INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Diplomarbeit

Design and implementation of a generic quality of service measurement and monitoring architecture

Cécile Neu

Aufgabensteller:	Prof. Dr. Heinz-Gerd Hegering
Betreuer:	Markus Garschhammer Bernhard Kempter
Abgabetermin:	01.11.2002

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Diplomarbeit

Design and implementation of a generic quality of service measurement and monitoring architecture

Cécile Neu

Aufgabensteller:	Prof. Dr. Heinz-Gerd Hegering
Betreuer:	Markus Garschhammer Bernhard Kempter
Abgabetermin:	01.11.2002

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

München, den 31.10.2002

. . . .
(Unterschrift des Kandidaten)

Abstract

Durch die Dienstorientiertheit heutiger IT-Infrastrukturen ist ein Bedarf nach vorhersehbarer und berechenbarer Dienstgüte entstanden. Während jedoch die heutigen praktische Ansätze zur Überwachung und Messung von Dienstgüte sehr technologiespezifisch sind, fordert die Heterogenität moderner Netzwerke einen generischen Ansatz. Moderne Dienstgütearchitekturen wie das ISO Quality of Service Framework behandeln die Frage der Dienstgütemessung jedoch nur sehr abstrakt und überlassen die Entscheidung über detaillierte Spezifikation und Implementierung den Benutzergemeinden.

Diese Arbeit stellt eine generische Architektur zur Messung und Überwachung von Dienstgüte vor. Das generische Konzept von Event Streams wird eingeführt, um die Interaktion am Dienstzugangspunkt zu modellieren. Ableitungen einer abstrakten Event-Oberklasse werden dazu verwendet, protokollspezifische Event Streams zu beschreiben.

Auf Grundlage des ISO Quality of Service Framework wird eine vereinigte Sicht auf Dienstgüte erarbeitet. Dabei wird Dienstgüte mit Hilfe von vier generischen Basisdimensionen modelliert. Durch eine geeignete Abbildung dieser Basisdimensionen auf protokollspezifische Messwerte kann diese Architektur an jede Art von Protokoll adaptiert werden.

Die Basisdimensionen werden durch Event-Korrelation bestimmt. Event-Korrelation definiert eine Abbildung zwischen zwei oder mehreren Event Streams auf Grundlage einer wohldefinierten Beziehung zwischen Events - zum Beispiel einer Identitätsbeziehung, wenn zwei Mitschnitte des selben Datenstroms korreliert werden.

Das Framework besteht aus drei konzeptuellen Blöcken: dem Datenkollektor, der einen Datenstrom mitschneidet und daraus einen entsprechenden Event Stream generiert, dem Datenkorrelator, der eine beliebige Anzahl von Event Streams korreliert und dem Datenanalysator, der eine statistische Analyse auf den korrelierten Event Streams durchführt und die QoS Basisdimensionen ausgibt.

Zudem wird das Konzept des Messkontextes eingeführt, um die Umstände zu modellieren und zu erhalten unter denen die Messung und Analyse durchgeführt werden. Kontext kann an jedem Punkt des Mess- und Analysevorganges gemessen werden und wird bis zum Datenanalysator propagiert, um dem Endbenutzer die Kontextinformation zur Verfügung zu stellen. Mittels eines Mechanismus namens Kontext-Backpropagation kann der Endbenutzer wiederum die Datenkollektion oder Korrelation beeinflussen.

Um zu veranschaulichen wie die Konzepte der Architektur in Software umgesetzt werden können, wird eine prototypische Implementierung vorgestellt. Diese Implementierung unterstützt eine repräsentative Auswahl an Event Typen für TCP/IP-Netze und hat ein benutzerfreundliches graphisches Interface.

Contents

List of Figures

Chapter 1

Introduction

In today's connected world, the concept of service quality has gained an importance never seen before. With the emergence of multimedia applications with high demands to network performance and the widespread use of networks and networked applications in business context, a demand for predictable and accountable service quality has arisen.

To this end, service level agreements between service providers and customers specify the service quality level expected by the customer and should be reliably provided. With the shift from network management to service management (see [33]), the handling of quality of service has become one of the foremost topics an integrated management has to face. To ensure the enforcement of SLAs, a detailed specification and later, a constant monitoring of quality of service is needed.

