INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Diplomarbeit

Abbildung von QoS-Parametern auf eine geschichtete Messarchitektur

Florian Uhl

Aufgabensteller: Prof. Dr. Heinz-Gerd Hegering
Betreuer: Markus Garschhammer

Abgabetermin: 06. November 2003

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Diplomarbeit

Abbildung von QoS-Parametern auf eine geschichtete Messarchitektur

Florian Uhl

Aufgabensteller: Prof. Dr. Heinz-Gerd Hegering
Betreuer: Markus Garschhammer

Abgabetermin: 06. November 2003

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

München, den 06. November 2003

. . . .

(Unterschrift des Kandidaten)

Zusammenfassung

Diese Diplomarbeit beschäftigt sich mit der Abbildung von QoS-Parametern auf eine geschichtete Messarchitektur mit Hilfe einer Abbildungssprache ('Eqosul'). Als Voraussetzung für diese Betrachtung benötige ich eine Sammlung von QoS-Parametern. Diese gewinne ich aus drei verschiedenen Quellen (ITU-T, IPPM und FIPA), ergänzt um mögliche weitere Parameter, die für die Fragestellung interessant sein könnten.

Desweiteren betrachte ich eine Liste von Protokollmechanismen und deren Auswirkungen auf QoS-Parameter. Die Ergebnisse dieser Teiluntersuchung verwende ich in einem 'sanity check' zur rudimentären Üeberprüfung der Modellierung und Abbildung der oben genannten QoS-Parameter.

Zuletzt beschäftigt sich diese Arbeit damit, inwieweit die Fragestellung vollständig abgedeckt wurde und welche weiterführenden Fragen verbleiben. Als Ergebnis lässt sich feststellen, daß zwar die betrachtete Messarchitektur generisch genug für die Modellierung und Messung der von mir betrachteten QoS-Parameter ist, jedoch sowohl die Definitionen der QoS-Parameter als auch der Protokollmechanismen noch zu große Spielräume bei der Modellierung offen lassen. Dies führt zu semantisch unterschiedlichen Implementierungen bei der Messung derselben QoS-Parameter.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einführung

1.1 Motivation

Das informatische Grundkonzept der Schichtenbildung und Verschattung ist im Bereich der Rechnernetze besonders deutlich bei der Implementierung von Protokollstacks zu beobachten. Für den Benutzer eines Protokolls sind lediglich funktionale Schnittstellen sichtbar, die eigentliche Implementierung ist transparent. Damit bleibt auch der Aufbau komplizierter Anwendungsprotokolle übersichtlich.

Mit der zunehmenden Betrachtung des **Quality of Service** (**QoS**) stellt sich aber eben diese Schichtung in sich gekapselter Protokolle als Problem dar: Eine detaillierte Analyse der Performance eines Protokollstacks ist auf Grund der verschatteten Implementierung nur schwer möglich. Ein genaue Darstellung der Mechanismen und Parameter, die den QoS eines Protokollstacks beeinflussen, wird aber als Voraussetzung zur Vorhersage des von der Implementierung erbrachten QoS gesehen.

Eine Möglichkeit, Informationen über die Performance und das Verhalten einer Protokollimplementierung zu ermitteln, besteht darin, die Implementierung als 'Black Box' zu betrachten und ihr Verhalten in Abhängigkeit unterschiedlicher Ausgangssituationen (z.B. verschiedene Eingangsdatenströme, unterschiedlich viele Instanzen, usw.) zu analysieren. Mit den gewonnenen Daten lässt sich ein Modell der Implementierung konstruieren, das dann zur Vorhersage des QoS verwendet werden kann. Dabei sind unterschiedlichste Modelle vom endlichen Automaten bis zum statistischen Prozess denkbar.

In dieser Diplomarbeit sollen zunächst mögliche Einflussgrössen der Implementierung auf bestimmte Parameter ermittelt werden, um grundsätzliche Möglichkeiten der QoS-Beeinflussung durch die Implementierung zu ermitteln. Das Verhalten eines Protokolls soll sowohl durch grundsätzliche Analyse von bekannten Protokollmechanismen wie Framing, Segmentierung, etc. als auch durch die empirische Betrachtung vorhandener Implementierungen ermittelt werden. Eine Ausdehnung der Untersuchung auf alle Schichten des OSI-Modells soll eine generische Betrachtung der QoS-Problematiken im Protokollstack und gleichzeitig eine Klassifizierung von QoS-Parametern und ihre Abhängigkeiten von den Implementierungen ermöglichen.

1.2 Fragestellung

Diese Diplomarbeit beschäftigt sich mit der Abbildung von QoS-Definitionen auf eine geschichtete Messarchitektur (siehe [Neu 02]).

Die hier verwendete und zu untersuchende Messarchitektur ist in [Neu 02] bereits ausreichend beschrieben worden. Daher beschränkt sich diese Arbeit bezüglich der Beschreibung auf eine kurze Einführung (Kapitel 2.1) und verweist für Details auf die genannte Arbeit.

Ich habe mich entschieden, QoS-Definitionen aus drei Quellen zu entnehmen. Zum einen werde ich die QoS-Parameter der **IP Performance Metrics Group** (**IPPM**) der **Internet Engineering Task Force** (**IETF**) [RFC 2330, RFC 2678, RFC 2679, RFC 2680, RFC 2681] betrachten. Desweiteren betrachte ich eine Auswahl der von der **International Telecommunications Union** (**ITU-T**) verwendeten Definitionen [X.641]. Zuletzt werde ich Teile der von der **Foundation for Intelligent Physical Agents** (**FIPA**) vorgeschlagenen QoS-Ontologie [FIPA QoS] verwenden. Dies ermöglicht auf der einen Seite die Betrachtung sehr eng definierter QoS-Parameter, da sich die IPPM lediglich mit den OSI-Schichten 3 und 4 und die FIPA lediglich mit mobilen Agenten beschäftigt. Auf der anderen Seite finden sich bei den von der ITU-T verwendeten Definitionen sehr generische Parameter wie z.B. 'capacity' die sich auf alle Schichten des OSI-Modells (und darüber hinaus) anwenden lassen.

Alle Quellen verwenden eigene (Semi-) Formalismen zur Beschreibung der QoS-Parameter. Ein Ziel dieser Arbeit ist es, die Abbildung der betrachteten QoS-Parameter auf die geschichtete Messarchitektur als (ausreichenden) Formalismus zur Beschreibung darzustellen. Dies ermöglicht es erst, QoS-Parameter verschiedener Quellen auf ihre Ähnlichkeit zu untersuchen. Selbst für Parameter aus der selben Quelle wird der Vergleich durch diese Abbildung auf unsere 3-schichtige Architektur erleichtert. Denn nach Abbildung auf die Messarchitektur können QoS-Parameter sehr leicht auf Übereinstimmungen in den 3 Modulen überprüft werden. Beispiel: QoS-Parameter A und B stimmen in den Modulen **dc** und **ec** überein, sie unterscheiden sich nur in **sp**. Durch diese Art von Vergleich lassen sich Klassen von QoS-Parametern bilden und der Anwender kann nun z.B. verschiedene Provider gegeneinander abgrenzen indem er die Bereitstellung bestimmter Klassen von QoS-Parametern überprüft.

Desweiteren erleichtert die Abbildung mittels des hier verwendeten Formalismus' auch die Vergleichbarkeit von QoS-Messungen. Da der Formalismus eine eindeutige Definition des Parameters darstellt -im Gegensatz zu den der Interpretation offenstehenden semiformalen oder umgangssprachlich definierten QoS-Parametern wie sie bis jetzt verwendet werden- werden Messungen verschiedener Personen oder Institutionen vergleichbar.

Zur Durchführung der Abbildung wird -wie erwähnt- eine eigens zu diesem Zweck entworfene Sprache namens Eqosul verwendet. Sie ermöglicht eine Abbildung der Definitionen auf die drei Blöcke der Messarchitektur ('data collector', 'event correlator' und 'statistic processing'). Eine Betrachtung von semantisch äquivalenten (aber syntaktisch unterschiedlichen) Abbildungsmöglichkeiten findet aus Zeitgründen nicht statt und würde den Rahmen der Arbeit sprengen. Sie verbleibt 'for further studies' und muss in späteren Arbeiten geklärt werden. Da die vorliegenden QoS-Parameter-Definitionen nicht eindeutig interpretierbar sind, können auch die mittels Eqosul erstellten Abbildungen nicht eindeutig sein. D.h. die Abbildung eines QoS-Parameters x mittels Eqosul kann mehreren Ergebnissen x1, x2, ... führen. Um dieses Problem zu vermeiden müsste eine Abbildungsvorschrift erstellt werden, die Regeln zur Abbildung zwischen den vorliegenden QoS-Parameter-Definitionen und Eqosul angibt. Auf der anderen Seite offenbart die Möglichkeit, einen QoS-Parameter in Eqosul auf mehr als eine Weise modellieren zu können die Spielräume in deren Definitionen.

Zuletzt werde ich die Abbildung einem 'sanity check' unterziehen um sicherzustellen, daß keine redundante oder unzureichende Kodierung stattfindet. Redundante Kodierung ist dadurch erkennbar, daß die Änderung eines Parameters der Messarchitektur zu keiner Änderung des gemessenen QoS-Parameters führt. Unzureichende Kodierung wird durch 'falsche' Messergebnisse erkannt (i.e. es wird ein anderer QoS-Parameter gemessen als intendiert). Als Vorüberlegung zu diesem *sanity check* werde ich die Auswirkungen ausgewählter Protokollmechanismen auf QoS-Parameter untersuchen.

Die Generik der Messarchitektur wird ausschließlich exemplarisch an Hand der betrachteten QoS-Parameter untersucht. Ich werde keine allgemeinen Aussagen über eine 'globale Generik' treffen.

Zusätzlich zu den offensichtlichen Ergebnissen dient diese Arbeit der Feststellung, welche QoS-Parameter inhärent 'ähnlich' sind. Dies wird durch Betrachtung der einzelnen Blöcke der Abbildung geschehen. Informell stellen sich zwei QoS-Parameter als ähnlich dar wenn sie, bei **dc** beginnend über **ec** zu **sp** hin, in einem möglichst 'späten' Modul voneinander abweichen.

1.3 Vorüberlegungen

Zu Beginn der Diplomarbeit wollte ich sicherstellen, daß die von mir erwarteten Auswirkungen von Protokollmechanismen auf QoS-Parameter überhaupt auftreten und messbar sind. Zudem sollten die dabei erzielten Messungen bereits erste Einblicke in die Lösung der Fragestellung geben. Zu diesem Zweck entwickelte ich ein prototypisches Messprogramm sowie einige Hilfsprogramme dazu. Dieses System nenne ich **proof of idea** (**poi**).

Die Idee der Programme ist denkbar einfach. Ein 'client' (auch Sender genannt, siehe A.1) auf Maschine A baut eine Verbindung zum 'server' (auch Empfänger genannt, siehe A.2) auf Maschine B auf. Er sendet in auf der Kommandozeile konfigurierbaren Abständen eine vordefinierte Anzahl an vorgegebenen Daten. Sowohl Sender als auch Empfänger zeichnen die inter-frame-delays oberhalb der OSI-Schicht 4 auf. Diese Daten werden, um den Einfluss von I/O-Operationen zu minimieren, während des Testlaufs in dynamisch alloziertem Speicher gehalten und erst nach Abschluss der Messung in eine Datei geschrieben.

Diese Messdaten können mittels einiger weniger, zumeist selbst erstellter Hilfsprogramme (siehe A.3, A.4, A.5, A.6 und A.7, zudem kamen 'gnuplot', 'tcpdump' und 'tethereal' zum Einsatz) bearbeitet und im Anschluss daran statistisch und graphisch aufbereitet werden. Anhand eines konkreten Beispiels (siehe Abbildung 1.2) werde ich das Ergebnis dieser Aufbereitung kurz erläutern.

Die x-Achse stellt den inter-frame-delay in **microseconds** (10^{-6} Sekunden), die Y-Achse die auf 1 normalisierte Anzahl an Paketen dar. Die rote Kurve steht für die Messung auf dem Sender, die grüne für die Messung auf dem Empfänger. Zudem enthält die Legende am rechten oberen Rand des Diagramms statistische Werte wie sie im Rahmen von sogenannten Boxplots vorkommen. Der erste Wert in Klammern ist das Minimum der gemessenen Zeiten, der letzte Wert das Maximum der gemessenen Zeiten. Die Werte zwischen den Pipes stellen den linken Whisker, das Viertelquartil, den Median, das Dreiviertelquartil und den rechten Whisker der gemessenen inter-frame-delays dar.

Die Tests zu den Ergebnissen, die in den Abbildungen 1.2 und 1.3 gezeigt werden, waren wie folgt aufgebaut: Szenario 1 (Abbildung 1.1 oben) stellt eine Verbindung zwischen Client und Server über das Loopback-Interface dar. Der Client wurde aufgerufen mit 'poi-client pcheger2 4711'. Szenario 2 (Abbildung 1.1 unten) stellt eine Verbindung zwischen Client und Server über mehrere Transitnetze mittels eines **Secure Shell** (**ssh**)-Tunnels dar. Der Client wurde ebenfalls mittels 'poi-client pcheger2 4711' aufgerufen. Jedoch wurde der Server auf Port 4712 konfiguriert und Port 4711 wurde über eine ssh-Verbindung zu ankh.is-a-geek.org:4712 umgeleitet. Von dort wurde die Verbindung ebenfalls wieder über eine ssh-Verbindung zurück auf pcheger2 auf Port 4712 geleitet. Dies stellt (aus Sicht des Endbenutzers) sicherlich einen funktional transparenten Austausch der involvierten Transitsysteme -sowie damit auch der OSI-Schichten 1-2- dar. Siehe hierzu Abbildung 1.1.

Das Ergebnis von Szenario 1 (Abbildung 1.2) war wie erwartet: Der Client sollte die Daten im Abstand von 500 microseconds senden. Er zeichnete tatsächliche inter-frame-delays von ca. 530 microseconds für die Mehrheit der Daten auf. Wie anhand der Whisker-Werte (526 microseconds für den linken und 533 microseconds für den rechten) ersichtlich, gab es nahezu keine Schwankungen des inter-frame-delays. Die (nahezu konstante) Differenz von etwa 30 microseconds zwischen dem angestrebten und dem tatsächlich erreichten inter-frame-delay führe ich auf den Overhead der Systemaufrufe und des Netzwerkstacks zurück. Da keine großen Datenmengen übermittelt wurden und auch die Abstände zwischen den gesendeten Daten (im Kontext des Betriebssystems) nicht sehr klein waren, führe ich die oberhalb des rechten Whiskers liegenden Werte überwiegend auf Effekte des Scheduling zurück. Die unterhalb des linken Whiskers liegenden Werte sind vermutlich Auswirkungen davon, daß eine Verzögerung im Versand des Pakets n zu einer (relativ zu Paket n) früheren Versendung von Paket $n + 1$ führt. Der Server empfing die Daten in diesem Szenario, ebenfalls wie erwartet, mit ähnlich geringen Abweichungen. Hier betrug der Wert des linken Whiskers 525 microseconds, der des rechten wiederum 533 microseconds. Median, Viertel- und Dreiviertelquartil sind identisch mit den Werten des Clients. Die Ausreißer oberhalb des rechten Whiskers sind etwas extremer, da der Server unter anderem auf Grund des 3-way-handshaking den ersten inter-frame-delay als sehr hoch errechnet. Dies ist ein bewusst in Kauf genommener Fehler des prototypischen

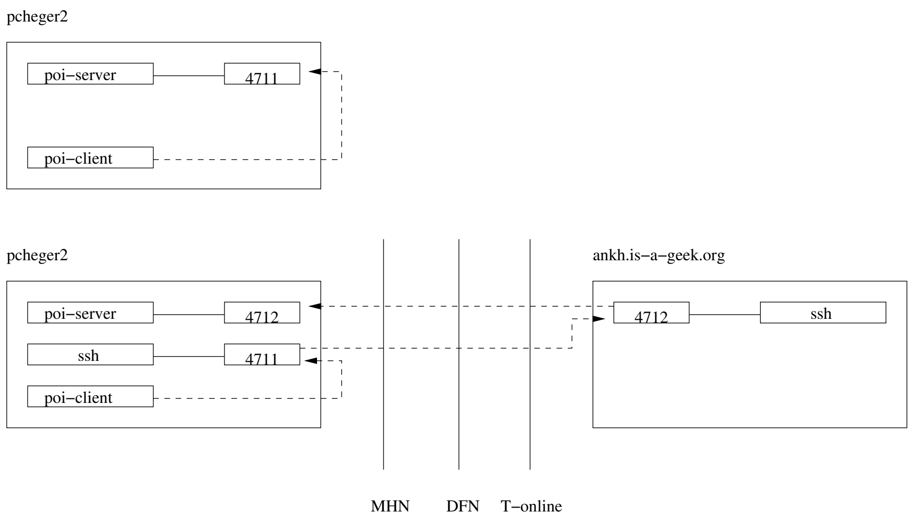

Abbildung 1.1: Austausch der Transitsysteme

Messprogramms, der jedoch nicht sehr stark in Erscheinung tritt. Dementsprechend sind die Extrema unterhalb des linken Whiskers teilweise noch kleiner als es beim Client der Fall war.

Betrachtet man nun das zweite Szenario (Abbildung 1.3), ergeben sich doch deutliche Unterschiede. Die vom Client gemessenen Werte unterscheiden sich zwar nur marginal (zwischen 1 und 4 microseconds sofern man die Extrema außer Acht lässt) von den in Szenario 1 vom Client ermittelten, die Messung des Servers jedoch weicht erheblich von denen des Servers in Szenario 1 ab. Die Werte für den linken und rechten Whisker, den Median und die beiden Quartile wurden alle mit dem Wert 4 microseconds ermittelt. Die Extrema oberhalb des rechten Whiskers waren jedoch sowohl in Quantität als auch in Qualität stärker ausgeprägt als es in Szenario 1 der Fall war. Dieses Ergebnis ist auf den ersten Blick völlig unerwartet, lässt sich meiner Ansicht nach aber wie folgt erklären: Der ssh-Tunnel verwendet sowohl Datenkomprimierung als auch Datenverschlüsselung. Datenkomprimierung ist am effektivsten wenn es auf möglichst große Blöcke von Daten angewendet wird. Auch Datenverschlüsselung wird üblicherweise mit Blöcken von Daten durchgeführt um nicht eine sogenannte 'known plain text'-Attacke mit Hilfe der dann notwendigen pad-Bytes zu ermöglichen. Es ist daher anzunehmen, daß der ssh-Client die Daten blockt. Unter dieser Annahme sind die Ergebnisse leicht erklärbar. Der ssh-Client wartet auf einen ausreichend gefüllten Puffer. Während dieser Füllzeit erreichen natürlich keine Daten den poi-Server. Es entsteht ein sehr großer inter-frame-delay. Ist der Puffer dann ausreichend gefüllt, werden die (völlig gleichförmigen) Daten sehr gut komprimiert und übertragen. Der ssh-Client dekomprimiert die Daten auf der Seite des poi-Servers und übergibt sie an diesen. Sehr viele Daten treffen 'auf einmal' beim poi-Server ein. Es entstehen natürlich sehr geringe inter-frame-delays für diese, im Block versandten, Daten. Der Wert von 4 microseconds ist vermutlich die Minimalzeit, die der Netzwerkstack des Empfängersystems selbst bei voller Queue benötigt, um die Daten an die oberen Schichten (5+ im OSI-System) weiterzureichen.

Im Ergebnis lässt sich feststellen, daß ein aus Sicht des Anwenders funktional transparenter Austausch der Transitsysteme zu doch sehr erheblichen Änderungen des gemessenen QoS 'inter-frame-delay' geführt hat. Dies wurde auch durch die Auswertung weiterer Testszenarien verdeutlicht.

Daher erscheint es sinnvoll und notwendig, sich im Rahmen einer Diplomarbeit mit einer solchen Fragestellung zu beschäftigen.

Abbildung 1.2: Szenario 1 – loopback-Verbindung

Abbildung 1.3: Szenario 2 – ssh-Tunnel

1.4 Vorgehensmodell

Schwerpunkt dieser Arbeit ist die Abbildung von QoS-Parametern auf [Neu 02]. Um diese Abbildung durchführen zu können, benötige ich eine Sammlung von QoS-Parametern. Dazu werde ich in Abschnitt 3.2 exemplarisch verschiedene Quellen von QoS-Parametern betrachten. Zusätzlich zu den QoS-Parametern benötige ich einen geeigneten Formalismus zur Abbildung dieser Parameter auf [Neu 02]. Hierzu verwende ich eine eigens entwickelte Abbildungssprache ([Gar 04]) die ich in Abschnitt 2.2 kurz vorstellen werde. Die Abbildung der QoS-Parameter auf [Neu 02] wird in Kapitel 4 beschrieben, eine kurze Beschreibung der Architektur selbst erfolgt in Abschnitt 2.1.

Anschließend werde ich diese Abbildung einem 'sanity check' unterziehen. Ich werde untersuchen, ob Veränderungen des QoS durch die Messarchitektur tatsächlich wahrgenommen werden. Grundlage dafür ist eine Untersuchung dessen, wie QoS-Parameter überhaupt beeinflusst werden können. Dazu werde ich in Abschnitt 3.3 die Auswirkungen von Protokollmechanismen auf QoS-Parameter exemplarisch untersuchen. Die Auswahl und Beschreibung der betrachteten Protokollmechanismen erfolgt im Abschnitt 3.1 zusammen mit der Betrachtung der QoS-Parameter-Definitionen.

Abbildung 1.4 stellt dieses Vorgehen graphisch dar.

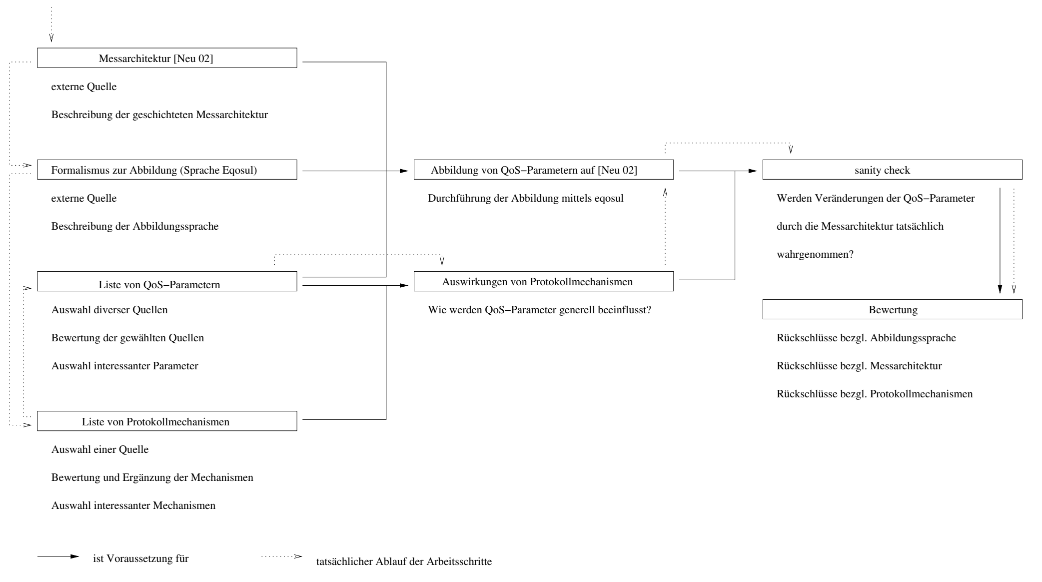

Abbildung 1.4: Vorgehensmodell

Kapitel 2

Grundlagen der Arbeit und verwendete Definitionen

2.1 Beschreibung der Messarchitektur

Die in dieser Arbeit betrachtete und verwendete Messarchitektur wurde von Cecile Neu im Rahmen ihrer Diplomarbeit 'Design and implementation of a generic quality of service measurement and monitoring architecture' [Neu 02] im Jahr 2002 entwickelt, beschrieben und prototypisch implementiert.

