nachricht und den Nachrichtentyp. In der

[2] Zur Implementierung von Threads werden POSIX-Threads verwendet.

"*Send*"-Methode wird nun entschieden, wie die Kommunikation zwischen dem Modul und dem Kern des Frameworkknotens geschehen soll. Dies kann zum einen eine Kommunikation per Socket sein. Zum anderen ist die Kommunikation per Warteschlange möglich. Dies bedeutet, dass mit Hilfe des Dispatchers nachgesehen wird, welches Modul für die Nachricht zuständig ist. Steht in der Zieladresse nicht das Protokoll "*local*", so wird mit Hilfe des Dispatchers aufgelöst, welches Modul für dieses Protokoll zuständig ist. Nun wird die Nachricht an die "*Receive*"-Funktion des zuständigen Moduls übergeben. Die Kommunikation für das sendende Modul ist damit erledigt. Handelt es sich bei dem Zielprotokoll um das "*local*" Protokoll, so wird mit Hilfe des Dispatchers das lokale Modul herausgesucht, welches für diese Nachricht zuständig ist. Dies geschieht auf Grund der in der Adresse angegebenen "ModulID". Die Nachricht wird nun der "*Receive*"-Funktion des zuständigen Moduls übergeben. Kann kein zuständiges Modul auf einem Knoten gefunden werden, wird die Nachricht an das Modul mit der Nummer 0 übergeben. Unter dieser "ModulID" registriert sich der Loader zusätzlich[3]. Nun kann der Loader entscheiden, was mit der Nachricht weiterhin geschehen soll. Diese Nachricht kann entweder gelöscht werden, oder der Loader kann an Hand des Nachrichttyps erkennen um welche Komponente es sich handelt und die Nachricht an diese Komponente weiterleiten. Ist keine Komponente für diesen Nachrichtentyp gestartet kann dieser die Komponente neu starten.

Ist die Nachricht nun in einem Modul in einer Warteschlange lokalisiert, so wartet diese darin so lange, bis sie von einem Modul verarbeitet werden kann. Soll die Nachricht verarbeitet werden, prüft das Modul an Hand einer Liste nach, welche Funktion für eine entsprechende Nachricht zuständig ist. Die Funktion wird nun mit der Nachricht als Parameter aufgerufen. Bevor ein Modul auf eine Nachricht reagieren kann, muss es bei sich selbst eine "*Callback*"-Funktion registrieren, welche eine bestimmte Nachricht verarbeitet. Ist eine solche Methode nicht für eine bestimmte Nachricht registriert, so wird diese verworfen. Es existiert aber auch ein Nachrichtentyp, der auf alle ankommenden Nachrichten reagiert. Dieser Nachrichtentyp muss ebenfalls registriert werden, ansonsten werden die Nachrichten gelöscht. Nachdem die Methode zur Bearbeitung einer Nachricht gefunden wurde, kann die Verarbeitung der Nachricht und dessen Inhalt von der "Rückruf"-Funktion durchgeführt werden. Diese Methode ist für die Freigabe des Speichers, den eine Nachricht belegt, verantwortlich. Hinzu kommt, dass Daten der Adressierung ebenfalls in Verantwortung der Callback-Funktion gelöscht werden müssen.

5.3 Aufbau eines allgemeinen Moduls

Das allgemeine Modulinterface erbringt die Grundfunktionen eines Moduls zur Kommunikation mit anderen Modulen im Framework. Dieses Interface ist in der C++ Klasse *Modulecore* gekapselt.

[3]Der Loader besitzt die feste ID 1.

Quellcode 5.1: Modulecore-Interface

```
class Modulecore{
  private:

    typedef list<struct RegMessages*> RegMsg;
    RegMsg RegisteredMessageTypes;

    ClientSocket *KommSock;
    Threads *sth;
    sem_t *x,*wsem,*rsem;
    int readercount, inq;
    queue<msginfo_t*> thqueue;

    struct RegMessages *GetMsgType(msgtype mtype,
                                   unsigned int modID);
    void RegisterVarPar(struct parameter *par);

  protected:

    ModuleInfo *ModInfo;
    Dispatcher *Disp;
    RPCMgr *RPCManager;
    unsigned int ModuleID;
    unsigned int Debug;
    addr SELF;

  public:

    Modulecore(void *dummy);
    virtual ~Modulecore();
    void Run();
        void Run(void *dummy);
    unsigned int GetModuleID();
    const char *GetModuleName();
    unsigned int GetDebugMode();
    void start_sockthread(void *dummy);
    virtual void RegisterMsgType(msgtype mtype,
                          unsigned int modID,
                          unsigned int flags,
                          TFunktor *function);
    virtual void UnRegisterMsgType(msgtype mtype,
                            unsigned int modID,
                            unsigned int flags);
    virtual void Send(msgtype mtype,
                 addr to,
                 msgsize msize,
```

```
46              dataptr mdata);
    virtual void Send(msgtype mtype,
48              addr to,
                addr from,
50              msgsize msize,
                dataptr mdata);
52  virtual void Send(msginfo_t* info);
    virtual void Receive(unsigned int nonce,
54                  unsigned int flags,
                    msgtype mtype,
56                  addr to,
                    addr from,
58                  msgsize msize,
                    dataptr mdata);
60  virtual void Receive(msginfo_t *info);
    virtual void StatusChange(void *dummy);
62  virtual void ShutdownMessage(msginfo_t* info);
    ModuleInfo *GetModuleInfo();
64 };
```

Dem Konstruktor der Klasse wird ein generischer Zeiger auf eine Datenstruktur übergeben, die verschiedene Daten zum Starten eines Moduls enthält. Dies sind zum einen ein Zeiger auf den Dispatcher, ModulID, Modulname, Startmethode ("Thread" oder "Fork"), Dateiname und ein Zeiger auf eine Struktur mit den dynamischen Variablen, die einem Modul in der Konfiguration übergeben werden. Alle diese Variablen werden lokal gespeichert. Auf die Variablen kann über die Klasse "ModuleInfo" zugegriffen werden. Wird ein Modul per "Fork" gestartet, so kommuniziert das Modul nicht mehr per Warteschlange mit den anderen Modulen eines Frameworkknotens, sondern per Socket. Dazu wird ein Thread gestartet, der auf neue Nachrichten per Socket lauscht und diese in die Warteschlange stellt. Die Warteschlange ist bei der Kommunikation per Socket nicht nötig, aber um die Verarbeitung der Nachrichten zu vereinfachen, werden diese ebenfalls in die Warteschlange gestellt. Somit ist eine einheitliche Entnahme der Nachrichten aus der Warteschlange möglich.

Diese Klasse implementiert die Hauptschleife eines Moduls. Die Schleife ist in der Funktion *Run* gekapselt. Darin wird eine Nachricht aus der Queue entnommen und in der Liste der registrierten Nachrichten nach einer Callback-Funktion gesucht. Diese Suche wird von der Funktion *GetMsgType* durchgeführt. Die implementierte Queue stammt aus der STL. Damit diese Queue auch in multithreaded Umgebungen funktioniert, wurde sie zur Sicherung gegen gemeinsamen Zugriff mit Semaphoren geschützt. Nachdem eine "*Callback*"-Funktion für eine Nachricht gefunden wurde, wird die "*Callback*"-Funktion aufgerufen und die Nachricht dieser Funktion übergeben.

Zur Registrierung und Deregistrierung von Nachrichtentypen stehen die beiden Funktionen *RegisterMsgType* und *UnRegisterMsgType* zur Verfügung. Die Ab-

bildung von Nachrichtentyp auf Behandlungsroutine wird durch eine Liste realisiert, die den Nachrichtentyp, die ModulID und den Zeiger auf die Callback-Funktion enthält.

Hinzu kommen die beiden grundlegenden Funktionen zum Senden und Empfangen von Nachrichten. In der *Send*-Funktion wird die Nachricht für das Versenden aufbereitet und mit Hilfe des Dispatchers das zuständige Modul zur Verarbeitung einer Nachricht gesucht. Der konkrete Nachrichtentransfer ist im Abschnitt 5.2 geschildert. Aufbereitung der Daten bedeutet in diesem Zusammenhang, dass bekannte Felder der Adressierung in Network-Byte-Order gewandelt werden, bevor diese übertragen werden. Bytes der Nachricht werden nicht berücksichtigt, da der Aufbau dieser nicht bekannt ist. Dafür muss die Funktion, die die Nachrichtendaten erstellt, selber sorgen. In dieser Funktion wird auch die Absenderadresse automatisch generiert. Die Adresse setzt sich aus dem Protokoll “*local*”, dem Knotennamen “*localhost*” und der eigenen ModulID zusammen. Die eigene ModulID ist dem Modul bekannt. Der Aufbau der Addressierung ist in Kapitel 4.5 erklärt.
In der *Receive*-Funktion werden diese entsprechenden Felder in der Adressierung wieder in Host-Byte-Order gewandelt. Konkret handelt es sich bei den Adressen um die ModulID, die jeweils verändert wird. Zum Schluss wird die Nachricht in die Queue gestellt.

5.4 Aufbau eines Funktionsmoduls

Dieser Abschnitt erklärt das Grundgerüst eines Funktionsmoduls. Dieses Grundgerüst bzw. Interface baut auf dem allgemeinen Modulinterface (class Modulecore) auf. Diese Klasse erweitert die Funktionalität des allgemeinen Modulinterfaces und verbirgt einige Details der Kommunikation zum eigentlichen Modul hin. Jedes Funktionsmodul sollte auf dieser Klasse basieren.

Quellcode 5.2: ModuleInterface-Interface

```
class ModuleInterface : public Modulecore{
  private:

    typedef list<struct depmods*> DependMods;
    DependMods InputMods;
    DependMods OutputMods;

    typedef list<struct paralist*> ParamSets;
    ParamSets Parameter;

    typedef list<struct mangle*> MangleList;
    MangleList InputHandler;

    bool CheckModuleReady();
    void BuildReadyMessage(struct depmods *dep);
    void MessageHandler(msginfo_t *info);
```

```
    addr GetAddrFromLocation(struct depmods *dep);
18
  protected:
20
    LoggerClient *Logger;
22
    unsigned int GetUUIDFromAddr(addr a);
24  unsigned short int GetModuleStatus();

26 public:

28  ModuleInterface(void *dummy);
    virtual void RegisterMsgType(msgtype mtype,
30                               unsigned int modID,
                                 TIfaceFunktor *function);
32  virtual void RegisterMsgType(msgtype mtype,
                                 unsigned int modID,
34                               unsigned int flags,
                                 TFunktor *function);
36  virtual void UnRegisterMsgType(msgtype mtype,
                                   unsigned int modID);
38  virtual void InputMangle(msginfo_t *info) {};
    virtual void OutputMangle(msginfo_t *info) {};
40  virtual void Send(msgtype mtype,
                 msgsize msize,
42               dataptr mdata);
    virtual void Send(IfaceMsg *info);
44  virtual void Send(msginfo_t *info);
    void *GetDependVar(unsigned int uuid,
46                     char *vname);
    void *GetCommonVar(char *vname);
48  void message_readymodule(msginfo_t *info);
    unsigned short int GetNumInputDep();
50  unsigned short int GetNumOutputDep();
    virtual ~ModuleInterface();
52 };
```

Im Konstruktor dieser Schnittstelle wird die Datenstruktur des Startparameters erweitert. Es handelt sich jetzt nicht mehr nur um die im vorhergehenden Abschnitt geschilderten Parameter. Es werden weitere Informationen, die z.B. die Abhängigkeit zwischen einzelnen funktionalen Modulen beschreiben, und solche, die in Abhängigkeit des Quell- oder Zielmoduls gesetzt sind, übergeben.

Hinzu kommt, dass die beiden Funktionen zur Registrierung und Löschung von Nachrichtentypen erweitert werden. Der Hintergrund dabei ist, dass die eigentliche Methoden des Moduls nicht wissen, woher die Nachrichten kommen, sondern nur noch Informationen in Form von verschiedenen Nachrichtentypen

empfängt. Informationen, die den Ursprung einer Nachricht (best. Modul) betreffen, müssen in den Nachrichten Daten selbst übertragen werden.

Um jedoch den Empfang und Versand von Nachrichten flexibel zu gestalten, wird bevor die Nachricht vom Nachrichtenhandler verarbeitet wird, die Nachricht durch die Funktion *InputMangle* verarbeitet. In dieser Funktion stehen Lokalisationsinformationen in Form der Quelladresse zur Verfügung. Hintergrund der Verarbeitung ist, dass mit Hilfe dieser Funktion eine Filterung der Nachrichten vorgenommen werden kann oder andere Parameter einer Nachricht verändert werden können, ohne dass die eigentliche Funktion des Moduls davon beeinträchtigt wird und somit davon unabhängig geschehen kann. Ebenso werden die Daten nicht mehr an einzelne Module gesendet. Vielmehr werden die Daten nur an die Module weiter gegeben, die laut Konfiguration eine Abhängigkeit vom Modul besitzen und die Daten weiterverarbeiten sollen. Bevor die Nachricht an ein Modul versendet wird, wird die Funktion *OutputMangle* aufgerufen. Hiermit wird eine Ausgangsfilterung erzielt. Es kann damit gesteuert werden, dass nicht jede Nachricht an jedes folgende Modul weitergeleitet wird, sondern nur an bestimmte.

In diesem Zusammenhang lassen sich auch Variablen für die verschiedenen Input- und Outputabhängigkeiten definieren, die je nach Quelle/Ziel definiert sind und abgefragt werden können. Dies wird mit der Funktion *GetDependVar* erreicht. In der Funktion *GetCommonVar* werden die Inhalte der allgemeinen Variablen aus der Basisklasse *Modulecore* zurückgeliefert.

Ein Modul, welches auf diese Klasse aufbaut, reagiert ebenfalls standardmäßig auf *Ready-Nachrichten* vom Loader und anderen Modulen, mit denen der im Kapitel 4.8 geschilderte Modulzustand verarbeitet und erstellt wird.

5.5 Start einer Konfiguration

Im Kapitel "Entwurf" ist geschildert, dass ein Loader eine komplette Konfiguration entgegennehmen und sie zur Ausführung bringen kann. Diese Konfiguration ist in Anhang A beschrieben. Sie kann vom Loader dem lokalen Dateisystem entnommen werden, kann als Nachricht erhalten werden oder sie wird aus dem Service Discovery geladen. Das Aussehen dieser Konfiguration ist in Anhang A beschrieben. Aus dieser Datei erstellt der Loader für jedes Modul, welches beschrieben wird, eine Modulstart-Nachricht und überträgt sie an den jeweiligen Loader auf dem Frameworkknoten, auf dem das Modul ausgeführt werden soll. Der jeweilige Loader nimmt diese Nachricht entgegen, lädt das Modul und bringt es zur Ausführung. Eine Modulstartnachricht besteht aus den folgenden Sektionen:

- Allgemeiner Abschnitt
- kein oder ein Inputdependency Abschnitt
- kein oder ein Outputdependency Abschnitt

- kein, ein oder mehrere Parameter Abschnitt(e)

Der komplette Aufbau dieser Nachricht ist in Anhang B definiert und dargestellt.

Nachdem für jede Nachricht alle Start-Nachrichten an die entsprechenden Loader versendet wurden, wird an alle Module, die keine Outputdependency-Module haben, eine *Ready*-Nachricht verschickt, da diese die letzten Module in einer Verarbeitungsreihe sind. Sie werden nicht an die Module direkt versendet, da der Loader, der die Konfiguration lädt, die Adresse noch nicht kennt. Er kennt nur den Knoten auf dem diese Ausgeführt werden sollen. Die Nachrichten werden vielmehr an den Loader des Frameworkknotens gesendet, auf dem das Modul ausgeführt werden soll. Jeder Loader führt eine Liste der Module, die lokal auf dem Frameworkknoten ausgeführt werden. Ein Modul wird dabei über seinen Modultyp und eine "ModulID" identifiziert, die in der *Ready*-Nachricht angegeben sind. Der lokale Loader schaut daraufhin in seiner Liste der gestarteten Nachrichten nach und versendet die Nachricht an das lokale Modul. Dieses Modul verzeichnet in seiner Liste der Output-abhängigen Module, dass sich ein Modul im *Ready*-Zustand befindet. Befinden sich nun alle Output-abhängigen Module im *Ready*-Zustand, so wird jeweils eine *Ready*-Nachricht an alle Module versendet, von denen das Modul Daten erhält. Die Behandlung des Zustandes und der *Ready*-Nachrichtenverarbeitung findet in der Funktion *message_readymodule* in der Klasse *ModuleInterface* statt. Da diese Funktion Lokalisationsinformationen des Absenders ermitteln kann, kennt dieses Modul nun auch automatisch die komplette Adresse eines anderen Moduls und nicht nur den Knoten.

Befinden sich nun alle Funktionsmodule einer Konfiguration im Ready-Zustand, kann das erste Modul in der Verarbeitungsreihe mit der Verarbeitung von Daten beginnen und diese an seine nachfolgenden Module versenden.

5.6 Benutzerdefinierte Variablen

Wie in den vorherigen Abschnitten angedeutet, lassen sich für jedes Modul beliebig viele benutzerspezifische Variablen speichern. Diese Variablen können entweder in der Konfigurationsdatei oder in dem Modul selbst definiert werden. Sie sind über das *ModuleInfo*-Interface zugänglich.

Quellcode 5.3: ModuleInfo-Interface

```
class ModuleInfo{
2  private:

4    typedef list<struct mod_info_t*> ModuleInformation;
     ModuleInformation ModInfo;
6    Modulecore *ModCore;

8    mod_info_t *GetInfo(char *vname);
```

```
10    public:

12      ModuleInfo();
        virtual ~ModuleInfo();
14      void std_cb_func(void *v);
        void message_listvar(msginfo_t *info);
16      void message_getvalue(msginfo_t *info);
        void message_setvalue(msginfo_t *info);
18      void RegisterVar(char *vname,
                         void *vdata,
20                       unsigned int vtype,
                         short unsigned int vaccess,
22                       TInfoFunktor *cb_func);
        template <typename vtype>
24      void RegisterVar(char *vname,
                         vtype vdata,
26                       unsigned int vtype,
                         short unsigned int vaccess,
28                       TInfoFunktor *cb_func);
        void UnRegisterVar(char *vname);
30      var_info_t *ListVar(unsigned short int *num);
        void *GetVarValue(char *vname);
32      void ChangeValue(char *vname,
                         void *vvalue);
34      void ChangeVar(char *vname,
                       unsigned int vtype,
36                     short unsigned int vaccess,
                       TInfoFunktor *cb_func);
38  };
```

In diesem Interface sind Methoden implementiert, die Variablen registrieren, löschen und die Werte der Variablen ändern können. Mit der Funktion *RegisterVar* werden Variablen registriert und der Zeiger auf die Daten übergeben. Es wird kein eigener Speicher für Variablen belegt. Dieser muss vor der Registrierung der Variablen schon reserviert worden sein. Für den schreibenden Zugriff auf die Variablen kann eine "*Callback*"-Funktion hinterlegt werden. Sie wird genau dann aufgerufen, wenn die Variable mit der Funktion *ChangeValue* geändert wird. Beim Zugriff auf eine Variable wird nur der Pointer auf die Daten zurückgeliefert. Eine Typisierung muss durch den Benutzer stattfinden. Es können auch der Zugriffsschutz, Typ und die "*Callback*"-Funktion zur Laufzeit für eine angegebene Variable geändert werden.