Today, the monitoring of service quality is often done implicitly using heterogeneous methods and gathering the relevant data from a multitude of different sources: traffic statistics can be gathered via SNMP, tools like Netsaint ([1]) give information about service availability etc. Analyzing the collected data and estimating the level of compliancy with the SLAs remains a human activity. The heterogeneity of both the underlying network technologies and the tools used for network management further complicate this process.

Considering the importance of QoS management, it is surprising that no unified QoS measurement architecture has emerged. The approaches so far have been either very abstract, mentioning the problem but leaving the details of the implementation to the end user, or too specific to be of use outside a very narrow range of application. The International Standards Organization QoS standard X.641 ([35]) identifies QoS monitoring as a mechanism of QoS management, but fails to specify a framework for QoS monitoring. On the other hand, the IP performance metric RFCs (RFC2330, [47]) give a very detailed discussion of the implementation of a performance metric for IP networks, but lack the background of a QoS architecture.

This thesis proposes a generic framework for the measurement of quality of service, providing a unified view on QoS and ensuring the comparability of service quality regardless of the technical details of the underlying network. The framework proposal will be concluded by an implementation of this framework for packet-switched networks as a proof-of-concept and in order to verify some of the assumptions made during this work.

Using a unified view on QoS lays the groundwork to developing a generic QoS measurement architecture, as the concepts used are not inherently limited to one networking technology, and thus the heterogeneity of today's networks can be adequately reflected. Furthermore, this raises the point of comparability: can a representation of service quality found such that the results permit not only a qualitative comparison but also a quantitative assessment of the relative performance of different network technologies. Finally, a generic measurement architecture is very adaptive and can easily accommodate emerging new technologies without needing a fundamental revision.

First, an overview of the most common definitions of QoS will be given in chapter 2. The QoS definition used in this thesis will be heavily based on the definition given in the ISO QoS standard, but it is important to see the heterogeneity present in this field to better contextualise this work. A representative choice of works on the subject of QoS measurement and monitoring, both from the academic and the industry world, is then presented and discussed. After this summary of the current state of the art, a list of requirements for the QoS measurement framework is formulated.

Chapter 3 presents the QoS measurement architecture. To achieve a high level of generality but keep the implementation easy, the concept of events is introduced to model network activity. While the processing of events is inherently protocol- and network-specific, the methods used remain generic, hiding the underlying heterogeneity from the architectural framework. The partitioning of the framework into three main aspects - event collection, event correlation and event analysis - further sustains this modular approach. Finally, the rules used in those steps are not hard-coded in the framework, but implementation-specific, preserving the generality of the approach.

After the presentation of the UML implementation model in chapter 4, chapter 5 discusses the technical details of the implementation: what kind of experimental setups can be used with this measurement architecture, what technical and methodical problems are likely to be encountered and how should they be solved.

Chapter 6 presents the final proof-of-concept implementation which can be used to measure the QoS of ICMP and TCP traffic in IP networks and discusses the code in more detail. Finally, chapter **??** shows some experimental results gained with this software.

Chapter 2

Quality of service

In the last years, numerous quality of service frameworks targeting different application fields - from generic QoS architecture proposals to very technical frameworks - have evolved. In this chapter, a broad overview of the state of the art regarding quality of service framework will be given and the respective definitions of QoS, focusing on their relevance for this work given. In the second section, a few representative approaches to the problem of quantifying and measuring quality of service will be presented, and their advantages and drawbacks discussed. Finally, I will attempt to show why a more generic and unified approach to QoS measurement is needed, and list the requirements such an approach has to fulfill. A discussion of QoS frameworks can also be found in [15].

2.1 Definitions

2.1.1 ISO QoS Framework

One of the fundamental works in this area is the QoS framework that was proposed by the International Standards Organization ([35]). This framework, built on the concepts of the Open System Interconnection management framework, defines the architectural principles, concepts and structures underlying the provision of quality of service in OSI networks. It provides a conceptual and functional framework only and is not 1 as an implementation specification.

In this framework, the concept of service is to be understood in a very broad sense as the interaction between entities and the provision of functions by entities, objects and applications. The model considers two classes of entities responsible for the management of QoS: the system QoS entities, which monitor and control the performance of the global system by interacting with specific layer QoS entities; and the layer QoS entities, which implement direct control of protocol entities and are responsible for negotiating service quality with inferior layer and exchanging information between layers.