Die Messarchitektur besteht aus 3 Schichten bzw. Modulen. Diese werden [Neu 02] 'data collector', 'data correlator' und 'data analysator' genannt. Auf Grund der tatsächlichen Funktionalität der Module verwende ich die Begriffe 'data collector', 'event correlator' und 'statistic processing'.

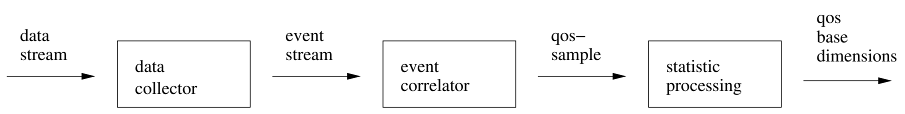

Abbildung 2.1: Zusammenwirken der Module

Der 'data collector' bzw. eine Vielzahl dieser (z.B. je eine Instanz pro Protokollschicht) dient zur Erhebung von Messdaten an der Dienst- bzw. Protokollschnittstelle. Als Ausgabe erscheint ein Strom (bzw. mehrere Ströme) von Ereignissen. Diese werden dann an den 'event correlator' weitergeleitet. Siehe hierzu Abbildung 2.2.

Der 'event correlator' dient dazu, die von verschiedenen 'data collector'-Instanzen gelieferten Ereignisströme zu korrelieren. Beispielsweise kann hier eine Korrelation zwischen einem fest getakteten 'Zeitstrom' und einer Reihe von 'ICMP echo request'- und 'ICMP echo reply'-Ereignissen hergestellt werden. Generell kann diese Korrelation vertikal zwischen verschiedenen Schichten des Protokollstacks, horizontal zwischen Ereignissströmen der selben Daten zu verschiedenen Zeiten oder semantisch geschehen. Diese korrelierten Ereignisströme nennen wir bereits 'qos sample'.

Zuletzt gelangen die korrelierten Daten in das 'statistic processing'-Modul. Dieses dient der statistischen wie optischen Aufbereitung der Daten. Z.B. kann hier eine Varianzberechnung erfolgen oder eine Normalverteilung erstellt werden.

Zudem verwendet die Messarchitektur das Konzept des Kontextes. Dieser kann zu jedem Zeitpunkt zwischen 'data collector' und 'statistic processing' ermittelt werden und wird dann in Richtung 'statistic processing' propagiert, so daß dem Benutzer Kontextinformationen zur Verfügung gestellt werden können.

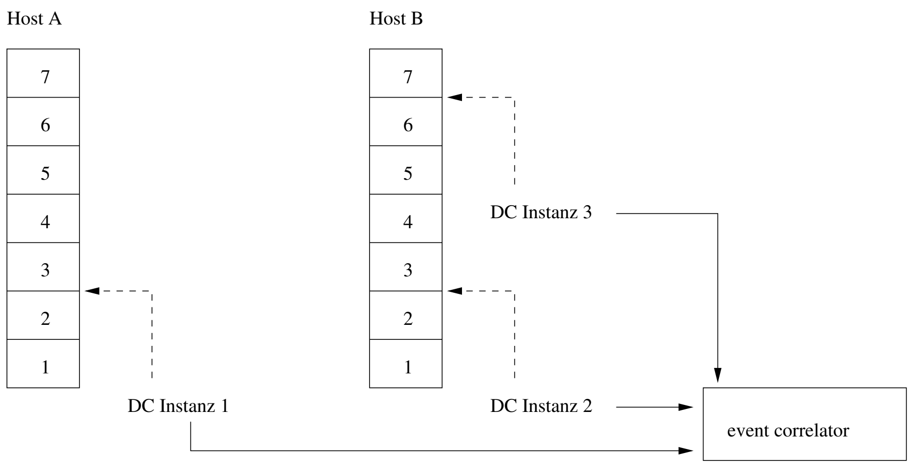

Abbildung 2.2: Beispiel für data collector-Konfiguration

Dieser vom Benutzer selbst definierbare Messkontext wird rückwärts vom 'statistic processing' in Richtung 'data collector' durch die Messarchitektur hindurch propagiert.

2.2 Die Abbildungssprache 'Eqosul'

Zur Abbildung der QoS-Parameter auf die geschichtete Messarchitektur verwenden wir eine Abbildungssprache, genannt Eqosul (siehe [Gar 04], sowie Anhang A.8 für die BNF).

Die zugrundeliegende Idee ist es, den Messmechanismus aus [Neu 02], i.e. die Module *data collector (dc)*, *event correlator (ec)* und *statistic processing (sp)* zu parametrisieren.

Die Abbildungssprache dient dazu, den QoS-Parameter auf die drei Blöcke 'data collector', 'event correlator' und 'statistic processing' abzubilden. Dazu werden 'interfaces' definiert (siehe Abbildung 2.3). Zwischen diesen finden sich dann sog. 'flows'. Diese stellen tatsächliche oder logische Datenströme dar, z.B. kann dies der Strom an Ethernet-Rahmen zwischen zwei Netzwerkkarten oder aber auch ein getakteter 'Zeitstrom' sein. Aufbauend auf diesen 'flows' definiert man mit Hilfe von regulären Ausdrücken Filter, auch 'events' genannt. Dies ist ausreichend um das Modul **dc** ausreichend zu parametrisieren. Für das Modul **ec** werden sodann noch 'samples' definiert. Diese entsprechen den korrelierten Eventströmen, die für das Modul **sp** benötigt werden und setzen bestimmte Samplestrategien ('jedes n-te', 'zufällig', ...) und -funktionen ('gaussverteilt', 'exponentialverteilt', ...) um. **sp** wiederum liefert zum jetzigen Zeitpunkt den statistisch aufbereiteten Wert des gemessenen QoS-Parameters zurück. Eine graphische Aufbereitung erfolgt zur Zeit noch nicht.

Eqosul befand sich während der Durchführung meiner Arbeit in Entwicklung. Insbesondere führten Teilergebnisse meiner Arbeit zu Änderungen an Eqosul. Aus diesem Grund werde ich an einigen Stellen (insbesondere in Kapitel 4) ein iteratives Vorgehen verwenden und beschreiben. Dies entspricht der konkreten Weiterentwicklung der Abbildungssprache während der Dauer meiner Diplomarbeit.

Zum Beispiel verwendete Eqosul zu Beginn den Begriff des 'Flows' (siehe Abbildung 2.3). Dieses Konzept wurde entfernt, da zum einen ein 'Flow' implizit bereits durch die beiden 'Interfaces' definiert ist und zum anderen Informationen verlorengegangen wären, unter anderem über die Übertragungsrichtung.

Für weitere Details zu Eqosul verweise ich auf [Gar 04].

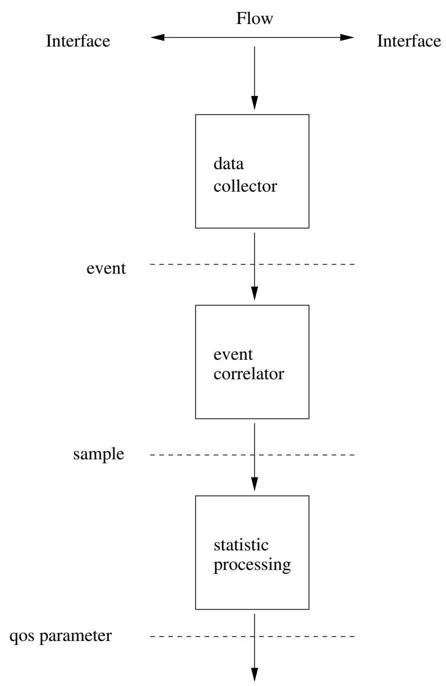

Abbildung 2.3: Zuordnung Eqosul-Objekte zu Modulen der Messarchitektur

2.3 Die verwendeten Standards

2.3.1 IPPM

Die QoS-Parameter der **IPPM** sind ausschließlich für den Bereich Internet, genauer nur für die Schichten 3 und 4 des OSI-Modells mit den Protokollen IP, TCP, UDP und ICMP anwendbar bzw. definiert. Dies macht sie einerseits zu nur sehr eingeschränkt verwendbaren QoS-Begriffen wenn es um die Betrachtung der Generik einer Messarchitektur geht. Zum anderen sind sie aber durch ihre klare Fokussierung und konkrete, am Anwendungsfall orientierte, Definition sehr gut geeignet, die Mächtigkeit sowohl der Messarchitektur als auch der verwendeten Abbildungssprache zu untersuchen. Trotz des engen Bereichs, den diese Dokumente abdecken, fehlt es -wie auch bei den anderen betrachteten QoS-Werken- an der Präzision.

Die Definitionen der IPPM-QoS-Parameter finden sich nicht in einem einzigen Werk. Stattdessen sind sie über mehrere **requests for comments** (**RFC**) verteilt die zum Teil sehr stark aufeinander aufbauen bzw. in weiten Teilen übereinstimmen.

In RFC 2330, 'Framework for IP Performance Metrics', findet sich das Rahmenwerk für die IPPM-QoS-Begriffe. Es definiert bestimmte Kriterien für die verwendeten Metriken, einige Grundbegriffe des Internets (um auf ein eindeutiges Vokabular zurückgreifen zu können), Konzepte zu den Themen 'Metrik' und 'Messmethodik' und diskutiert Probleme wie Messungenauigkeiten und -fehler. Weiterhin wird das Thema aufgegriffen, wie Metriken aus anderen, elementareren 'zusammengesetzt' werden können.

RFC 2678, 'IPPM Metrics for Measuring Connectivity' beschäftigt sich mit dem Begriff der 'connectivity'. Da sämtliche QoS-Begriffe (angewandt auf den Bereich Internet) natürlich eine Erreichbarkeit der Systeme voraussetzen, ist dieser QoS-Begriff von einiger Bedeutung für unsere weiteren Betrachtungen. Ausgehend von 'type-p-instantenous connectivity' und auf dieser aufbauend werden in dieser RFC auch komplexere Arten der 'connectivity', insbesondere über einen längeren Zeitraum gemessene, definiert und diskutiert.

RFC 2679, 'A One-way Delay Metric for IPPM', beginnt mit dem Begriff 'Type-P-One-way-Delay' und baut auf diesem Begriff Definitionen für ein Sample, genannt 'Type-P-One-way-Delay-Poisson-Stream', und gewisse Statistiken auf. Im Aufbau ähnelt es daher sehr stark unserer Messarchitektur.

Mit dem Begriff des Paketverlusts beschäftigt sich RFC 2680, 'A One-way Packet Loss Metric for IPPM'. Der Aufbau der RFC entspricht weitestgehend [RFC 2679].

RFC 2681, 'A Round-trip Delay Metric for IPPM', baut unmittelbar auf [RFC 2680] auf und entwickelt den Begriff weiter. Aufbau und Inhalt entsprechen größtenteils dem von [RFC 2680].

2.3.2 ITU-T

Das von der **ITU-T** 'Information Technology - Quality of Service: Framework' genannte Rahmenwerk definiert, ganz im Gegensatz zum IPPM-Rahmenwerk, selbst bereits einiges an QoS-Parametern. Es ist somit viel weniger ein Rahmenwerk als ein Anfang einer Ontologie. Wie später noch detaillierter betrachtet, leiden die Definitionen der QoS-Parameter jedoch häufig daran, nicht exakt spezifiziert zu sein. Dies ist einerseits ein Problem der viel größeren Generik des ITU-T-Werks, andererseits aber ein klares Versäumnis der Autoren bzw. verabschiedenden Gremien. Dies macht es aber sehr schwierig, tatsächlich vergleichbare Aussagen zu treffen, da dazu präzise Definitionen der Begrifflichkeiten und QoS-Parameter notwendig wären. Hier ist noch eine Lücke zu füllen.

2.3.3 FIPA

Die von der **FIPA** vorgeschlagene 'Quality of Service Ontology Specification' bietet, wie auch die ITU-T-Spezifikation, einen weiteren Einblick in QoS-Begriffe außerhalb der Internetwelt. Sie beschäftigt sich ausschließlich mit QoS-Parametern für den von der FIPA verwendeten 'Message Transport Service'. Um sich veränderten Umständen anpassen zu können, muß ein Agent in der Lage sein, Änderungen in seiner Umgebung auch tatsächlich wahrnehmen und quantifizieren zu können. Desweiteren muß er mit anderen, kooperierenden Agenten Informationen über QoS-Parameter austauschen können. Nur so ist bei einer Kooperation die Einhaltung seiner ursprünglichen Schranken zu gewährleisten. Hierzu ist eine präzise Definition der Begriffe und Parameter unumgänglich. Die FIPA-Spezifikation hat sich dies als Ziel gesetzt. Wie ich später noch zeigen werde (siehe 3.2), sind die von der FIPA verwendeten QoS-Begriffe leider in vielerlei Hinsicht alles andere als präzise. Daher wird das gesteckte Ziel meiner Ansicht nach nicht erreicht.

Kapitel 3

Auswirkungen von Protokollmechanismen auf QoS-Parameter

Dieses Kapitel beschäftigt sich mit der Frage, wie QoS-Parameter generell beeinflusst werden. Inwiefern zeigen bestimmte (beliebige aber feste) Protokollmechanismen Auswirkungen auf spezifische (beliebige aber feste) QoS-Parameter? Das Ergebnis dieser Untersuchungen geht in den in Kapitel 5 durchgeführten *sanity check* ein, der unter anderem überprüfen soll, inwieweit die Messarchitektur Änderungen des QoS-Parameters tatsächlich wahrnimmt. Zudem dient es dazu, ein besseres Verständnis für das Verhalten von Protokollen zu erhalten.

Zuerst wähle ich eine Quelle für Protokollmechanismen aus. Sodann bewerte und ergänze ich diese Liste der Mechanismen. Für die QoS-Parameter habe ich diverse Quellen (u.a. [X.641] und [RFC 2330]) bereits in Kapitel 2.3 ausgewählt, genauer betrachtet und bewertet. In diesem Kapitel werde ich daher lediglich noch die Auswahl der betrachteten QoS-Parameter vornehmen.

Das Problem der Auswirkungen von Protokollmechanismen auf QoS-Parameter spannt eine Matrix mit den Achsen 'Protokollmechanismus' und 'QoS-Parameter' auf. Da die Mächtigkeit der Matrix aber eine Untersuchung jedes einzelnen Feldes verbietet, werde ich nur ausgewählte Felder betrachten. Dazu wähle ich exemplarisch 'interessante' Protokollmechanismen und QoS-Parameter aus und betrachte anschließend die konkreten Auswirkungen des Mechanismus auf den QoS.

Schließlich werde ich die in diesem Kapitel gefundenen Ergebnisse zusammenfassen und meine Schlussfolgerungen präsentieren.

3.1 Protokollmechanismen

Auf der Seite der Protokollmechanismen betrachte ich eine Auswahl der in [Black] S. 23ff. angegebenen, inhaltsunabhängigen. Diese dienen quasi als Bausteine um komplexere Protokolle aufzubauen. Ich nehme an, daß diese ausreichen, die gesamte Funktionalität des OSI-Stacks zu simulieren (ohne Beweis). Alle anderen Protokollmechanismen sind somit Kompositionen aus den von mir betrachteten. Da sich diese Protokollmechanismen ausschließlich mit Verbindungen (bzw. Kanälen) oder aber mit PDUs und SDUs unabhängig von deren Inhalt beschäftigen, nehme ich exemplarisch noch zwei inhaltsabhängige hinzu: 'data compression' und 'data encryption'.

Wie ich gleich darlegen werde, stützen sich leider einige dieser Protokollmechanismen auf andere ab, sind also nicht für sich betrachtet einsetzbar. Von daher ist die angedachte Betrachtung '1 Protokollmechanismus vs. 1 QoS-Parameter' so nicht möglich. Ich werde darauf bei den betroffenen Mechanismen jeweils noch konkret eingehen.

Am besten für meine Betrachtungen geeignet wäre eine Sammlung von Protokollmechanismen die, ähnlich der Basis eines Vektorraums, linear unabhängig voneinander sind und durch die jedes Protokoll vollständig (sozusagen als Linearkombination) darstellbar wäre. Dies ist bei [Black] nicht der Fall. Zum einen fehlen die inhaltsabhängigen Protokollmechanismen, zum anderen sind die von ihm geschilderten Mechanismen keineswegs linear unabhängig. Dennoch dient diese Sammlung meiner Ansicht nach als geeignete Basis für eine Untersuchung der Auswirkungen von Protokollmechanismen auf QoS-Parameter, da diese beiden Forderungen (lineare Unabhängigkeit und Vollständigkeit) zwar 'angenehm' und systematisch 'schön' aber für die gemachten Aussagen nicht zwingend notwendig sind. Ich verwende bewusst dennoch [Black] um einen ersten Einblick in die Problematik zu erhalten, auch wenn die Abhängigkeiten zwischen Protokollmechanismen und QoS-Parametern damit nicht so gut darstellbar sind, wie es zu wünschen wäre.

Im Folgenden werde ich kurz auf jeden einzelnen der von [Black] verwendeten Protokollmechanismen eingehen, die verwendete Definition nennen und, wo nötig, eine Abbildung zur Verdeutlichung angeben. Die Abbildungen sind unmittelbar [Black] entnommen und verwenden '()' als Symbol für Verbindungen. Ansonsten ist die Semantik meines Erachtens nach selbsterklärend.

Mechanismus multiplexing / demultiplexing

Definition 'an n-layer function (and its reverse function) that uses one n-1-connection to support multiple n-connections'

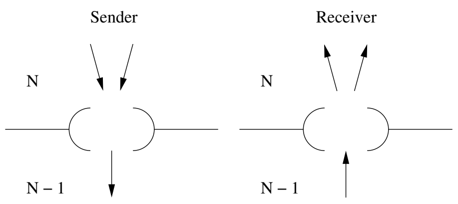

Abbildung 3.1: 'multiplexing' und 'demultiplexing' nach Black, p. 24

Mechanismus splitting / recombining

Definition 'an n-layer function (and its reverse function) that uses more than one n-1-connection to support an n-connection'

Kommentar Ich weiche hier von der Definition in [Black] ab. Black verwendet hier 'multiple n-connections'. Ich halte dies nicht für sinnvoll. splitting/recombining sind meiner Ansicht nach besser als Komplement zu multiplexing/demultiplexing zu sehen.

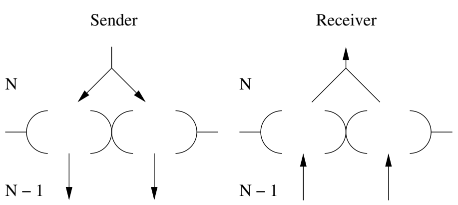

Abbildung 3.2: 'splitting' und 'recombining' nach Black, p. 24

Mechanismus segmenting / reassembling

Definition 'an n-entity function (and its reverse function) that maps an n-service data unit into multiple n-protocol data units'

Kommentar Damit die Empfangsseite der Kommunikation die 'service data unit' wieder zusammensetzen kann, benötigt sie mindestens den 'buffering'-Mechanismus.

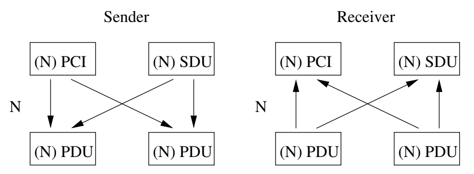

Abbildung 3.3: ’segmenting’ und ’reassembling’ nach Black, p. 24

Mechanismus blocking / deblocking

Definition ’an n-entity function (and its reverse function) that maps multiple n-service data units into one n-protocol data unit’

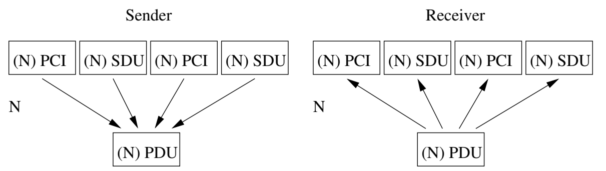

Abbildung 3.4: ’blocking’ und ’deblocking’ nach Black, p. 25

Mechanismus concatenation / separation

Definition ’an n-entity function (and its reverse function) that maps multiple n-protocol data units into one n-1-service data unit’

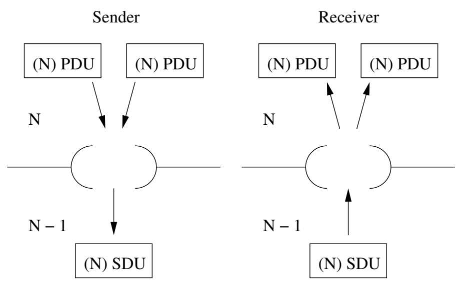

Abbildung 3.5: ’concatenation’ und ’separation’ nach Black, p. 25

Mechanismus protocol identifier

Definition ’an identifier between communicating entities to select a protocol to be used in the logical connection’

Mechanismus centralized/decentralized multi-endpoint connection

Definition ’the ability to associate data from one to many and/or many to one connection endpoints’

Mechanismus flow control

Definition ’a function to control the flow of data between adjacent layers or within a layer’

Kommentar Dieser Protokollmechanismus ist leider kein elementarer, genauer gesagt kann er nicht sinnvoll alleine vorkommen. Ein Protokoll, daß keinen anderen Protokollmechanismus verwendet als ’flow control’ ist (zur Datenübertragung) nicht vorstellbar. Zumindest einer der beiden Kommunikationspartner muss zusätzlich noch ’buffering’ einsetzen (sonst gehen Daten entweder schon beim Sender oder beim Empfänger verloren).

Mechanismus sequencing

Definition 'an n-layer function that preserves the order of data units submitted into it'

Kommentar Auch dieser Protokollmechanismus ist nicht elementar. Genau wie 'flow control' stützt auch er sich auf 'buffering' ab. Ansonsten wäre ein Verdrehen der Reihenfolge nicht korrigierbar. Unter der Annahme, daß Pakete nicht nur direkt in falscher Reihenfolge ankommen sondern auch verloren gehen oder beschädigt werden können (dies führt natürlich auch zu einer falschen Empfangsreihenfolge), benötigen wir zudem noch 'acknowledgement'.