5.7 RPC-Behandlung

Im Abschnitt 5.2 ist die allgemeine Funktionsweise des Datenaustausches erklärt. Ein Datenaustausch kann auch als *Remote-Procedure-Call* (RPC) oh-

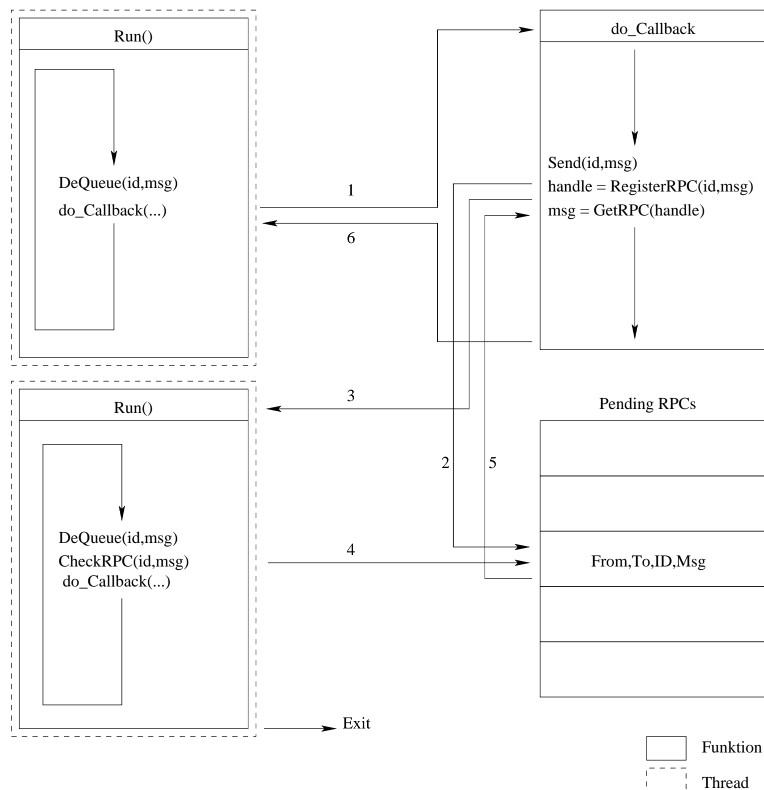

Abbildung 5.1: Funktionsweise des RPC-Managers

ne Rückgabewert interpretiert werden. Folglich muss es auch einen Mechanismus geben, mit dem RPCs erstellt werden können, die Rückgabewerte liefern. Jedoch war dies mit der bisherigen Verarbeitungsweise einer ankommenden Nachricht nicht möglich. Wird zu einem Zeitpunkt eine Nachricht verarbeitet, können keine anderen Nachrichten bearbeitet werden. Es können nur weitere Nachrichten in die Eingangswarteschlange integriert werden. Die Verarbeitung der ankommenden Nachrichten läuft somit synchron. Wird nun in dieser Umgebung eine RPC mit Rückgabewert aufgerufen, so kann die Funktion aufgerufen werden, aber sie liefert niemals einen Wert zurück, da die Nachrichtenverarbeitung blockiert ist. Zur Vermeidung wurde die Nachrichtenverarbeitung um einen RPC-Manager erweitert. Die Funktionsweise des RPC-Managers ist in Abbildung 5.1 dargestellt. Der RPC-Manager integriert sich an verschiedenen Punkten der Nachrichtenverarbeitung. Der Ablauf einer synchronen RPC ist in der Abbildung 5.1 gezeigt. In Schritt 1 wird eine Nachricht aus der Eingangswarteschlange entnommen und über den Callback-Mechanismus behandelt. In dieser Nachrichtenbehandlungsroutine werden nun Daten einer RPC benötigt. Dazu versendet die Funktion eine Nachricht an die entfernte Prozedur. Nun wird in Schritt 2 ein Eintrag in einer Liste für RPCs erstellt. In diesem Eintrag ist hinterlegt, auf welche Nachricht gewartet werden soll. Im 3. Schritt erzeugt der RPC-Manager einen neuen Thread zur Verarbeitung der ankommenden Nachrichten. Damit wird sichergestellt, dass auch weiterhin Nachrichten aus der Warteschlange entnommen und verarbeitet werden können. Nachdem der

neue Thread erstellt und ausgeführt wurde, blockiert sich die Funktion mit Hilfe einer Semaphore selbst. Nun werden Nachrichten, die in der Warteschlange stehen, von dem neuen Thread abgearbeitet. Befindet sich eine Antwort auf eine RPC unter den Nachrichten, bemerkt dies der RPC-Manager in Schritt 4. Handelt es sich um eine RPC-Antwort, so wird die Nachricht in dem Eintrag der wartenden RPCs hinterlegt und die entsprechende Semaphore deblockiert. Nun kann die RPC im 5. Schritt die Nachricht aus dem Puffer nehmen und verarbeiten. Nachdem die RPC verarbeitet ist und die "*Callback*"-Funktion ebenfalls beendet ist, wird der Thread, der vom RPC-Manager erzeugt wurde, beendet. Dies geschieht über den Rückgabe-Code der *CheckRPC*-Funktion. Denn nun kann die eigentliche Schleife des Moduls weiterlaufen, da der RPC Aufruf beendet wurde. Im 6. Schritt kehrt die "*Callback*"-Funktion zurück und es kann eine neue Nachricht aus der Warteschlange entnommen werden. Findet hingegen ein neuer RPC-Aufruf in einer "*Callback*"-Funktion statt, so beginnt der RPC-Mechanismus bei Schritt 2.

5.8 Logger

In Abschnitt 4.4.7 wird ein Logger auf Basis einer DHT erörtert. Ebenso wird die Funktionsweise des Loggers dort erklärt.
Entgegen diesem Entwurf ist der Logger nicht auf Basis einer DHT implementiert, da die Implementierung einer DHT den Umfang dieser Arbeit überschritten hätte[4]. Die Log-Einträge werden stattdessen an das Syslog-Interface von Linux/Unix übergeben. Des Weiteren besteht der Logger aus zwei Teilen; einem Clientteil und einem Serverteil. Der Serverteil implementiert das Syslog-Interface und nimmt die Log-Nachrichten aus dem Framework entgegen. Der Clientteil der Loggers ist in das Interface eines Modules eingebettet. Damit ist es möglich, dass innerhalb eines Moduls Log-Nachrichten durch einen Methodenaufruf erzeugt werden. Im Hintergrund wird aus diesem Aufruf eine Nachricht erstellt und an den Serverteil gesendet. Der Logger muss auf jedem Knoten ausgeführt werden. Er kann über Variablen so gesteuert werden, dass die Speicherung der Nachrichten nur auf einem Knoten stattfindet, ein lokaler Logger nur als Proxy dient und die Nachrichten einfach weiterreicht. In der Tabelle 5.8 werden die Variablen des Loggers beschrieben. Es handelt sich bei diesem Parametern ausschließlich um Umgebungsvariablen des Serverteils. Der Clientteil besitzt keine Umgebungsvariablen.

5.9 Beispiel-Modul

Anhand des TCP-Analyse Moduls wird die Implementierung eines Moduls beispielhaft gezeigt. Der funktionelle Umfang dieses Modul entstand im Rahmen einer Studienarbeit (siehe auch [Zülc04]). Diese Arbeit wurde an das Framework soweit angepasst. In dem folgenden Code-Auszug wird beispielhaft der funktionale Teil eines Moduls gezeigt.

[4]Es handelt hierbei um eine prototypische Implementierung, in der nicht alle Funktionen implementiert sind

Tabelle 5.2: Variablendefinitionen des Logger

Parameter	Typ	Beschreibung
location	VAR_STRING	Angabe auf welchem Knoten die Speicherung der Nachrichten stattfindet. Handelt es sich um den Eintrag *local* findet die Speicherung lokal statt und die folgenden Variablen müssen definiert sein: *prefix, options, facility*. Handelt es sich um den Eintrag *remote*, so muss die Variable *host* eingetragen sein.
prefix	VAR_STRING	Definiert einen Präfix, der vor alle Einträge geschrieben wird.
options	VAR_STRING	ist eine mit Leerzeichen getrennte Liste der folgenden Werte: LOG_NDELAY, LOG_NOWAIT, LOG_PID, LOG_CONS, LOG_PERROR, LOG_ODELAY.
facility	VAR_STRING	Gibt das Ziel der Einträge an. Der Wert muss aus einem der Facility-Werte des Syslog-Interfaces bestehen.
host	VAR_STRING	Gibt den Knotennamen an auf dem die Nachrichten gespeichert werden sollen.

Quellcode 5.4: Modulbeispiel anhand des TCP-Analyse Moduls (1)

```
TCPAnalyse::TCPAnalyse(void *dummy) :
                ModuleInterface(dummy) {
        module_init(this);
        RegisterMsgType(MSG_PCAP_PACKET,
                ModuleID,
                wrap(this,&TCPAnalyse::message_packet_recv));
}

void TCPAnalyse::InputMangle(msginfo_t* info) {
        stringstream *msg;
        unsigned int protocols;

        if (info->mdata) {
                msg = info->mdata;
                msg->read((char*)&protocols,4);
                if (protocols & PROTO_TCP != PROTO_TCP) {
                        printf("Paket ohne TCP-Header empfangen");
                        delete info;
                        info = NULL;
                }
        }
```

```
22 }

24 void TCPAnalyse::message_packet_recv(IfaceMsg* info) {
       struct timeval ts;
26     char *hdrs;
       unsigned short int ilen, tlen;
28     unsigned short int glen = 0;
       unsigned int protocols;
30     stringstream *msg = info->mdata;

32     msg->read((char*)&protocols,4);
       msg->read((char*)&ts,8);
34     msg->read((char*)&ilen,2);
       glen = ilen;
36     msg->read((char*)&tlen,2);
       glen += tlen;
38     hdrs = new char[glen];
       msg->read((char*)hdrs,glen);
40
       module_run(this,ts,(struct iphdr*)hdrs);
42
       delete hdrs;
44     delete info;
   }
46
   TCPAnalyse::~TCPAnalyse() {
48     UnRegisterMsgType(MSG_PCAP_PACKET, ModuleID);
       module_shutdown();
50 }
```

In dem folgenden Auszug werden die drei Methoden implementiert, die der Loader aufruft, wenn das Modul gestartet wird. Diese drei Methoden müssen in der entsprechenden Header-Datei als *extern "C"* definiert werden, damit der dynamische Loader die Symbole in der Shared-Library findet und kein sogenanntes *Namemangling*, wie in C++ üblich, stattfindet.

Quellcode 5.5: Modulbeispiel anhand des TCP-Analyse Moduls (2)

```
  #include "tcpanalyzer.h"
2 #include "TCPAnalyse.h"

4 void Module_tcp_Init(Library *lib,void *dummy) {
      if (lib) {
6         try {
              lib->SetClass(new TCPAnalyse(dummy));
8         } catch (...) {
              throw Modul_Exception(MODUL_UNKNOWN,
10                                  lib->GetFilename());
```

```
        }
12    }
}
14
void Module_tcp_Run(Library *lib, void *dummy) {
16    if (lib) {
        try {
18            lib->GetClass()->Run();
        } catch (...) {
20            throw Modul_Exception(MODUL_UNKNOWN,
                                     lib->GetFilename());
22        }
    }
24 }

26 void Module_tcp_Done(Library *lib, void * dummy) {
    if (lib) {
28        try {
            delete(lib->GetClass());
30        } catch (...) {
            throw Modul_Exception(MODUL_UNKNOWN,
32                                     lib->GetFilename());
        }
34    }
}
```

Wie in den Code-Ausschnitten zu sehen ist, reichen diese Funktionen aus, um ein vollständiges Modul zu erhalten. Dazu wird eine neue Klasse erzeugt, die von der Klasse *ModuleInterface* abgeleitet wird. In dieser Klasse müssen für ein einfaches Modul nur der Konstruktor und der Destruktor überschrieben werden. In dem Konstruktor wird eine Behandlungsmethode für die Nachricht vom TYP MSG_PCAP_PACKET registriert. Dabei wird die Klassenmethode angegeben, die aufgerufen werden soll, wenn eine solche Nachricht vom Modul empfangen wird. Im Destruktor der Klasse wird die Registrierung rückgängig gemacht und das Modul wird beendet. Die beiden anderen Methoden dienen der Bearbeitung der Nachrichten. In der Methode *InputMangle* findet eine Filterung der Nachrichten statt. Der Filter löscht alle Nachrichten, die keine TCP-Header Informationen enthalten. Damit wird sichergestellt, dass das Modul nur Nachrichten verarbeitet, in denen TCP-Header Informationen enthalten sind. In der eigentlichen Nachrichtenbehandlungsfunktion werden empfangene Nachrichten interpretiert und die Verarbeitung der Daten wird vorgenommen. Dieses Modul erzeugt wiederum eigene Nachrichten, die an weitergehende Module gesendet werden können. Der Aufbau der Nachrichten leitet sich aus dem Aufbau der Einträge ab, die in der Studienarbeit in Dateien geschrieben wurden. Diese Nachrichten wurden um Kontextinformation, d. h. um welche Verbindung es sich handelt, erweitert.

5.10 Exception-Generierung

In dem Framework sind zur Behandlung von Fehlern *Exceptions* implementiert. Diese *Exceptions* werden ausgelöst, wenn in einem Modul Fehler auftreten. Wird eine Exception ausgelöst, so erscheint der Auslöser der Exception auf dem Bildschirm. Alle Exceptions sind von der Basisklasse *class Base_Exception* abgeleitet. Eine beispielhafte Anwendung der Exceptions ist im vorherigen Abschnitt zu erkennen. Dort werden alle Exceptions abgefangen, die in einem Modul ausgelöst werden können. Es werden für folgende Ereignisse Exceptions erzeugt und abgefangen:

- Socket-Fehler (lesen, schreiben, erzeugen)
- Modul-Fehler (alle während der Ausführung)
- Modul-Laden (Library laden, Symbole nicht gefunden, ...)
- Threaderzeugung
- Konfiguration-Laden (Datei nicht gefunden,...)
- Pcap-Fehler (keine Rechte, Fehler im Filter,...)

Exceptions, die in Socket-Fehlern, Modul-Laden oder Konfiguration-Laden ihre Ursache haben, werden in Nachrichten an den lokalen Loader umgewandelt. Dieser hat nun die Möglichkeit, mit Hilfe dieser Ereignisse geeignete Maßnahmen durchzuführen, um andere Module oder Knoten von dem Fehler in Kenntnis zu setzen und evtl. ihn selbst zu beheben. Bei einer Socket-Exception (z.B. Socket kann nicht geöffnet werden) versucht ein Modul erst dreimal einen Socket zu öffnen (Zwischen jedem Versuch wartet das Modul 30 Sekunden). Kann auch nach dem 3. Versuch keine Verbindung geöffnet werden, wird eine weitere Exception ausgelöst.

Um eigene Exceptions zu implementieren, ist eine Klasse zu erzeugen, die von einer der folgenden Klassen abgeleitet ist:

- Base_Exception
- ERRNO_Exception
- Fault_Exception

Ein Beispiel für eine Exceptionklasse ist in dem folgenden Codeausschnitt zu finden:

Quellcode 5.6: Beispiel Exception an Hand der Modul-Exception

```
class Modul_Exception : public Fault_Exception {
    public:
        Modul_Exception(int cause, const char *s) :
```

```
        Fault_Exception(cause, mkwhat(cause,s)) {};

    private:
        string mkwhat(int cause,const char *s) {
            ostringstream out;

            if (cause == MODUL_UNKNOWN) {
                out << "Modul: ... " << s << endl;
            }
            if (cause == MODUL_INITFAILED) {
                out << "Modul: ... " << s << endl;
            }
            if (cause == MODUL_RUNFAILED) {
                out << "Modul: ... " << s << endl;
            }
            if (cause == MODUL_DONEFAILED) {
                out << "Modul: ... " << s << endl;
            }
            return out.str();
        };
};
```

Auf den Fehlerzeichenkette kann per *const char *what()* zugegriffen werden. Zusätzlich steht in den jeweiligen abgeleiteten Klassen von *ERRNO_Exception* und *Fault_Exception* die Funktion "*const int ExceptionCode()*" zur Verfügung. Sie hat im ersten Fall immer den Wert der Variablen *errno*, im zweiten Fall einen benutzerspezifischen Wert.

5.11 Management

Aus dem Kapitel Entwurf geht hervor, dass ein Management des Frameworks sowohl ein Management für einen Knoten als auch für ein Modul existiert. Das Management für ein Modul ist in das Modul selbst integriert. Überwiegend handelt es sich dabei um Funktionen zum Auslesen oder Setzen von Variablenwerten. Diese Grundfunktionen für das Management sind in jedes Modul integriert. Erweiterte Managementfunktionen sind von dem Modul selbst zu implementieren. Es wird dafür kein spezielles Interface bereitgestellt. Das Management lässt sich per Nachrichtengenerierung und Reaktion auf Nachrichten implementieren.
Zum anderen gibt es ein Management für einen Knoten selbst. Dieses Management ist durch ein eigenes Modul repräsentiert. Es wird in Kapitel 4.9.2 geschildert. Um auf das Management zuzugreifen und einen Knoten zu managen, existiert ein Client-Programm, welches sich per Socket mit einem Knoten verbindet und sowohl den Knoten also auch alle anderen Knoten administrieren kann. Das Managementprogramm besteht aus einem TCL-Interpreter[5] und

[5] www.tcl.tk

Grundbefehlen, die darin implementiert wurden. Das Managementprogramm lässt sich durch Bibliotheken erweitern und somit mit neuen Befehlen ergänzen. Die Bibliotheken lassen sich entweder per Konfiguration zum Programmstart laden, oder sie lassen sich zur Laufzeit per Kommando nachladen. Die Bibliotheken müssen dabei folgendes Interface implementieren, um geladen werden zu können.