Quality of service is characterized by a few key concepts. The QoS characteristics designate any aspect of system QoS which can be identified and quantified. This models the actual behavior of a system, not the result of a measurement.

QoS categories are a policy leading to the choice of a particular set of QoS characteristics for the implementation of different QoS requirements targeting different user groups. Examples of QoS categories would be: security, ease of use, flexibility / extensibility etc. The user requirements are expressed as a set of QoS requirements, which can consist of QoS parameters (the requirements between entities) or QoS context (modeling the requirements within one entity). Those requirements are then realized by QoS mechanisms or management functions: for instance, establishing QoS for a set of QoS characteristics or maintaining actual QoS as close as possible to target QoS. While the standard presents a set of QoS characteristics of general importance, it leaves the selection of a suitable subset of characteristics for actual measurement and monitoring of QoS to the user communities. Also, the set of characteristics presented is not exhaustive, and leaves open a wide range of further specialization for many characteristics. From the ISO QoS standard ([35]):

> This QOS framework specifically excludes the detailed specification of QOS mechanisms. It is not the intent ... to serve as an implementation specification, to be a basis for appraising the conformance of implementations, or to define particular services and protocols. Rather, it provides a conceptual and functional framework for QOS.

2.1.2 ITU-TS and ATM QoS

The concept of quality of service used in ATM networks corresponds to the ITU-T (International Telecommunication Union) Recommendation I.350 [34] 1 QoS and network performance in digital networks. Starting from the definition of QoS as the degree of satisfaction experienced by the service user, quality of service is then characterized as service support, service operability and service integrity performance. A clear distinction is made between quality of service and network performance: while the first is a user-oriented service attribute, focusing on user-observable effects, the latter is defined as a provider oriented, connection element attribute, focusing on planing, development, operation and maintenance of a network. More importantly, network performance is defined as an end-to-end characteristic of the network, while quality of service is to be observed between service access points.

Based on this recommendation, the ATM Forum has defined a set of cell transfer performance parameters which serve as a basis for measuring the QoS provided by the network. User quality expectations are matched to four service classes, which have been defined on the basis of three type of requirements: the need for between the sender and the receiver, constant or variable bit rate and connection-oriented or connectionless transfer. The AAL (ATM adaption layer) then matches the service class requirements onto the ATM layer, where service quality is measured using parameters such as cell error ratio, cell loss ratio, cell transfer delay etc. (see [10]).

This QoS framework is quite interesting because it begins with a user-centric concept of service quality and provides a mapping onto network-centric performance parameters. The obvious drawback is that this approach is only valid for ATM networks and was not conceived with a more abstract and generic framework in mind. While the parameter mapping might provide some interesting insights for the implementation part of this thesis, the QoS framework is of no direct concern.

2.1.3 Quality of service in IP networks

Traditionally, traffic in IP networks only had one type of service quality class available: that of point-to-point, best effort delivery. Traffic is processed as quickly as possible, but there are no guarantees as to actual delivery, available bandwidth or timeliness. Even though the IPv4 standard envisioned different IP precedence classes using the TOS (type of service) header field (see [9]), this never found widespread acceptance. In the last years, two new developments in the field of IP QoS - Integrated Services and Differentiated Services - have succeeded into bringing differentiated service classes to IP networks. Both the intserv and the diffserv approach, as well as the IETF QoS manager will now be presented and evaluated.

Integrated services and RSVP

The integrated services model, as specified by the IETF intserv working group in [13], provides mechanisms for resource reservation and admission control. By reserving bandwidth and router resources along the communication path using the RSVP protocol ([12]), applications can request a specific QoS which is then provided to the packet stream. So far, two service quality classes have been defined: controlled load, targeted towards application requiring reliable and enhanced best effort delivery, and guaranteed service for application requiring fixed delay bounds. No framework for the implementation of those service quality classes is provided.

Differentiated services

The differentiated services model, as specified in [45] and [11], uses the TOS header field as DS field and a set of per-hop-behaviors to realize different traffic classes. In IPv6, the corresponding header field used is the traffic class field. Diffserv is significantly different from the intserv model. As the classification, marking and policing of packets is only necessary at the boundary of networks, the ISP core routers need not implement as much functionality as RSVP-enabled routers. Thus, the implementation and deployment of differentiated services is easier. Furthermore, while intserv works on network layer 4, diffserv acts on network layer 3.