Mechanismus acknowledgement

Definition 'an n-layer function that a receiving n-entity uses to acknowledge a protocol data unit from a sending n-entity'

Kommentar Damit dieser Protokollmechanismus sinnvoll eingesetzt werden kann (i.e. der sendende Partner auch tatsächlich ein nicht bestätigtes Paket neu senden kann). benötige ich ebenfalls wieder den 'buffering'-Mechanismus.

Mechanismus reset

Definition 'an n-layer function that returns the correspondent n-entities to a state, with a possible loss or duplication of data'

Mechanismus data compression/data decompression

Definition 'a function (and its reverse) that reduces redundancy and increases entropy in the data (with the purpose of reducing the amount of transferred data)'

Kommentar Da für einen effizienten Einsatz von Datenkomprimierung große Blöcke an Ausgangsdaten zusammengefasst werden sollten, benötigen wir 'buffering' auf der einen Seite und die Mechanismen blocking/deblocking (bei 'normal' komprimierbaren Daten, da ich in diesem Fall mehrere service data units in eine protocol data unit verpacken kann) bzw. segmenting/reassembling (bei nahezu nicht komprimierbaren Daten, die z.T. auf Grund der notwendigen Header real zu mehr übertragenen Daten führen als ohne Komprimierung).

Mechanismus data encryption/data decryption

Definition 'a function (and its reverse) that maps data from one alphabet to another'

Kommentar Da Datenverschlüsselung auch am Besten mit Blöcken von Daten operiert (weil sog. Padbytes einen 'known plain text'-Angriff ermöglichen), benötigt auch dieser Protokollmechanismus möglichst die Mechanismen 'buffering, 'blocking/deblocking' und 'segmenting/reassembling'.

3.2 QoS-Parameter

Wie bereits in den Abschnitten 1.2 und 2.3 geschildert, betrachte ich QoS-Parameter aus den Quellen IPPM ([RFC 2330] [RFC 2678] [RFC 2679] [RFC 2680] [RFC 2681]), ITU-T [X.641] und FIPA [FIPA QoS]. Diese Auswahl beinhaltet sowohl sehr eng gefasste QoS-Parameter, die sich nur auf einem sehr eng begrenzten Feld einsetzen lassen als auch sehr abstrakte und allgemein gehaltene QoS-Parameter die in nahezu jedem Feld verwendet werden können. Zudem werde ich weitere QoS-Parameter betrachten, die mir von Interesse schienen.

Ich werde in diesem Abschnitt nur kurz auf die jeweiligen QoS-Parameter eingehen, um einen Überblick über die ausgewählten und später näher betrachteten QoS-Parameter zu vermitteln. Die Definition werde ich nur in Ausschnitten angeben um ein grundlegendes Verständnis für die ausgewählten QoS-Parameter zu vermitteln. Für die exakte Definition verweise ich, sofern vorhanden, auf die jeweiligen Originalquellen.

Parameter *Verbindungsaufbauzeit*

Quelle -

Definition Die Zeit die benötigt wird, eine Verbindung aufzubauen, i.e. die Zeit zwischen Initiierung der Verbindung und Etablierung der Verbindung.

Typ Zeit

Parameter *user information throughput*

Quelle [X.641], 6.3.3.16, S. 18

Definition An amount of user information transferred in a period of time.

Typ Daten/Zeit

Parameter *TCP-One-Way-Delay*

Quelle [RFC 2679], Abschnitt 3, S. 3 ff.

Definition For a real number dT, „the TCP-One-way-Delay from Src to Dst at T is dT" means that Src sent the first bit of a TCP packet to Dst at wire-time T and that Dst received the last bit of that packet at wire-time T+dT.

Typ Zeit

Parameter *UDP-One-Way-Packet-Loss*

Quelle [RFC 2680], Abschnitt 2, S. 3 ff.

Definition „The UDP-One-way-Packet-Loss from Src to Dst at T is 0" means that Src sent the first bit of a UDP packet to Dst at wire-time T and that Dst received that packet. „The UDP-One-way-Packet-Loss from Src to Dst at T is 1" means that Src sent the first bit of a UDP packet to Dst at wire-time T and that Dst did not receive that packet.

Typ Wahrscheinlichkeit

Parameter *IP Packet Delay Variation*

Quelle [RFC 3393], Abschnitt 2, S. 4 ff.

Definition The ipdv is the difference between the one-way-delay of the selected packets.

Typ Zeit

Parameter *ICMP-Round-Trip-Delay (ping)*

Quelle -

Definition Die Zeit zwischen Versand eines ICMP-Pakets vom Typ *echo_request* und dem Eintreffen eines ICMP-Pakets vom Typ *echo_reply* mit der selben ID.

Typ Zeit

Parameter *WWW-Seitenaufbauzeit*

Quelle -

Definition Die Zeit zwischen Abschicken der Anforderung zum Darstellen einer im WWW verfügbaren Seite durch den Benutzer und vollständig dargestellter Seite.

Typ Zeit

Parameter *Framefehlerrate bei Ethernet*

Quelle -

Definition Das Verhältnis 'nicht korrekt empfangene Ethernet-Rahmen' zu 'insgesamt verschickte Ethernet-Rahmen' zwischen zwei durch Ethernet verbundenen Rechner.

Typ Wahrscheinlichkeit

Parameter *Wahrscheinlichkeit der Verbindungsablehnung bei TCP*

Quelle -

Definition Das Verhältnis 'abgelehnte TCP-Verbindungen' zu 'insgesamt initiierte TCP-Verbindungen'.

Typ Wahrscheinlichkeit

Parameter *rtt*

Quelle [FIPA QoS], S. 3

Definition The round trip time which is the time required for a data segment to be transmitted to a peer entity and a corresponding acknowledgement sent back to the originating entity.

Typ Zeit

Parameter *delay*

Quelle [FIPA QoS], S. 3

Definition The (nominal) time required for a data segment to be trasmitted to a peer entity.

Typ Zeit

3.3 Auswirkungen / 'Die Matrix' en detail

Für die Auswahl der Kreuzungspunkte (i.e. der konkreten Protokollmechanismus/QoS-Paarungen) war es mir wichtig, mehrere Klassen von Fällen auszuwählen. Zum einen Fälle, in denen ein Effekt offensichtlich ist. Zudem Paarungen bei denen kein Effekt erkennbar ist. Und zu letzt diejenigen Fälle, bei denen eine Auswirkung deshalb nicht feststellbar ist, weil der Protokollmechanismus mit anderen (häufig nur impliziten) Prämissen definiert wurde als der QoS-Parameter (z.B. one-to-one-Verbindung auf der einen Seite und one-to-many-Verbindung auf der anderen).

Betrachten wir nun, inwieweit ein beliebiger (aber fester) Protokollmechanismus Auswirkungen auf einen beliebigen (aber festen) QoS-Parameter zeigt. Wir vermerken eines der beiden möglichen Ergebnisse für die Matrix. '+' bedeutet, daß eine Parameteränderung des Protokollmechanismus' sich auf das Messergebnis unseres QoS-Parameters auswirkt. Ein '-' bedeutet, daß dies nicht der Fall ist.

Sofern nicht anders angegeben, geben nicht spezifizierte Schichten die Daten ohne jegliche zeitliche wie inhaltliche Beeinflussung nach oben bzw. unten weiter (sie verhalten sich also mitunter transparent zu unserer Messung). Diese Annahme ist natürlich nicht korrekt, aber für die Durchführung der Untersuchung notwendig.

Für die Fälle, in denen QoS-Parameter nicht präzise genug definiert sind, werde ich diese geeignet erweitern und gegebenenfalls eine der möglichen Erweiterungen auswählen um dann auf dieser Grundlage fortzufahren. Eine Diskussion der Mehrdeutigkeit führe ich nicht durch.

Desweiteren gilt die Annahme, daß der **dc** aus technischer Sicht alles messen kann. Dies wird gerade bei QoS-Parametern oberhalb der Schicht 7 in der Realität nicht ohne weiteres zu leisten sein, ist aber für die Arbeit als Annahme notwendig.

3.3.1 'protocol identifier' vs. 'TCP-Instantaneous-Unidirectional-Connectivity'

Beispiel 1: Messung der TCP-Instantaneous-Unidirectional-Connectivity zwischen Quelle q und Senke s. Dies wird als Erfolg gewertet, wenn ein von q gesendetes TCP-Paket mit gesetztem SYN-Flag und

Zielport 22 bei s ankommt. Dies sei der Fall, da ein dazwischenliegendes Transitsystem ausschließlich Port 25 filtere.

Beispiel 2: Messung der TCP-Instantaneous-Unidirectional-Connectivity zwischen Quelle q und Senke s. Dies wird als Erfolg gewertet, wenn ein von q gesendetes TCP-Paket mit gesetztem SYN-Flag und Zielport 25 bei s ankommt. Dies sei nicht der Fall, da ein dazwischenliegendes Transitsystem Port 25 filtere.

Ganz offensichtlich besteht intuitiv eine TCP-connectivity zwischen q und s. Jedoch bei ungünstiger Wahl der Testparameter (und das sind in diesem Fall nicht einmal Parameter des Protokollmechanismus' selbst) werden falsche Ergebnisse erzielt. Grund hierfür ist, dass der gemessene QoS-Parameter unterspezifiziert ist. Eine TCP-Verbindung benötigt immer auch die Angabe eines Ports (auf jeder Seite). Somit muss dies unbedingt auch in die Definition eines QoS-Parameters, der TCP-connectivity messen soll. Zur Umgehung des Problems könnte man natürlich die Messung auf alle möglichen Ports ausweiten und dies als Erfolg zählen sobald ein einziges Paket bei s ankommt. Dies hat jedoch gravierende Nachteile bezüglich der Dauer der Messung sowie des übertragenen Datenvolumens.

Betrachten wir nun den QoS-Parameter als geeignet erweitert (ausreichend präzise spezifiziert). Ändert man nun den (einzigen) Parameter des Protokollmechanismus' z.B. auf UDP ab, lässt sich dieser QoS-Parameter überhaupt nicht mehr messen. Im Ergebnis ein klares '+'.

3.3.2 'segmenting/reassembling' vs. 'user information throughput'

Dieser QoS-Parameter ist derart definiert, daß ein Einfluss niedrigerer Schichten ausgeschlossen ist. Daher bleibt lediglich zu prüfen, ob die Verwendung des Protokollmechanismus' auf der selben Schicht wie der QoS-Parameter einen Einfluss auf denselben hat. Man stelle sich einen Benutzer vor, der bei der Ausgabe 'segmenting' verwendet, mitunter also die Daten bzw. das Bearbeitungsergebnis blockweise von sich gibt. Selbstverständlich hat dies dann Einfluss auf den QoS-Parameter auf Grund des Paketisierungsoverheads (bzw. der notwendigen Wartezeit um Blöcke füllen zu können). Im Ergebnis also ein '+'.

3.3.3 'centralized/decentralized multi-endpoint-connection' vs. 'one-way delay'

Beispiel 1: Messe den one-way delay zwischen Host h und Ziel z1.

Beispiel 2: Messe den one-way delay zwischen Host h und Ziel z1, z2 und z3 (z.B. Broadcast).

Hier stellt sich das Problem, daß der QoS-Parameter 'one-way delay' (implizit) nur für one-to-one-connections definiert ist. Was also ist das Ergebnis der Messung 'one-way delay' für Beispiel 2? Mitunter lässt sich also entweder der QoS-Parameter überhaupt nicht messen für Protokolle, die den Protokollmechanismus 'centralized/decentralized multi-endpoint-connection' einsetzen oder der QoS-Parameter muss 'geeignet' erweitert werden. Als Beispiel hierfür nehme man das Minimum der Verzögerung der drei 'Verbindungen'. Der Einfluss einer Änderung der Parameter des Mechanismus ist dann jedoch nicht a priori quantifizierbar (a posteriori natürlich schon). Im Ergebnis erhält man entweder ein 'n/a' oder ein '+'.

3.3.4 'flow control' vs. 'packet delay variation'

Unter der Annahme, daß ein Protokoll, das 'flow control' einsetzt, zugleich auch 'buffering' einsetzt (siehe die Erläuterungen hierzu weiter oben), lässt sich sofort feststellen, daß der Protokollmechanismus den QoS-Parameter beeinflusst (Buffer auf Empfängerseite voll, Datenfluss wird gestoppt, 'packet delay variation' erhöht sich). Im Ergebnis erhalten wir ein '+'.

3.3.5 'sequencing' vs. 'connectivity'

Dieser Protokollmechanismus hat keinerlei Einfluss auf diesen QoS-Parameter. Sequencing beschäftigt sich mit dem Inhalt der Verbindung, den Daten. Connectivity mit dem Zustandekommen der Verbindung. Sequencing kann somit erst dann Einfluss auf QoS-Parameter nehmen, wenn eine Verbindung besteht. Dann aber ist connectivity bereits gegeben. Das Ergebnis lautet somit '-'.

3.3.6 'sequencing' vs. 'packet delay variation'

Dieser Protokollmechanismus hat dann einen Einfluss auf diesen QoS-Parameter wenn z.B. Pakete verloren gegangen, beschädigt worden sind oder niedrigere Schichten des Stacks keine Reihenfolgesicherung bieten und die Reihenfolge der Pakete vertauscht wurde. Dann muss die Reihenfolge wiederhergestellt werden und dazu muss das beschädigte oder verlorene Datenpaket zumindest einmal neu geschickt werden (oder die Pakete die vor dem falsch einsortierten empfangen, logisch aber nach ihm einzusortieren sind). Im Ergebnis '+'.

3.3.7 'acknowledgement' vs. 'round-trip delay'

Der 'round-trip delay' wird von diesem Protokollmechanismus z.B. dann beeinträchtigt, wenn eine Paketbestätigung verlorengegangen ist. Der sendende Rechner darf dann (logisch betrachtet) die Antwort auf eben jenes Paket nicht verwerten und schickt das Paket nach Ablauf des Timeouts erneut. Der round-trip delay kann dann erst nach Eintreffen sowohl der Bestätigung für das gesendete Paket als auch des Antwortpakets berechnet werden. Ergebnis '+'.

3.3.8 'reset' vs. 'round-trip delay'

Hier ist das Ergebnis ein '+'. Sende q ein Paket zu s. S setze die Verbindung unmittelbar nach dem Empfang des Pakets, mitunter vor Absenden der Antwort, zurück. Q muss nach Ablauf eines Timeouts erneut ein Paket schicken (oder das Ergebniss wird auf 'unendlich' gesetzt für den Fall, daß Timeouts nicht zum Einsatz kommen).

3.4 Ergebniszusammenfassung

Im Ergebnis ergibt sich, daß der jetzige Zustand im Bereich QoS schlecht ist. Insbesondere treten Defizite auf bei der Definition von Protokollmechanismen und QoS-Parametern.

Zur Betrachtung der Protokollmechanismen muss zuerst eine Sammlung von Protokollmechanismus-Definitionen vorliegen. Diese Sammlung muss vollständig in dem Sinne sein, daß sich alle Protokolle durch Kombination der aufgeführten Mechanismen abbilden bzw. darstellen lassen. Diese Sammlung muss aber auch präzise genug sein, eine eindeutige algorithmische Darstellung zu ermöglichen. An beiden Anforderungen scheitert es bis heute.

Die von mir verwendete Sammlung aus [Black] ist nicht vollständig, da sie inhaltsabhängige Mechanismen nicht einmal in ihrer allgemeinsten Form anspricht. Zudem sind, wie oben gezeigt, viele der Mechanismen nicht unabhängig voneinander betrachtbar. Für eine klare und präzise Betrachtung der Auswirkungen von Protokollmechanismen ist es aber unabdingbar, daß die Mechanismen elementarer Natur sind.

[Black] verwendet desweiteren auch keine präzisen und eindeutigen Definitionen der Protokollmechanismen. Eine Umsetzung auf Algorithmen ist damit nicht mehr ohne weiteres möglich. Genau dies aber ist notwendig um überhaupt (insbesondere vergleichbar) Auswirkungen messen zu können.

Es ist somit klar, daß auch Protokollmechanismen durch formale und nicht semi-formale oder gar umgangssprachliche Definitionen festgelegt werden müssen, will man nicht lediglich sehr allgemeine Aussagen treffen.

Was die QoS-Mechanismen betrifft, so tritt bei allen von mir verwendeten Quellen das selbe Problem auf: fehlende Präzision.

Wie bereits bei den Protokollmechanismen festgestellt, bedarf es auch bei den QoS-Parametern einer präzisen, formalen Definition. Es ist untragbar, daß QoS-Parameter zu großen Teilen 'eigenständig erweitert' werden müssen, um überhaupt eine lauffähige Messumgebung zu erhalten. Unter diesem Gesichtspunkt ist es natürlich so gut wie unmöglich, Messungen von QoS-Parametern vergleichbar zu machen. Was Provider A als Messung der 'packet delay variation' bezeichnet, muss (semantisch) höchstens in Ansätzen mit der von Provider B durchgeführten Messung der 'packet delay variation' übereinstimmen. Wie bereits in Kapitel 1 erläutert, ist heute aber gerade die Vergleichbarkeit über Providergrenzen hinweg ein elementarer Anspruch an den Bereich 'QoS'. Dies wird zur Zeit aber trotz eines anderslautenden Anspruchs weder durch die QoS-Definitionen von IPPM, ITU-T oder FIPA gewährleistet.

Zudem kommt hinzu, daß einige QoS-Parameter derart definiert sind, daß der Einfluss bestimmter Protokollmechanismen **per Definition** implizit oder gar explizit ausgeschlossen wird. Dies verhindert die Betrachtung der Auswirkungen eben jener Protokollmechanismen auf bestimmte QoS-Parameter.

Kapitel 4

Abbildung auf die Messarchitektur

Im folgenden Kapitel werde ich mit Hilfe der Abbildungssprache 'Eqosul' (siehe 2.2) ausgewählte QoS-Parameter auf eine geschichtete Messarchitektur (siehe [Neu 02]) abbilden.

Die Auswahl der QoS-Parameter berücksichtigt zum einen eine Auswahl an Parametern über möglichst alle Schichten des OSI-Modells (und darüber hinaus), zum anderen werde ich einen Teil der in Abschnitt 3.3 betrachteten Parameter hier ebenfalls behandeln. Dies erleichtert mir in Kapitel 5 die Überprüfung der Abbildungsmethodik, da für diese QoS-Parameter ja bereits Vorüberlegungen bezüglich möglicher Einflußfaktoren stattgefunden haben.

Ich werde zu jedem QoS-Parameter ein Sequenzdiagramm angeben, das die Modellierung des Interfaces incl. der Events verdeutlicht. Die Syntax und Semantik der Diagramme ist an UML angelehnt (siehe [Oes 98], S. 306ff.), ich füge jedoch folgende Ergänzung hinzu: Der in Abbildung 4.1 als erstes angeführte Pfeil (*syn*) entspricht UML und steht für einen Methodenaufruf. Der zweite Pfeil ist eine Ergänzung. An den Enden stehen jeweils *Ereignisse*, hier also die Ereignisse *syn_sent* und *syn_received*. Diese entsprechen zusammen semantisch *syn*. Der Vorteil dieser Notation ist es jedoch zum einen, daß wir an Interfaces auftretende Ereignisse klar graphisch darstellen können und sich diese den entsprechenden Ereignisdefinitionen in Eqosul zuordnen lassen. Zum anderen ermöglicht diese Änderung auch die Darstellung von Verlusten, wie dies z.B. der dritte Pfeil zeigt. Da wir bei unseren QoS-Parametern in der Mehrzahl der Fälle asynchrone Kommunikation betrachten, wird die Darstellung der Sachverhalte durch diese Ergänzung erheblich vereinfacht. Desweiteren besteht durch diese Ergänzungen die Möglichkeit, nur einen Teil der Kommunikationsbeziehung zu modellieren sofern die beim zweiten Kommunikationspartner auftretenden Ereignisse keine Rolle spielen und nicht explizit betrachtet werden müssen. Dies vereinfacht und verbessert die Modellierung.

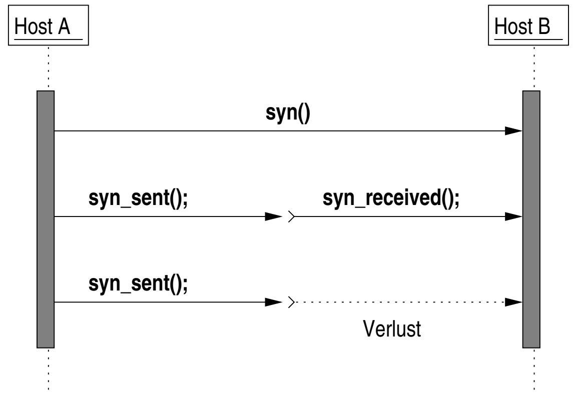

Abbildung 4.1: Ergänzung der UML-Syntax und -Semantik

4.1 Verbindungsaufbauzeit

Zuerst betrachte ich den QoS-Parameter 'Verbindungsaufbauzeit'. In einer ersten Iteration werde ich diese Betrachtung TCP-spezifisch durchführen.

Am Interface treten die folgenden Ereignisse auf: *syn_sent*, *syn_received*, *syn_ack_sent*, *syn_ack_received*, *ack_sent*, *ack_received*. Diese Ereignisse entsprechen dem Ablauf des 3-Wege-Handshakes von TCP. Nach Abschluß des 3-Wege-Handshakes ist der Verbindungsaufbau unter TCP abgeschlossen. Somit ist es ausreichend, den *data collector* von [Neu 02] bzw. den Interface-Abschnitt unter Eqosul entsprechend zu parametrisieren. Die nachfolgende Datenübertragung ist für die Messung des QoS-Parameters 'Verbindungsaufbauzeit' nicht von Belang und wird daher nicht modelliert.

Für den Abschnitt 'Event' der Eqosul-Definition musste jedoch ein Problem gelöst werden. Zur eindeutigen Identifikation der Kommunikationspartner einer TCP-Verbindung benötige ich je Partner das Tupel (Host, Port). Eqosul selbst bot jedoch nur die Möglichkeit, einfache Werte als Identifikatoren zu verwenden. Ich ergänze dies um die Möglichkeit, auch Tupel angeben zu können. Somit definiere ich die Ereignisse *tcp_syn_sent* als Startereignis, *tcp_syn_ack_received* als 'Kontrollereignis', das sicherstellt, daß das SYN/ACK-Paket gesendet und empfangen wurde und das Ereignis *ack_received* als den Verbindungsaufbau abschließendes Ereignis.

Da der Verbindungsaufbau mit *syn_sent* beginnt und mit *ack_received* abgeschlossen ist, benötige ich zur Messung der Verbindungsaufbauzeit im Sample-Abschnitt von Eqosul lediglich die Zeitdifferenz zwischen *syn_sent* und *ack_received*. Als Randbedingung gilt jedoch, daß dazwischen noch *tcp_syn_ack_received* stattfinden muß. Dies ist im Sample-Abschnitt von Eqosul entsprechend zu berücksichtigen. Es müssen die drei Ereignisse *syn_sent*, *syn_ack_received* und *ack_received* mit zueinander passenden Parametern (Host, Port) auftreten bevor eine Berechnung stattfinden kann. Zur Berechnung selbst genügen jedoch die Zeitstempel von *syn_sent* und *ack_received*.