Quellcode 5.7: Management-Bibliothek Interface

```
void Command_mloader_Loading(LibLoad *lib,
                             Tcl_Interp *dummy) {
    if (dummy) {
        Tcl_CreateCommand(dummy,
                          "help_loader",
                          cmd_loader_help,
                          (ClientData)0,
                          (Tcl_CmdDeleteProc*)NULL);
    }
}

void Command_mloader_UnLoading(LibLoad *lib,
                               Tcl_Interp *dummy) {
    if (dummy) {
        Tcl_DeleteCommand("help_loader");
    }
}
```

Innerhalb dieser Methoden können neue TCL-Kommandos implementiert und registiert werden.

Für die Kommunikation mit einem Knoten steht ein Socket (Skt) zur Verfügung, der genutzt werden kann. Über den Socket werden Nachrichten gesendet, die das gleiche Format haben wie die Nachrichten innerhalb des Frameworks. So ist es einfach, Nachrichten zu erzeugen, zu versenden und zu empfangen. Da der Management-Client eine Erweiterung eines TCL-Interpreters darstellt, ist es möglich diesen Client in einer Script Datei anzugeben und eine Sequenz von Befehlen daran anzuschließen. Ein typisches Script sieht wie folgt aus:

Quellcode 5.8: TCL-Script des Management-Clients

```
#! ./manager

loadcmd mod_msmux.so
loadcmd mod_mloader.so

connect 10.15.0.2

smuxconnect 10.10.2.10 4711
smuxconnect 141.3.41.250 4711
```

```
10 smuxconnect 141.4.41.250 4711 router.bzuelch.de

12 loadconfig withvgl.cfg
```

Dieses Script startet zuerst das Managementprogramm. Danach werden zusätzliche Kommandos aus Bibliotheken geladen. Dies geschieht mit Hilfe des Kommandos *loadcmd*. Nachdem die beiden Bibliotheken geladen wurden, stehen weitere Befehle für den *Loader* und *SocketMultiplexer* zur Verfügung. Anschließend wird eine Verbindung zu einem Knoten aufgebaut (*connect*. Es muss sichergestellt sein, dass auf der angegebenen IP-Adresse ein Knoten ausgeführt wird. Anschließend werden Verbindungen zwischen drei Knoten aufgebaut. Damit ist das Framework in der Lage Nachrichten am entfernte Knoten zu senden. Zum Verbinden müssen IP-Adresse und Port-Nummer angegeben werden. Wird zusätzlich noch ein Knotenname angegeben wird dieser Befehl auf diesem Knoten ausgeführt. Vorher muss aber eine Verbindung zu diesem Knoten existieren, sonst werden Nachrichten an diesen Knoten gelöscht, da sie nicht zugestellt werden können. Als letzter Befehl wird eine Konfigurationsdatei dem Loader übergeben, diese wird vom Loader geladen und die Module startet. Danach beendet sich das Script und das Framework verrichtet seine Arbeit im Hintergrund.
Wird das Management-Programm ohne ein Script direkt aufgerufen, so stehen noch weitere Befehle zur Verwaltung des Managements zur Verfügung. Eine Übersicht über diese Befehle sind mit den Befehlen *help* oder *help*_**MODUL** zu bekommen[6].

[6] Es sind die Befehle help, help_loader und help_smux verfügbar.

6. Ergebnisse

In diesem Kapitel werden die Ergebnisse der Implementierung des in Kapitel 4.1 entworfenen Frameworks beschrieben. Dazu werden einige kurze Beispiele vorgestellt, wie das Framework die Anforderung erfüllt und als Grundlage für skalierbare verteilte Messungen und Analysen dient. In den weiteren Teilen des Kapitels werden Ergebnisse einiger Messungen und Analysen anhand von implementierten Modulen präsentiert.

6.1 Framework

Wie aus dem Entwurf des Framworks hervorgeht, soll es zur skalierbaren und verteilten Analyse von Internetverkehr eingesetzt werden. Dazu wurde eine Software entworfen und realisiert. In dieser Software wird ein Rahmenprogramm implementiert, welches eine verteilte Messung und Analyse ermöglichen kann. Der Entwurf in Kapitel 4.1 geht über die realisierte Variante hinaus. In der Implementierung fehlen die Mechanismen, die die Verteilung einzelner Module automatisieren. Das bedeutet; der Ort eines Moduls muss fest per Konfiguration vorgegeben sein, damit das Modul auf einem Knoten gestartet wird. Es findet eine Suche nach einem geeignetem Knoten statt. Hiefür fehlt eine gemeinsame Datenhaltung, die es gestattet, eine Suche über geeignete Knoten durchzuführen. Um die Verteilung von einzelnen Funktionsmodulen vorzunehmen, wurde ein einfaches Kommunuikationsmodul auf Basis von TCP-Sockets implementiert. Dieses Modul kann Verbindung zu verschiedenen Knoten aufbauen. Jedoch ist kein Routing implementiert, welches die Kommunikation transitiv über mehrere Knoten hinweg erlauben würde. Damit einzelne Knoten miteinander kommunizieren können, ist eine direkte Verbindung zweier Knoten nötig. Des Weiteren existiert der Loader, der eine angegebene Konfiguration laden kann. Eine Konfiguration besteht aus einer Anzahl von Modulen, die für eine Analyse benötigt werden. Für jedes Modul wird in dieser Konfiguration festgelegt, mit welchen anderen Modulen es kommuniziert und auf welchem

Knoten es gestartet werden soll. Des Weiteren kann ein Funktionsmodul beliebige Variablen registrieren und kann auf diese in dem Modul über ein Interface zugreifen. Der Aufbau einer Konfiguration ist in Anhang A geschildert. Dort ist auch eine Beispielkonfiguration gezeigt. Weitere Konfigurationen finden sich auf der beiliegenden CD. Zu dem eigentlichen Knotenprogramm wurde ein Managementprogramm entwickelt, welches sich mit einem beliebigen Knoten verbinden kann. Ist es mit einem Knoten verbunden, können über dieses Programm verschiedene Parameter abgerufen werden. Es sind jedoch nicht alle Parameter zur Administration implementiert.

Um eine verteilte Messung und Analyse durchzuführen, ist auf verschiedenen Knoten das Framework zu starten. Jede Instanz eines Knotens kann durch eine Startkonfigurationsdatei an einen Rechner angepasst werden. Ist keine Konfiguration als Kommandozeilenparameter angegeben, so wird die Standardkonfigurationsdatei gesucht und geöffnet. Nachdem auf mehreren Knoten das Framwork ausgeführt wird, kann eine Konfiguration geladen werden. Die Konfiguration beinhaltet die Module, die zur Messung und Analyse notwendig sind. Es sind folgende Funktionsmodule implementiert:

- PCap - Paketerfassung
- TCP-Analyse - Analyse von TCP-Strömen [1]
- TCP-Sampler - Sampling von RTT und TTL - Messwerten
- Differenz - Berechnung der Differenz zweier RTT oder TTL-Werte
- sog. Korrelator - Suche nach Veränderungen in RTT oder TTL-Werten

Hinzu kommen zu diesen Funktionsmodulen noch einige Module, die die berechneten Ergebnisse in Dateien ausgeben können.

Sind nun die angegeben Module einer Konfiguration auf den verschiedenen Knoten gestartet, kann das erste Modul in einer Analysekette mit der Ausführung seiner Funktionen beginnen. Dies wird über den Modulzustand gesteuert, der in Kapitel 4.8 definiert wurde.[2] Nun werden die von einem Modul erzeugten Nachrichten entlang der Analysekette gesendet, d.h. die von einem Modul erzeugten Nachrichten werden an alle Folgemodule weitergereicht, die auf Daten von diesem Modul warten. Die Module greifen wie in einem Lego-Baukasten ineinander und bilden, bestimmt durch die Konfiguration, eine gesamte Messung und Analyse.

6.2 Analysebespiel

In der Abbildung 6.1 ist ein beispielhafter Aufbau einer Konfiguration gezeigt. Darin werden Pakete mit dem *Pcap-Modul*[3] erfasst und an ein *TCP-Analyse-Modul* weitergleitet. Dieses führt eine Analyse des ankommenden TCP-Stroms

[1] Dieses Modul wurde zu großen Anteilen aus der Studienarbeit [Zülc04] übernommen.

[2] So wartet das Pcap-Modul z.B. darauf, dass alle Module gestartet sind, bevor es mit der Erfassung der Pakete beginnt.

[3] siehe dazu auch in Kapitel 4.10.1

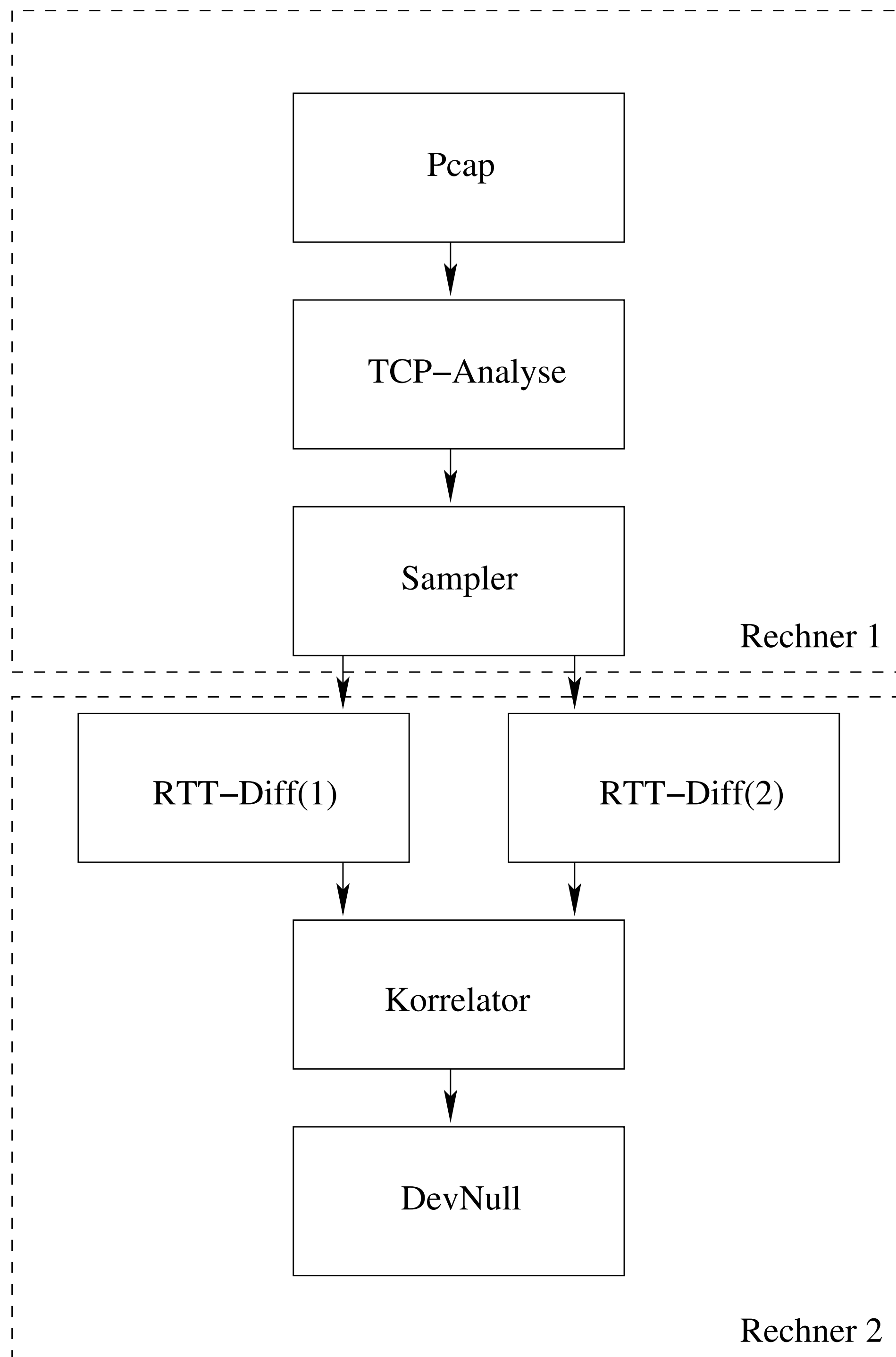

Abbildung 6.1: Analyseszenario für eine Analye

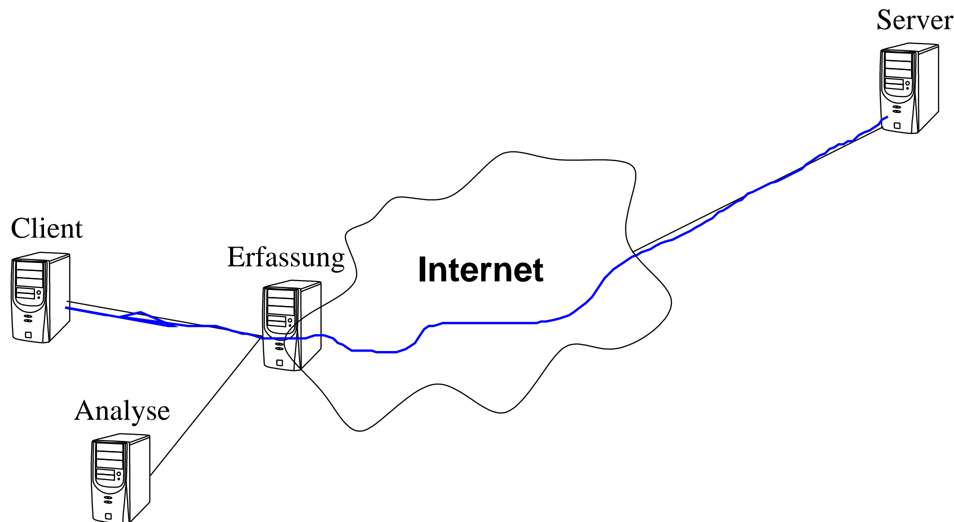

Abbildung 6.2: Messaufbau

durch. Die Funktionsweise des Moduls entspricht der, in [Zülc04] geschilderten. Nach der Analyse eines TCP-Stroms werden die Ergebnisse der TTL- und RTT-Berechnung an einen Sampler weitergereicht. Der Sampler reduziert die Datenmenge der berechneten TTL- und RTT-Werte in dem er nicht mehr jeden Wert weiterreicht, sondern nur noch jeden x-ten Wert. Dieser Wert kann über die Konfiguration gesteuert werden. Nachdem der Sampler passiert ist, werden die Pakete, die nicht gelöscht wurden, an die folgenden *Differenz-Erzeuger* weitergereicht. Jeder dieser *Differenz-Erzeuger* berechnet die Differenz zwischen dem aktuellen RTT- oder TTL-Wert und einem per Konfiguration vorgegebenen älteren Wert. Es können beliebig viele dieser Module parallel gestartet werden und somit gleichzeitig die Differenz zwischen mehreren verschiedenen Werten berechnet werden. Diese Berechnung dient der Erfassung von Änderungen in den Werten von TTL und RTT über die Zeit hinweg und soll somit Auskunft darüber geben, in wieweit sich die TTL- und RTT-Werte über die Zeit ändern. Überschreiten die Differenzen einen konfigurierbaren Wert, so kann von einer RTT-Änderung oder TTL-Änderung ausgegangen werden. Die Änderung kann verschiedene Ursachen haben. Die Ursachen werden in Kapitel 3.6 erörtert. Um nun von dieser RTT oder TTL-Änderung sprechen zu können, werden die Ergebnisse in dem sog. Korrelator verglichen und ausgewertet. Da jeder *Differenz-Erzeuger* die Differenz mit einem verschiedenen Wert berechnet, wird nun überprüft, ob eine Änderung eines Wertes nur bei einer Differenz erkannt wurde, oder ob mehrere Differenzen eine Änderung aufweisen. Dazu wird in dem Korrelator angegeben, um wieviel Prozent sich die Werte ändern müssen, um als Änderung erkannt zu werden. Ist der Wert zu klein gewählt, werden vermeintliche Änderungen erkannt, die jedoch keine sind, da sie einem Rauschen der Ergebnisse unterworfen sind und eigentlich keine Änderung darstellen. Hinzu kommt ein Wert, der angibt, bei wie vielen Differenzen eine Änderung erkannt werden muss. Die in Abbildung 6.1 gezeigte Analyse wur-

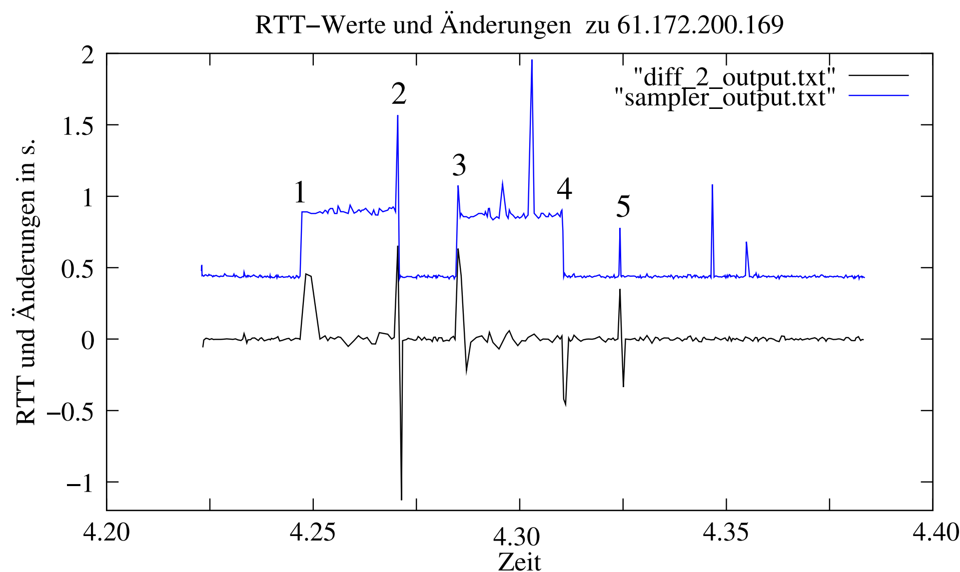

Abbildung 6.3: Messergebnisse des Samplers und des Differenz-Erzeugers

de gegen den Internet Server *mirrors.geekbone.org (61.172.200.169)* getestet. Dazu wurde der in Abbildung 6.2 gezeigte Messaufbau genutzt. Der Client, der den Download durchführt, ist über ein Ethernet-LAN mit einem Router verbunden. Dieser ist per DSL[4] an das Internet angebunden. Gleichzeitig dient der Router als Rechner für die Erfassung der Pakete. Somit entspricht dieser Rechner dem "Rechner 1" in der Abb. 6.1. Als "Rechner 2" fungiert der als "Analyse" gekennzeichnete Rechner. In der Abbildung 6.3 ist in den beiden Messverläufen zum einen der Verlauf der RTT und zum anderen der Verlauf der Änderung der RTT aufgetragen. Der Verlauf der RTT wurde einem Sampler und die Änderung der RTT-Werte einem der Differenz-Erzeuger entnommen. Bei dieser Messung wurden die Module mit folgenden Parametern gestartet:

- Sampler: jedes 15. TCP-Paket je Richtung wird weitergleitet.
- Differenz-Erzeuger: Berechnung der Differenz zu je 2,4,6 und 8 Pakete(n) älter als das aktuelle Paket.
- Korrelator: Änderung der RTT um mind. 8 Prozent des alten Wertes gegenüber dem aktuellen Wert.