IETF QoS Manager

In [20], an early work on an IETF QoS manager for the integrated services protocol suite is presented. The QoS manager is an abstract management layer separating the core QoS semantics from the details of the implementation, thus enabling applications to negotiate QoS requirements without knowing the details of any specific network service. This interesting approach seems not to have been developed any further and thus will not be discussed here.

Discussion

The quality of services concepts used by the integrated services model are very simple and network-centric. From [13]:

> The core service model is concerned almost exclusively with the time-of-delivery of packets. Thus, per-packet delay is the central quantity about which the network makes quality of service commitments.

The emphasis lies on real-time QoS, characterized by end-to-end packet delay and controlled link sharing.

The differentiated services model uses a flexible definition of quality of service (see [11]):

> A "service" defines some significant characteristics of packet transmission in one direction across a set of one or more paths within a network. These characteristics may be specified in quantitative or statistical terms of throughput, delay, jitter, and/or loss, or may otherwise be specified in terms of some relative priority of access to network resources.

The network traffic is then classified according to rules defined in the TCA (traffic conditioning agreement, which "specifies the forwarding service a customer should receive", [30]). Still, both approaches lack the back-up of a larger QoS framework. Both QoS definitions apply only to packet switched networks, and while the diffserv approach is more flexible it still leaves the definition of what can actually be conceived as the QoS of traffic to the TCA.

2.1.4 TINA

The TINA (Telecommunication Information Networking Architecture) consortium, formed by network operators, telecommunication equipment suppliers and computer suppliers was working on the definition of a software architecture to support flexible introduction and integrated management of new services in the telecommunication area. As of 31.12.2000, work on TINA-C has ceased. It is still discussed here as the software architecture proposed features an interesting approach to the description of service quality. A more in-depth description of the TINA-C architecture can be found in [27] and on the TINA-C web site ([2]).

The service concept used here is very broad and covers both the traditional concept of telecommunication services (services provided by network operators etc. to customers) and management services (needed for the operation and administration of telecommunication services or networks).

The TINA-C software architecture aims to be independent of underlying technical aspects of the network as well as interoperable across different network domains. It is divided in four technical areas: Computing Architecture, Management Architecture, Service Architecture and Network Architecture. Quality of service is defined 2 the context of the Computing Architecture, which provides a set of modeling concepts as a basis for the interoperability of telecommunication software.

Using the Object Definition Language, an extension of the Object Management Group Interface Definition Language, quality of service is defined as an object attribute of operations and stream flows to specify non-functional aspects. The goal here is to provide QoS transparency: applications do not have to deal with complex resource management mechanisms, as these are not handled by the application layer.

From [50]:

> The quality of service attributes associated with an operation or a flow address the timing constraints, degree of parallelism, availability guarantees and so forth, to be provided by that component.

The TINA-C architecture only provides the concepts necessary to associate quality of service statements with operations and flows. What it does not provide are the semantics of those specifications. I.e., if the service attribute "bandwidth" is assigned the value "2 Mbits", this does state whether the value should be interpreted as a mandatory capability, a mean value to be expected, or the maximum bandwidth capability of the interface.

2.1.5 Discussion

The relevance of these QoS architectures for this thesis will now be evaluated and discussed. A generic and technology-unspecific QoS definition is essential, as the aim of this thesis is to present a generic measurement architecture for QoS. Still, the QoS definition needs to be flexible enough to allow an easy mapping onto low-level, network-specific characteristics without posing a restriction as to the type of network where it is applicable.

None of the QoS definitions presented so far has provided a unified, high-level QoS concept. Both the ATM QoS and the intserv / diffserv approaches for packet switched networks are highly technology-specific, although the definition of QoS as the degree of user satisfaction presented in the ATM QoS standard maybe comes closest to a unified high-level QoS definition. The TINA-C approach, while quite interesting, will not be considered in this work as development has ceased. Furthermore, while the modeling of service quality as object attributes is a very interesting approach, it is much too specific to be of relevance here. Finally, the ISO QoS standard prepares the groundwork for a unified definition of QoS but leaves the details to the implementation.