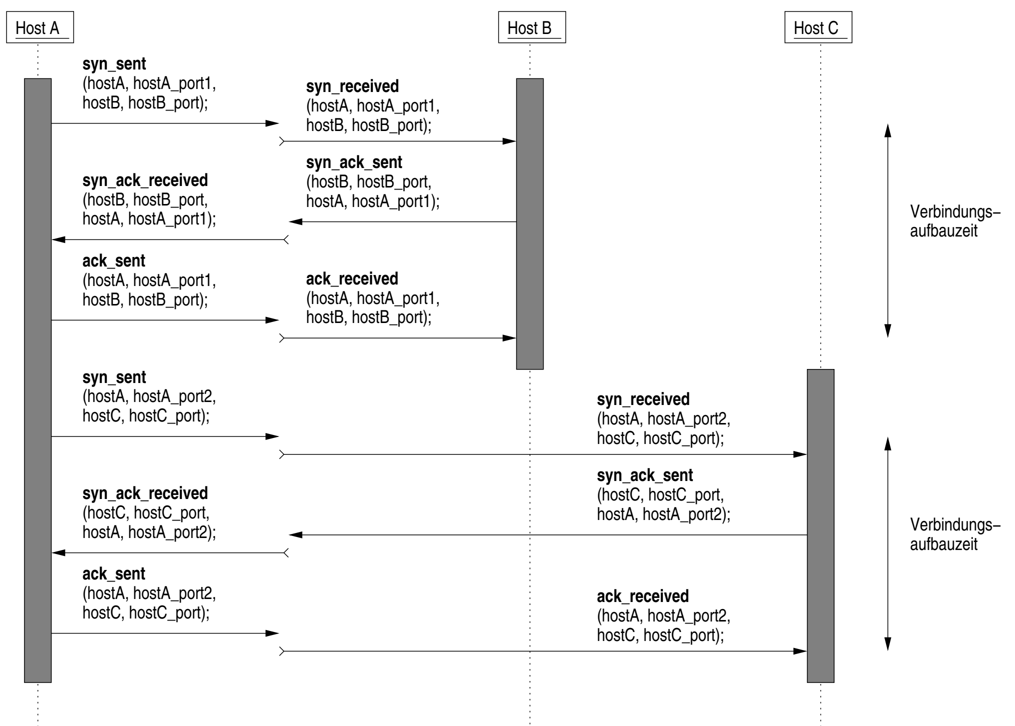

Abbildung 4.2: Interface-Modell für 'Verbindungsaufbauzeit' (TCP)

```
// Verbindungsaufbauzeit (am Beispiel TCP)
```

```

INTERFACE tcp {
  syn_sent     ( src:    BYTE[4],
                 src_port: BYTE[2],
                 dest:   BYTE[2],
                 dest_port: BYTE[2] );
  syn_received ( src:    BYTE[4],
                 src_port: BYTE[2],
                 dest:   BYTE[2],
                 dest_port: BYTE[2] );
  syn_ack_sent ( src:    BYTE[4],
                 src_port: BYTE[2],
                 dest:   BYTE[2],
                 dest_port: BYTE[2] );
  syn_ack_received ( src: BYTE[4],
                 src_port: BYTE[2],
                 dest:   BYTE[2],
                 dest_port: BYTE[2] );
  ack_sent     ( src:    BYTE[4],
                 src_port: BYTE[2],
                 dest:   BYTE[2],
                 dest_port: BYTE[2] );
  ack_received ( src:    BYTE[4],
                 src_port: BYTE[2],
                 dest:   BYTE[2],
                 dest_port: BYTE[2] );
  data_sent    ( src:    BYTE[4],
                 src_port: BYTE[2],
                 dest:   BYTE[2],
                 dest_port: BYTE[2],
                 data:   BYTE[] );
  data_received ( src:   BYTE[4],
                 src_port: BYTE[2],
                 dest:   BYTE[2],
                 dest_port: BYTE[2],
                 data:   BYTE[] );
};

// Eqosul-Ergaenzung: Tupel (src, src_port, dest, dest_port) zur eindeutigen
// Identifikation
EVENT tcp_syn_sent (tcp.syn_sent, (src, src_port, dest, dest_port)) {};
EVENT tcp_syn_ack_received (tcp.syn_ack_received, (src, src_port, dest, dest_port)) {};
EVENT tcp_ack_received (tcp.ack_received, (src, src_port, dest, dest_port)) {};

SAMPLE tcp_vaz_sample {
  USES
    // tcp_syn_sent beginnt den Verbindungsaufbau
    EVERY tcp_syn_sent AS a;
    // 'Kontrollereignis' um das zweite Drittel des 3-Wege-Handshakes abbilden zu
    // koennen
    EVERY tcp_syn_ack_received AS b;
    // tcp_ack_received schliesst den Verbindungsaufbau ab
    EVERY tcp_ack_received AS c;
  COMPUTES a.myMatch(c).time - a.hisMatch(c).time UNIT NULL;
};

QOS_PARAMETER tcp_verbindungsaufbauzeit {
  SAMPLE 1 TIMES FROM tcp_vaz_sample TO PRODUCE NORMALIZED_DISTRIBUTION;
```

```
};
```

Als nächstes abstrahiere ich vom TCP-Fall zum allgemeineren Fall eines Verbindungsaufbaus auf Betriebssystem-Ebene unabhängig vom darunterliegenden (konkreten) Protokoll.

Dazu genügt es, nur die API der Seite, die die Verbindung initiiert, zu betrachten. Der Kommunikationspartner wird nur noch durch *target* repräsentiert, eine genaue Modellierung seiner API ist nicht mehr notwendig. Das betrachtete Interface hat nur zwei Ereignisse, *connect* und *connect_return*. Der gesuchte Wert ist dann die Zeitdifferenz zwischen diesen beiden Aufrufen.

Im Abschnitt 'Interface' benötige ich den im TCP-Fall notwendigen, zusätzlichen Parameter zur eindeutigen Identifikation der Kommunikationspartner nicht mehr. Der Parameter 'target' legt eindeutig das Ziel fest. Sollte ein darunterliegendes Protokoll mehrere Werte zur Identifikation benötigen, lässt sich 'target' z.B. als Hash über diese Parameter verstehen. Der Sender wiederum ist bereits durch das Interface eindeutig festgelegt und kann in der Angabe der API entfallen.

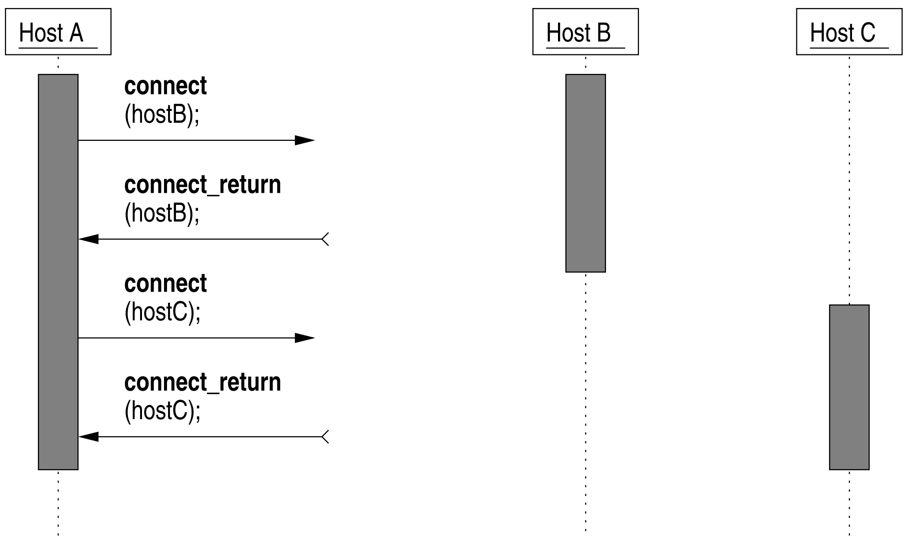

Abbildung 4.3: Interface-Modell für 'Verbindungsaufbauzeit' (OS)

```
// Verbindungsaufbauzeit (OS-Sicht)

INTERFACE os_connect {
 connect      ( target:  BYTE[] );
 connect_return ( target: BYTE[] );
};

EVENT connect_initiate (os_connect.connect, target) {};
EVENT connect_complete (os_connect.connect_return, target) {};

SAMPLE os_vaz_sample {
  USES
    EVERY connect_initiate AS a;
    EVERY connect_complete AS b;
  COMPUTES a.myMatch(b).time - a.hisMatch(b).time UNIT sec;
};

QOS_PARAMETER os_verbindungsaufbauzeit {
 SAMPLE 1 TIMES FROM os_vaz_sample TO PRODUCE STUB_DISTRIBUTION;
};
```

4.2 user information throughput

Als Nächstes betrachte ich den QoS-Parameter 'user information throughput' (siehe [X.641]). Auch hier werde ich in der Modellierung nur eine Seite betrachten, den Benutzer. Beim Benutzer können die Ereignisse *data_receive* und *data_processed* auftreten.

Die von mir gedachte Modellierung derart, daß im *data_processed*-Aufruf die Menge der verarbeiteten Daten übergeben und später im Sample ausgewertet werden soll, wird von Eqosul nicht unterstützt. Eventspezifische Datenstrukturen können im *event correlator* nicht abgebildet werden. Dies ist gewollt um unter anderem eine leichtere Vergleichbarkeit von Korrelationsmustern zu gewährleisten. Eqosul kennt als Ereignisattribute nur *ID* und *time*. Daher werde ich im Folgenden *data_processed* pro verarbeiteter Dateneinheit (z.B. Byte) generieren und anschließend die Queuefunktion *.count* zur Aufsummierung verwenden.

Zur Bestimmung einer zeitabhängigen Rate benötige ich in Eqosul ein weiteres Interface, das mir zu festgelegten Zeitpunkten Ereignisse liefert. Ich nenne es in diesem Beispiel 'clock'. Als Ereignis kennt es lediglich *second*.

Wie an diesem Beispiel deutlich wird, lassen sich auch QoS-Parameter auf die Messarchitektur abbilden, deren konkrete Messung technisch nicht oder nur unzureichend durchzuführen ist. Der QoS-Parameter *user information throughput* liegt oberhalb des OSI-Modells. Solche Messungen lassen sich mit Hilfe von Eqosul noch modellieren, mit [Neu 02] eventuell aber nicht messen.

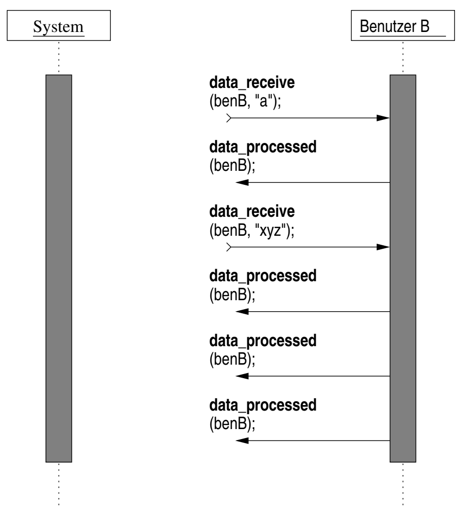

Abbildung 4.4: Interface-Modell für 'user information throughput'

```
// 'user␣information␣throughput'

  INTERFACE data_crunch {
    data_receive ( target: BYTE[],
                   data:     BYTE[] );
    data_processed ( target: BYTE[] );
  };

  INTERFACE clock {
    second     ();
  };

  EVENT data_sent (data_crunch.data_receive, target) {};
  EVENT data_done (data_crunch.data_processed, target) {};
  EVENT tick  (clock.second, NULL) {};
```

```

SAMPLE uit {
  USES
    EVERY data_done AS a;
    EVERY tick  AS b;
  COMPUTES a.count UNIT bit/sec;
};

QOS_PARAMETER user_information_throughput {
 SAMPLE 1 TIMES FROM uit TO PRODUCE NORMALIZED_DISTRIBUTION;
};
```

4.3 TCP-One-Way-Delay

Der QoS-Parameter *TCP-One-Way-Delay* ([RFC 2679]) ist sehr einfach zu modellieren. Das Interface besitzt zwei Ereignistypen: *tcp_send*, das beim Sender auftritt, und *tcp_receive*, das beim Empfänger auftritt. Gemessen wird die Zeitdifferenz zwischen diesen.

Da [RFC 2679] für den QoS-Parameter *TCP-one-way-delay* den Parameter 'Port' nicht kennt, werde ich dies hier entsprechend ohne Port modellieren. Es bleibt mithin der konkreten Implementierung des *data collector* überlassen, hier geeignete Werte zu bestimmen. Der QoS-Parameter *TCP-one-way-delay* ist somit unterspezifiziert, denn er misst zwischen zwei Kommunikationspartnern, die TCP einsetzen und TCP benötigt immer den Parameter 'Port'. Bei 'falscher' oder auch nur anderer Wahl des Parameters 'Port' durch denjenigen, der den *data collector* implementiert, können sich die gemessenen Werte erheblich unterscheiden. Beispiel: Implementierung 1 verwendet Port 25 (SMTP), der Zielrechner z hat aber auf diesem Port keinen offenen Socket. Implementierung 2 verwendet Port 22 (SSH), der Zielrechner z stellt auf diesem Port einen Dienst zur Verfügung. Implementierung 1 wird immer den Wert 'unendlich' liefern, Implementierung 2 einen 'tatsächlich gemessenen' Wert. Beide Implementierungen messen 'den selben QoS-Parameter', liefern jedoch völlig andere Werte.

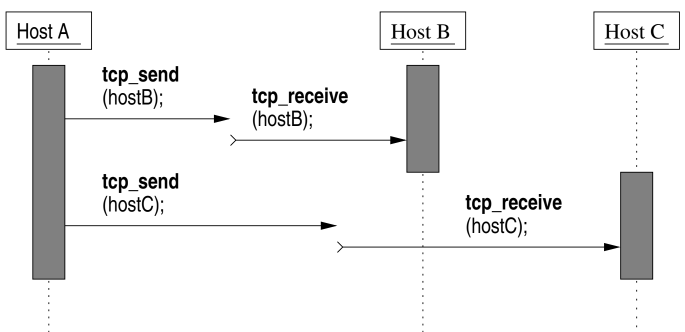

Abbildung 4.5: Interface-Modell für 'TCP-one-way-delay'

```
// TCP-One-Way-Delay (RFC 2679)

INTERFACE tcp {
 tcp_send   ( target:  BYTE[] );
 tcp_receive ( target:  BYTE[] );
};

EVENT start (tcp.tcp_send, target) {};
EVENT end (tcp.tcp_receive, target) {};

```

```
SAMPLE towd {
  USES
    EVERY start AS a;
    EVERY end AS b;
  COMPUTES a.myMatch(b).time - a.hisMatch(b).time UNIT sec;
  a.count COMPUTED EVENTS;
};

QOS_PARAMETER tcp_one_way_delay {
 SAMPLE 1 TIMES FROM towd TO PRODUCE NORMALIZED_DISTRIBUTION;
};
```

4.4 UDP-One-Way-Packet-Loss

Meine Idee zur Messung des *UPD-One-Way-Packet-Loss* basiert auf dem Quotienten 'erfolgreich empfangene Datagramme' zu 'insgesamt versendete Datagramme' und berechnet den Kehrwert dazu. Dies ist sowohl in der Modellierung des QoS-Parameters als auch in der praktischen Messung leichter als der direkte Ansatz über den Quotienten 'nicht oder fehlerhaft empfangene Datagramme' zu 'insgesamt empfangene Datagramme'. Zudem ist es mit Eqosul zur Zeit nicht möglich, Timeouts zu modellieren. Zu beachten ist hier, daß dieses *inverse Sampling* andere Werte ergeben wird als *zeitbasiertes Sampling*. Dies muss bei Vergleichen von Messungen berücksichtigt werden. In [RFC 2680] wird, wie schon in [RFC 2679], erneut auf die Angabe des Ports bei der Definition des QoS-Parameters verzichtet. Ich werde diese Unterspezifikation bei der Modellierung in Eqosul übernehmen.

Da ich in diesem Ansatz nur korrekt empfangene Datagramme betrachte, müssen implementierungsspezifische Details zur Bestimmung des Begriffs 'fehlerhaft' (wie z.B. UDP timeouts) nicht beachtet werden. Diese müssten sehr wohl modelliert werden, würde ich einen direkten Ansatz wählen, der das Verhältnis 'nicht oder fehlerhaft empfangene Datagramme' zu 'insgesamt empfangene Datagramme' unmittelbar berechnet.

Da Eqosul nur die Möglichkeit bot, einen einzelnen Wert vom Sample an das *statistic processing* weiterzugeben, musste Eqosul dahingehend erweitert werden, die Anzahl der betrachteten API-Aufrufe ebenfalls an das *statistic processing* weiterzugeben. Andernfalls kann keine sinnvolle Verarbeitung der Daten erfolgen. Es fehlt der Bezug, um mehrere gesampelte Werte auf der selben Skala normieren zu können. Hierzu ein Beispiel: Nehmen wir an, wir würden das Verhältnis von fehlerhaft gesendeten zu korrekt gesendeten TCP-Paketen messen. Beim ersten Sample-Vorgang erhalten wir den Wert 0,4. Beim zweiten Sample-Vorgang den Wert 0,8. Ohne eine Aussage darüber, über wievielen Ereignissen diese beiden Werte jeweils berechnet wurden, ist eine Zusammenfassung der Werte zu einem globalen Wert nicht möglich. Das erste Sample mag auf einer Messung mit insgesamt 100 Paketen beruhen, das zweite auf einer Messung mit 10 Paketen. Dies muß offensichtlich Eingang in die statistische Aufbereitung finden. Daher wurde das Sample in Eqosul ergänzt um die Angabe *n COMPUTED EVENTS*.

```
// UDP-One-Way-Packet-Loss (RFC 2680)

INTERFACE udp {
 udp_send    ( target:  BYTE[] );
 udp_receive ( target:  BYTE[] );
};

EVENT sent (udp.udp_send, target) {};
EVENT received (udp.udp_receive, target) {};

SAMPLE uowpl {
  USES
    EVERY sent AS a;
```

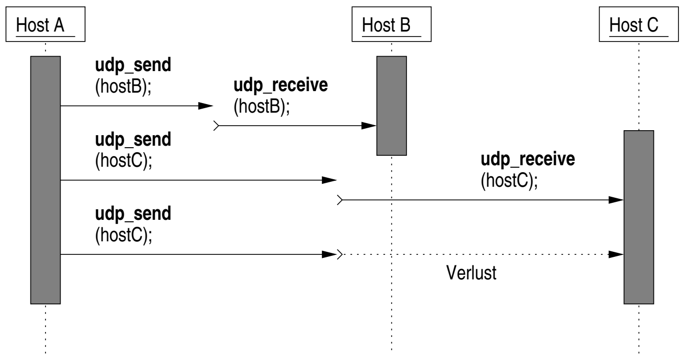

Abbildung 4.6: Interface-Modell für 'UDP-One-Way-Packet-Loss'

```
      EVERY received AS b;
    COMPUTES (1 - b.count/a.count) UNIT NULL;
    a.count COMPUTED EVENTS;
  };

  QOS_PARAMETER udp_one_way_packet_loss {
   SAMPLE 1 TIMES FROM uowpl TO PRODUCE NORMALIZED_DISTRIBUTION;
  };
```

4.5 TCP-Packet-Delay-Variation

Der in [RFC 3393] definierte QoS-Parameter *TCP-Packet-Delay-Variation*, baut auf [RFC 2679] auf. Es gelten daher die in Abschnitt 4.3 gemachten Feststellungen bzgl. Unterspezifizierung auch hier.

TCP-Packet-Delay-Variation verwendet weitgehend die selbe Messmethodik wie auch *TCP-One-Way-Delay* (siehe Abschnitt 4.3). Daher ist die Modellierung der Interfaces und Events identisch zu der bei *TCP-One-Way-Delay* vorgenommenen. Unterschiede treten auf im 'Sample'-Abschnitt und im 'QoS-Parameter'-Abschnitt.