Wie in der Abbildung 6.3 zu sehen ist, konnten einige Ereignisse beobachtet werden. Die Verbindung hatte zu Beginn eine stabile RTT von ca. 440ms. Zu Zeitpunkt 1 fand ein Sprung der RTT auf ca. 900ms. statt. Zu diesem Zeitpunkt wurde ein Paralleldownload zu einem anderen Server im Internet gestartet, so dass die komplette DSL-Leitung ausgelastet war. Diese Änderung ist auch deutlich im Differenz-Erzeuger daran zu erkennen, dass es zu diesem Zeitpunkt

[4]Der DSL-Anschluss verfügt über eine Übertragungsrate von maximal 1MBit/s.

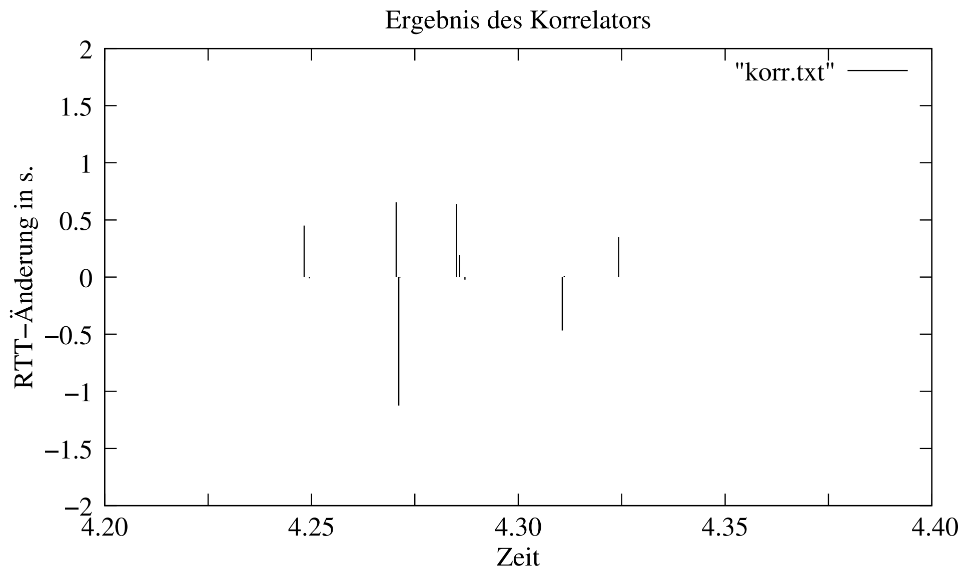

Abbildung 6.4: Messergebnisse des Korrelators

einen Peak in der Änderung der RTT-Werte gegeben hat. Danach (zwischen Zeitpunkt 1 und 2) ist festzustellen, dass die RTT-Werte sich stärker ändern als zu Beginn der Verbindung. Zum Zeitpunkt 2 wurde der Paralleldownload abgebrochen und dir RTT fiel auf das ursprüngliche Niveau zurück. Zwischen Zeitpunkt 3 und 4 wurde erneut ein Paralleldownload gestartet. Er lastete die Kapazität der Internetverbindung erneut aus, dies ist an einem schwankendem Verlauf der RTT-Werte zu erkennen. Zum Zeitpunkt 5 wurde eine Ping-Flood auf den Paketerfassungsrechner gestartet, was an einer kurzen Änderung der RTT zu erkennen ist. Aus Abbildung 6.4 ist der Einfluß des Korrelators auf die obrige Messung zu ersehen. Dieser erzeugt nur eine Ausgabe (in dieser Konfiguration), wenn mind. drei der vier Differenz-Erzeuger eine Differenz von mind. 8 Prozent zu einer vergangenen RTT berechnet haben. Die Ergebnisse des Korrelators können nun noch in einen einheitlichen Wert gewandelt werden, der als Event dienen kann. Im Event werden Informationen über Zeitpunkt und Änderungsrichtung zusätzlich gespeichert.

6.3 Verteilte Messung

In einer weiteren Konfiguration wurde eine verteile Messung und Analyse durchgeführt. Bei dieser Konfigurtation waren drei Rechner beteiligt. Zwei Rechner befanden sich hinter einem DSL-Anschluss. Ein weiterer Rechner ist über das Universitätsnetz angeschlossen und verfügt über einen schnellen Zugang zum Internet. Die beteiligten Rechner sind in Abbildung 6.6 dargestellt. Dabei sind die Rechner, die als Analyse, Client und Erfassung gekennzeichnet sind, diejenigen, die hinter dem DSL-Anschluss installiert sind. Die als "Erfassung 2" und "Client 2" gekennzeichneten Rechner befinden sich innerhalb des Universitätsnetzes. Auf drei dieser Rechner wird das Framework ausgeführt.

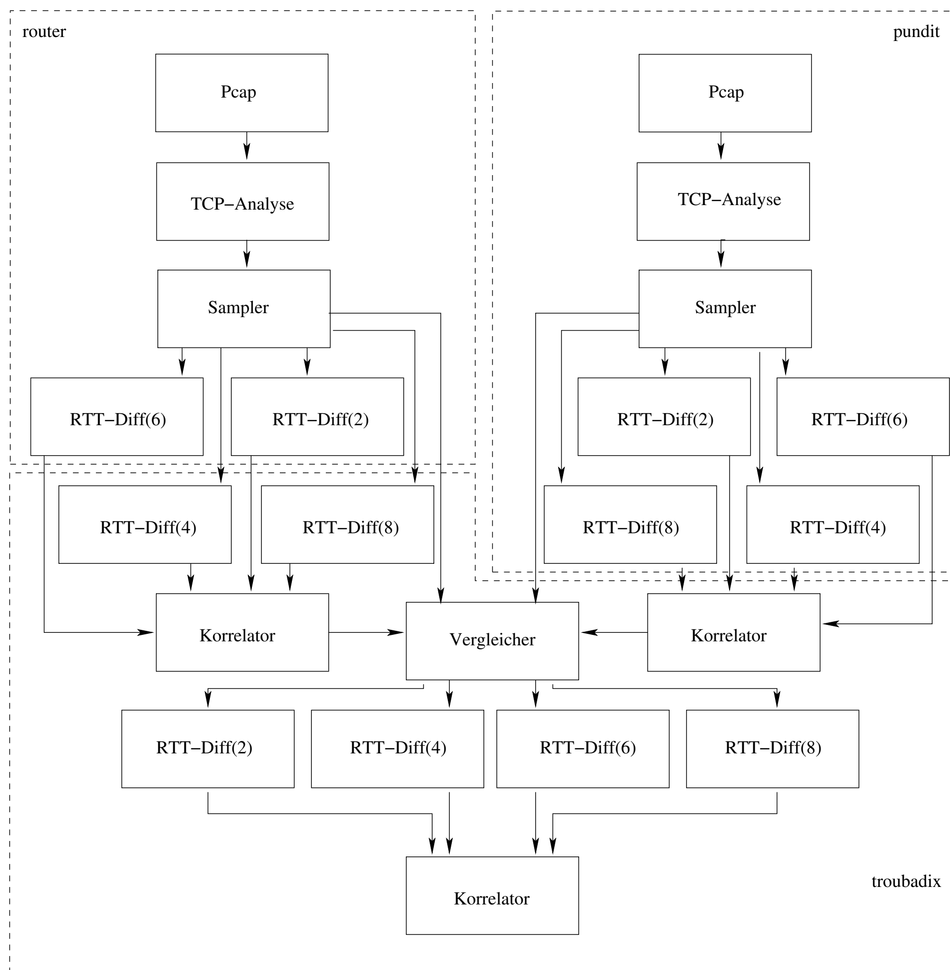

Abbildung 6.5: Konfiguration für eine verteilte Messung und Analyse

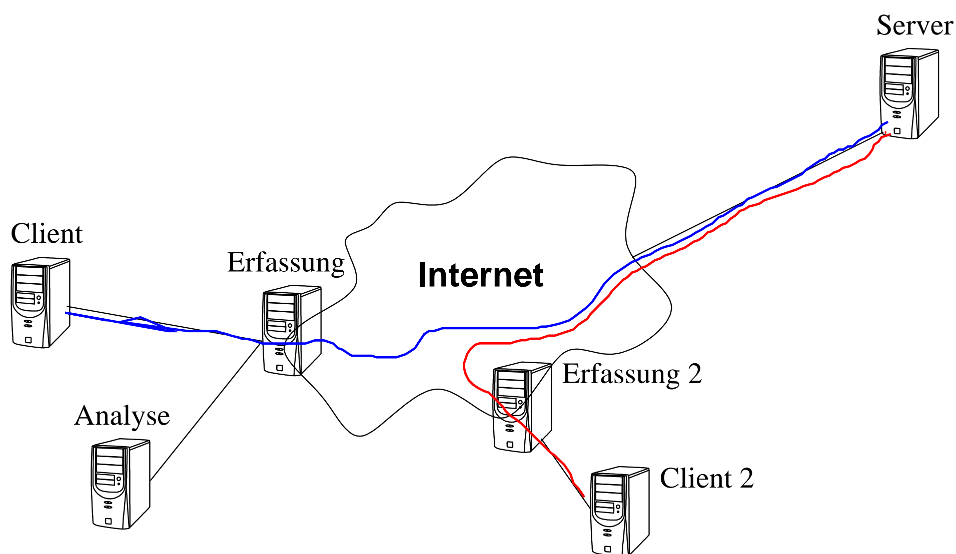

Abbildung 6.6: beteiligte Rechner in der verteilten Messung

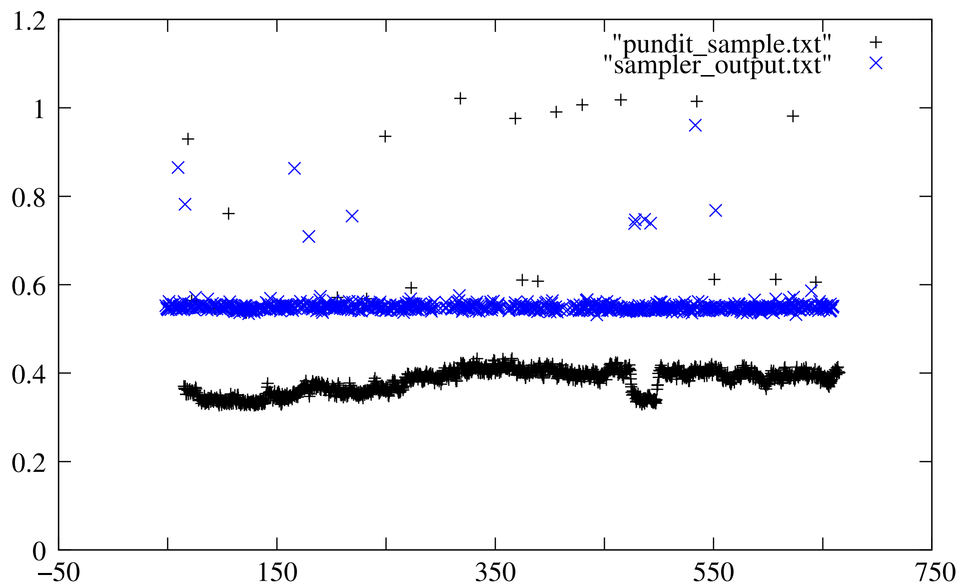

Abbildung 6.7: Überlagerung der Samplerergebnisse der RTT

Dies sind die Rechner Analyse (troubadix), Erfassung (Router) und "Erfassung 2" (Pundit). Die jeweils auf den Rechnern gestarteten Module und ihre Verknüpfungen gehen aus der Abbildung 6.5 hervor. Bei dieser Konfiguration werden in zwei verschiedenen Rechnern Pakete mit Hilfe des Pcap-Moduls erfasst. Die erfassten Pakete werden dann mit dem TCP-Analyse-Modul analysiert. Danach werden die Ergebnisse durch einen Sampler in ihrem Umfang reduziert. Anschließend werden die RTT-Werte auf ihre zeitliche Änderung hin untersucht, die durch die RTT-Differenz-Erzeuger berechnet werden. Es folgt eine Auswertung dieser Werte in den jeweiligen Korrelatoren. Der Korrelator meldet ein Ereignis, wenn in bei mind. drei der vier Differenz-Erzeuger eine Veränderung der RTT vorliegt. Die Veränderung der RTT. kann je nach Wunsch eingestellt und verändert werden. Diese wird jeweils auf den beiden erfassenden Systemen durchgeführt. Erst die gemeinsame Analyse der beiden untersuchten Ströme findet auf dem dritten Rechner statt. Auf dem dritten Rechner wird sowohl untersucht, ob Ereignisse der jeweiligen Korrelatoren zu gleichen Zeitpunkten auftreten, oder ob sie in einer Verbindung miteinander stehen.

Diese Konfiguration wurde gegen den Internetserver *mirrors.geekbone.org* getestet. In Abbildung 6.7 ist der Verlauf der RTT über den Messzeitraum ca. 600 Sekungen zu erkennen. Der obere Verlauf der RTT wurde auf dem Knoten *router* erfasst, der untere auf dem Rechner *pundit*. Der obere Verlauf der RTT ist über die gesammte Messzeit konstant. Der untere Verlauf zeigt einen leichten Anstieg bis ca. zum Messzeitpunkt 350s. Diese Veränderung wurde jedoch nicht detektiert, da sie für die eingestellten Werte in der Konfiguration zu gering war. Jedoch wurde die RTT-Veränderung bei ca. 500s. erkannt. Sie ist als Ausschnitt in Abbildung 6.8 erkennbar. Die Veränderung war stark genug, um erkannt zu werden. Die Abbildung zeigt in den schwarzen Balken,

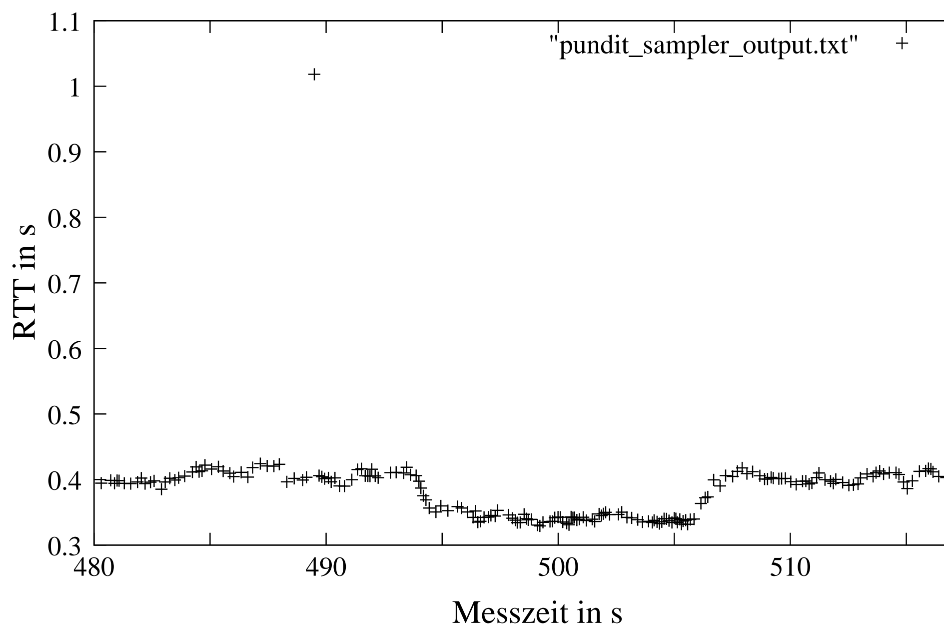

Abbildung 6.8: Sampler Ergebnis einer Knotens

dass dort erst eine Veränderung nach unten (bei 489s.) stattgefunden hat und eine Veränderung nach oben bei ca. 507s. Wie jedoch aus der Abbildung 6.7 hervorgeht, fand auf Rechner *router* keine Veränderung statt. Es sind aber mehrere deutliche RTT-Veränderungen zum Wert von ca. 800ms. zu erkennen. Sie wurden ebenfalls detektiert und sind als graue Impulse in Abbildung 6.9 zu erkennen. Jedoch wurde keine Korrelation dieser Ergebnisse mit den detektierten Ergebnissen für den unteren RTT-Verlauf (Rechner *pundit*) erkannt. Die Auswertung der TTL-Werte ergab, dass ihre Veränderungen der RTT auf dem Rechner *pundit* nicht auf einen Routenwechsel zurück zuführen sind. Die TTL-Werte blieben über den gesamten Messzeitraum konstant.