Instead of trying to find a unified, high-level QoS definition, the concept of service quality used in this work will be heavily based on the ISO QoS standard and consider only technology-based QoS characteristics as presented in [17]. As will be shown in section 3.3.2, the exact definition of high-level, end-user

QoS is irrelevant for this work. By using a suitable subset of technology-based QoS characteristics, a mapping from those QoS parameters onto high-level QoS is always possible. This will be discussed in more detail in chapter 3 and especially section 3.3.2.

2.2 Measuring and monitoring quality of service

With service orientation gaining more and more importance today, monitoring and thus precise measuring of quality of service has become one of the foremost topics network management has to face. Comparability of network services becomes possible only if a quantitative assessment of the services is offered. Service level agreements between service providers and customers rely on mechanisms for specifying levels of QoS which are to be expected and, later, monitored during operation. Network management needs tools to keep the quality of service provided by a network under surveillance.

Still, no unified approach to the monitoring and especially quantification of quality of service has emerged. With QoS and QoS monitoring being an inherent part of ATM networks, the topic of monitoring quality of service in multimedia networks has not lead to new developments in the field, but provides interesting approaches to the problem of defining a QoS metric, and overcoming the gap between user-centric and network-centric QoS. In the IP world, notable developments have been made by the IETF's Internet Protocol Provider Metrics working group. Finally, a research project on measuring and modeling service quality in packet switched networks is presented.

After a short presentation, these approaches will be rated according to the following criteria:

- Use of passive or active measurement methods? Passive measurement methods are to be preferred, as they do not disrupt or influence normal network operations and thus provide unbiased and more accurate measurements.
- End-to-end or network centric measurement? Due to the nature and scope of this work, only end-to-end approaches are of interest. The network is considered to be a black box, whose internal workings are of no concern. The traffic flows are measured and analyzed only across the protocol or the service interface.
- QoS definition used? A network-independent, high-level QoS definition is to be preferred, as this thesis aim towards a generic, technology-independent measurement interface. Ideally, a flexible mapping of high-level QoS parameters onto network-level QoS parameters should be provided.

2.2.1 QoS in multimedia networks

As a quality of service metric is already present in ATM networks through the definition of cell transfer performance parameters by the ATM Forum (see

[26]), the difficulties posed by measuring network performance in multimedia networks are not those of finding a suitable set of performance parameters, but rather of mapping user expectations to the already present set of measurement parameters. The existing mechanisms for measuring QoS being network-centric, a technique for quantifying user-centric QoS - which is often expressed in very vague terms - and mapping it onto network-centric QoS is needed.

[28] presents a method of QoS mapping between the user's video quality preference and the bandwidth required. User preference, in this case, can be a fast but coarse video transmission, or a slow but clear transmission. These are mapped onto three QoS parameters: spatial resolution, SNR[1] resolution and time resolution using mean opinion score evaluation to quantify the user's perceived video quality. Finally, the bandwidth required to support a given set of QoS parameters is determined by analyzing MPEG2 streams.

[48] holds that service quality should be measured according to its impact on final entities - that is, the applications, their users or the service provider themselves. To this end, the value of a given QoS characteristic is classified in one of 5 zones, denoting gradual degradation or improvement of perceived service quality. The authors also present a prototypical implementation of a QoS monitor.

[37] evaluates the performance of multiplexer service disciplines, which control traffic flows through ATM switches and are important in provisioning QoS. A quantitative comparison of the performance of common service discipline algorithms shows that a performance evaluation based on QoS parameters such as cell loss ratio, end-to-end delay and delay jitter do not adequately reflect user satisfaction. A new performance metric - the unified QoS metric - is presented. It uses a more user-centric approach by comparing delivered QoS to expected QoS.

All of these works are very specific and none presents a generic approach to the problem of measuring quality of service independently of the underlying network. While the question of mapping user-centric QoS onto network-centric QoS is an interesting one, it is never considered outside a given, network-specific QoS framework - that is, the ITU-TS framework in [28] and [37], or the ISO-QOS framework applied to packet switched networks in [48]. No effort is made to provide a generic interface to map high-level QoS concepts onto network-centric QoS, regardless of the type of network involved.

2.2.2 IPPM (IP Performance Metrics) RFC2330

While the 3 as a whole has been standardized since its beginning, efforts in standardizing 3 measurements were late in coming. The necessity for an 3 performance measurement framework led to the creation of the IP provider metrics working group by the IETF (see also [31]). The goals of this working group can be quickly summarized as follows:

[1]Signal to noise ratio

- to define specific metrics and procedures for the accurate measurement of connectivity, delay variations, loss patterns etc.
- to produce a MIB to retrieve results of IPPM metrics from existing network management systems and facilitate the communication of such metrics between existing management entities.