Will man den QoS-Parameter wie spezifiziert berechnen, stösst man auf ein Problem. Der QoS-Parameter ist spezifiziert als Differenz zweier Zeitwerte (siehe 't1' und 't2' in Abbildung 4.7). Dies lässt sich zwar in Eqosul zunächst formulieren, liefert jedoch nicht das erwartete Ergebnis. Denn jeweils der zweite Zeitwert (im Beispiel 't2') muss in der Eventqueue verbleiben um bei der Berechnung des nächsten Samples erneut zur Verfügung zu stehen (dann in der Rolle von 't1'). Eqosul entfernt aber nach Abschluss der Berechnung eines Samples alle angegebenen Events aus der Eventqueue. Zudem stehen die Events nicht in den Queues, wie sie beim nächsten Durchlauf benötigt würden. Der Wert 't2' (genauer: die beiden Events aus denen sich 't2' errechnet) müssten nach der Berechnung von den Queues c und d in die Queues a und b verschoben werden. Also müsste Eqosul um Mechanismen zur Queuebearbeitung erweitert werden. Dies widerspricht aber dem Grundgedanken, die mit Eqosul erstellten QoS-Parameterspezifikationen so einfach und lesbar wie möglich zu halten. Gerade im angegebenen Beispiel wäre das tatsächliche Verhalten von Eqosul nicht mehr leicht zu verstehen. Eqosul würde sich von einer Beschreibungssprache zu einer (partiellen) Programmiersprache entwickeln. Und die komplexeren Spezifikationen führen auch zu einer schlechteren Vergleichbarkeit der QoS-Parameterdefinitionen insgesamt. Diese Art der Erweiterung von Eqosul ist daher vermutlich abzulehnen.

```
// TCP-Packet-Delay-Variation (RFC 3393)
// -- Achtung: so nicht moeglich --

  INTERFACE tcp {
```

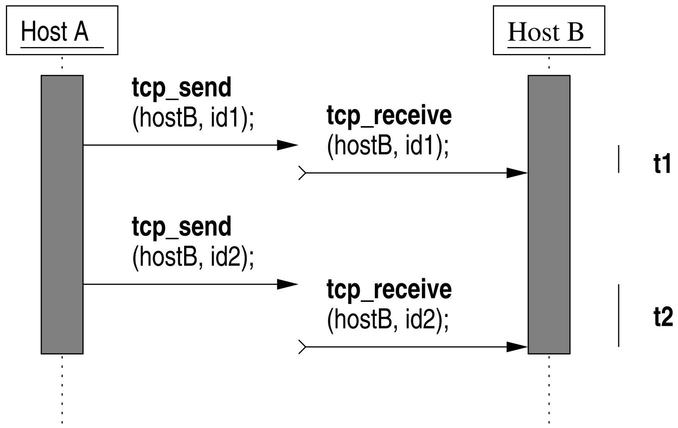

Abbildung 4.7: Interface-Modell für 'TCP-Packet-Delay-Variation'

```
   tcp_send   ( target:  BYTE[],
                ID:      BYTE[] );
   tcp_receive ( target: BYTE[],
                ID:      BYTE[] );
 };

 EVENT start1 (tcp.tcp_send, (target,id1)) {};
 EVENT end1 (tcp.tcp_receive, (target,id1)) {};
 EVENT start2 (tcp.tcp_send, (target,id2)) {};
 EVENT end2 (tcp.tcp_receive, (target,id2)) {};

 SAMPLE towd {
   USES
     EVERY start1 AS a;
     EVERY end1 AS b;
     EVERY start2 AS c;
     EVERY end2 AS d;
   // t1 = a.myMatch(b).time - a.hisMatch(b).time
   // t2 = c.myMatch(d).time - c.hisMatch(d).time
   COMPUTES (c.myMatch(d).time - c.hisMatch(d).time) - (a.myMatch(b).time - a.hisMatch(b).time) UNIT sec;
   a.count COMPUTED EVENTS;
 };

 QOS_PARAMETER tcp_packet_delay_variation {
  SAMPLE 1 TIMES FROM towd TO PRODUCE NORMALIZED_DISTRIBUTION;
 };
```

Man könnte Eqosul jedoch dahingehend erweitern, daß es eine Varianzberechnung unmittelbar im *statistic processing* ermöglicht. Es ist jedoch zu beachten, daß dieses Vorgehen nicht den selben Wert berechnet, wie es in [RFC 3393] spezifiziert ist, denn der dort angegebene Mechanismus berechnet keine Varianz über einer Stichprobe. Dann würde die Eqosul-Spezifikation wie folgt aussehen:

```
// TCP-Packet-Delay-Variation (RFC 3393) -- 2. Ansatz

 INTERFACE tcp {
   tcp_send   ( target:  BYTE[],
                ID:      BYTE[] );
   tcp_receive ( target: BYTE[],
                ID:      BYTE[] );
 };
```

```
EVENT start (tcp.tcp_send, (target,id)) {};
EVENT end (tcp.tcp_receive, (target,id)) {};

SAMPLE towd {
  USES
    EVERY start AS a;
    EVERY end AS b;
  COMPUTES a.myMatch(b).time - a.hisMatch(b).time UNIT sec;
  a.count COMPUTED EVENTS;
};

QOS_PARAMETER tcp_packet_delay_variation {
 // Eqosul-Erweiterung: "DERIVE␣VARIANZ" zur Ableitung der Varianz
 // Liefert einen einzigen Wert zurueck: Die Varianz ueber einer Stichprobe der Laenge 200
 SAMPLE 200 TIMES FROM towd TO DERIVE VARIANZ
};
```

4.6 ICMP-Round-Trip-Delay (ping)

Zur Messung des QoS-Parameters *ICMP-Round-Trip-Delay (ping)* benutze ich ein einfaches Interface-/Ereignis-Modell (siehe Abbildung 4.8) mit je zwei Ereignissen, *echo_request_send* und *echo_reply_receive*. *echo_request_send* tritt beim Senden eines ICMP-Pakets vom Typ 'echo request' auf, *echo_reply_receive* beim Empfang eines ICMP-Pakets vom Typ 'echo reply'. Zur Identifizierung bzw. Zuordnung der Antwortpakete zu den Anfragepaketen wird eine eindeutige ID generiert und in den beiden Ereignissen übergeben. Gemessen wird die Zeitdifferenz zwischen diesen zwei Ereignissen. Nicht betrachtet werden die (impliziten) Ereignisse *echo_request_receive* und *echo_reply_send*. Da es sich um einen Round-Trip-Wert handelt, sind für unsere Messung nur die Ereignisse auf Host A relevant.

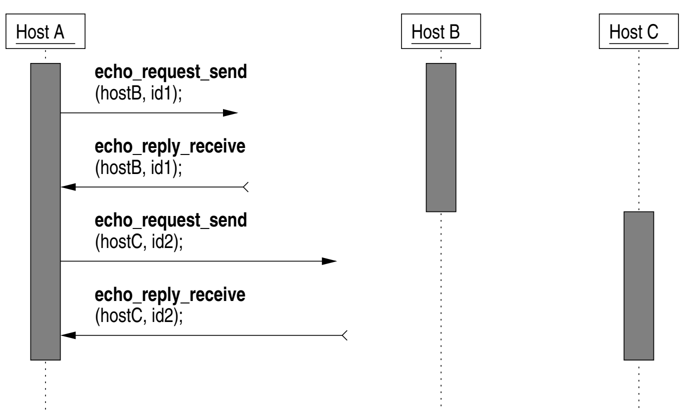

Abbildung 4.8: Interface-Modell für 'ICMP-Round-Trip-Delay (ping)'

```
// ICMP round trip delay

 INTERFACE icmp {
  echo_request_send (dest:  BYTE[4],
                ident: BYTE[] ); // eindeutige Identifizierung
  echo_reply_receive (dest:  BYTE[4],
                ident: BYTE[] ); // eindeutige Identifizierung
 };
```

```
EVENT icmp_request (icmp.echo_request_send, ident){};
EVENT icmp_reply (icmp.echo_reply_receive, ident){};

SAMPLE irtd {
  USES
        EVERY icmp_request AS a;
        EVERY icmp_reply AS b;
  COMPUTES a.myMatch(b).time - a.hisMatch(b).time UNIT sec;
  1 COMPUTED EVENTS;
};

QOS_PARAMETER icmp_round_trip_delay {
  SAMPLE 1 TIMES FROM irtd TO PRODUCE NORMALIZED_DISTRIBUTION;
};
```

4.7 WWW-Seitenaufbauzeit

Betrachten wir einen weiteren QoS-Parameter der oberhalb des OSI-Schichtenmodells liegt, die *WWW-Seitenaufbauzeit*. Darunter verstehe ich die Zeit zwischen Abschicken der URL und vom Browser fertig gerenderter Seite. Nicht gemeint ist die reine Renderingzeit des Browsers, diese ist für den Benutzer von geringerer Bedeutung und in dem von mir bestimmten QoS-Parameter enthalten.

Bei der Interface- und Ereignismodellierung (siehe Abbildung 4.9) kommen nur zwei Ereignisse vor, *eingabe_schicken* vom Benutzer zum Browser und *seitenaufbau_fertig* in der Rückrichtung. Gemessen wird die Zeitdifferenz zwischen diesen beiden beim Anwender auftretenden Ereignissen. Nicht betrachtet werden die beim Browser auftretenden Ereignisse. Diese sind, wie schon in Abschnitt 4.6, für unsere Betrachtung unerheblich.

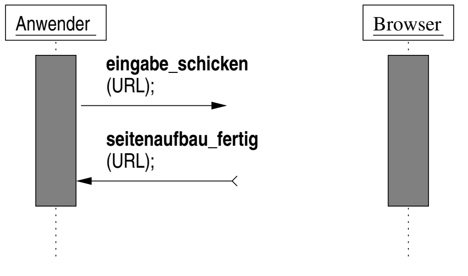

Abbildung 4.9: Interface-Modell für 'WWW-Seitenaufbauzeit'

```
// WWW-Seitenaufbauzeit

  INTERFACE anwender {
    eingabe_schicken (url: BYTE[] );
    seitenaufbau_fertig (url:  BYTE[] );
  };

  EVENT eingabe (anwender.eingabe_schicken, url) {};
  EVENT ausgabe (anwender.seitenaufbau_fertig, url) {};

  SAMPLE wsaz {
    USES
          EVERY eingabe AS a;
          EVERY ausgabe AS b;
    COMPUTES a.myMatch(b).time - a.hisMatch(b).time UNIT sec;
```

```
  1 COMPUTED EVENTS;
};

QOS_PARAMETER www_seitenaufbauzeit {
  SAMPLE 1 TIMES FROM wsaz TO PRODUCE NORMALIZED_DISTRIBUTION;
};
```

4.8 Framefehlerrate bei Ethernet

Zur Modellierung des QoS-Parameters *Framefehlerrate bei Ethernet* werde ich analog zu Abschnitt 4.4 vorgehen und den Kehrwert von 'Anzahl korrekt empfangener Rahmen' zu 'Anzahl insgesamt versendeter Rahmen' zur Berechnung verwenden. Ich behandele also fehlerhaft versendete, verlorengegangene und fehlerhaft übertragene bzw. fehlerhaft empfangene (i.e. auf dem Empfängersystem beschädigte oder inkorrekt interpretierte) Rahmen gleich.

Das Interface *ether* kennt zwei Ereignisse, *frame_send* und *frame_receive*. Der erstere tritt bei (auch fehlerhaftem) Versand eines Ethernet-Rahmens auf, der zweite bei korrektem Empfang eines Ethernet-Rahmens. Der einzige für diese Betrachtung relevante Parameter dieser Ereignisse ist jeweils 'MAC' zur eindeutigen Identifizierung des Kommunikationspartners. Bei der Betrachtung dieses QoS-Parameters sind bei meiner Modellierung die restlichen im Rahmen enthaltenen Daten (z.B. die Payload) nicht von Bedeutung.

Hier tritt eine Problematik zu Tage, die ich der Klarheit halber bei den bisherigen QoS-Parametern nicht beachtet habe, da diese ohnehin unterspezifiziert waren (siehe 4.3, 4.4 und 4.5) oder ohnehin nur 'einseitig' auf einer Seite der Kommunikationsbeziehung Ereignisse betrachtet haben (siehe 4.6): Modelliert sind logische point-to-point-Verbindungen zwischen den beiden Kommunikationspartnern, tatsächlich aber nutzen diese in der Regel *shared media*, über die der Datenverkehr mehrerer kommunizierender Partner verfügbar ist. Es muss daher auf Empfängerseite sichergestellt werden (mittels geeigneter Eventfilter), daß nur Ereignisse vom intendierten Kommunikationspartner betrachtet werden. Beispiel (siehe Abbildung 4.10): Host B empfängt üblicherweise nicht nur Daten von Host A über sein Ethernetinterface, sondern auch von anderen Hosts. Daher muss „hostA" im Event als Filter modelliert werden, sonst kommt es zu fehlerhaften Berechnungen. Es ist gerade nicht gewünscht die von anderen Hosts gesendeten Ethernetrahmen mit auszuwerten, sondern lediglich die von Host A gesendeten. Dies ist in der Modellierung zu berücksichtigen.

Als ersten Lösungsansatz kann man diesen Filter in den Eventnamen einfließen lassen. Also z.B. die Events wie folgt benennen: *frame_receive_hostA*, *frame_receive_hostB*, etc. Dies führt jedoch zu einem erheblich komplexeren *event correlator* und zu einer geringeren Vergleichbarkeit von QoS-Parametern.

Als zweiten Lösungsansatz ließe sich das Event parametrisieren (z.B. 'EVENT example (api.aufruf, (id-element1, id-element2)) (src: BYTE[1]) {}'). Dies führt zu einer 'Eventschar'. Dies wiederum führt zu einer Schar von Korrelatoren. Zu Beginn einer konkreten Messung müsste das System dann die konkreten Werte vom Benutzer erfragen um aus diesen Scharen jeweils eindeutige Instanzen auswählen bzw. generieren zu können. Zur Zeit ist dies in Eqosul nicht möglich, ich werde im weiteren Verlauf dieses Problem daher nicht mehr behandeln.

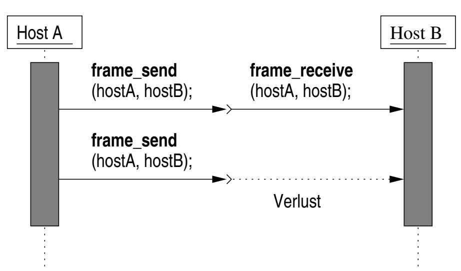

Abbildung 4.10: Interface-Modell für 'Framefehlerrate bei Ethernet'

```
// Framefehlerrate bei Ethernet

  INTERFACE ether {
    frame_send ( vonMAC:  BYTE[6],
                 anMAC:   BYTE[6] );
    frame_receive ( vonMAC: BYTE[6],
                 anMAC:   BYTE[6] );
  };

  EVENT sent (ether.frame_send, (vonMac, anMAC)) {};
  EVENT received (ether.frame_receive, (vonMAC, anMAC)) {};

  SAMPLE efr {
    USES
      EVERY sent AS a;
      EVERY received AS b;
    COMPUTES (1 - b.count/a.count) UNIT NULL;
    a.count COMPUTED EVENTS;
  };

  QOS_PARAMETER ethernet_fehlerrate {
    SAMPLE 1 TIMES FROM efr TO PRODUCE NORMALIZED_DISTRIBUTION;
  };
```

4.9 Wahrscheinlichkeit der Verbindungsablehnung bei TCP

In einem ersten Schritt modelliere ich den QoS-Parameter *Wahrscheinlichkeit der Verbindungsablehnung bei TCP* wie folgt:

Das Interface *tcp* kennt nur zwei Ereignisse, *tcp_initiate* und *tcp_established*. *tcp_initiate* tritt auf wenn der Host eine Verbindung aufbauen möchte, *tcp_established* wenn die Verbindung erfolgreich aufgebaut wurde. Wie schon in den Abschnitten 4.4 und 4.8 bilde ich auch hier den Kehrwert zu 'Anzahl erfolgreich etablierter Verbindungen' zu 'Anzahl insgesamt initiierter Verbindungen' zur Bestimmung des gesuchten QoS-Parameters. Mitunter fallen dann aber auch Ereignisse wie 'SYN-Paket verlorengegangen' unter den gemessenen Wert, nicht jedoch in die informelle Semantik des Ausdrucks 'Wahrscheinlichkeit der Verbindungsablehnung bei TCP' (da hier keine Ablehnung seitens des angesprochenen Zielrechners vorliegt sondern ein Übertragungsfehler). Aus Sicht des Benutzers dürfte dies keine Rolle spielen, da dieser am Zustandekommen der Verbindung interessiert ist, nicht an der genauen Fehlerursache.

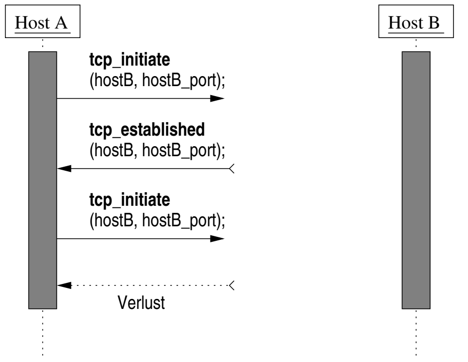

Abbildung 4.11: Interface-Modell für 'Wahrscheinlichkeit der Verbindungsablehnung bei TCP' (1. Variante)

```
// Wahrscheinlichkeit der Verbindungsablehnung bei TCP

  INTERFACE tcp {
   tcp_initiate ( host:  BYTE[],
                  port:   BYTE[] );
   tcp_established ( host: BYTE[],
                  port:   BYTE[] );
  };

  EVENT initiate (tcp.tcp_initiate, (host, port)) {};
  EVENT established (tcp.tcp_established, (host, port)) {};

  SAMPLE vaw {
    USES
      EVERY initiate AS a;
      EVERY established AS b;
    COMPUTES (1 - b.count/a.count) UNIT NULL;
    1 COMPUTED EVENTS;
  };

  QOS_PARAMETER tcp_verbindungsablehnungs_wahrscheinlichkeit {
   SAMPLE 1 TIMES FROM vaw TO PRODUCE NORMALIZED_DISTRIBUTION;
  };
```

Betrachtet man nun den Parameter *Wahrscheinlichkeit der Verbindungsablehnung bei TCP* wie er bei präziser, an der unmittelbaren Wortbedeutung („...ablehnung") orientierter Lesart zu verstehen wäre, ergibt sich folgende Modellierung:

Das Interface *tcp* besitzt nun die zwei Ereignisse *tcp_syn_sent* und *tcp_rst_received*. *tcp_syn_sent* tritt auf wenn der die Verbindung initiierende Rechner ein TCP-Paket mit dem SYN-Flag schickt (Beginn des 3-Wege-Handshakes). *tcp_rst_received* tritt auf wenn ein TCP-Paket mit dem RST-Flag (und passendem Tupel (host,port)) empfangen wird. Dies ist die *explizite* Verbindungsablehnung. Siehe hierzu Abbildung 4.12. Der QoS-Parameter lässt sich durch einfache Verhältnisbildung zwischen den empfangenen *tcp_rst_received*-Ereignissen und den gesendeten *tcp_syn_sent*-Ereignissen ermitteln. Das Sample gibt entweder den Wert '0' oder den Wert '1' zurück, je nachdem ob zu einem speziellen SYN-Paket ein RST-Paket empfangen wurde oder nicht. Daher kann hier '1 COMPUTED EVENTS' verwendet werden, da das Sample in der Tat nicht über mehrere Werte berechnet wird.

Nicht modelliert wird damit der Fall einer impliziten Verbindungsablehnung (der Zielrechner verwirft z.B. das eingehende SYN-Paket ohne ein RST-Paket zu versenden) oder, wie oben genannt, verlorengegangene Pakete.

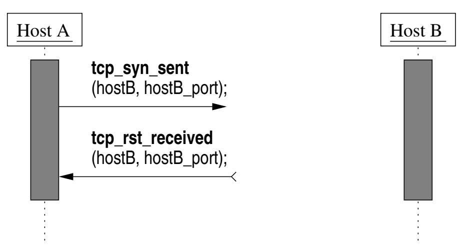

Abbildung 4.12: Interface-Modell für 'Wahrscheinlichkeit der Verbindungsablehnung bei TCP' (2. Variante)

```
// Wahrscheinlichkeit der Verbindungsablehnung bei TCP (2. Variante)

  INTERFACE tcp {
```

```
  tcp_syn_sent ( host:   BYTE[],
                 port:   BYTE[] );
  tcp_rst_received ( host: BYTE[],
                 port:   BYTE[] );
};

EVENT syn (tcp.tcp_syn_sent, (host, port)) {};
EVENT rst (tcp.tcp_rst_received, (host, port)) {};

SAMPLE vaw2 {
  USES
    EVERY syn AS a;
    EVERY rst AS b;
  COMPUTES (b.count/a.count) UNIT NULL;
  1 COMPUTED EVENTS;
};

QOS_PARAMETER tcp_verbindungsablehnungs_wahrscheinlichkeit2 {
 SAMPLE 1 TIMES FROM vaw2 TO PRODUCE NORMALIZED_DISTRIBUTION;
};
```

4.10 rtt (FIPA)

Der in [FIPA QoS], S. 3 definierte QoS-Parameter *rtt* (round trip time) lässt sich wie folgt modellieren und auf [Neu 02] abbilden:

Am Interface *data_transfer* tritt das Ereignis *data_send*, welches den Versand von Daten modelliert, und das Ereignis *data_confirmed*, welches den Eingang der Bestätigung modelliert, auf. Der QoS-Parameter ergibt sich als Zeitdifferenz zwischen diesen beiden Ereignissen.

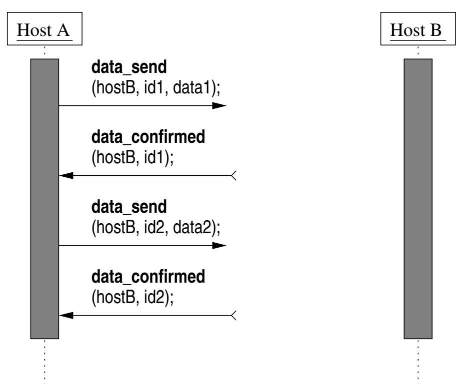

Abbildung 4.13: Interface-Modell für 'round trip time' (FIPA)

```
// rtt (FIPA -- round trip time)

 INTERFACE data_transfer {
  data_send  ( target:  BYTE[],
               ident:  BYTE[],
               data:   BYTE[] );
  data_confirmed ( target:  BYTE[],
               ident:  BYTE[] );
 };
```

```

EVENT sent  (data_transfer.data_send, ident) {};
EVENT confirmed (data_transfer.data_confirmed, ident) {};

SAMPLE rtt {
  USES
        EVERY sent AS a;
        EVERY confirmed AS b;
  COMPUTES a.myMatch(b).time - a.hisMatch(b).time UNIT sec;
  1 COMPUTED EVENTS;
};

QOS_PARAMETER fipa_rtt {
  SAMPLE 1 TIMES FROM rtt TO PRODUCE NORMALIZED_DISTRIBUTION;
};
```

4.11 delay (FIPA)

Der von der FIPA *delay* genannte QoS-Parameter lässt sich sehr einfach wie folgt modellieren.

Das Interface kennt zwei Ereignisse, *data_send* und *data_receive*, die beim Versenden bzw. Empfang eines Datenpakets ausgelöst werden. Diese besitzen jeweils 4 Parameter, *from*, *to*, *ident* und *data*. *data* stellt die versendeten Nutzdaten dar, *ident* einen eindeutigen Identifikator für diese Datenversendung, *to* die Zieladresse (gegebenenfalls als Tupel wenn dies durch das konkrete Protokoll notwendig sein sollte) und *from* die Absenderadresse. Berechnet wird schlicht die Zeitdifferenz zwischen dem Absenden der Daten und dem Eintreffen der Daten.

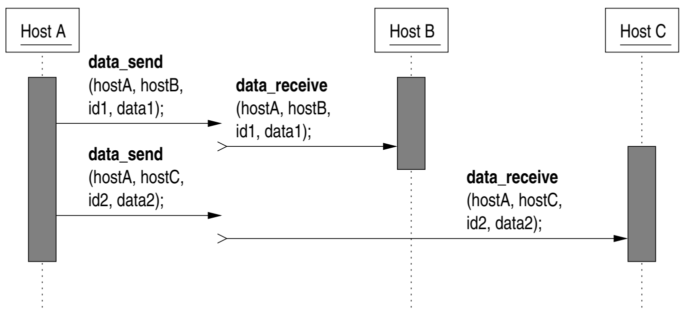

Abbildung 4.14: Interface-Modell für 'delay' (FIPA)

```
// delay (FIPA)

 INTERFACE data_transfer {
  data_send    ( from:   BYTE[],
                 to:     BYTE[],
                 ident:  BYTE[4],
                 data:   BYTE[] );
  data_receive ( from:   BYTE[],
                 to:     BYTE[],
                 ident:  BYTE[4],
                 data:   BYTE[] );
 };

 EVENT sent  (data_transfer.data_send, (from, to, ident)) {};
 EVENT received (data_transfer.data_receive, (from, to, ident)) {};
```

```

SAMPLE del {
  USES
    EVERY sent AS a;
    EVERY received AS b;
  COMPUTES a.myMatch(b).time - a.hisMatch(b).time UNIT sec;
  a.count COMPUTED EVENTS;
};

QOS_PARAMETER fipa_delay {
 SAMPLE 1 TIMES FROM del TO PRODUCE NORMALIZED_DISTRIBUTION;
};
```

4.12 Zusammenfassung

Ich komme nun zur Zusammenfassung der Ergebnisse dieses Kapitels.

Zur Beantwortung der Frage, welche QoS einander ähnlich sind bzw. zur selben „Klasse" gehören müssen bestimmte Kriterien der Bewertung überprüft werden. Da ich drei der 11 QoS-Parameter auf zwei verschiedene Arten modelliert habe, ergibt sich in Summe die Zahl 14 für die QoS-Parameter.

12 mal wurde der QoS-Parameter mit 2 Queues im Sample modelliert, 1 mal mit 3 Queues und 1 mal mit 4 Queues.