6.4 Beeinflussung der Analyse

In Kapitel 3.4 wird auf die Möglichkeit der Beeinflussung von Messungen den durch die Messung verursachten Verkehr hingewiesen. In Abbildung 6.10 ist eine solche Beeinflussung zu erkennen. Der obere Verlauf der RTTs hat, gemessen auf dem Rechner *router*, einige Schwankungen in den RTTs. Sie korrelieren mit den sprunghaften Anstiegen der RTTs im unteren Verlauf, gemessen auf dem Rechner *pundit*. Bei dieser Messung wurde ein Rechner in den USA vermessen (*mirrors.usc.edu*). Er lastete die DSL-Verbindung allein aus. Die RTTs, gemessen auf dem Rechner *router*, wurden immer dann geringer, wenn auf dem anderen Rechner die RTTs anstiegen. Es konnte jedoch eine direkte Verknüpfung dieser beiden Ereignisse ausgeschlossen werden. Eine Vergleichsmessung, die nur auf dem Rechner *router* durchgeführt wurde ergab, dass die RTT über den gesamten Messzeitraum konstant blieb. Damit bleibt nur die Möglichkeit einer Beeinflussung durch den Verkehr, der durch die Messung auf dem Rechner *pundit* verursacht wurde. Dieser Verkehr wurde nur dann geringer, wenn

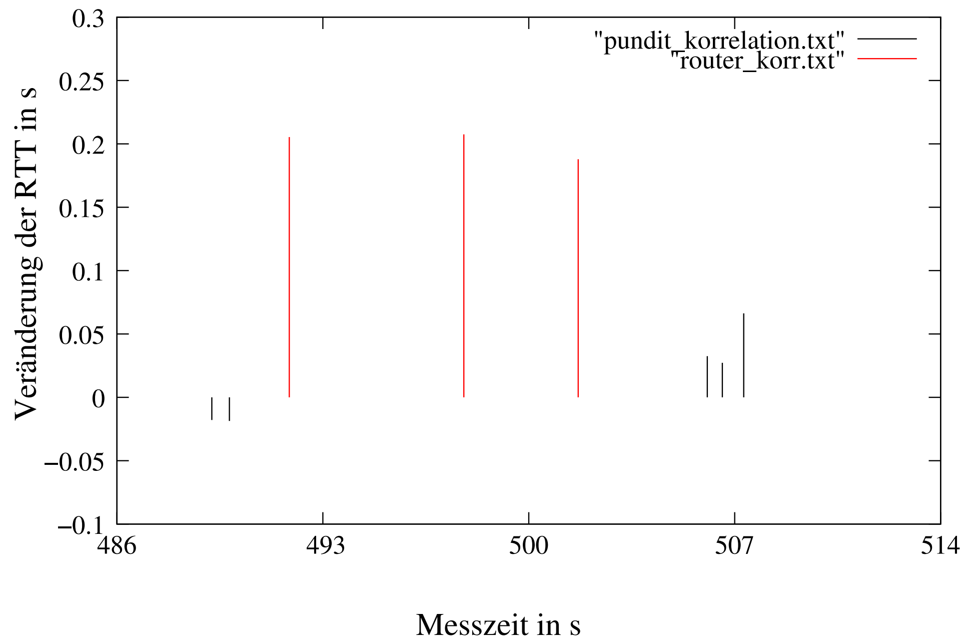

Abbildung 6.9: Überlagerung der Korrelationsergebnisse

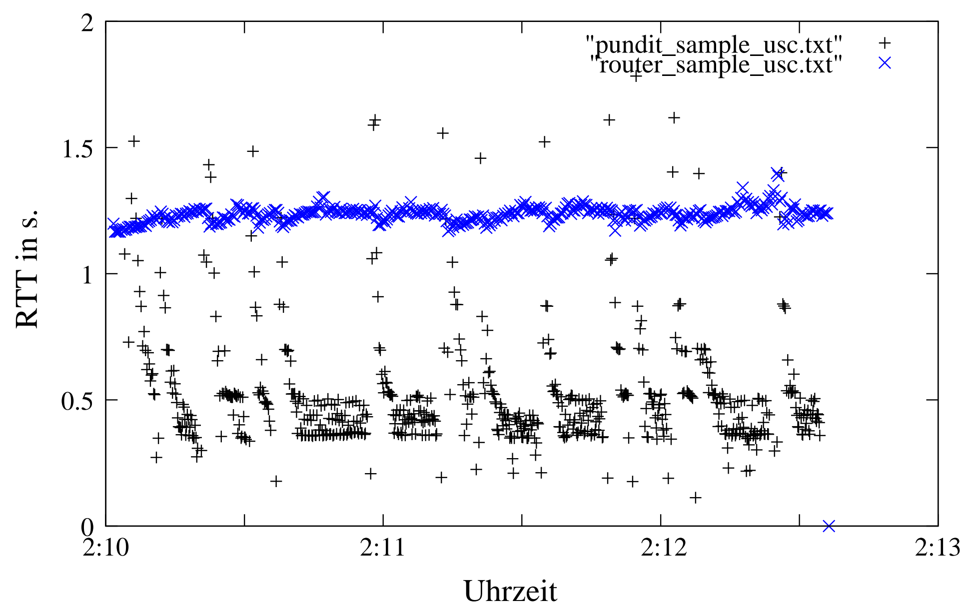

Abbildung 6.10: Überlagerung der Samplerergebnisse router/pundit - usc.edu

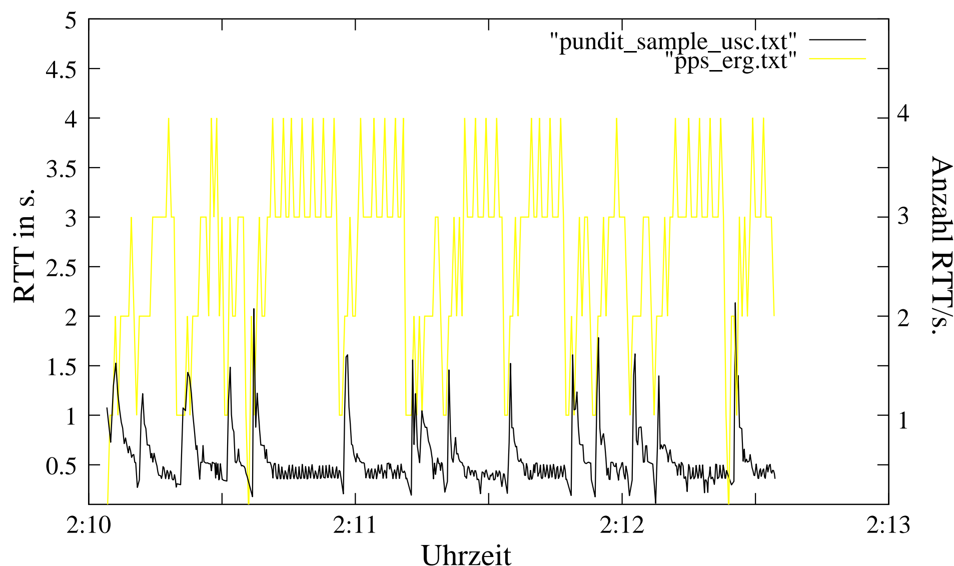

Abbildung 6.11: Samplerergebnis router - usc.edu

die RTT dieses Rechners auf ca. 1.5s. anstieg. Zu diesem wurden keine Ergebnisse erzeugt und an den Frameworkknoten *troubadix* gesendet. Dieser Schluss wird durch die Abbildung 6.11 verdeutlicht. Eine ebenfalls durchgeführte Vergleichsmessung auf dem Rechner *pundit* ergab, dass der Verlauf der RTT dort ähnlich war, wie in der Abbildung zu erkennen ist. Die im RTT-Verlauf auf *pundit* erkennbaren Sprünge der RTT von ca. 500ms. auf ca. 1500ms. sind sehr wahrscheinlich auf Paketverluste oder Queuing-Effekte in Routern zurückzuführen. Dies wird deutlich, wenn man sich die Abschätzung des durch die Messung und Analyse erzeugten Verkehrs betrachtet. Durch die Analyse auf dem Knoten "*pundit*" wird ein Messverkehr von ca. 8 KByte/s, unter der Beachtung der Konfiguration in Abbildung 6.6, erzeugt. Der Messverkehr setzt sich aus 20 Byte IP-Header, 40 Byte TCP-Header, 60 Byte Nachrichten-Header des Frameworks und ca. 50 Byte Nachrichtendaten. Bei der genutzten Konfiguration wurden ca. 11 x 4 x 170 Byte/s vom Rechner "*pundit*" an den Rechner "*troubadix*" gesendet. Die Anzahl der gesendeten Messdatenpakete ging, wie aus Abbildung 6.11 ersichtlich, genau dann zurück, wenn die RTT bei der Messung auf dem Rechner "*pundit*" einen Anstieg auf ca. 1.5s. hatte. Damit verringerte sich die Übertragung auf ein Paket pro Sekunde. Die Messdatenraten verringerte sich damit auf ein viertel der vorhergehenden Rate. Wiederum mit der verringerten Messdatenrate reduzierte sich ebenfalls die RTT auf dem Rechner "*router*". Die RTT steigt wieder an, genau dann wenn die Messdatenrate wieder steigt. Somit liegt hier die Vermutung nahe, dass diese Schwankungen der RTT tatsächlich vom zusätzlichen Messverkehr erzeugt wurde.

test

7. Zusammenfassung und Ausblick

Ziel dieser Arbeit war es, einen Prototyp zur skalierbaren verteilten Messung und Analyse von Internetverkehr zu entwerfen und implementieren. Anhand von Beispielmessungen wurde im Anschluss gezeigt, dass es möglich ist, in einer verteilten Umgebung Internetverkehr skalierbar zu messen und zu analysieren. In einem ersten Schritt wurden verschiedene Aspekte untersucht, die für eine verteilte Messung und Analyse von Internetverkehr wichtig erscheinen. In der Untersuchung der Analysemöglichkeiten wurde dargelegt, dass sich viele verschiedene Möglichkeiten bieten, Internetverkehr zu analysieren. Des Weiteren wurde untersucht, in wieweit Overlay- und Peer-to-Peer Strukturen in die Analyse von Internetverkehr mit einbezogen werden können. Ebenso wurde untersucht, wie sich aus den "rohen"-Messwerten konkrete Ereignisse und Events generieren lassen. Ausserdem wurden Schwierigkeiten bei der Interpretierung von Messergebnissen und deren Bedeutung für Ereignisse analysiert.

Die Implementierung eines Prototyps zur skalierbaren und verteilten Messung von Internetverkehr hat gezeigt, dass es möglich ist, Internetverkehr auf sehr flexible und skalierbare Weise zu messen und analysieren. Mit dieser Implementierung wurde gezeigt, dass es möglich ist, verschiedene Verkehrsströme parallel zu untersuchen. Hierbei stammen die Messdaten nicht nur aus einer Messquelle, sondern aus verschiedenen Quellen, die an unterschiedlichen Orten im Internet lokalisiert sind. Ebenso zeigt dieses Framework, dass die Auswertung der Messdaten nicht an einem zentralen Ort stattfindt, sondern über mehrere Knoten verteilt werden kann. Die Messungen und die Ergebnisse, die an Hand von Beispielmodulen durchgeführt wurden, zeigen, dass die Möglichkeit besteht, Internetverkehr auf verschiedenen Knoten zu analysieren und dass dies nicht nur auf einen Knoten beschränkt bleiben muss.

Diese Implementierung stellt jedoch nur einen Prototyp dar und bietet noch vielfältige Erweiterungsmöglichkeiten, die über den Rahmen dieser Arbeit hinausgehen würden. So ist es möglich, Teile des Frameworks auf Router oder

andere Netz-Hardware zu portieren, um weitere Quellen für die Messwerterzeugung hinzuzufügen. Eine andere Möglichkeit besteht darin, weitere Module zu entwickeln, die andere Messquellen (wie SNMP) nutzbar machen. Andererseits könnten die Ergebnisse der Analyse und Aggregation verwenden werden, um mit Hilfe der gewonnenen Ergebnisse Einfluss auf den untersuchten Verkehr auszuüben. Erweiterungen können auch in der Implementierung neuer Analyse- und Aggregationsmodule bestehen, die weitere Analysen auf den Ergebnissen durchführen. Ein weiterer Aspekt ist die Implementierung von Overlay- und Peer-to-Peer Modulen, mit denen das Framework erweitert werden kann und dessen Möglichkeiten ausgebaut werden. So kann mit einem zentralen Service-Discovery, welches z.B. auf einer verteilten Hashtabelle basiert die Möglichkeit geschaffen werden, die Verteilung der Module anhand der aktuellen Last auf dem Knoten durchzuführen.
Das implementierte Programm kann ausserdem als Basis für eine skalierbare und verteilte Simulation verschiedenster neuen Dienste dienen. Es sind dazu nur geringfügige Erweiterungen notwendig. Als Beispiel soll die Simulation von Overlay-Netzen beschrieben werden. Da in das Programm in einer modularen Form entworfen und entwickelt wurde, können neue Module implementiert werden, die Nachrichten untereinander austauschen. Ein solche Komponente kann somit auch ein Overlay-Knoten implementieren. Diese Kommunikation der Komponenten geschieht mit Hilfe der Nachrichtenschnittstelle des Frameworks. Ist ein solches Modul einmal implementiert, können beliebig viele Instanzen auf Frameworkknoten gestartet werden. Je nach Anzahl der Overlay-Knoten, die simuliert werden sollen, können beliebig viele Knoten hinzugenommen werden und darauf weitere Instanzen gestartet werden. Somit stellt diese Entwicklung ein skalierbare und flexible Möglichkeit dar Simulationen durchzuführen und disese zu implementieren.

A. XML-Konfiguration

A.1 Aufbau einer Konfigurationsdatei

Quellcode A.1: Beispiel einer Konfigurationsdatei

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<configuration cname="TCPAnalyse" cid="298345987">
    <description>
    Diese Koonfiguration erfasst TCP-Pakete und gibt diese aus
    </description>
    <module modname="Input␣Test" uuid="1878234542">
        <filename>mod_pcap.so</filename>
        <modtype>Input</modtype>
        <description>
        Dieses Modul erfasst TCP-Pakete und versendet diese.
        </description>
        <node type="remote">trubadix.bzuelch.de</node>
        <startmode>thread</startmode>
        <depend>
            <output paramid="1684677">
                <id location="remote"
                    addr="trubadix.bzuelch.de"
                    modtype="Output">1877531542
                </id>
                <id location="remote"
                    addr="router.bzuelch.de"
                    modtype="Output">2345542
                </id>
            </output>
        </depend>
        <parameter id="1877531542">
            <param name="capture_source"
```

```
28                          type="string"
                            access="read">10.15.0.0/24
30                  </param>
                    <param name="capture_source_port"
32                          type="string"
                            access="read">any
34                  </param>
                    <param name="capture_protocol"
36                          type="string"
                            access="read">tcp
38                  </param>
                    <param name="capture_destination"
40                          type="string"
                            access="read">141.3.0.0/16
42                  </param>
                    <param name="capture_destination_port"
44                          type="integer"
                            access="read">80
46                  </param>
                </parameter>
48              <parameter id="2345542">
                    <param name="capture_source"
50                          type="string"
                            access="read">10.15.0.0/24
52                  </param>
                    <param name="capture_source_port"
54                          type="string"
                            access="read">any</param>
56                  <param name="capture_protocol"
                            type="string"
58                          access="read">tcp
                    </param>
60                  <param name="capture_destination"
                            type="string"
62                          access="read">129.13.0.0/16
                    </param>
64                  <param name="capture_destination_port"
                            type="integer"
66                          access="read">80
                    </param>
68              </parameter>
            </module>
70          <module modname="Output␣Test" uuid="1877531542">
                <filename>mod_tcp.so</filename>
72              <modtype>Output</modtype>
                <description>
```

```
74          Dieses Modul äempfngt Pakete und gibt diese aus.
            </description>
76          <node type="remote">router.bzuelch.de</node>
            <startmode>thread</startmode>
78          <depend>
                <input paramid="1878234542">
80                  <id location="remote"
                         addr="trubadix.bzuelch.de"
82                       modtype="Input">1878234542
                    </id>
84              </input>
            </depend>
86          <parameter id="1878234542">
                <param name="print_packets_typ"
88                     type="string"
                       access="read">destination_ip
90              </param>
                <param name="destination_ip"
92                     type="string"
                       access="read">10.15.0.2
94              </param>
            </parameter>
96      </module>
        <module modname="Output␣Test" uuid="2345542">
98          <filename>mod_tcp.so</filename>
            <modtype>Output</modtype>
100         <description>
            Dieses Modul äempfngt Pakete und gibt diese aus.
102         </description>
            <node type="remote">trubadix.bzuelch.de</node>
104         <startmode>thread</startmode>
            <depend>
106             <input paramid="2345542">
                    <id location="remote"
108                      addr="trubadix.bzuelch.de"
                         modtype="Input">1878234542</id>
110             </input>
            </depend>
112         <parameter id="2345542">
                <param name="print_packets_typ"
114                    type="string"
                       access="read">destination_net
116             </param>
                <param name="destination_net"
118                    type="string"
                       access="read">141.3.0.0/16
```