The broader goal, as stated in [46], is to aid capacity planning and troubleshooting in large networks and enable customers to compare the performance of different providers, or to rate the service quality they are experiencing. In a series of RFCs (see RFC2678 [41] through RFC2681 [8]), the concept of metrics is defined more clearly, resulting in a distinction between analytical metrics (defined in terms of abstract, theoretical properties of components) and experimental metrics (directly defined by measurements). Furthermore, propositions are made regarding measurement strategies, measurement infrastructure, error evaluation etc.

While these RFCs provide a solid groundwork for experimental methodology, their relevance is strictly limited to IP networks. As the goal of the IPPM working group is to standardize 3 measurements and not to provide yet another QoS framework, no attempt is made to tie this work into a more generic, less network-specific QoS architecture.

2.2.3 Fasbender

In [24], the author presents an analytical characterization of the service quality of packet switched networks, geared towards the collection of sufficient data for a realistic simulation of a packet switched network. By assessing the network performance on an end-to-end basis, it is possible to gain a description of the characteristics of packet switched networks without considering the inner workings of such a network, or the detailed interactions of application software and transport protocols.

Using a slightly modified version of the *ping* tool for active measurements (that is, probe packets are injected into the network using a random-additive-sampling distribution), round trip times and one way trip time of IP packets are collected and evaluated. The tool is used to survey quality of service of network connections between the RWTH Aachen and a representative choice of network hosts. Thus, network characteristics are described from an end user's point of view.

Although the experimental methodology presented in this work is quite interesting, its focus does not lie primarily on QoS monitoring. The use of an active measurement technique skews the observed results, but the impact on system and network performance is not estimated. The QoS concept used in this work is very vague. Basically, QoS is considered to be sufficiently described by round trip times, one way trip times and packet loss rates. The approach is centered on packet switched networks and there is no effort to tie this view on QoS into a more abstract, high-level QoS architecture. Finally, the emphasis of the work lies more on providing a mathematically accurate description of the observed traffic characteristics than on measuring service quality in the sense of the ISO QoS framework.

2.2.4 Discussion

This section gave a broad overview of different 1 to the question of QoS measurement and monitoring, ranging from more theoretical works concerning the mapping of user-centric QoS onto network-centric parameters in ATM networks to the practical work of the IPPM, and Fasbender. The lack of a generic framework to back up the measurement methodology is common to all those 1.

The multimedia-related works discussed all use very specific definitions of QoS which are not applicable outside the narrow field of multimedia applications in ATM networks. The IPPM framework and the experimental work presented by Fasbender focus both on technical aspects of the question without attempting to tie the approaches into a bigger background of a generic measurement architecture. While inspiration for the practical aspects of implementing such a generic measurement architecture can be gained from those approaches, they clearly do not provide a satisfying outlook on the field of QoS measurement frameworks.

2.3 A new approach?

Why try to design a generic QoS measurement architecture? Section 1 gave a quick overview of the problem. With the increasing complexity and heterogeneity of today's networks, integrated management concepts are sought out to ease network management - but no work has been attempted into the direction of a generic, integrated approach to QoS management and monitoring.

By using a common QoS concept and well-defined interfaces, a generic QoS measurement architecture hides the heterogeneity of the underlying networks and provides the user with a unified interface, thus reducing the complexity of QoS management, and ensuring the comparability of QoS measurements.

2.3.1 Requirements

As a conclusion, the requirements that have emerged for the QoS measurement framework and its implementation will be listed.

The measurement framework should use a generic, high-level QoS definition like the one presented in the ISO QoS standard X.641 ([35]). A suitable mapping onto protocol-specific measurands will have to be specified. The use of a generic QoS definition ensures the generality of the approach and makes it possible to define unified interfaces.

The underlying network heterogeneity should be hidden by using a generic description of traffic. As with the QoS definition, a mapping of this generic model onto more technology-specific elements of network traffic will be needed, but this description allows the definition of generic methods.