7 mal berechnete der QoS-Parameter einen Zeitwert, 4 mal eine Wahrscheinlichkeit, 2 mal eine Varianz (s.o. Abschnitt 4.5 für Details) und 1 mal ein Durchsatz pro Zeit.

7 der QoS-Parameter wurden einseitig modelliert, i.e. ohne detaillierte Darstellung der Ereignisse auf Seite des Kommunikationspartners. 7 QoS-Parameter wurden zweiseitig modelliert.

Die QoS-Parameter *Verbindungsaufbauzeit (OS)* (Abschnitt 4.1), *ICMP-Round-Trip-Delay* (Abschnitt 4.6), *WWW-Seitenaufbauzeit* (Abschnitt 4.7) und *rtt (FIPA)* (Abschnitt 4.10) stellen die Klasse der einseitig mit 2 Queues gemessene Zeitwerte dar. Diese *round trip time*-Werte können beispielsweise von Providern nicht oder nur sehr erschwert als QoS-Parameter angeboten werden da beide Messpunkte ('Interfaces') auf der selben Seite, beim Benutzer liegen. Diese QoS-Parameter bilden aber eine für den Benutzer wichtige Klasse, da sie die 'Responsiveness' der Netzanbindung des Benutzers beschreiben.

Die Klasse der *one way delays* bilden die beiden QoS-Werte *TCP-One-Way-Delay* und *delay (FIPA)*. Zu ihrer Messung muss eine Zugangsmöglichkeit auf beiden zur Verbindung gehörigen Interfaces bestehen. Sie sind aus Sicht des Benutzers nicht so wichtig wie die vorige Klasse, da sie in *round trip time*-Werte beinhaltet sind, bzw. der Benutzer die *round trip time*-Werte wissen möchte und daher an den Grundbausteinen der *one way delays* nur ein geringes Interesse hat.

Für den Endnutzer noch von großer Bedeutung sind die Durchsatzraten, da die Durchsatzrate bei großen Datentransfers wichtiger als die Verbindungsaufbau- und abbauzeiten wird und die transferierten Datenmengen konstant zunehmen. Aus dieser Klasse habe ich *user information throughput* betrachtet.

Die Fehlerraten (*UDP-One-Way-Packet-Loss*, Abschnitt 4.4, *Framefehlerrate bei Ethernet*, 4.8 und *Wahrscheinlichkeit der Verbindungsablehnung*, Abschnitt 4.9) sind für den Benutzer häufig nur indirekt interessant, da sie andere (unmittelbar interessante) Werte wie die Verbindungsaufbauzeit oder den Datendurchsatz beeinflussen.

Sehr deutlich geworden ist auch, daß diejenigen QoS-Parameter die mit Eqosul auf mehrere Arten modelliert werden können, die grössten Freiheitsgrade in ihrer (umgansspraсhlichen oder semiformalen) Definition aufweisen. Je Präziser ein QoS-Parameter definiert ist, desto weniger Freiräume bleiben bei der Modellierung mittels Eqosul. Zudem stellt eine Modellierung eines Parameters eine formale Spezifikation dar, die nur wenige Unklarheiten in der Interpretation zulässt. Die verbleibenden Freiheiten stammen aus dem hohen Abstraktionsgrad der Modellierung. Dieser hohe Abstraktionsgrad wiederum erlaubt eine leichtere Einteilung in Klassen bzw. Vergleichbarkeit zwischen den QoS-Parametern.

Kapitel 5

sanity check der Abbildung

In Kapitel 4 habe ich mittels Eqosul ([Gar 04]) Abbildungen von QoS-Parametern auf die geschichtete Messarchitektur [Neu 02] vorgenommen. In diesem Kapitel werde ich Methoden diskutieren, diese Abbildungen zu überprüfen. Mit diesen Methoden soll festgestellt werden, ob Änderungen am QoS-Parameter auch tatsächlich Auswirkungen auf den gemessenen Wert zeigen, ob die Messarchitektur also tatsächlich den QoS-Parameter misst.

5.1 Sequenzdiagramme

Eine Möglichkeit der Überprüfung ist es, die Auswirkungen verschiedener Protokollmechanismen in den in Kapitel 4 dargestellten Sequenzdiagrammen abzubilden. Dies ist für einen *sanity check*, der nur rudimentär Fehler in der Modellierung (oder in der Messarchitektur) entdecken soll, ausreichend. Eine vollständige Prüfung bzw. Verifikation der Messarchitektur und der Abbildungssprache ist damit weder möglich noch beabsichtigt.

Die durch die Sequenzdiagramme dargestellten QoS-Parameter stellen ihrerseits Protokolle dar, denn es handelt sich um „Vereinbarungen über den geordneten Ablauf einer Kommunikation [...] Entsprechend der aufeinander aufbauenden Schichten bei der Übertragung von Informationen wird für jede Ebene ein eigenes Protokoll vereinbart, dessen Realisierung sich auf das Protokoll der nächst tieferen Ebene abstützt." (Def. 'Protokoll', [Duden Inform.], S. 555). Diese QoS-Parameter stützen sich auf Protokolle niedrigerer Schichten ab. Zum Beispiel stützt sich der QoS-Parameter *TCP-One-Way-Delay* auf die Protokolle TCP, IP, Ethernet, etc. ab.

Diese Protokolle nutzen (jeweils) eine bestimmte Menge an Protokollmechanismen, also z.B. multiplexing, buffering, etc. und stützen sich ihrerseits auf Protokolle niedrigerer Schichten ab.

Im Ergebnis erhalten wir einen (Ausschnitt des) OSI-Stack mit dem betrachteten QoS-Parameter als Schicht n und den genutzten Protokollen in den Schichten darunter.

Für unsere Betrachtung ungünstig ist jedoch genau die Tatsache, daß jedes einzelne Protokoll in diesem Stack bestimmte Protokollmechanismen einsetzt. Dabei kann es vorkommen, daß verschiedene Schichten dieses Stacks denselben Protokollmechanismus einsetzen. Beispielsweise nutzt TCP den Mechanismus 'multiplexing' von IP. IP wiederum nutzt das (stochastische) Multiplexing von Ethernet. Wir erhalten eine Protokollmechanismen-Hierarchie bzw. -Schichtung. Die Auswirkungen dieser Protokollmechanismen überlagern sich und lassen es nicht zu, eindeutig die Auswirkung des auf Schicht n untersuchten Protokollmechanismus auf den QoS-Parameter zu ermitteln.

Diese Überlegung bestätigt meine bereits im Kapitel 1 angesprochene Behauptung, die Transparenz des OSI-Stacks sei lediglich aus funktionaler Sicht gegeben, nicht jedoch im Hinblick auf QoS-Parameter.

Diese problematische Überlagerung der Auswirkungen von Protokollmechanismen wird noch dadurch erschwert, daß die unterhalb des betrachteten QoS-Parameters (oder zumindest die in Schicht n durch den QoS-Parameter selbst) verwendeten Protokollmechanismen in den Sequenzdiagrammen modelliert werden müssten um ihre Auswirkungen überhaupt untersuchen zu können.

Dies widerspricht aber zum einen grundsätzlich der Idee, möglichst abstrakt und von konkreten Implementierungen losgelöst zu modellieren, zum anderen würden die Sequenzdiagramme eine Komplexität erreichen, die der Idee eines einfachen *sanity check* zuwiderlaufen würde.

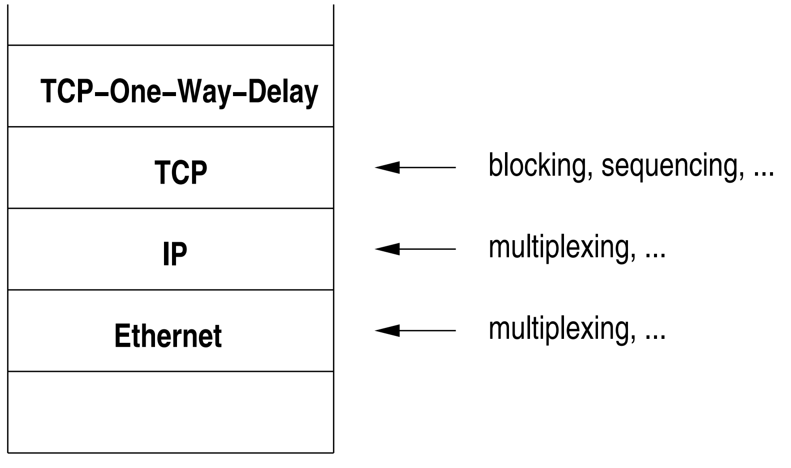

Abbildung 5.1: Schichtung der Protokolle unterhalb eines QoS-Parameters

Im Ergebnis lässt sich feststellen, daß dieser Ansatz wohl nicht geeignet sein wird, sinnvolle Aussagen über die Modellierung, die Sprache oder die Messarchitektur zu treffen.

5.2 Java-Testlauf

Eine weitere Möglichkeit der Überprüfung bietet sich unmittelbar aus dem von Eqosul generierten Java-Code heraus an.

Eqosul soll aus der QoS-Parameterdefinition lauffähigen Javacode erzeugen. Die Module **ec** und **sp** können von Eqosul vollständig erzeugt werden, das Modul **dc** kann nur als Rahmen erzeugt werden, da es die implementierungsspezifischen Details enthält und diese durch unsere abstrakte Modellierung gerade nicht festgelegt werden und den konkreten Code enthalten müssen um an den konkret für die Messungen herangezogenen Interfaces die Messdaten zu erfassen. Da diese Module ohnehin manuell implementiert werden müssen, bietet es sich an, dem Programm nicht von außen Daten einzuspielen (z.B. über einen Leitungssimulator) sondern von den **dc**-Modulen direkt künstliche Ereignisse erzeugen zu lassen.

Auf diese Art lassen sich die Eingangsdaten sehr genau kontrollieren, was eine wichtige Voraussetzung für Aussagen über die Genauigkeit von Eqosul darstellt. Siehe dazu Abbildung 5.2.

Wurde der QoS-Parameter korrekt modelliert und arbeitet Eqosul den Erwartungen gemäß, so sollten sich nur geringfügige Abweichungen zwischen den Eingangsdaten und den von Eqosul ausgegebenen Daten bemerkbar machen.

Meiner Meinung nach (ohne Beweis) sollten die Graphen der Messdaten nicht deutlich in ihrer Form verändert worden sein. Zudem sollten Verschiebungen (sowohl entlang der Ordinate als auch entlang der Abszisse) möglichst klein gegen die Messwerte sein. Da es sich aber bei unserer Untersuchung hier nur um einen *sanity check* und keine exakte statistische Betrachtung handelt, ist jedoch eine informelle Aussage über die 'annähernde Gleichheit der Eingangs- und Ausgangsdaten' ausreichend (siehe Abbildung 5.2 als Beispiel für eine verfälschende Messung).

Leider war es mir bis zum Zeitpunkt der Abgabe dieser Arbeit mit Eqosul noch nicht möglich, für die betrachteten QoS-Parameter lauffähigen Code zu generieren. Eqosul erzeugt zum jetzigen Zeitpunkt weder den Code für das **sp**-Modul noch wird der Code für das **ec**-Modul vollständig generiert. Es ist jedoch abzusehen, daß dies in Kürze möglich sein wird.

Als Nachteil dieser Methode empfinde ich es, daß Eqosul dabei durch von Eqosul generiertem (Rahmen-) Code überprüft wird, i.e. eine zu enge Bindung zwischen testendem und getestetem Code besteht. Systematische Fehler in der Codeerzeugung würden den Test ad absurdum führen.

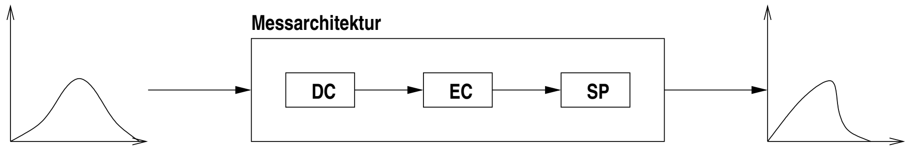

Abbildung 5.2: Verfälschung der Messdaten durch die Messarchitektur

Kapitel 6

Zusammenfassung und Ausblick

Diese Diplomarbeit hat zuerst einen kurzen Einblick in die geschichtete Messarchitektur gegeben, die im Rahmen dieser Arbeit zur Messung von QoS-Parametern verwendet wurde. Sodann wurde eine Abbildungssprache als geeigneter Formalismus vorgestellt, um generell QoS-Parameter auf diese Messarchitektur abzubilden. Anschließend wurde aus mehreren Quellen eine Liste von QoS-Parametern ausgewählt und beschrieben.

Anhand der Liste von QoS-Parametern sowie einer Liste von Protokollmechanismen, die ebenfalls kurz vorgestellt wurde, habe ich eine Betrachtung durchgeführt, wie QoS-Parameter generell durch allgemeine Protokollmechanismen beeinflusst werden (können).

Im Anschluss daran habe ich mit Hilfe der Abbildungssprache die ausgewählten QoS-Parameter auf die geschichtete Messarchitektur abgebildet und am konkreten Beispiel notwendige Erweiterungen für diese Abbildungssprache postuliert sowie bestimmte Grenzen aufgezeigt.

Zuletzt habe ich zwei Methoden beschrieben, wie diese Abbildung einem sogenannten *sanity check* unterzogen werden könnte und inwieweit diese Ansätze tatsächlich für solch eine Überprüfung geeignet sind.

Im Hinblick auf die Abbildungssprache hat sich gezeigt, daß sie für die betrachteten Fälle generisch genug ist, um diese modellieren zu können. Einzige Ausnahme ist der QoS-Parameter *TCP-Packet-Delay-Variation* (siehe Abschnitt 4.5), der es nötig machen würde, Queue-Manipulationsoperationen in Eqosul einzuführen. Dies würde aber diese Beschreibungssprache um Allgemeinprogrammiersprachliche Konzepte erweitern und dadurch die bis jetzt übersichtlichen, deklarativen QoS-Beschreibungen (in der Mehrzahl der Fälle) unnötig unübersichtlich machen.

Zudem hat sich gezeigt, daß Eqosul nicht nur hinreichend generisch für die Abbildung von QoS-Parametern auf die betrachtete Messarchitektur ist, sondern auch einen geeigneten Formalismus darstellt, um QoS-Parameter präzise zu beschreiben. Dies wiederum offenbart Ungenauigkeiten und Spielräume in der Interpretation der umgangssprachlich oder semiformal definierten Parameter. Dies ist aber Voraussetzung sollen QoS-Merkmale zum Beispiel über Providergrenzen hinweg zur Verfügung stehen. Denn nur bei exakter (i.e. formaler) Definition der QoS-Parameter kann sichergestellt werden, daß auch tatsächlich alle Beteiligten den selben Wert bereitstellen bzw. erwarten.

Eqosul kann darüberhinaus auch für QoS-Parameter verwendet werden, die 'oberhalb' des OSI-Stacks liegen bzw. nicht mit üblichen technischen Ansätzen gemessen werden können. Bei geeigneter Modellierung stellt dies für Eqosul keine großen Probleme dar. Es können sich jedoch Probleme bei der konkreten Umsetzung dieser Modellierungen in Codefragmente ergeben.

Weiterhin hat sich erwiesen, daß die betrachteten QoS-Rahmenwerke, -Definitionen und -Ontologien unbrauchbar sind, sollten sie über Organisationsgrenzen hinweg zum Einsatz kommen. Die dort vorgenommenen Definitionen sind -selbst bei Werken die nur einen sehr engen Ausschnitt der Wirklichkeit wie z.B. mobile Agenten eines einzigen Firmenkonsortiums betreffen- allesamt nicht präzise genug formuliert um als Grundlage für die organisationsübergreifende Bereitstellung von QoS-Parametern oder den ad hoc-Wechsel (bzw. die ad hoc-Auswahl) eines Providers auf Grund angebotener QoS-Merkmale zu dienen. Selbst Werke, die zur Definition eines einzigen QoS-Parameters 10 Seiten aufwenden (siehe die Definitionen der IPPM), erreichen auf Grund fehlender formaler Beschreibungsmechanismen nicht die notwendige Präzision.

Schließlich hat sich gezeigt, daß die Messarchitektur, wie die Abbildungssprache, einen hohen Grad an Generik besitzt und alle betrachteten QoS-Parameter mit Hilfe dieses Schemas abgedeckt werden können.

Da meine Betrachtungen nur exemplarischer Natur waren, bleibt zu untersuchen, ob sich nicht doch QoS-Parameter finden bzw. definieren lassen, die durch die Messarchitektur bzw. Eqosul nicht abbildbar sind.

Im Hinblick auf Protokollmechanismen bleibt der Beweis schuldig, daß die von mir verwendete (bzw. irgendeine) Sammlung von Mechanismen alle denkbaren Protokolle abdecken kann. Selbst für den eingeschränkten Fall, der die Daten unabhängig von ihren Werten behandelt (i.e. datenneutral ist und höchstens über der Länge der Daten operiert) ist dies fraglich. Aus diesem Grund bleibt auch die Frage ungeklärt, inwieweit aus gemessenen QoS-Parametern auf die verwendeten Protokollmechanismen geschlossen werden kann, also die 'Implementierungs-black-box' entschlüsselt werden kann.

Bestätigt hat sich meiner Ansicht nach die Vermutung, daß sich der OSI-Stack zwar aus Sicht des Anwenders funktional transparent verhält, sich mit Blick auf QoS-Parameter aber keineswegs transparent verhält.

Anhang A

Listings der Hilfsprogramme

A.1 poi-client.c

```c
/* $Id: poi_client.c,v 1.1 2003/04/03 11:53:52 uhlf Exp $ */

/* ------------------------------------------------------------------------------- */

/* But why worry about style? Who cares what a program looks like if it works?
   Doesn't it take too much time to make it look pretty? Aren't the rules
   arbitrary anyway? The answer is that well-written code is easier to read and
   to understand, almost surely has fewer errors, and is likely to be smaller
   than code that has been carelessly tossed together and never polished. (...)
   Sloppy code is bad code - not just awkward and hard to read, but often
   broken.

   Brian W. Kernighan & Rob Pike in "The Practice of Programming" */

/* ------------------------------------------------------------------------------- */

#include <errno.h>
#include <netdb.h>
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sysexits.h>
#include <time.h>
#include <sys/time.h>
#include <unistd.h>

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include "debug_module.h"
#include "log_module.h"

/* ------------------------------------------------------------------------------- */
/* function prototypes */
```

```
void busywait(void);
long timestamp(void);
int main(int argc, const char * argv[]);

/* ------------------------------------------------------------------------------ */

const  char *daten = "a";
       char *rechner = "localhost"; /* default falls nicht Wert als Parameter uebergeben */
unsigned int count = 5000;   /* default falls nicht Wert als Parameter uebergeben */
unsigned int delay = 0;      /* default falls nicht Wert als Parameter uebergeben */
unsigned int port = 4711;    /* default falls nicht Wert als Parameter uebergeben */

/* ------------------------------------------------------------------------------ */

void busywait(void) {

  long diff;
  long jetzt;
  long startzeit;

  /* Mikrosekunden */
  startzeit = timestamp();

  while (1) {

    jetzt = timestamp();

    diff = jetzt - startzeit;

    if (diff < 0) {
      diff = 1000000 + diff; /* Sekundenumbruch */
      debug("[Client] Sekundenumbruch in busywait()\n");
      debug("[Client] diff=%lu, jetzt=%lu, startzeit=%lu\n", diff, jetzt, startzeit);
    }

    if (diff > delay)
      break;

  } /* while */

  return;

} /* busywait() */

/* ------------------------------------------------------------------------------ */

long timestamp(void) {

  struct timeval tv;

  gettimeofday(&tv, NULL);

  return tv.tv_usec;

} /* timestamp() */

/* ------------------------------------------------------------------------------ */
```

```
int main(int argc, const char * argv[]) {

  char logfile[30];
  int i=0;
  int Socket;
  long diff=0;
  long *p;
  long stempel_alt=0;
  long stempel_neu=0;
  struct sockaddr_in dest_addr;
  struct hostent *host;

  /* ----- ACHTUNG -----

    Wenn nur ein Parameter uebergeben wird, ist das die Portnummer.
    Bei zwei Parametern ist der erste der Rechnername und der _zweite_
    der Port (und der dritte dann der Delay)

    ----- ACHTUNG ----- */

  if (argc == 1) {
    ;
  } else if (argc == 2) {
    port = atoi(argv[1]);
  } else if (argc == 3) {
    port = atoi(argv[2]);
    rechner = (char *) argv[1];
  } else if (argc == 4) {
    delay = atoi(argv[3]);
    port = atoi(argv[2]);
    rechner = (char *) argv[1];
  } else {
    printf("Syntax:␣␣␣%s\n", argv[0]);
    printf("␣␣␣␣␣␣␣␣␣␣%s␣<port>\n", argv[0]);
    printf("␣␣␣␣␣␣␣␣␣␣%s␣<host>␣<port>\n", argv[0]);
    printf("␣␣␣␣␣␣␣␣␣␣%s␣<host>␣<port>␣<delay>\n", argv[0]);
    printf("Defaults:␣Host=%s,␣Port=%u,␣Count=%u,␣Delay=%u\n", rechner, port, count, delay);
    exit(EX_USAGE);
  }
  printf("[Client]␣Einstellungen:␣Host=%s,␣Port=%u,␣Count=%u,␣Delay=%u\n", rechner, port, count, delay);
  sprintf(logfile, "log_c%u_d%u.dat", count, delay);

  debug("[Client]␣Versuche␣Rechnernamen␣auszuwerten\n");
  host = gethostbyname(rechner);
  if (host == NULL) {
    herror("[Client]␣Problem␣bei␣der␣Auswertung␣des␣Hostnamens");
    fflush(stdout);
    exit(EX_UNAVAILABLE);
  }

  debug("[Client]␣Versuche␣Socket␣anzulegen\n");
  if ((Socket=socket(PF_INET, SOCK_STREAM, 0)) < 0) {
    perror("[Client]␣Kann␣Socket␣nicht␣anlegen");
    exit(EX_OSERR);
  }
  debug("[Client]␣Socket␣angelegt\n");
```

```
  dest_addr.sin_family = AF_INET;
  dest_addr.sin_port = htons(port);
  dest_addr.sin_addr = *((struct in_addr *)host->h_addr);
  memset(&(dest_addr.sin_zero), '\0', 8);

  debug("[Client]␣Versuche␣connect()␣durchzufuehren\n");
  if (connect(Socket, (struct sockaddr *) &dest_addr, sizeof(struct sockaddr_in)) < 0) {
      if (errno == ECONNREFUSED) {
        perror("[Client]␣Verbindung␣vom␣Zielcomputer␣abgelehnt␣...␣evtl.␣kein␣Server␣am␣Laufen");
      } else {
        perror("[Client]␣Kann␣keinen␣connect()␣zum␣Server␣durchfuehren");
      } /* if (errno == ... */
      exit(EX_OSERR);
  } /* if */

  p = (long *) calloc(count, sizeof(long));
  if (p == NULL) {
    perror("[Client]␣calloc()␣ging␣daneben");
    exit(EX_OSERR);
  }

  stempel_neu = timestamp();
  debug("[Client]␣Sende␣Anfrage(n)␣an␣Server\n");
  for (i=0; i<count; i++) {
    if (write(Socket, daten, 2) < 0) {
      perror("[Client]␣Kann␣nicht␣auf␣Socket␣schreiben");
      free(p);
      exit(EX_OSERR);
    }
    stempel_alt = stempel_neu;
    stempel_neu = timestamp();
    diff = stempel_neu - stempel_alt;
    if (diff < 0) {
      diff = 1000000 + diff; /* Sekundenumbruch */
    }
    p[i] = diff;
    debug("[Client]␣%lu␣%lu␣%lu\n", stempel_alt, stempel_neu, diff);
    if (delay > 0) {
      debug("[Client]␣busywait()\n");
      busywait();
      debug("[Client]␣busywait()␣vorbei\n");
    }
  } /* for */

  log_open(logfile);
  for (i=0; i<count; i++) {
    log("%lu\n", p[i]);
  }
  log_close();

  debug("[Client]␣Anfrage␣an␣Server␣durchgefuehrt\n");

  free(p);
  return(EX_OK);

} /* main */
```