```
120             </param>
            </parameter>
122         <parameter id="1878234542">
                <param name="print_packets_typ"
124                    type="string"
                       access="read">destination_net
126             </param>
                <param name="destination_net"
128                    type="string"
                       access="read" >129.13.0.0/16
130             </param>
            </parameter>
132     </module>
</configuration>
```

Am vorstehenden Beispiel dieser Konfigurationsdatei soll der Aufbau einer Konfigurationsdatei zum Laden und Ausführen von Messungen und Analysen veranschaulicht werden.

In Zeile 2 der Datei wird der Name und die ID der Konfiguration festgelegt. Dabei ist der Name optional, wird er nicht angegeben, so ist der Name gleich der ID der Konfiguration. Mit Hilfe der ID wird die Konfiguration eindeutig bestimmt und ist mit dieser ID während der Laufzeit verfügbar. In den nächsten drei Zeilen folgt die verbale Beschreibung der Konfiguration. Diese ist ebenfalls optional und muss nicht angegeben werden. Nach dieser allgemeinen Beschreibung der Konfiguration folgt der Abschnitt mit den Modulen. Es folgen nun ein oder mehrere Abschnitte zur Beschreibung von einzelnen Modulen der Konfiguration. Die Beschreibung eines Moduls ist in den Zeilen 6 - 69, 70 - 96 und 97 - 132 zu finden. Es folgt nun die Beschreibung eines Moduls anhand des Moduls in den Zeilen 6 - 69. Ein Modul ist aus drei Abschnitten aufgebaut. Zuerst existiert eine allgemeine Sektion (Zeilen 7 - 13). Dieser Abschnitt kann zwei Unterabschnitte *input* und *output* enthalten. Beiden Unterabschnitte beschreiben die Input- und Outputabhängigkeiten eines Moduls (d. h. Module, von denen das Modul Nachrichten empfängt und an die es Nachrichten versendet). In jedem dieser Unterabschnitte sind ein oder mehrere abhängige Module beschrieben. Danach folgt der Abschnitt, in welchem die Abhängigkeiten zu anderen in der Konfiguration beschriebenen, Modulen dargelegt werden (Zeilen 14 - 25). Abschließend besteht die Beschreibung eines Moduls aus einer oder mehreren Parametersektionen (Zeilen 26 - 68). Wiederum besteht ein Parameterabschnitt aus mind. einem Parameter.
Ein Modul wird durch seinen Namen und seine ID innerhalb einer Konfiguration identifiziert, wobei die ID Vorrang hat. Im allgemeinen Teil eines Moduls ist der Dateiname angegeben, in dem sich der Ausführungscode des Moduls befindet. Des Weiteren ist der Typ des Moduls spezifiziert. Diese Typen sind an den funktionalen Entwurf des Frameworks gekoppelt, bzw. die Module sollten als Typ einen der dort verwendeten Namen tragen. Damit wird den Modulen eine Grundstruktur aufgeprägt und gewährleistet, dass der Input nur von

Modulen eines bestimmten Typs kommen kann. In der allgemeinen Sektion eines Moduls folgt nun dessen verbale Beschreibung. Sie ist optional und muss nicht angegeben werden. Nun folgt noch die Angabe des Knotens auf dem das Modul ausgeführt werden soll (Zeile 12). Als Typ lassen sich hier *remote* und *auto* wählen. Bei *auto* wird kein Wert innerhalb des XML-Tag angegeben. Damit wird dem Loader mitgeteilt, dass er einen geeigneten Knoten auf Grund einer Metrik aus Abschnitt 4.6.1 suchen soll. Bei der Option *remote* muss als Wert für den Tag entweder ein Hostname oder eine IP-Adresse angegeben werden. Abgeschlossen wird der allgemeine Abschnitt durch den Startmodus eines Moduls. Dieser kann entweder *thread*, *fork* oder *binary* sein. Bei *thread* wird das Modul als neuer Thread innerhalb des Knotens gestartet und ausgeführt. Während bei *fork* das Modul in einen neuen Prozess geforkt wird. Bei dieser Option findet die Kommunikation mit dem Knoten über einen Socket statt. Die dritte Option *binary* bedeutet, dass das Modul komplett in einer eigenen ausführbaren Datei vorliegt und seperat gestartet werden muss. Sie soll dazu genutzt werden einzelne Module in einem anderen Benutzerkontext auszuführen; z.B. benötigt das Pcap-Modul "root"-Benutzer-Rechte, um Pakete erfassen zu können. Ist diese Option gewählt, so wird die in *filename* angegebene Datei mit Hilfe von *exec* ausgeführt. Die Kommunikation mit dem Knoten erfolgt ebenfalls über einen Socket.

Es folgt nun die Beschreibung der Modulsabhängigkeiten für ein einzelnes Modul. Dieser Abschnitt wird mit dem Tag <depend> ... </depend> umschlossen. Darin wird nochmals zwischen Eingangs- und Ausgangsabhängigkeiten unterschieden. Jedoch ist jeweils ein abhängiges Modul von vornherein bestimmt. Für jede der beiden Abhängigkeiten kann eine Parameter-ID angegeben werden. Diese Parameter werden dann genutzt, wenn Nachrichten von einem Modul kommen, welches nicht in der Abhängigkeitsliste auftaucht. Wird keine ID angegeben, werden die Parameterwerte in den allgemeinen Parametern des Moduls gesucht[1]. Ein Eintrag in der Abhängigkeitsliste sieht wie folgt aus (Zeilen 16 - 19). Der Eintrag hat immer einen Wert. Dieser Wert ist eine ID eines anderen Moduls. Dieses muss in der Konfiguration vorkommen. Hinzu kommen Informationen, wo das Modul zu finden ist (*addr* und *location*). Handelt es sich um ein *auto*-Lokationsmodul, so muss mit Hilfe des Service Discovery das gestartete Modul lokalisiert werden. Des Weiteren wird der Modultyp angegeben, von dem Nachrichten kommen oder zu dem Nachrichten gesendet werden. Bei *auto* ist *addr* leer; ansonsten enthält *addr* entweder eine IP-Adresse oder einen Hostnamen. Ein Hostname wird im Loader auf die interne Adressierung umgewandelt und kann dann zur Kommunikation genutzt werden.

Der letzte Abschnitt einer Modulbeschreibung besteht aus den Parametern. Die Parameter-Werte können in Abhängigkeit des Quell-/Zielmoduls gesetzt werden. Dies kann in vielen Situationen erwünscht sein. Ein Parameterabschnitt ist durch seine ID gekennzeichnet und muss auf eine ID bei den abhängigen Modulen passen. Außerdem kann es einen Parameterabschnitt geben,

[1] Eine Beschreibung was allgemeine Parameter sind, folgt weiter unten bei der Beschreibung der Parameterabschnitte.

dessen ID auf die moduleigene ID passt. Dann werden die in dem Abschnitt enthaltenen Parameter als allgemeine Parameter aufgefasst und in dem Parameterspeicher eines Moduls gespeichert. Für jede andere ID wird ein eigener Parameterspeicher erzeugt. Bei einem Parameterspeicher handelt es sich um ein *ModuleInfo*-Objekt. In jedem Parameterabschnitt sind ein oder mehrere Parameter eingeschlossen. Fehlt ein Parameterabschnitt für eine ID, die in der Abhängigkeitsliste erwähnt wird, so werden die Parameter aus dem “allgemeinen”-Parametern genutzt. Ein Parametereintrag ist wie folgt aufgebaut. Zuerst enthält er einen Namen. Mit diesem wird die Variable adressiert. Es folgt der Typ der Variablen. Es sind *string* und *integer* zugelassen. Andere Angaben werden als String interpretiert. Ein Parameter wird abschließend durch seine Zugriffsrechte charakterisiert. Eine Variable kann entweder das Zugriffsrecht *read* oder *readwrite* besitzen. Innerhalb eines Parameter-Tags ist der Wert der Variablen gespeichert. Ein Beispiel für einen Parameter ist in den Zeilen 27 - 30 zu finden.
Damit ist ein Modul charakterisiert und steht zur Ausführung bereit. Die Abhängigkeiten der einzelnen Module beschränken sich nur auf funktionale Module. Interaktionen mit den Verwaltungsmodulen auf einem Knoten sind davon nicht betroffen.

A.2 Allgemeine Festlegungen

Die folgenden Attribute und Tags der Konfigurationsdatei sind optional und müssen nicht angegeben werden. Zusätzlich werden die Defaultwerte angegeben, wenn diese nicht in der Datei vorhanden sind.

Attribute / Tag	Defaultwert	Beschreibung
configuration->cname	“	Wert von cid
configuration->description	“	Textuelle Beschreibung der Konfiguration
module->description	“	Textuelle Beschreibung des Moduls
depend->(input/output)	“	Es muss kein Input/Output Tag angegeben werden, wenn keine entsprechenden Abhängigkeiten vorhanden sind.
depend->input->location	remote	'remote' bedeutet, dass ein Hostname angegeben wird.
depend->output->location	remote	'remote' bedeutet, dass ein Hostname angegeben wird.
parameter->param->access	read	alle Variablen sind per default nur lesbar, wenn diese auch schreibbar sein sollen, muss dies angegeben werden.
nodetype	remote	

Alle anderen Attribute und Tags sind verpflichtend und müssen angegeben werden. Die Konfigurationsdatei wird in ISO-8859-1 kodiert. Alle IDs sind vom Typ unsigned int (4 Oktett) und liegen im Bereich von 1 bis $2^{3}2 - 1$. Die Parametertypen sind momentan auf die beiden Typen *string* und *integer* beschränkt; andere Angaben werden als *string* interpretiert.

B. Nachrichten-Typen

Die folgenden Nachrichtentypen sind vordefinierte Nachrichten, die zwischen den einzelnen Modulen ausgetauscht werden. In diesem Kapitel sollen die einzelnen Nachrichten beschrieben und deren Aufbau definiert werden. Die Nachrichten lassen sich in zwei Klassen einteilen. Zum einen sind dies solche, die vom Framework definiert sind, um einzelne Module zu starten, zu beenden und zu administrieren.Zum anderen gibt es Nachrichten, die von den einzelnen Funktionsmodulen definiert und zwischen ihnen ausgetauscht werden. Diese Nachrichten werden, soweit bekannt, im zweiten Teil des Kapitels beschrieben. Im ersten erfolgt die Definition der Nachrichten, die von den einzelnen Modulen generiert werden, welche zum Ausführen des Frameworks notwendig sind.

B.1 Allgemeine Definitionen

In diesem Abschnitt sollen einige grundlegende Richtlinien zur Kodierung der einzelnen Nachrichten angegeben werden. Der prinzipielle Aufbau einer Nach-

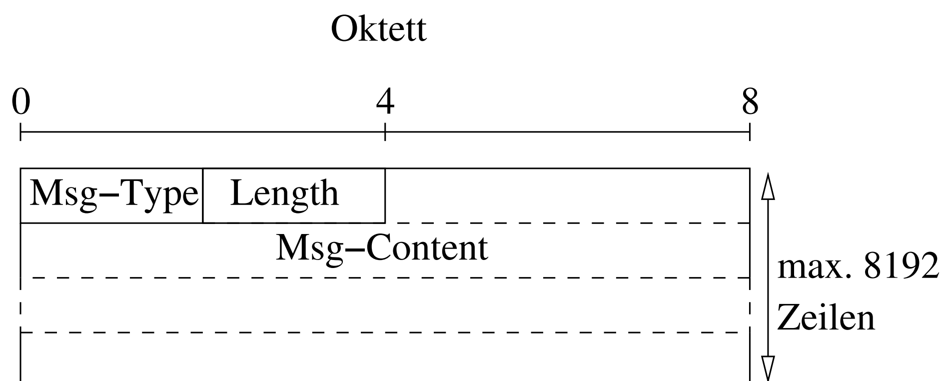

Abbildung B.1: Aufbau einer Nachricht

richt ist in Abbildung B.1 dargestellt. Eine Nachricht kann maximal 65536 Oktett lang sein. Darin eingeschlossen ist der Nachrichtentyp und die Längenangabe, so dass effektiv 65532 Oktett für Nachrichten Inhalt erhalten bleiben. Die Adressen sind unabhängig von den Nachrichten definiert.

1. Alle Nachrichten sind Byte (Oktett) orientiert. Alle Felder einer Nachricht sind Vielfache eines Byte (Oktett).
2. Alle *Integer (unsigned int)* werden im *Big-Endian* Format in einer Nachricht genutzt. Dies entspricht der *Network Byteorder*.
3. Alle *Short Integer (short int)* werden im *Big-Endian* Format in einer Nachricht genutzt. Dies entspricht der *Network Byteorder*.
4. Jeder String setzt sich aus einer Längenangabe in Byte (Oktett) in *Big-Endian* Format und der folgenden Zeichenkette zusammen. Bei der Längenangabe handelt es sich um einen *Short Integer*. Die einzelnen Zeichen der Zeichenkette sind entweder ISO 8859-1 oder ISO 8859-1 kodiert.
5. Ein Nachrichtentyp besteht aus einem *Short Integer* im *Big-Endian* Format. Die Tabelle B.2 listet die definierten Nachrichtentypen auf. Die einzelnen Formate der Nachrichten werden in den Abschnitten B.2 und B.3 definiert.
6. Für fest vordefinierte Nachrichten werden keine Typangaben vor einem Inhaltsfeld gemacht. Für Nachrichten, die nicht definiert sind, wird vor jedes Feld eine Typangabe gesetzt; diese sind in Tabelle B.1 definiert.
7. Eine Struktur setzt sich aus einer Längenangabe (Integer) gefolgt von den einzelnen Einträgen zusammen. Ein Eintrag besteht aus einer Typangabe (Short Integer), Längenangabe (Short Integer) und den eigentlichen Daten zusammen. Die *Integer* und *Short Integer* folgen den obrigen Angaben. Es lassen sich auch geschachtelte Strukturen definieren. Die Typangabe geht aus Tabelle B.1 hervor.
8. Alle Nachrichten müssen so definiert werden, dass sie transparent per Socket und per Warteschlange (zwischen mehreren Threads) transportiert werden können. Es findet kein Marshaling/Unmarshaling der Nachrichtendaten statt. Einzig die Quell- und Zieladressen einer Nachricht werden für den Transport gemarshalt.

Tabelle B.2: Definierte Nachrichtentypen

Bezeichner	Wert	Beschreibung
MSG_NOTDEF	0	Reservierter Nachrichtentyp.
MSG_FETCHALL_MSG	0xFFFF	alle Nachrichten werden verarbeitet.

Bezeichner	Wert	Beschreibung
MSG_MODPING_MSG	0xFFFE	schickt eine "Ping"-Nachricht zwischen zwei Modulen und wertet die RTT des Pings aus.
MSG_LOADER_STARTMODULE	0x01	Startet ein Modul.
MSG_LOADER_STARTCONFIG	0x02	Startet eine komplette Konfiguration.
MSG_LOADER_STOPMODULE	0x03	Beendet ein Modul. Sendet eine Nachricht an den Loader. Darin wird diesem mitgeteilt, welches Modul beendet werden soll.
MSG_LOADER_STOPCONFIG	0x04	Beendet eine laufende Konfiguration.
MSG_LOADER_LISTDISPATCHER	0x05	Listet alle im Dispatcher registrierten Module auf.
MSG_LOADER_LISTPROTOCOL	0x06	Listet alle registrierten Protokolle auf.
MSG_LOADER_NEWMODID	0x07	Anforderung einer neuen ModulID beim Loader. Diese Funktion wird von SMux benötigt.
MSG_MODULE_READY	0x08	Nachricht, dass sich ein Modul im Ready-Zustand befindet.
MSG_LOADER_CONFIGFAILED	0x09	Nachricht, wenn das Starten einer Konfiguration fehlgeschlagen ist.
MSG_LOADER_MODULEFAILED	0x0A	Nachricht an einen Loader, dass das Laden eines Moduls fehlgeschlagen ist.
MSG_LOADER_REQUUID	0x0B	Anforderung an einen Loader, eine neue UUID für ein Modul zu erzeugen.
MSG_MODULE_BREAK	0x0C	Nachricht zwischen Modulen, um diesen mitzuteilen, das sich ein Modul im "Break"-Zustand befindet.
MSG_LOADER_LISTSTARTMODULES	0x0D	Listet alle vom Loader gestarteten Module auf.
MSG_LOADER_MODSHUTDOWN	0x0E	Nachricht von einem Modul an den Loader. Darin teilt das Modul mit dass es sich beendet.

Bezeichner	Wert	Beschreibung
MSG_LOADER_LISTCONFIGS	0x0F	Listet alle gestarteten Konfigurationen des aktuellen Loaders auf.
MSG_LOADER_LISTCONFIGMODS	0x10	Liste alle Module einer Konfiguration auf.
MSG_LOADER_SHUTDOWNNODE	0x11	Nachricht an einen Loader. Mit dieser Nachricht wird ein Knoten beendet.
MSG_LOADER_NODESHUTDOWN	012	Nachricht von einem Loader an alle verbundenen Knoten, dass der aktuelle Knoten beendet wird.
MSG_MODULE_CLOSE	0x13	Nachricht zwischen Modulen, damit wird angezeigt, dass sich das aktuelle Module beendet.
MSG_LOGGER_MSG	0x20	Logging-Nachricht, die an den Logger gesendet wird.
MSG_MGMT_SOCKETFAILED	0x40	Nachricht vom Management zum einem Loader, darin wird mirgeteilt, dass der Managementserver keinen Socket erzeugen kann. Der Aufbau der Nachricht entspricht exakt der MSG_SMUX_SOCKETFAILED-Nachricht.
MSG_SMUX_CONNECTSMUX	0x50	Nachricht zum Verbinden mit einem anderen SMux.
MSG_SMUX_SOCKETFAILED	0x51	Nachricht vom SMux an den Loader, dass er sich nicht mit einem anderen SMux verbinden konnte oder die Verbindung abgebrochen ist.
MSG_SMUX_LISTSOCKETS	0x52	Listet alle Socketverbindungen zu anderen Knoten auf.
MSG_SMUX_DISCONNECT	0x53	Beendet eine Verbindung zu einem anderen Knoten.
MSG_MODINFO_LISTVAR	0x60	Listet die registrierten Variablen eines Moduls auf.
MSG_MODINFO_GETVALUE	0x61	Liefert den Wert einer Variablen zurück.
MSG_MODINFO_SETVALUE	0x62	Setzt den Wert einer Variablen.

Bezeichner	Wert	Beschreibung
MSG_MODCORE_SHUTDOWN	0x70	Beendet ein Modul.
MSG_MODCORE_LISTHANDLER	0x71	Listet alle regitrierten Nachrichtenhandler eines Moduls auf.
MSG_MODCORE_LISTQUEUE	0x72	Listet alle Einträge der Eingangsqueue auf.
MSG_MODIFACE_LISTDEPMOD	0x73	Listet alle abhängigen Module auf.
MSG_PCAP_PACKET	0x80	mitgelauschte Paketdaten des PCAP-Moduls.
MSG_PCAP_GETCOMMAND	0x81	Liefert den Filterstring des PCAP-Modulszurück.
MSG_TCPANALYSE_RTT	0x90	RTT-Werte einer TCP-Verbindung, die mit dem TCP-Analysator berechnet wurden.
MSG_TCPANALYSE_RAW	0x91	Die wichtigsten Paket-Werte aus einem IP und TCP-Header.
MSG_TCPANALYSE_INTERPKT	0x92	Versendet Informationen über die Zeit zwischen zwei aufeinander folgenden Paketen.
MSG_TCPANALYSE_DUPPKT	0x93	Versendet Informationen über den Grund und den Ort eines Paketverlustes.
MSG_TCPANALYSE_SUM	0x94	Summaryinformationen, nach dem Beenden einer Verbindung.
MSG_TCPANALYSE_SACK	0x95	Informationen über SACK-Ereignisse werden in dieser Nachricht mitgeteilt.
MSG_DIFF_RTT	0xA0	Informationen über die Differenz zwischen zwei hintereinander folgenden RTT-Werten.
MSG_DIFF_TTL	0xA1	Informationen über die Differenz zwischen zwei hintereinander folgenden TTL-Werten.
MSG_DIFFTLV_RTT	0xD2	TLV-Nachricht zum File-writer.
MSG_DIFFTLV_TTL	0xD3	TLV-Nachricht zum File-writer.

Bezeichner	Wert	Beschreibung
MSG_KORRELATOR_RTT	0xB0	Informationen über zeitliche Veränderungen der RTT-Werte.
MSG_KORRELATOR_TTL	0xB1	Informationen über zeitliche Veränderungen der TTL-Werte.
MSG_KORRTLV_RTT	0xD4	TLV-Nachricht zum Filewriter.
MSG_KORRTLV_TTL	0xD5	TLV-Nachricht zum Filewriter.
MSG_TCPANATLV_RTT	0xD0	TLV-Nachricht zum Filewriter.
MSG_TCPANATLV_TTL	0xD1	TLV-Nachricht zum Filewriter.
MSG_VERGL_RTTCH	0xE1	Informationen über Veränderungen der TTL-Werte, die auf verschiedenen Knoten gemessen wurden..
MSG_VERGL_RTTDIFF	0xE2	Informationen die Differenz zwischen RTT-Werten von verschiedenen Hosts.

B.2 Frameworknachrichten

B.2.1 Loader

B.2.1.1 MSG_LOADER_STARTMODULE

Die Modulstartnachricht ist aus vier Abschnitten (Sections) aufgebaut. Dies sind die:

- Allgemeiner-Abschnitt
- kein oder ein Inputdependency-Abschnitt
- kein oder ein Outputdependency-Abschnitt
- kein, ein oder mehrere Parameter-Abschnitt(e)

Die einzelnen Abschnitte sind wie folgt aufgebaut:

Quellcode B.1: Allgemeiner-Abschnitt der Modulstartnachricht