Accordingly, the layer of the OSI stack at which traffic was captured, should be hidden by the measurement methods. Using unified methods allows unified views and unified interfaces. Technical details like the layer of a capture or the protocol used in network traffic are relegated to the implementation.

The implementation should follow the paradigm of unification proposed by the measurement framework. This means the implementation should be as modular as possible, enabling users to add new functionality (i.e. the support of new types of protocols or the evaluation of other types of service quality) easily and quickly, without reimplementing the bulk of the software. The computation should be done generically with reusable methods, keeping the logic of the analysis in exchangeable rule sets. Finally, the impact of the measurement process on the system should be kept as low as possible to avoid falsifying the results. A quantification of the error level introduced is also desirable.

Chapter 3

Architectural view

As seen in chapter 2, the field of QoS frameworks and QoS measurement methodology is very heterogenous. There is no unified definition of what high-level, user-centric QoS is, and while the subject of QoS measurement has received much attention in the industry as well as in academia, the work presented remains very technology-specific. There does not seem to be any active effort to develop a unified architecture for the measurement of quality of service.

The goal of this thesis is to present a framework for the measurement of quality of service that provides a unified view on QoS: by using an abstract and generic definition of quality of service and providing a non-ambiguous mapping onto network-technology and layer-specific QoS parameters, the underlying heterogeneity can be hidden. Section 2.3.1 listed the requirements that have been identified for this QoS measurement architecture. After an outline of the basic idea, the architectural framework will be presented in section 3.1. The elements of the architectural framework are discussed in sections 3.5.1 through 3.5.3.

High-level QoS parameters like "available bandwidth" or "reliability" can be broken down into several different QoS parameters depending on the layer at which the service quality is considered, and depending on the type of network and application that is considered. While "reliability" might be the complete transfer of a HTML page when considering HTTP over a TCP/IP network at layer seven, it could be the lossless transmission of a live video stream over an ATM network, or the frame loss that occurs over a wireless LAN (IEEE 802.11b) connection. The same high-level QoS parameter comes to describe very different situations when broken down at network level. Still, there remains a certain comparability: a lossy video transmission, a bad WLAN-connection, an interrupted web page transfer all appear as insufficient service quality to the end user and can indeed be traced back to similar causes, a loss of data at a certain point and layer in the network.

But how can those different service qualities be measured without having to implement a totally new measurement framework each time? The differences outlined might seem huge, but a more careful look onto measurement techniques shows similarities between all approaches. Basically, traffic is intercepted or

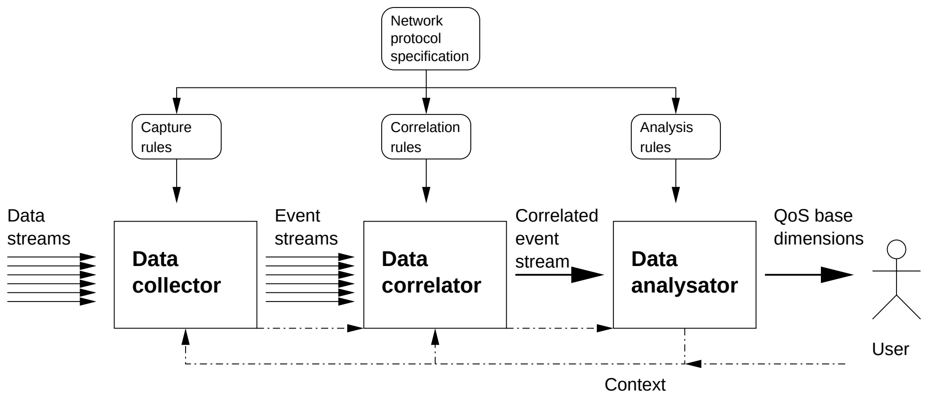


Figure 3.1: The measurement architecture

captured at one or many points in the network, and 1 in order to obtain the service quality information needed. The mechanisms of the analysis will often be very similar: bandwidth usage measurements involve some kind of counting, be it cells, packets or frames; reliability can be defined in term of loss factors etc. If a suitable and generic representation of network traffic can be found, and unified interfaces to the data gathering and analysis processes defined, then a generic QoS measurement architecture can be built, keeping the framework abstract and generic, with an implementation than is easy to customize to new protocols.