A.2 poi-server.c

```
/* $Id: poi_server.c,v 1.1 2003/04/03 11:33:28 uhlf Exp $

  2003-03-10, Florian Uhl

*/

/* ------------------------------------------------------------------------------ */

#include <stdarg.h>
#include <stdio.h>
#include <string.h>
#include <sysexits.h>
#ifdef __linux
  #include <sys/time.h>
#else
  #include <time.h>
#endif
#include <unistd.h>

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <errno.h>

#include <fcntl.h>
#include <sys/stat.h>

#include <stdlib.h>

#include "debug_module.h"
#include "log_module.h"

/* ------------------------------------------------------------------------------ */

  static      char *logfile = "log_c5000.dat";
  static      char *pidfile = "poi_server.pid";
  static      char *portfile = "poi_server.port";
  static      char *rechner = "pcheger2";
              int  log_desc;
  static unsigned int count = 5000;
  static unsigned int port = 4711;
              long *p;

/* ------------------------------------------------------------------------------ */

void ende (int exit_code) {

  debug ("[Server]␣Loesche␣PID-Datei\n");
  unlink(pidfile);
  debug ("[Server]␣Loesche␣Port-Datei\n");
  unlink(portfile);
  debug ("[Server]␣Schliesse␣Log-Datei\n");
  log_close();
  debug ("[Server]␣Programm␣terminiert\n");
```

```
  exit(exit_code);

} /* ende */

/* ------------------------------------------------------------------------------- */

void pid_file (void) {

  char buf[sizeof(pid_t)];
  int descriptor;
  pid_t pid;

  pid = getpid();
  debug("[Server] PID = %u\n", pid);

  umask(S_IWGRP|S_IWOTH); /* entspricht 022 */

  debug("[Server] Versuche PID-Datei zu erzeugen/oeffnen\n");
  #ifdef __linux
    descriptor = open(pidfile, O_RDWR|O_CREAT|O_TRUNC, S_IRUSR|S_IRGRP|S_IROTH);
  #else
    descriptor = open(pidfile, O_RDWR|O_CREAT|O_TRUNC|O_NOFOLLOW, S_IRUSR|S_IRGRP|S_IROTH);
  #endif

  sprintf(buf, "%u", pid);

  write(descriptor, buf, sizeof(pid_t)+1);

  debug("[Server] Versuche PID-Datei zu schliessen\n");
  close(descriptor);

} /* pid_file() */

/* ------------------------------------------------------------------------------- */

void port_file (void) {

  char buf[sizeof(pid_t)];
  int descriptor;

  umask(S_IWGRP|S_IWOTH); /* entspricht 022 */

  debug("[Server] Versuche Port-Datei zu erzeugen/oeffnen\n");
  #ifdef __linux
    descriptor = open(portfile, O_RDWR|O_CREAT|O_TRUNC, S_IRUSR|S_IRGRP|S_IROTH);
  #else
    descriptor = open(portfile, O_RDWR|O_CREAT|O_TRUNC|O_NOFOLLOW, S_IRUSR|S_IRGRP|S_IROTH);
  #endif

  sprintf(buf, "%u", port);

  write(descriptor, buf, sizeof(port)+1);

  debug("[Server] Versuche Port-Datei zu schliessen\n");
  close(descriptor);

} /* port_file() */
```

```c
/* ------------------------------------------------------------------------------- */

long timestamp (void) {

  struct timeval tv;

  gettimeofday(&tv, NULL);

  return tv.tv_usec;

} /* timestamp */

/* ------------------------------------------------------------------------------- */

int main(int argc, char **argv) {

         char     eingabe[2];
         socklen_t addrlen;
         int      err;
  extern int      errno;
         int      i=0;
  unsigned int    j;
         int      Socket;
         int      WorkSocket;
         long     diff=0;
         long     stempel_alt=0;
         long     stempel_neu=0;
  struct sockaddr_in socketaddress;

  pid_file();
  log_open(logfile);

  /* ----- */

  debug("[Server] Versuche socket()\n");

  Socket=socket(PF_INET, SOCK_STREAM, 0);

  if (Socket < 0) {
    perror("[Server] socket() fehlgeschlagen");
    ende(EX_OSERR);
  } else {
    debug("[Server] socket() erfolgreich (Socket = %u)\n", Socket);
  }

  /* ----- */

  socketaddress.sin_family = AF_INET;
  socketaddress.sin_addr.s_addr = INADDR_ANY;
  socketaddress.sin_port = htons(port);

  /* ----- */

  debug("[Server] Versuche bind()\n");

  err=bind(Socket, (struct sockaddr *) &socketaddress, sizeof(socketaddress));

  while (err < 0) {
```

```
    socketaddress.sin_port += 1;
    if (socketaddress.sin_port > 49151) {
      perror("[Server]␣Kann␣bind()␣fuer␣Socket␣ueberhaupt␣nicht␣durchfuehren");
      ende(EX_UNAVAILABLE);
    } /* if */
    err=bind(Socket, (struct sockaddr *) &socketaddress, sizeof(socketaddress));
  } /* while */

  debug("[Server]␣bind()␣erfolgreich␣(rechner␣=␣%s,␣port␣=␣%u)\n", rechner, ntohs(socketaddress.sin_port));

  port_file();

  /* ----- */

  debug("[Server]␣Versuche␣listen()\n");

  err=listen(Socket, 1);

  if (err < 0) {
    perror("[Server]␣listen()␣fehlgeschlagen");
    ende(EX_OSERR);
  }
  else {
    debug("[Server]␣listen()␣erfolgreich␣(err␣=␣%u)\n", err);
  }

  /* ----- */

  debug("[Server]␣Versuche␣accept()\n");

  WorkSocket=accept(Socket, (struct sockaddr *) &socketaddress, &addrlen);

  if (WorkSocket < 0) {
    perror("[Server]␣accept()␣fehlgeschlagen");
    debug("[Server]␣accept()␣fehlgeschlagen␣(errno␣=␣%u)\n", errno);
    ende(EX_OSERR);
  } else {
    debug("[Server]␣accept()␣erfolgreich␣(WorkSocket␣=␣%u)\n", WorkSocket);
  }

  /* ----- */

  debug("[Server]␣Warteschleife␣beginnt\n");

  p = (long *) calloc(count, sizeof(long));
  if (p == NULL) {
    perror("[Server]␣calloc()␣ging␣daneben");
    exit(EX_OSERR);
  }

  stempel_neu = timestamp();
  for (j=0; j<count; j++) {
    i = read(WorkSocket, &eingabe[0], 2);
    if (i > 0) {
      stempel_alt = stempel_neu;
      stempel_neu = timestamp();
      diff = stempel_neu - stempel_alt;
      if (diff < 0) {
```

```
        diff = 1000000 + diff; /* Sekundenumbruch */
      }
      p[j] = diff;
      debug("[Server]␣%lu␣%lu␣%lu\n", stempel_alt, stempel_neu, diff);
    } /* if (i ... */
    eingabe[0] = '\0';
    eingabe[1] = '\0';
  } /* for */

  for (j=0; j<count; j++) {
    log("%lu\n", p[j]);
  }

  debug ("[Server]␣Gebe␣calloc()-Speicher␣frei\n");
  free(p);

  /* ----- */

  ende(EX_OK);

} /* main */
```

A.3 make-gp

```
#!/bin/tcsh -fb

# $Id: make_gp,v 1.10 2003/04/16 10:47:16 uhlf Exp $
# Florian Uhl, 2003

# ---------------------------------------------------------------------------

setenv PATH '' && rehash

set base_dir = '/users/stud/uhlf/poi/stats'

set cat    = '/bin/cat'
set gnuplot = '/usr/bin/gnuplot'
set mkdir  = '/bin/mkdir'
set sed    = '/bin/sed'

set parser  = 'gnuparse.pl'
set mparser = 'median.pl'
set tcpparser = 'tcpparse.pl'

set temp_dir = '/tmp/uhlf'
set temp_file = '/tmp/uhlf/gnuplot.temp'

# ---------------------------------------------------------------------------

  cd ${base_dir} && echo $PWD

  ${mkdir} -p ${temp_dir} || exit;

  cd sender && echo $PWD

  echo "--"
  foreach data ( log_*.dat )
```

```
set png = `echo ${data} | ${sed} -e 's/\.dat/\.png/'`

# Ueberspringen wenn schon vorhanden (Systemlast!)
if ( -f ${png} ) then
  # echo "${png} existiert, ueberspringe ${data} vollstaendig"
  echo "${png}␣existiert"
  echo "--"
  continue
endif

set gp = `echo ${data} | ${sed} -e 's/\.dat/\.gp/'`

# Ueberspringen wenn schon vorhanden (Systemlast!)
if ( -f ${gp} ) then
  # echo "${gp} existiert, ueberspringe Erzeugung"
  echo "${gp}␣existiert"
else
  ${cat} ${data} | ../${parser} > ${gp}
endif

set sender_median = `${cat} ${data} | ../${mparser}`
echo "${data}:␣sender␣hat␣${sender_median}"

set tcp = `echo ${data} | ${sed} -e 's/log/tcp/'`

if ( -f ${tcp} ) then
  set sender_tcp_median = `${cat} ${tcp} | ../${tcpparser}`
  echo "${tcp}:␣sender␣hat␣${sender_tcp_median}"
endif

cd ../empfaenger

# Ueberspringen wenn schon vorhanden (Systemlast!)
if ( -f ${gp} ) then
  # echo "${gp} existiert, ueberspringe Erzeugung"
  echo "${gp}␣existiert"
else
  ${cat} ${data} | ../${parser} > ${gp}
endif

set empfaenger_median = `${cat} ${data} | ../${mparser}`
echo "${data}:␣empfaenger␣hat␣${empfaenger_median}"

if ( -f ${tcp} ) then
  set empfaenger_tcp_median = `${cat} ${tcp} | ../${tcpparser}`
  echo "${tcp}:␣empfaenger␣hat␣${empfaenger_tcp_median}"
endif

cd ../sender

echo "set␣terminal␣png␣color;" > ${temp_file}
echo "set␣output␣'${png}';" >> ${temp_file}
echo "set␣logscale␣y;" >> ${temp_file}

echo -n "plot␣[0:20000]␣[0.0002:1.0000]␣'${gp}'" >> ${temp_file}
echo -n "␣title␣'sender␣poi␣(${sender_median})'" >> ${temp_file}
echo -n "␣with␣histeps␣lt␣1,␣" >> ${temp_file}
```

```

    echo -n "'../empfaenger/${gp}'" >> ${temp_file}
    echo -n "␣title␣'empfaenger␣poi␣(${empfaenger_median})'" >> ${temp_file}
    echo -n "␣with␣histeps␣lt␣2" >> ${temp_file}

    if ( -f ${tcp} && -f ../empfaenger/${tcp} ) then
      echo -n "␣,␣0␣title␣'sender_tcp␣(${sender_tcp_median})'" >> ${temp_file}
      echo -n "␣with␣histeps␣lt␣3" >> ${temp_file}
      echo -n "␣,␣0␣title␣'empfaenger_tcp␣(${empfaenger_tcp_median})'" >> ${temp_file}
      echo "␣with␣histeps␣lt␣4" >> ${temp_file}
    else
      echo "␣" >> ${temp_file}
    endif

    ${gnuplot} < ${temp_file}

    echo "--"

  end # log file loop

# -------
# the end
# -------
```

A.4 median.pl

```
#!/usr/local/mnm/bin/perl -Tw

# $Id: median.pl,v 1.5 2003/04/15 14:00:56 uhlf Exp $
# Florian Uhl, 2003

# Nimmt eine Lise von x inter-frame-delays von STDIN und gibt
# auf STDOUT den Median, 1/4-il und 3/4-il aus.

use strict;
use POSIX;

# ----------------------------------------------------------------------------

my $i       = 0;
my $num_samples = 0; # Anzahl eingelesener Werte

my @values   = ();
my @sortedvalues = ();

my $min_value = 999999999;
my $max_value = 0;

my $median = 0;
my $mi_l   = 0;
my $mi_u   = 0;

my $_14    = 0;
my $_14_l  = 0;
my $_14_u  = 0;

my $_34    = 0;
```

```
my $_34_l  = 0;
my $_34_u  = 0;

my $inter_quantil = 0;
my $left_search = 0;
my $left_whisker = "";
my $right_search = 0;
my $right_whisker = "";

my $ergebnis = "";

# ----------------------------------------------------------------------------

# -- Initialisierung fuer @values und @sortedvalues nicht noetig
#    (werden tatsaechlich von Anfang bis Ende gefuellt)

# -- STDIN zeilenweise bearbeiten ...
# ... und die Daten einfach nacheinander in das Array schreiben
while(<STDIN>) {

  # "trailing newline" entfernen
  chop;

  if ( $_ < ${min_value} ) {
    ${min_value} = $_;
#     print STDERR "Neues Minimum! ($_)\n";
  }

  if ( $_ > ${max_value} ) {
    ${max_value} = $_;
#     print STDERR "Neues Maximum! ($_)\n";
  }

  $values[${num_samples}] = $_;

  ${num_samples} += 1;

}

#  print STDERR "Anzahl eingelesener Werte: ${num_samples}\n";

  if ( (${num_samples}/4.0) != int(${num_samples}/4.0) ) {
    print STDERR "Achtung!␣Berechnung␣nicht␣korrekt,␣da␣\$num_samples␣";
    print STDERR "nicht␣ganzzahlig␣durch␣4␣teilbar.␣Offsets␣fuer␣Viertel-␣";
    print STDERR "und␣Dreiviertel-Quantil␣vermutlich␣falsch.\n";
  }

# ----------------------------------------------------------------------------
# Sortieren ...

  @sortedvalues = sort {$a <=> $b} @values;

# ----------------------------------------------------------------------------
# Median et al.

  # Ich brauche gar keine Fallunterscheidung hier.
  # Bei ungerader Anzahl an Samples fallen $mi_l und $mi_u zusammen.
```

```
  ${mi_l} = floor((${num_samples} - 1) / 2.0);
  ${mi_u} = ceil((${num_samples} - 1) / 2.0);

#  print STDERR "\$mi_l = ${mi_l}\n";
#  print STDERR "\$mi_u = ${mi_u}\n";

  ${median} = ($sortedvalues[${mi_l}]+$sortedvalues[${mi_u}])/2;

#  print STDERR "\$median = ${median}\n";

# ----------------------------------------------------------------------------

# 1/4-il

  ${_14_l} = floor((${num_samples} - 1) / 4.0);
  ${_14_u} = ceil((${num_samples} - 1) / 4.0);

#  print STDERR "\$_14_l = ${_14_l}\n";
#  print STDERR "\$_14_u = ${_14_u}\n";

  ${_14} = ($sortedvalues[${_14_l}]+$sortedvalues[${_14_u}])/2;

#  print STDERR "\$_14 = ${_14}\n";

# ----------------------------------------------------------------------------

# 3/4-il

  ${_34_l} = ${num_samples} - ceil((${num_samples} - 1) / 4.0) - 1;
  ${_34_u} = ${num_samples} - floor((${num_samples} - 1) / 4.0) - 1;

#  print STDERR "\$_34_l = ${_34_l}\n";
#  print STDERR "\$_34_u = ${_34_u}\n";

  ${_34} = ($sortedvalues[${_34_l}]+$sortedvalues[${_34_u}])/2;

#  print STDERR "\$_34 = ${_34}\n";

# ----------------------------------------------------------------------------

  ${inter_quantil} = ${_34} - ${_14};

# ----------------------------------------------------------------------------

# linker (unterer) Whisker

  ${left_search} = ${_14} - 1.5*${inter_quantil};

  if ( ${left_search} <= ${min_value} ) {
    ${left_whisker} = ${min_value};
  } else {
    ${left_whisker} = "??";
    FOR: for ( ${i} = 0; ${i} < ${num_samples}; ${i}++ ) {
      if ( ${left_search} <= $sortedvalues[$i] ) {
        ${left_whisker} = $sortedvalues[$i];
        last FOR;
      } # inner if
    } # for loop
```

```
  } # outer if

# -----------------------------------------------------------------------------

# rechter (oberer) Whisker

  ${right_search} = ${_34} + 1.5*${inter_quantil};
  if ( ${right_search} >= ${max_value} ) {
    ${right_whisker} = ${max_value};
  } else {
    ${right_whisker} = "??";
    FOR: for ( ${i} = 0; ${i} < ${num_samples}; ${i}++ ) {
      if ( $sortedvalues[$i] > ${right_search} ) {
        ${right_whisker} = $sortedvalues[${i}-1];
        last FOR;
      } # inner if
    } # for loop
  }

# -----------------------------------------------------------------------------

  ${ergebnis} = "${min_value}␣|␣${left_whisker}␣${_14}␣${median}␣${_34}␣${right_whisker}␣|␣${max_value}";

  print STDOUT ${ergebnis};

# -- "The End"
exit(0);
```

A.5 gnuparse.pl

```
#!/usr/local/mnm/bin/perl -Tw

# $Id: gnuparse.pl,v 1.3 2003/04/15 15:13:41 uhlf Exp $
# Florian Uhl, 2003

# Nimmt eine Lise von 5000 inter-frame-delays von STDIN und gibt
# auf STDOUT fuer ein mit gnuplot erstelltes Histogramm geeignete
# Daten aus.
# (natuerlich ist der fixe Wert von 5000 nicht das Ziel, sondern eine
#  variable Menge)

use strict;

# -----------------------------------------------------------------------------

my $i;
my $j;

my $lower_x = 0;
my $upper_x = 20000;

my $lower_y = 0;
my $upper_y = 5000;
my $num_samples = 0; # Anzahl eingelesener Werte

my $bucket_size = 50;
my $num_buckets;
```

```

my $bucket;
my @buckets = ();
my @sortedbuckets = ();

my $lower_boundary;

# ----------------------------------------------------------------------------

# -- Zuerst berechnen wir die Anzahl an buckets die benoetigt werden
${num_buckets} = (${upper_x} - ${lower_x}) / ${bucket_size};
# print STDERR "Benoetigen (maximal) ${num_buckets} buckets\n";

# -- Initialisiere das Array @buckets
for ( ${i} = 0; ${i} < ${num_buckets}; ${i}++) {
  $buckets[${i}] = 0;
}

# -- STDIN zeilenweise bearbeiten ...
# ... und die Daten in Buckets "sortieren" (naja, zaehlen)
while(<STDIN>) {

  ${num_samples} += 1;

  # "trailing newline" entfernen
  chop;

  if ( $_ >= ${upper_x} ) {
#     print STDERR "Ausreisser! ($_)\n";
    $bucket = ${num_buckets} - 1; # "Ausreisser" zaehlen wir im letzten
                             # Bucket mit
  } else {
    $bucket = int($_ / $bucket_size); # 'poor man's floor() implementation'
                                # = ganzzahliger Anteil der Division
  }

 if ($buckets[$bucket] == 0) {
#     print STDERR "Neues bucket 'aufgemacht'\n";
  }

  $buckets[$bucket] += 1;

}

#  print STDERR "Anzahl eingelesener Werte: ${num_samples}\n";
  if ( ${num_samples} > ${upper_y} ) {
    print STDERR "Mehr␣Samples␣aus␣Datei␣eingelesen␣(${num_samples})␣als␣erwartet␣(${upper_y})!\n";
  }

# ----------------------------------------------------------------------------
# Ueber alle Buckets iterieren und Werte in gnuplot-geeigneter Form ausgeben

for ( ${i} = 0; ${i} < ${num_buckets}; ${i}++ ) {

  # Untere x-Grenze
  $lower_boundary = ${i} * ${bucket_size};

  # In der Mitte des Buckets einen Punkt setzen
```

```
  ${j} = ${lower_boundary} + (${bucket_size}/2);

  print STDOUT "${j}␣" . $buckets[$i]/${num_samples} . "\n";

}

# ------------------------------------------------------------------------------

# -- "The End"
exit(0);
```

A.6 png2ps

```
#!/bin/tcsh -fb

# ------------------------------------------------------------------------------

setenv PATH '' && rehash

set base_dir='/users/stud/uhlf/uhl03/stats'

set convert='/usr/bin/convert'
set nice='/usr/bin/nice'
set sed='/bin/sed'

# ------------------------------------------------------------------------------

  cd ${base_dir}/sender && echo $PWD
  foreach png ( *.png )

    echo -n ${png}
    set ps=`echo ${png} | ${sed} -e 's/\.png/\.ps/'`
    if !( -f ${ps} ) then
      ${nice} -n 20 ${convert} -rotate 90 ${png} ps:${ps}
      echo "␣"
    else
      echo "␣(skipped)"
    endif

  end
```