```
struct msg_start_module {
    unsigned short int mlen;
    char *modulename;
    unsigned int uuid;
```

Tabelle B.1: Typdefinitionen

Bezeichner	Wert	Beschreibung
MSG_EINTR_NOTDEF	0	Undefinierter Wert
MSG_EINTR_INT	1	Integer der Länge 4 Oktett
MSG_EINTR_SINT	2	Short Integer der Länge 2 Oktett
MSG_EINTR_STRING	3	Zeichenkette, Längenfeld für Länge der Zeichenkette in Short Integer
MSG_EINTR_STRUCT	4	Struktur aus mehreren einzelnen Einträgen. Enthält ein Längenfeld des Typs Integer gefolgt von eigentlichen Einträgen
MSG_EINTR_USERDEF	5	Benutzerspezifischer Eintrag, mit einer Längenangabe in Short Integer.

```
    unsigned int startmethod;
6   unsigned short int flen;
    char *filename;
8   unsigned short int molen;
    char *modtype;
10  unsigned short int numindepend;
    unsigned short int numoutdepend;
12  unsigned short int numparaset;
};
```

Quellcode B.2: Dependency-Abschnitt der Modulstartnachricht

```
struct dependentry {
2   unsigned int uuid;
    unsigned short int mlen;
4   char *modaddr;
    unsigned short int tlen;
6   char *modtype;
    unsigned short int mnlen;
8   char *modulename;
};
```

Quellcode B.3: Parameter-Abschnitt der Modulstartnachricht

```
struct parasection {
2   unsigne dint uuid;
    unsigned short int numentry;
4 };

6 struct paramentry {
    unsigned short int nlen;
8   char *name;
    unsigned short int type;
```

```
10      unsigned short int access;
        unsigned short int vallen;
12      unsigned short int vlen;
        char *value;
14  };
```

Die einzelnen Abschnitte werden in einem Puffer zusammengestellt und dann in einer binären Nachricht übertragen.

B.2.1.2 MSG_LOADER_STARTCONFIG

Quellcode B.4: Aufbau der Startconfig-Nachricht

```
    struct msg_start_config {
2       unsigned short int mode;
        unsigned int len;
4       char *filename;
    };
```

Diese Nachricht enthält den Modus, von wo der Loader sich die Konfiguration besorgen soll. Der Modus kann folgende Werte annehmen:
0 = Konfigurationsdatei per Service Discovery holen
1 = Konfigurationsdatei aus dem lokalen Dateisystem laden
2 = Konfigurationsdatei als Anhang in der Nachricht

Befindet sich die Onfiguration im Anhang der Nachricht, so hat der Wert *len* einen Wert ungleich 0 und danach beginnen die Daten der Konfigurationsdatei.

B.2.1.3 MSG_LOADER_LISTCONFIGS

Diese Nachricht wird vom Management-Client erzeugt und an den Loader gesendet. Diese Nachricht enthält in ihrer Anforderung an den Loader keine Daten, so dass nur die Antwort gezeigt wird.

Quellcode B.5: Aufbau der MSG_LOADER_LISTCONFIGS-Nachricht

```
    struct cfgmod {
2       unsigned int uuid;
        unsigned int flags;
4       unsigned short int nummods;
        unsigned short int nlen;
6       char *name;
        unsigned short int flen;
8       char *filename;
    };
10
    struct msg_ld_listconfig {
12      unsigned short int numconfigs;
        struct cfgmod mod[];
14  };
```

B.2.1.4 MSG_LOADER_LISTCONFIGMODS

Dia Anforderung enthält die UUID (unsigned int) als Daten der Nachricht. In der Antwort finden sich folgende Parameter;

Quellcode B.6: Aufbau der MSG_LOADEr_LISTCONFIGMODS-Nachricht

```
struct modentry {
    unsigned int uuid;
    unsigned short int nlen;
    char *name;
    unsigned short int alen;
    char *addr;
    unsigned short int tlen;
    char *type;
    unsigned short int ideps;
    unsigned short int odeps;
};

struct msg_ld_lstcmods {
    unsigned short int numentry;
    struct modentry mods[];
};
```

B.2.1.5 MSG_LOADER_STOPCONFIG

Diese Nachricht enthält nur die UUID (unsigned int) der Konfiguration, die beendet werden soll. Auf Grund dieser Nachricht werden an alle enthaltenen Module die MSG_MODCORE_SHUTDOWN-Nachrichten versendet.

B.2.1.6 MSG_LOADER_LISTDISPATCHER

Quellcode B.7: Aufbauf der MSG_LOADER_LISTDISPATCHER-Nachricht

```
struct disp_entry {
    unsigned int ModuleID;
    unsigned int nlen;
    char *modname;
};
struct lst_dipatcher {
    unsigned short int numentry;
    struct dips_entry entries[];
};
```

B.2.1.7 MSG_LOADER_LISTPROTOCOL

Quellcode B.8: Aufbau der MSG_LOADER_LISTPROTOCOLS-Nachricht

```
struct proto_entry {
    unsigned short int plen;
    char *protocol;
```

```
    unsigned short int nlen;
    char *modname;
    unsigned int ModuleID;
};

struct msg_ld_lstdisp {
    unsigned short int numentry;
    struct proto_entry entries[];
};
```

B.2.1.8 MSG_MODULE_READY

Quellcode B.9: Aufbau der Modul-Ready-Nachricht

```
struct msg_mod_ready {
    unsigned int uuid;
    unsigned int ModID;
    unsigned short int mlen;
    char *modname;
    unsigned short int tlen;
    char *modtype;
    unsigned short int alen;
    char *addr;
};
```

Diese Nachricht wird in dieser Form vom Loader und den einzelnen Modulen versendet.

B.2.1.9 MSG_LOADER_MODULEFAILED

Diese Nachricht wird von Loadern erzeut, wenn der Start eines Moduls fehlgeschlagen ist.

Quellcode B.10: Aufbau der Modulefailed-Nachricht

```
struct msg_mod_fail {
    unsigned short int exceptioncode;
    unsigned int ModuleID;
    unsigned short int mlen;
    char *modulename;
    unsigned int uuid;
    unsigned short int startmode;
    unsigned short int flen;
    char *filename;
};
```

B.2.2 Logger

B.2.2.1 MSG_LOGGER_MSG

Quellcode B.11: Aufbau der Logger-Nachricht

```
struct logger_msg {
    unsigned short int nlen;
    char *nodename;
    unsigned short int modlen;
    char *modname;
    unsigned short int priority;
    unsigned short int mlen;
    char *logmsg;
};
```

B.2.3 Kernmodul

B.2.3.1 MSG_MODCORE_SHUTDOWN

Diese Nachricht ist leer und enthält keine Informationen. Das alleinige Empfangen dieser Nachricht beendet ein Modul.

B.2.3.2 MSG_MODCORE_LISTHANDLER

Quellcode B.12: Aufbau der Zeige registrierte Nachrichtenhandler-Nachricht

```
struct hndentry {
    unsigned short int mtype;
    unsigned int ModuleID;
    unsigned int Flags;
};

struct modcore_lsthand {
    unsigned short int anzhandler;
    struct hndentry entries[];
};
```

B.2.3.3 MSG_MODCORE_LISTQUEUE

Quellcode B.13: Aufbau der Zeige Queueinhalt-Nachricht

```
struct modcore_lstqueue {
    unsigned short int fplen;
    char *fprotocol;
    unsigned short int fnlen;
    char *fnode;
    unsigned int fModID;
    unsigned short int tplen;
    char *tprotocol;
    unsigned short int tnlen;
    char *tnode;
    unsigned int tModID;
    unsigned short int MsgType;
```

```
    unsigned int Msize;
    unsigned int Flags;
};
```

B.2.3.4 MSG_MODIFACE_LISTDEPMOD

Quellcode B.14: Aufbau der abhängige Module-Nachricht

```
struct depentry {
    unsigned short int alen;
    char *addr;
    unsigned short int mtlen;
    char *modtype;
    unsigned short int nlen;
    char *modname;
    unsigned int uuid;
    unsigned int ModuleID;
    unsigned short int Ready;
};

struct modiface_depmod {
    unsigned short int inum;
    unsigned short int onum;
    struct depentry ientries[];
    struct depentry oentries[];
};
```

B.2.4 Module-Management

B.2.4.1 MSG_MODINFO_GETVALUE

Diese Nachricht fordert den Inhalt einer Variablen an.

Quellcode B.15: Aufbau der ModuleInfo-GetValue-Nachricht (Req.)

```
struct msg_modinfo_getvalue_req {
    unsigned short int len;
    char *varname;
};
```

Es wird folgende Antwort generiert:

Quellcode B.16: Aufbau der ModuleInfo–Nachricht (Resp.)

```
struct msg_modinfo_getvalue_resp{
    unsigned short int len;
    char *varname;
    unsigned short int vartype;
    unsigned short int veraccess;
    unsigned short int varlen;
    char *data;
};
```

B.2.4.2 MSG_MODINFO_LISTVAR

Die Nachricht muss als Aufforderung zur Auflistung der registrierten Variablen an ein Modul gesendet werden. Der optionale Filter dient dazu nur Variablen zurückzuliefern, die entweder nur lesend (readonly) oder lesend und schreibend (readwrite) sind.

Quellcode B.17: Aufbau der ModuleInfo-Listvariable-Nachricht (Req.)

```
struct msg_modinfo_listvar_req {
    unsigned short int filter;
};
```

Als Antwortnachricht wird die folgende Nachricht generiert. Diese enthält einen Kopf, in dem die Anzahl der folgenden Variablen hinterlegt ist. Danach folgen *num_vars* Einträge der zweiten Struktur.

Quellcode B.18: Aufbau der ModuleInfo-ListVariable-Nachricht (Resp.)

```
struct msg_modinfo_listvar_resp {
    unsigned short int msg_num;
    unsigned short int num_vars;
    char *vars;
};

struct msg_modinfo_listvar_entry {
    unsigned short int len;
    char *vname;
    unsigned short int vartype;
    unsinged short int varaccess;
};
```

Diese Nachricht wird an den Sender der Aufforderung zurück gesendet.

B.2.4.3 MSG_MODINFO_SETVALUE

Nachricht ist nicht implementiert. Der Nachrichtentyp wurde jedoch reserviert.

B.2.5 Socket-Multiplexer

B.2.5.1 MSG_SMUX_CONNECTSMUX

Quellcode B.19: Aufbau der SMux ConnectSMux-Nachricht

```
struct msg_consmux {
    unsigned short int plen;
    char *protocol;
    unsigned short int hlen;
    char *hostname;
    unsigned short int port;
```

B.2.5.2 MSG_SMUX_SOCKETFAILED

Die Nachricht hat einen allgemeinen Abschnitt. Diesem folgt ein Abschnitt, der je nach Urache des Fehlers anders ausgeprägt ist. Es sind folgende Fehlerursachen definiert:

- SERROR_SERVERFAILED = 1
- SERROR_CLIENTFAILED = 2
- SERROR_REMOTEEND = 3
- SERROR_LOCALEND = 4

Quellcode B.20: Aufbau der SMux Socketfailed-Nachricht

```
struct smux_err {
    unsigned int cause;
    int errno;
    void *errdescription;
};

struct smux_err_server {
    unsigned short int protocol;
    unsigned short int port;
};

struct smux_err_client {
    unsigned short int hlen;
    char *hostname;
    unsigned short int protocol;
    unsigned short int port;
};
```

Hinter der *errdescription* verbirgt sich jeweils einer der beiden Abschnitte *smux_err_server* oder *smux_err_client*. Wobei letzeres auch für die Fehler *SERROR_CLIENTFAILED, SERROR_REMOTEEND und SERROR_LOCALEND* benutzt wird.

B.2.5.3 MSG_SMUX_LISTSOCKETS

Quellcode B.21: Parameter-Abschnitt der Modulstartnachricht

```
struct smux_peerentry {
    unsigned short int hlen;
    char *host;
    unsigned short int modus;
    unsigned short int ModuleID;
    unsigned short int plen;
```

```
    char *peer_ip;
8 };

10 struct msg_smux_listpeers {
    unsigned short int numpeers;
12  struct smux_peerentry entries[];
};
```

B.2.5.4 MSG_SMUX_DISCONNECT

Quellcode B.22: Parameter-Abschnitt der Modulstartnachricht

B.3 Funktionsmodule

B.3.1 Pcap-Modul

B.3.1.1 MSG_PCAP_PACKET

Quellcode B.23: Aufbau der Pcap-Paket-Nachricht

```
struct pcap_pkt {
2   unsigned int usedprotocols;
    unsigned short int iplen;
4   unsigned short len tcpudplen;
    struct iphdr iph;
6   union {
        struct tcphdr tcp;
8       struct udphdr udp;
    };
10 };
```

Diese Nachticht enthält erfasste Paketdaten von TCP oder UDP-Paketströmen als Payload von IP-Paketen.

B.3.1.2 MSG_PCAP_GETCOMMAND

Quellcode B.24: Aufbau der Pcap-GetCommand-Nachricht

```
struct pacp_getcmd {
2   unsigned short int clen;
    char* cmd;
4 };
```

für das setzen eines neuen Filter-Strings wird der gleiche Nachrichtenaufbau werwendet. Es ändert sich nur der Nachrichtentyp.

B.3.2 TCP-Analyse

B.3.2.1 MSG_TCPANALYSE_RTT

Quellcode B.25: Aufbau der TCPAnalyse-RTT-Nachricht

```
struct tcpanalyse_rtt {
    unsigned int sourceip;
    unsigned int destip;
    unsigned short int sourceport;
    unsigned short int destport;
    struct timeval tv;
    unsigned short int dir;
    double rtt;
    double estrtt;
};
```

B.3.2.2 MSG_TCPANALYSE_RAW

Quellcode B.26: Aufbau der TCPAnalyse-RAW-Nachricht

```
struct tcpanalse_raw {
    unsigned int sourceip;
    unsigned int destip;
    unsigned short int sourceport;
    unsigned short int destport;
    struct timeval tv;
    unsigned short int dir;
        unsigned char ttl;
    unsigned int sip;          //host-bytorder
    unsigned int dip;          //host-byteorder
    unsigned short int sport //host-byteorder
    unsigned short int dport //host-byteorder
    unsigned int seq;
    unsigned int ack_seq;
    unsigned short int win;
    unsigned short int flags;
};
```

B.3.2.3 MSG_TCPANALYSE_INTERPKT

Quellcode B.27: Aufbau der TCPAnalyse-INTERPKT-Nachricht

```
struct tcpanalyse_interpkt {
    unsigned int sourceip;
    unsigned int destip;
    unsigned short int sourceport;
    unsigned short int destport;
    struct timeval tv;
    unsigned short int dir;
    struct timeval ipkt;
};
```

B.3.2.4 MSG_TCPANALYSE_DUPPKT

Quellcode B.28: Aufbau der TCPAnalyse-DUPPKT-Nachricht

```
struct tcpanalyse_duppkt {
    unsigned int sourceip;
    unsigned int destip;
    unsigned short int sourceport;
    unsigned short int destport;
    struct timeval tv;
    unsigned short int dir;
    unsigned short int type;
    unsigned short int cause;
};
```

B.3.2.5 MSG_TCPANALYSE_SACK

Quellcode B.29: Aufbau der TCPAnalyse-SACK-Nachricht

```
struct tcpanalyse_duppkt {
    unsigned int sourceip;
    unsigned int destip;
    unsigned short int sourceport;
    unsigned short int destport;
    struct timeval tv;
    unsigned short int dir;
    unsigned short int type;
    unsigned short int cause;
};
```

B.3.3 Differenz-Modul

B.3.3.1 MSG_DIFF_RTT

Quellcode B.30: Aufbau der Differenz-RTT-Nachricht

```
struct msg_rttdiff {
    unsigned int sip,dip;
    unsigned short int dir;
    unsigned int numback;
    double rtt;
    double rttdiff;
    double estrtt;
    double estrttdiff;
};
```

B.3.3.2 MSG_DIFF_TTL

Quellcode B.31: Aufbau der Differenz-TTL-Nachricht

```
struct msg_ttldiff {
    unsigned int sip,dip;
    unsigned short int dir;
```

```
4     unsigned int numback;
      u_int8_t ttl;
6     u_int8_t ttldiff;
  };
```

B.3.4 Korrelator

B.3.4.1 MSG_KORRELATOR_RTT

Quellcode B.32: Aufbau der Korrelator-RTT-Nachricht