A.7 tcpparse.pl

```
#!/usr/local/mnm/bin/perl -Tw

# $Id: tcpparse.pl,v 1.5 2003/04/15 14:18:12 uhlf Exp $
# Florian Uhl, 2003

# Nimmt den output von tcpdump und erzeugt daraus Eingaben fuer gnuplot.

use strict;
use POSIX;

# ------------------------------------------------------------------------------

my $i = 0;
```

```

my $num_records = 0;

my $timestamp = "";
my @timestamps = ();

my @delays= ();
my @sorteddelays = ();

my $alt_s = 0;
my $alt_m = 0;
my $neu_s = 0;
my $neu_m = 0;

my $diff = 0;

my $mindelay = 0;
my $maxdelay = 0;

my $mi_u = 0;
my $mi_l = 0;
my $median = 0;

# ----------------------------------------------------------------------------

# Beispiel:

# 1050330231.242402 10.0.1.1.1108 > 10.0.0.1.4711: S 1038859089:1038859089(0) win 5840 <mss 1460,sackOK,timestamp 173825762 0,nop,wscale 0> (DF)
# 1050330231.243002 10.0.1.1.1108 > 10.0.0.1.4711: . ack 1014268746 win 5840 <nop,nop,timestamp 173825762 173859022> (DF)
# 1050330231.243402 10.0.1.1.1108 > 10.0.0.1.4711: P 0:2(2) ack 1 win 5840 <nop,nop,timestamp 173825762 173859022> (DF)
# (...)
# 1050330231.276345 10.0.1.1.1108 > 10.0.0.1.4711: . 5118:6566(1448) ack 1 win 5840 <nop,nop,timestamp 173825766 173859024> (DF)
# (...)
# 1050330231.502961 10.0.1.1.1108 > 10.0.0.1.4711: F 10000:10000(0) ack 1 win 5840 <nop,nop,timestamp 173825788 173859032> (DF)
# 1050330231.545705 10.0.1.1.1108 > 10.0.0.1.4711: . ack 2 win 5840 <nop,nop,timestamp 173825793 173859052> (DF)

# -- STDIN zeilenweise bearbeiten ...
LOOP: while(<STDIN>) {

  # "trailing newline" entfernen
  chop;

  if ( $_ =~ /^(..........\.......) (.*): (.*)\((.*)\) ack/ ) {

    # ueberspringen wenn 0 Bytes uebertragen wurden (wieso kann das passieren?)
    if ( ${4} == 0 ) {
      next LOOP;
    }

    # gibt die Zeitstempel aus
#    print STDOUT "${1}\n";

    $timestamps[${num_records}] = ${1};
    ${num_records}++;

    # gibt die Menge an uebertragenen Bytes in diesem Rahmen an (z.B. zur Ueberpruefung mit 'bc')
#    print STDERR "${4}\n";
```

```
  }

}

#  print STDERR "${num_records} Zeilen matchen\n";

# -----------------------------------------------------------------------------

# Beispiel:

# 1050330231.502961
# 1050330231.545705

  # print STDERR "Unsortiert:\n";
  ( ${alt_s}, ${alt_m} ) = split(/\./, $timestamps[0], 2);
  for ( ${i}=1; ${i} < ${num_records}; ${i}++ ) {

    ( ${neu_s}, ${neu_m} ) = split(/\./, $timestamps[${i}], 2);
    ${diff} = (${neu_s} - ${alt_s})*1000000 + (${neu_m} - ${alt_m});
    $delays[${i}-1] = ${diff};
    # print STDERR "${diff}\n";
    ${alt_s} = ${neu_s};
    ${alt_m} = ${neu_m};

  }

# -----------------------------------------------------------------------------

  # Jetzt den ganzen Senf sortieren ...
  @sorteddelays = sort {$a <=> $b} @delays;

  # Trivial weil sortiert ...
  ${mindelay} = $sorteddelays[0];
  ${maxdelay} = $sorteddelays[${num_records}-2];

  # print STDERR "Sortiert:\n";
  for ( ${i}=0; ${i} < (${num_records}-1); ${i}++ ) {
    # print STDERR "$sorteddelays[${i}]\n";
  }

  # Bei ungerader Anzahl an Samples fallen $mi_l und $mi_u zusammen.
  ${mi_l} = floor((${num_records} - 2) / 2.0);
  ${mi_u} = ceil((${num_records} - 2) / 2.0);

  # Arithmetisches Mittel ...
  ${median} = ($sorteddelays[${mi_l}]+$sorteddelays[${mi_u}])/2;

  # print STDERR "\$median = ${median}, \$mindelay = ${mindelay}, \$maxdelay = ${maxdelay}\n";
  print STDOUT ${mindelay} . "␣|␣??␣??␣" . ${median} . "␣??␣??␣|␣" . ${maxdelay} . "\n";

# -----------------------------------------------------------------------------

exit(0);

# ------------
# -- "The End"
# ------------
```

A.8 BNF zur Abbildungssprache Eqosul

```
//Syntax von eQoSuL (Stand 6.11.2003)

// Basisdefinitionen

  <LBRACE>                := "{"

  <RBRACE>                := "}"

  <END_STATEMENT>         := ";"

  <NULL>                  := "NULL"

  <UNIT>                  := "bit" "bit/sec" "sec"|"%"|"NULL"

  <QOS_PARAMETER_KEYWORD >   := "QOS_PARAMETER"

  <SAMPLE_KEYWORD>        := "SAMPLE"

  <EVENT_KEYWORD>         := "EVENT"

  <FLOW_KEYWORD>          := "FLOW"

  <INTERFACE_KEYWORD>     := "INTERFACE"

  <USES_KEYWORD>          := "USES"

  <COMPUTES_KEYWORD>      := "COMPUTES"

  <UNIT_KEYWORD>          := "UNIT"

  <AS_KEYWORD>            := "AS"

  <EVENTS_COMPUTED_KEYWORD>   := "COMPUTED␣EVENTS"

  <EVERY_KEYWORD>         := "EVERY"

  <RANDOM_KEYWORD>        := "RANDOM"

  <BETWEEN_KEYWORD>       := "BETWEEN"

  <AND_KEYWORD>           := "AND"

  <INTERFACE_DIR>         := "SEND"|"REC"

  <COUNT_DEF>             := "count"

  <MYMATCH_DEF>           := "myMatch"

  <HISMATCH_DEF>          := "hisMatch"

  <TIME_DEF>              := "time"

  <ID_DEF>                := "id"
```

```
<DISTRIBUTION>          := "NORMALIZED_DISTRIBUTION"
                         | "CULMULATIVE_DISTRIBUTION"


<ALPHA_CONSTANT>        := ("'"(["a"-"z","A"-"Z","0"-"9","_","-"])*"'")


<ARRAY_DEFINITION>      := "["["1"-"9"](["0"-"9"])*"]"
                         | "[]"

<NUM_CONSTANT>          := (( ["0"-"9"] )+)
                         | ("-" ( ["0"-"9"] )+)
                         | ( ( ["0"-"9"] )+ "." ( ["0"-"9"])+)
                         | ( "-"( ["0"-"9"] )+ "." ( ["0"-"9"])+)




<plusOperand>           := "+"

<minusOperand>          := "-"

<multOperand>           := "*"

<divOperand>            := "\"

<QUESTIONMARK>          := "?"


<FUNCTION_ID>           := "left_equal_id"
                         | "right_equal_id"
                         | "count"
                         | "time_dif"

<COMPARE>               := "<"|"<="|"=="|"!="|">="|">"



<CONSTANT_ARRAY>        := "["(<NUM_CONSTANT>|<ALPHA_CONSTANT>)(","(<NUM_CONSTANT>|<ALPHA_CONSTANT>))*"]">



<BIT>                   := "BIT"

<BYTE>                  := "BYTE"

<CHAR>                  := "CHAR"


<PARAMETERS>            := "("<IDENTIFIER>":"(["a"-"z"])*(","<IDENTIFIER>":"(["a"-"z"])*)*")"




<EVAL_FUNCTION>         := "size"
                         | "value"
```

```
 <IDENTIFIER>           := ["a"-"z","A"-"Z"](["a"-"z","A"-"Z","_","0"-"9"])*


 <COMMENT>              := "//" (~["\n"])*"\n"


 <TYPE>                 := <BIT> <ARRAY_DEFINITION>
                         | <BIT>

                         | <BYTE> <ARRAY_DEFINITION>
                         | <BYTE>

                         | <CHAR> <ARRAY_DEFINITION>
                         | <CHAR>


// ein gültiges Statement der Sprache

 <eQoSuL>               := ( <COMMENT>
                            | <Qos_Parameter>
                            | <Sample>
                            | <Event>
                            | <Interface>
                          )*
                          <EOF>


 <Qos_Parameter>        := <QOS_PARAMETER_KEYWORD> <IDENTIFIER> (<COMMENT>)*
                            <LBRACE>
                              "SAMPLE" <NUM_CONSTANT> "TIMES␣FROM" <IDENTIFIER>
                              "TO␣PRODUCE" <DISTRIBUTION> <END_STATEMENT> (<COMMENT>)*
                            <RBRACE>
                          <END_STATEMENT>
```

```
<Sample> := <SAMPLE_KEYWORD> <Sample_Identifier> (<COMMENT>)*

           <LBRACE>

             <USES_KEYWORD>
             (
               <SamplingStrategy> <Event_Identifier>
               <AS_KEYWORD> <Queue_Identifier> <END_STATEMENT>
               (<COMMENT>)*
             )*
             (<COMMENT>)*

             <COMPUTES_KEYWORD> <Term> <UNIT_KEYWORD> (<UNIT>|<NULL>) <END_STATEMENT>
             (<COMMENT>)*

           <RBRACE>

         <END_STATEMENT>


    //zugehörige Teildefinitionen

    <Queue_Identifier> := <IDENTIFIER>
    <Event_Identifier> := <IDENTIFIER>


    <SamplingStrategy> := <EVERY_KEYWORD>
                   | <RANDOM_KEYWORD> "(" <NUM_CONSTANT> (","<NUM_CONSTANT>)*")"







    <Term> := ("(")* //fakultativ eine Klammer
            ( //erstmal muss etwas kommen, was einen Event liefert mit
              // einer Methode hinten dran, der eine Zahl liefert
              (
               <EqualID>
               // jetzt muss eine Methode kommen
               <numGivingEventFunction>
              )

              | //oder eine Zahl selber

              (
               <numConstant>
              )

              | //oder ein Variablenname mit einer Methode hinten dran

              (
               <anId>
               <numGivingQueueFunction>
              )
```

```
              )

              (")")* //fakultativ eine Klammer -
                   //Klammerung wird damit nicht geprüft

              ( //dann ein Operator
                <operand>
                ("(")*
                  <Term>
                (")")*
              )* //beliebig oft



      <operand> := <plusOperand>
              | <minusOperand>
              | <multOperand>
              | <divOperand>


      <anId> := <IDENTIFIER>



      //eine Sammlung aller Funktionen,
      //die auf eine Queue angewandt werden können und
      //eine Zahl zurückgeben

      <numGivingQueueFunction> := "."<COUNT_DEF>
                           //hier können weitere Funktionen
                           //ergänzt werden



      //Sammlng von Functionen, die auf einen Event angewandt
      //werden können und eine Zahl zurückgeben

      <numGivingEventFunction> := <time>
                          | <id>
                           //hier können weitere Funktionen
                           //ergänzt werden
      <time> := "."<TIME_DEF>


      <id> := "."<ID_DEF>



      // sucht erstes Eventpaar mit gleicher ID in Q1 und Q2
      // liefert das Event aus Q1 (myMatch) oder Q2 (hisMatch)
      //Argumente: zwei Queues Q1 und Q2

      <EqualID> := <Queue_Identifier>"." (<MYMATCH_DEF>|<HISMATCH_DEF>)
                "(" <Queue_Identifier> ")"

      <Queue_Identifier> := <IDENTIFIER>
```

```
<Event> := <EVENT_KEYWORD> <Event_Identifier>
          "("
            <Interface_Identifier>"."<Api_Identifier>","
            (
              (
                 <idBuilder>
                 | (
                    //alternative Spezifikation eines Tuples
                    //für die Generierung der id
                    "("
                      <idBuilder> ("," <idBuilder>)*
                    ")"
                   )
              )
              | <NULL>
            )
          ")"
          (<COMMENT>)*


          <LBRACE> (<COMMENT>)*
            [<Filter>] (<COMMENT>)*
            (<COMMENT>)*
          <RBRACE>
        <END_STATEMENT>



    <Filter> := "(" <SingleFilter> ("," <SingleFilter>)* ")"

    <SingleFilter> := (<NUM_CONSTANT>|<ALPHA_CONSTANT>|<IDENTIFIER>)
                 | (<COMPARE>(<NUM_CONSTANT>|<ALPHA_CONSTANT>|<IDENTIFIER>))
                 | (<EVAL_FUNCTION><COMPARE>(<NUM_CONSTANT>|<ALPHA_CONSTANT>|<IDENTIFIER>))
                 | (RegExp())


<Interface_Identifier> := <IDENTIFIER>
<Api_Identifier> := <IDENTIFIER>
<idBuilder> := <IDENTIFIER>









<Interface> := <INTERFACE_KEYWORD> <Interface_Identifier> (<COMMENT>)*
             <LBRACE> (<COMMENT>)*
              (
                <Api_Identifier>
                "("
                   (
                    <Attribute_Identifier> ":" <type> (<COMMENT>)*
                    (
                      "," (<COMMENT>)*
```

```
                        <Attribute_Identifier> ":" <type> (<COMMENT>)*
                      )*
                    )? //kann auch weggelassen werden
                 ")" <END_STATEMENT> (<COMMENT>)*
              )* (<COMMENT>)*
            <RBRACE>
          <END_STATEMENT>


<Attribute_Identifier> := <IDENTIFIER>
```

A.9 Erläuterungen zu eQoSuL

Der nachfolgende Text ist ohne (grössere) inhaltliche Änderungen von M. Garschhammer übernommen. Er dient lediglich als Kurzeinführung dazu, die Konzepte der Sprache Eqosul genauer zu beschreiben. Ich bitte die kaputten Referenzen zu entschuldigen.

Idee: Sprache, die eine Dienstgütemerkmal beschreibt und zugleich als Leitfaden für die implementierung / messung dienen kann, wenn daraus Code (in einer typischen Programmiersprache) generiert wurde.

Die einzelnen Elemente der Sprache orientieren sich direkt am Spezifikationsmodell, wie es in Abschnitt xy eingeführt wurde. Die Hauptelemente von eQoSuL werden im folgenden an Hand von Beispielen vorgestellt. Der folgende Abschnitt **??** gibt eine streng formale Beschreibung von eQoSuL in Form einer BNF [**?**].

eQoSuL bietet die vier Konstrukte `INTERFACE`, `EVENT`, `SAMPLE` und `QOS_PARAMETER`. Die Konstrukte können in beliebiger Reihenfolge und Anzahl verwendet werden. Teilweise beziehen sich die Konstrukte aufeinander, wie in der folgenden Darstellung noch verdeutlicht werden wird. Es müssen jedoch nicht alle Bezüge in einer Spezifikation (einer Datei) aufgelöst werden. Um die Wiederverwendbarkeit bestehender Teildefinitionen zu erleichtern verwendet eQoSuL eine Repository bestehender Definitionen an Hand dessen Querbezuge, sofern möglich, aufgelöst werden. Im Folgenden werden die vier Konstrukte in der typischen Definitionsreihenfolge beispielhaft beschrieben. In den Beispielen werden Bezeichner *kursiv* dargestellt.

INTERFACE In diesem Konstrukt wird ein für die Messung eines Dienstgütemerkmals notwendiges Interfaces beschrieben. Dabei wird die am Interface angebotene Funktionalität durch (mögliche) API-Aufrufe spezifiziert. Ein API-Aufruf kann verschiedene Parameter haben, die zusammen mit dem API-Namen spezifiziert werden. Das Beispiel zeigt die Spezifikation des IP-Interfaces, wie es sich einem Betriebssystembenutzer für gewöhnlich bietet.

```
INTERFACE ip4_to_user {
  // primitiven aus der spec RFC 791 page 32
  send ( src: BYTE[4], // source address
          dst: BYTE[4], // destination address
         prot: BIT[8], // protocol
          TOS: BIT[8], // type of service
          TTL: BIT[8], // time to live
         Data: BYTE[], // Data to send
           Id: BIT[16], // Identifier
           DF: BIT[1], // Fragment
```

```
        opt: BIT[]); // option data

  rec ( src: BYTE, // source address
       send: BYTE[4], // destination address
       prot: BIT[8], // protocol
        TOS: BIT[8], // type of service
       Data: BYTE[], // Data to send
        opt: BIT[]); // option data
};
```

Aus dem Beispiel ist erkennbar, dass von der Sprache typische Konstrukte zur Typisierung von Variablen geboten werden. Variablen können auch als Felder, beliebiger oder fester Länge definiert werden. Die möglichen Variablentypen sind allerdings eingeschränkt, um möglichst vergleichbare Interfacespezifikationen zu erhalten. Eine detaillierte Übersicht über die in eQoSuL möglichen Datentypen liefert der folgenden Abschnitt **??**.

EVENT Nach der Spezifikation eines oder mehrere Interfaces können Events (Ereignisse) festgelegt werden, die an einem Interface auftreten. Dazu wird dem Aufruf einer bestimmten API-Funktion des Interfaces ein Event zugeordnet.

```
EVENT ip_frame_sent (ip4_to_user.send, NULL) {};
```

Im Bespiel wird das Event `ip_frame_sent` ausgelöst, wenn am Interface `ip4_to_user` die Funktion `send` aufgerufen wurde. Zusätlich wird angegeben, ob und auf welche Weise der erzeugte Event mit einem Identifikator versehen werden soll. `NULL` bedeutet dabei, dass auf die Festlegung eines Identifikators verzichtet wird. Zur Berechnung kann jedes Attribut aus dem API-Aufruf verwendet werden, aus dem der Event generiert wurde. Mit der Spezifikation

```
EVENT ip_frame_rec  (ip4_to_user.rec, prot) {};
```

werden dem entsprechende Events erzeugt, die als Identifikator den Protocolidentifikator aus IP verwenden. Dieser wird nicht eins-zu-eins aus der API übernommen sonder durch eine Hash-Funktion berechnet. Damit ist es auch möglich, ein Tupel von API-Attributen zur Bildung des Identifikators anzugeben.

In vielen Fällen ist es notwendig bei der Erzeugung von Events den API-Aufruf auf bestimmte Eigenschaften, z.B. die Länge eines Datenfeldes zu prüfen. Zu diesem Zweck bietet eQoSuL ein Filterstatement (in den bisherigen Beispielen nicht verwendet), in dem Bedingungen für die einzelnen Attribute eines API-Aufrufs in Form eines regulären Ausdrucks angegeben werden können.

```
EVENT short_ip_frame_sent (ip4_to_user.send, NULL)(src: BYTE[4]) {
  // und jetzt ein Muster für die Parameterleiste angeben
  (src,*,*,*,*,size<30,*,*,*)
};
```


```
EVENT short_ip_frame_sent (ip4_to_user.send, NULL)(src: BYTE[4]) {
  // und jetzt ein Muster für die Parameterleiste angeben
  (src,*,*,*,*,size<30,*,*,*)
};
```


Obiges Beispiel zeigt eine Filtermaske, die eine Eventerzeugung nur bei Datagrammen mit einem Payloadfeld zuläßt, das nicht mehr als 30 Bytes enthält.

SAMPLE Im logisch nächsten Schritt (eQoSuL legt keine Reihenfolge der Definitionen fest) wird ein Sample definiert. In dieser Defininition wird die Vorschrift zur Berechnung eines einzigen Wertes eines Dienstgüteparameters festgelegt. Die hier berechneten Werte können dann einer statistischen Weiterverarbeitung (in eQoSuL im Konstrukt QOS_PARAMETER definiert) unterzogen werden.

Nach der Festlegung eines Names für das Sample wird nach dem Schlüsselwort USES bestimmt, welche Events zur Berechnung des Samples verwendet werden. Für die Berechnung von Werten geht eQoSuL dabei von folgender Modellvorstellung aus: Die nach USES angegebenen Events werden in Puffern abgelegt, die durch das Schlüsselwort AS frei benannt werden können.

Nach diesen Festlegungen wird die eigentliche Rechenvorschrift hinter dem Schlüsselwort COMPUTES angegeben und durch die Angabe der Einheit UNIT, in der der berechnete Wert zu interpretieren ist, abgeschlossen. Für die statistische Verarbeitung wird zudem vor dem Schlüsselwort COMPUTED EVENTS angegeben, wieviele Ereignisse in die Berechnung des Wertes eingehen, damit beispielsweise nachträgliche Normierungen durchgeführt werden können.

```
SAMPLE ip_one_way_delay {
  USES
  EVERY ip_frame_sent AS a;
  EVERY ip_frame_rec AS b;
  COMPUTES a.myMatch(b).time - a.hisMatch(b).time UNIT sec;
  1 COMPUTED EVENTS;
};
```

QOS_PARAMETER Mit dem Konstrukt QOS_PARAMETER wird bestimmt, wie ein Dienstgütemerkmal durch die statistische Aggregation von Samples dargestellt wird. Dazu wird nach dem Schlüsselwort SAMPLE angegeben, wieviele Samples welchen Types verarbeitet werden sollen. Weiterhin wird anch dem Schlüsselwort TO PRODUCE festgelgt welche statistische Funktion auf die Samples angewandt werden soll.

```
QOS_PARAMETER ip_owd_distribution {
  SAMPLE 100 TIMES FROM ip_one_way_delay
                   TO PRODUCE NORMALIZED_DISTRIBUTION;
};
```

Abkürzungsverzeichnis

BNF Bacchus-Naur-Form

dc data collector

ec event correlator

FIPA Foundation for Intelligent Physical Agents

ICMP internet control message protocol

IETF Internet Engineering Task Force

IP internet protocol

IPPM IP Performance Metrics Group

ITU-T International Telecommunications Union

microseconds 10^{-6} Sekunden

PCI protocol control information

PDU protocol data unit

poi proof of idea

QoS Quality of Service

RFC request for comments

SDU service data unit

sp statistic processing

ssh secure shell

TCP transmission control protocol

UDP user datagram protocol

Literaturverzeichnis

[Black] BLACK: *OSI - A Model for Computer Communications Standards*, 1991.

[Duden Inform.] BI-WISS.-VERL. (HRSG.), LEKTORAT DES: *Duden Informatik - 2. Auflage*. Dudenverlag Mannheim, Leipzig, Wien, Zuerich, 1993.

[FIPA QoS] INTELLIGENT PHYSICAL AGENTS, FOUNDATION FOR: *Quality of Service Ontology Specification*, December 2002, http://www.fipa.org/specs/fipa00094/SC00094A.pdf .

[Gar 04] GARSCHHAMMER, M.: *Abbildungssprache*, to be published.

[Neu 02] NEU, C.: *Design and implementation of a generic quality of service measurement and monitoring architecture*. Diplomarbeit, LMU, 2002.

[Oes 98] OESTEREICH, B.: *Objektorientierte Softwareentwicklung*. Oldenbourg Verlag, ISBN 3-486-24787-5, 1998.

[RFC 2330] ALMES, MAHDAVI, MATHIS und PAXSON: *RFC 2330: Framework for IP Performance Metrics*, 1998, http://www.ietf.org/rfc/rfc2330.txt .

[RFC 2678] MAHDAVI: *RFC 2678: IPPM Metrics for Measuring Connectivity*, 1999, http://www.ietf.org/rfc/rfc2678.txt .

[RFC 2679] ALMES, KALIDINDI und ZEKAUSKAS: *RFC 2679: A One-way Delay Metric for IPPM*, 1999, http://www.ietf.org/rfc/rfc2679.txt .

[RFC 2680] ALMES, KALIDINDI und ZEKAUSKAS: *RFC 2680: A One-way Packet Loss Metric for IPPM*, 1999, http://www.ietf.org/rfc/rfc2680.txt .

[RFC 2681] ALMES, KALIDINDI und ZEKAUSKAS: *RFC 2681: A Round-trip Delay Metric for IPPM*, 1999, http://www.ietf.org/rfc/rfc2681.txt .

[RFC 3393] DEMICHELIS und CHIMENTO: *IP Packet Delay Variation Metric for IP Performance Metrics*, 2002.

[X.641] ITU-T: *Information Technology - Quality of Service: Framework. ITU-T Recommendation X.641*, 1997.

Index