```
  struct msg_rttkorr {
2     unsigned int sip;
      unsigned int dip;
4     struct timeval zeit;
      unsigned short int dir;
6     unsigned int flags;
      double rtt;
8     double ortt;
      double diffrtt;
10 };
```

B.3.4.2 MSG_KORRELATOR_TTL

Quellcode B.33: Aufbau der Korrelator-TTL-Nachricht

```
  struct msg_ttlkorr {
2     unsigned int sip;
      unsigned int dip;
4     struct timeval zeit;
      unsigned short int dir;
6     unsigned int flags;
      u_int8_t ttl;
8     u_int8_t ottl;
      u_int8_t diffttl;
10 };
```

B.3.5 Vergleicher

Der Vergleicher erzeugt bei dem Vergleich zweier RTT-Wertströme die Nachricht MSG_TCPANALYSE_RTT. Der Unterschied besteht jedoch darin, dass es sich bei den RTT und EstRTT-Werten um Differenzen zwischen zwei oder mehreren RTT-Strömen handelt. Mit der Erzeugung dieser Nachricht ist es möglich den Datenstrom wieder an einen Sampler oder Differenz-Erzeuger zu senden.

B.3.6 FileWriter

B.3.6.1 TLV-Nachricht

Quellcode B.34: Allgemeiner Aufbau einer TLV-Nachricht für den Filewriter

```
struct tlventry {
    unsigned short int type;
    unsigned short int len;
    void *data;
};

struct file_tlv {
    unsigned short int numtlvs;
    struct tlventry entries[];
};
```

Für den Filewriter sind folgende Typen mit den entsprechenden Längen in der Tablle B.3 definiert.

Tabelle B.3: Typen und Längen für eine TLV-Nachricht

Typ	Länge
TYPE_SINTEGER	4
TYPE_UINTEGER	4
TYPE_STRING	$0 < len < 65535$
TYPE_DOUBLE	8
TYPE_USHORT	2
TYPE_SSHORT	2
TYPE_BYTE	1

C. Konfiguration des Hauptprogramms

Bei der Konfigurationsdatei für das Hauptprogramm handelt es sich um eine XML-Datei. Zum Parsen der Konfigurationsdatei wird eine C++ Vartiante der *libxml2* verwendet. Diese Library nennt sich *libxml++*. Es wird die Version 2.6.x der Bibliothek benutzt. Die Konfigurationsdatei enthält eine Liste der Module, die zu Beginn eines Knotens geladen werden sollen. Des Weiteren enthält sie für jedes Modul spezifische Parameter, die ein Modul benötigt um ausgeführt zu werden. Es soll nun die Struktur der Konfigurationsdatei beschrieben werden. Zuerst beinhaltet diese Konfiguration ein *Startbehavior*-Tag

Quellcode C.1: Konfiguration des Startverhaltens

```
<startbehavior mode="infront">
    <option name="debug"/>
</startbehavior>
```

Dieser Tag enthält das Attribute “mode”. Mit diesem Attribut wird dem Programm mitgeteilt, ob dieses im Vordergrund (infront) oder Hintergrund (daemon) laufen soll. Als Subtag kann ein *Startbehavior* ein oder mehrere *option*-Tags enthalten. Momentan ist nur ein *option*-Tag definiert. Es trägt den Namen *debug* und zeigt dem Programm an, ob mehr oder weniger Ausgaben auf den Bildschirm gemacht werden sollen. Nun folgt der *modules*-Tag Abschnitt.

Quellcode C.2: Liste der Module

```
<modules modulepath="/path/to/modules/">
    <module ... >
     ...
    </module>
     ...
</modules>
```

In diesem Tag sind die Module eingeschlossen, die beim Start des Hauptprogramms ausgeführt werden sollen. Außerdem enthält dieser Tag das Attribute *modulepath*. Damit wird der Pfadpräfix festgelegt, in dem alle Module zu finden sind. In einem oder mehreren folgenden *module*-Tags werden die einzelnen Module beschrieben, die ausgeführt werden sollen.

Quellcode C.3: Modulbeschreibung

```
<module mame="Modulename" filename="filename.so">
    <parameter ...></parameter>
    ...
</module>
```

Ein *module*-Tag charakterisiert ein Modul. Zum einen gibt das Attribut *name* den Modulnamen an. Zum anderen wird im Attribut *filename* der Dateiname des Moduls hinterlegt. Jedes *module*-Tag kann keinen, einen oder mehrere Parameter-Tags aufnehmen, die Parameter des Programms beschreiben und festlegen. Ein *parameter*-Tag ist immer wie folgt aufgebaut:

Quellcode C.4: Parametereintrag innerhalb eines Moduls

```
<parameter name="Varname"
           type="VarType"
           access="Accright">Value
</parameter>
```

Der Tag enthält die drei Attribute *name, type* und *access*. Dabei gibt *name* den Namen des Parameters an. Über diesen kann später im Modul per ModuleInfo-Objekt auf den Inhalt der Variablen zugegriffen werden. *Type* gibt den Typ der Variablen an. Dies kann entweder *string* oder *integer* (unsigned int) sein. In *access* wird das Zugriffsrecht auf diese Variable im Programm festgelegt. Dies kann entweder *read* oder *readwrite* sein. Wird im Modul auf eine nur lesende Variable schreibend zugegriffen, so wird der Wert der Variablen nicht geändert. Ein Variable, die das Schreibrecht besitzt, kann auch verändert werden. Eingeschlossen im öffnenden und schließenden *parameter*-Tag befindet sich der Wert der Variablen. Eine vollständige Konfiguration hat damit nur die folgende Gestalt:

Quellcode C.5: Konfiguration des Hauptprogramms

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<config>
  <startbehavior mode="infront">
       <option name="debug"/>
  </startbehavior>
  <modules
    modulepath="/home/bjoern/projekte/Diplomarbeit/source/bin/">
  <module name="Loader"
           filename="mod_loader.so">
       <parameter name="moduleID"
```

```
                    type="integer"
12                  access="read">
          1
14        </parameter>
          <parameter name="modpath"
16                  type="string"
                    access="read">
18        /home/bjoern/projekte/Diplomarbeit/source/bin/
          </parameter>
20        <parameter name="configpath"
                    type="string"
22                  access="write">
          /home/bjoern/projekte/Diplomarbeit/source/config/
24        </parameter>
      </module>
26    <module name="Socket␣Multiplexer"
              filename="mod_smux.so">
28        <parameter name="moduleID"
                    type="integer"
30                  access="read">
          2
32        </parameter>
          <parameter name="proto"
34                  type="string"
                    access="read">
36        tcp
          </parameter>
38        <parameter name="port"
                    type="integer"
40                  access="read">
          4711
42        </parameter>
      </module>
44   <module name="Logger"
              filename="mod_logger.so">
46        <parameter name="location"
                    type="string"
48                  access="read">
          local
50        </parameter>
          <parameter name="prefix"
52                  type="string"
                    access="read">
54        </parameter>
          <parameter name="options"
56                  type="string"
```

```
                              access="read">
58            LOG_NDELAY LOG_NOWAIT
              </parameter>
60            <parameter name="facility"
                              type="string"
62                            access="read">
              LOG_LOCAL7
64            </parameter>
        </module>
66      </module>
        <module name="Knoten␣Management"
68                 filename="mod_management.so">
                       <parameter name="port"
70                            type="integer"
                              access="read">
72            47110
              </parameter>
74      </module>
    </modules>
76 </config>
```

Literatur

[ABCKM^{+}04] Brent Chun Andy Bavier, Mic Bowman, David Culler, Scott Karlin, Steve Muir, Larry Peterson, Timothy Roscoe, Tammo Spalink und Mike Wawrzoniak. Operating System Support for Planetary-Scale Services. In *roceedings of the First Symposium on Network Systems Design and Implementation (NSDI)*, März 2004.

[ABKM01a] D. Andersen, H. Balakrishnan, M. Kaashoek und R. Morris. The case for resilient overlay networks. In *Proceedings of the 8th Annual Workshop on Hot Topics in Operating Systems (HotOSVIII) , May 2001*, Mai 2001.

[ABKM01b] David Andersen, Hari Balakrishnan, M. Frans Kaashoek und Robert Morris. Resilient Overlay Networks. In *Proc. 18th ACM Symposium on Operating Systems Principles*, Banff, Canada, October 2001.

[AlPS99] M. Allman, V. Paxson und W. Stevens. TCP Congestion Control. RFC 2581 (Proposed Standard), IETF, Apr. 1999.

[bala96] TCP Behavior of a Busy Internet Server: Analysis and Improvements. In H. Balakrishnam und et. al. (Hrsg.), *Proceedings of IEEE INFOCOMM 1998*, 1996, S. 252–262.

[bolo93] End-to-End Packet Delay and Loss Behavior in the Internet. In J-C. Bolot (Hrsg.), *Proceedings of SIGCOMM 1993*, Sept. 1993, S. 289–298.

[BrJa88] R. Braden und V. Jacobson. TCP Extentions for long-delay paths. RFC 1072 (UNKNOWN), IETF, Okt. 1988.

[CeWF] Cellatoglu, Worrall und Fabri. Performance of RTP/UDP/IP header compression in cellular networks. www.ee.surrey.ac.uk/Personal/S.Worrall/Publications/lcs2k_crtp.pdf.

[CLCo04] Jiri Navratil Connie Logg und Les Cottrell. Correlating Internet Performance Changes and Route Changes to Assist in Troubleshooting from an End-user Perspective. In *Proceedings of 5th*

Passive and Active Measurement Workshop (PAM 2004), Juan-les-Pins, France, April 2004.

[Comb] Gerald Combs. Ethereal. www.ethereal.com.

[DATi03] Goujun Jin Deb Agarwal, José María González und Brian Tierney. An Infrastructure for Passive Network Monitoring of Application Data Streams. In *Proceedings of Passive an Active Measurement Workshop 2003 (PAM 2003), La Jolla, Kalifornien/USA*, April 2003.

[Dege01] M. Degermark. Requirements for robust IP/UDP/RTP header compression. RFC 3096 (Proposed Standard), IETF, Juli 2001.

[DeHi98] S. Deering und R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460 (Draft Standard), IETF, Dez. 1998.

[DENP97] M. Degermark, M. Engan, B. Nordgren und S. Pink. Low-loss TCP/IP Header Compression for Wireless Networks. In *Proceesings of MobiCom '96*. ACM, 1997, S. 375 – 387.

[DeNP99] M. Degermark, B. Nordgren und S. Pink. IP Header Compression. RFC 2507 (Proposed Standard), IETF, Feb. 1999.

[Deut96] P. Deutsch. DEFLATE Compressed Data Format Specification version 1.3. RFC 1951 (INFORMATIONAL), IETF, Mai 1996.

[dhts02] Security Considerations for Peer-to-Peer Distributed Hash Tables. In E. Sit und R. Morris (Hrsg.), *Proceedings of IPTPS 2002*, 2002.

[eal.01] C. Bormann et al. RObust Header Compression (ROHC): Framework and four profiles: RTP, UDP, ESP, and uncompressed. RFC 3095 (Proposed Standard), IETF, Jul. 2001.

[EaRS02] D. Eastlake, J. Reagle und D. Solo. (Extensible Markup Language) XML-Signature Syntax and Processing. RFC 3275 (Draft Standard), IETF, März 2002.

[FHSZ02] Thomas Fuhrmann, Till Harbaum, Marcus Schöller und Martina Zitterbart. AMnet 2.0: An Improved Architecture for Programmable Networks. In *Proceedings of the International Workshop on Active Networks IWAN2002, Zürich, Switzerland*, December 2002.

[Hein99] Mathias Hein. *TCP/IP Internet-Protokolle im professionellen Einsatz*. International Thomson Publishing. 1999.

[Henz99] Norbert Henze. *Stochastik für Einsteiger*. Vieweg-Verlag. 1999.

[HoAd87] M. R. Horton und R. Adams. Standard for interchange of USENET messages. RFC 1036 (UNKNOWN), IETF, Dez. 1987.

[HoMi01] U. Hofmann und I. Milouche. Distributed Measurement and Monitoring in IP Networks. In *Proceedings of SCI 2001 /ISAS 2001 Orlando 7/2001*, Jul. 2001.

[HSSW+02] A. Hess, M. Schöller, G. Schäfer, A. Wolisz und M. Zitterbart. A dynamic and flexible Access Control and Resource Monitoring Mechanism for Active Nodes. In *Proceedings of the 5th International Conference on Open Architectures and Network Programming (OPENARCH) (Short Paper Session)*, 2002.

[Jacoa] V. Jacobson. PCAP. www.tcpdump.org.

[Jacob] V. Jacobson. tcpdump. www.tcpdump.org.

[Jacoc] V. Jacobson. Traceroute. ftp.ee.lbl.gov/traceroute.tar.Z.

[JFSJ90] J.Case, M. Fedor, M. Schoffstall und J.Davin. A Simple Network Manaagement Protocol (SNMP). RFC 1157 (HISTORIC), IETF, Mai 1990.

[JMRW93] J.Case, K. McCloghrie, M. Rose und S. Waldbusser. Indroduction to version 2 of the Internet-standard Network Management Framwork. RFC 1441 (Proposed Standard), IETF, April 1993.

[Jons04] L-E. Jonsson. RObust Header Compression (ROHC): A Compression Profile for IP. RFC 3843 (Proposed Standard), IETF, 2004.

[KiDa01] Olaf Kirch und Terry Dawson. *Linux - Wegweiser für Netzwerker*. O Reilly Verlag. 2001.

[KLLL+97] David Karger, Eric Lehman, Tom Leighton, Mathhew Levine, Daniel Lewin und Rina Panigrahy. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *ACM Symposium on Theory of Computing*, Mai 1997, S. 654–663.

[LeZi97] A. Lempel und J. Ziv. A Universal Algorithm for Sequential Data. *IEEE Transactions on Information Theory* 23(3), 1997, S. 337–343.

[mell02] Measureing IP and TCP Behavior with Tstat. In F. Neri M. Mellia, R. Lo Cigno (Hrsg.), *Proceedings of IEEE GLOBECOM 2002*, 2002.

[MHHLS+02] Joseph M. Hellerstein Matthew Harren, Ryan Huebsch, Boon T. Loo, Scott Shenker und Ion Stoica. Complex Queries in DHT-based Peer-to-Peer Networks. In *Proceedings for the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02) , Cambridge, USA*, März 2002.

[MMFR96] M. Mathis, J. Mahdavi, S. Floyd und A. Romanow. TCP Selective Acknowledgement Options. RFC 2018 (Proposed Standard), IETF, Okt. 1996.

[PACR02] Larry Peterson, Tom Anderson, David Culler und Timothy Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proceedings of HotNets–I*, Princeton, New Jersey, October 2002.

[padh01] On Inferring TCP Behavior. In J. Padhye und S. Floyd (Hrsg.), *Proceedings of ACM SIGGCOM 2001*, 2001.

[PAMM98] V. Paxson, G. Almes, K. Mahdavi und M. Mathis. Framework for IP Performance Metrics. RFC 2330 (Informational), IETF, May. 1998.

[Paul04] Jr. Paul Penfield. Compression, Feb. 2004.

[Paxs97] Vern E. Paxson. *Measurements and Analysis of End-to-End Internet Dynamics*. PhD Dissertation, University of California, Lawrence Berkeley National Laboratory, April 1997.

[paxs98] An Architecture for Large-Scale Internet Measurement. In V. Paxson et al. (Hrsg.), *IEEE Communications, 1998*, Aug. 1998, S. 48–54.

[paxs01] Automated Packet Trace Analysis of TCP Implementations. In V. Paxson (Hrsg.), *Proceedings of ACM SIGGCOMM 1997, Cannes, France*, 2001, S. 167 – 179.

[Peuh01] M. Peuhkuri. A method to compress and anonymize packet traces. In *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, Nov. 2001, S. 257 – 263.

[Post81] J. Postel. Transmission Control Protocol. RFC 793 (Standard), IETF, Sep. 1981.

[RHLLS+03] Joseph M. Hellerstein Ryan Huebsch, Nick Lanham, Boon Thau Loo, Scott Shenker und Ion Stoica. Querying the Internet with PIER. In *Proceedings of 29th Conference on Very large Databases 2003, Berlin*, Sept. 2003.

[RoPe03] T. Roscoe und L. Peterson. PlanetLab: A Simple Common Sensor Interface for PlanetLab. Draft, März 2003.

[RoVC01] E. Rosen, A. Viswanathan und R. Callon. Multiprotocol Label Switching Architecture. RFC 3031 (Proposed Standard), IETF, Jan. 2001.

[Schn96] Bruce Schneier. *Angewandte Kryptographie*. Addison-Wesley Verlag. 1996.

[Scho02] R. Schollmeier. A Definition of Peer-to-Peer Networking for Classification of Peer-To-Peer Architectures and Applications. In *Proceedings of the First International Conference on Peer-to-Peer Computing (P2P'01)*. IEEE, 2002.

[Schä03] Günter Schäfer. *Netzsicherheit*. dpunkt.verlag. 2003.

[Sedg97] Robert Sedgewick. *Algorithmen*. Addison-Wesley Verlag. 1997.

[Shir00] Clay Shirky. What is P2P... And What Isn't? www.openp2p.com/pub/a/p2p/2000/11/shirky1-whatisp2p.html, Nov. 2000.

[SiRi04] C. Simpson und G. Riley. NETI@home: A Distributed Approach to Collecting End-to-End Network Performance Measurements. In *Proceedings of 5th Passive and Active Measurement Workshop (PAM 2004), Juan-les-Pins, France*, April 2004.

[SMKK+01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek und Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of ACM SIGCOMM*. ACM Press, 2001, S. 149–160.

[Tane00] Andrew S. Tanenbaum. *Computernetzwerke*. Pearson Studium. 2000.

[Trev03] Luca Trevisan. Data Compression via Huffman Coding. U.C. Berkeley, 2003.

[WiDP99] B. Wijnen, D.Harrington und R. Presuhn. An Architecture for Describing SNMP Management Frameworks. RFC 2571 (Draft Standard), IETF, April 1999.

[WiDP02] B. Wijnen, D.Harrington und R. Presuhn. An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks. RFC 3411 (Standard), IETF, Dez. 2002.

[Zülc04] B. Zülch. Entwicklung eines FlexiNet-Dienstes zur passiven Messung von TCP-Verbindungeigenschaften. Studienarbeit, Institut für Telematik, Fakultät für Informatik, Universität Karlsruhe (TH), März 2004.