3.1 Overview

The QoS measurement architecture proposed consists of three conceptual blocks: the data collector, which captures a data stream at the service access point or between the layers of the protocol stack, performs some data preprocessing and presents the collected data in a suitable representation as event stream; the data correlator, which correlates the event streams provided by the collector, and finally the data analysator, which performs a statistical analysis on the correlated event streams, and presents the QoS parameters (which will be called QoS base dimensions) computed at a common interface. This QoS information could then for example be used to provide an end user with a suitable high-level QoS representation.

To further refine the process of extracting QoS information from the data stream captured, contextual information gained from the data stream and, later, the correlation, is propagated to the data analysator. In reverse, contextual requirements given by the user are propagated back to the data collector and correlator. This gives the end-user the option to influence the accuracy, freshness, reliability etc. of the QoS information he receives.

Since the architecture is generic, detailed information about the inner workings of network protocols is not part of the architectural framework itself, but part of the implementation. Figure 3.1 shows how protocol information can be made available to the three blocks by the use of suitable description rules. In the implementation, this could mean using a suitable symbolic language to express data capture or correlation rules, or simply having the protocol-specific parts of the software encapsulated in modules, so that the support of a new protocol can easily be added without touching the core software.

Why the subdivision in these three conceptual blocks? This architectural decisions reflect the logical flow of information through the measurement process:

1. collecting a number of data streams (that is, capturing interaction at the SAP)
2. parsing the data streams to extract relevant information and produce an event stream (a more abstract and unified representation of the data stream captured)
3. performing a correlation on the event streams in order to be able to 2 the level of similarity or difference later on (in the example discussed earlier, to be able to number the packet loss of a TCP/IP network, one must correlate the sent data with the received data so that missing packets can be identified through their sequence number)
4. finally, performing a statistical analysis of the event stream to compute QoS data and providing this information at a unified interface

The data collector corresponds to point 1 and 2, the data correlator to point 3 and finally, the data analysator corresponds to point 4. As will be seen in 3.5.1, the data collector itself is further subdivided into a recorder (corresponding to point 1) and an event generator (point 2). Since capturing network traffic and generating event streams are very closely related operations, and since the data capture offers little complexity by itself, they are modeled as one conceptual block.

This modular approach also opens up the possibility to implement the architecture in a distributed fashion: i.e., having n collector processes on m networked machines and one central correlator / analysator. This reduces the system load on the collector machines - which is a benefit since minimizing impact on system performance has been phrased as a requirement in section 2.3.1 - while giving the user the benefit of a centralized data evaluation. The measurement process can also be visualized as a pipeline, as shown in figure 3.2, with arbitrary timing delays between the steps. While one user might request the QoS data to be provided in real-time (or as close as possible), another one might prefer to gather data at one time and perform the correlation and analysis process a few hours or days later. With a suitable approach to serializing the event stream produced by the collector, such a delayed data evaluation is possible.

The three interfaces, namely between the collector and the correlator, between the correlator and the analysator, and between the analysator and the end user, will now be discussed.

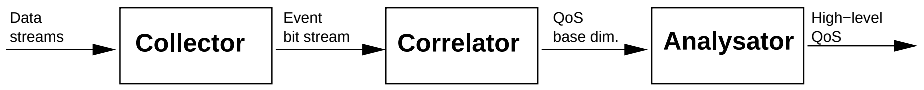


Figure 3.2: The measurement framework as pipeline

The collector provides the correlator with a stream of events. Although these events are derived from an abstract event class, they are inherently protocol-specific. The interface is generic, but the specific event classes and the event stream format are protocol-specific.

The correlator provides the analysator with a correlated event stream: the event streams generated by the collector are 1 in order to identify matching events. The correlation process is discussed in section 3.5.2. The correlated event stream provided by the correlator is simply the set of event streams that was generated by the collector plus the event mapping determined by the correlator.

Finally, the analysator has a unified interface, using a form of QoS representation based on the ISO QoS standard ([35]). Service quality is characterized through the usage of a small set of QoS base dimensions, which are protocol-independent but can be conveniently mapped onto protocol-specific measurands. Thus, the representation of QoS used at the interface between analysator and end user is always the same, regardless of the underlying network technology. The architecture could be extended to incorporate yet another level of correlation and analysis between the analysator and the end user in order to give the end user a more user friendly representation of service quality. This will be discussed briefly in 3.5.4, but since there is no common and generic definition of what exactly end user QoS is, there will be no in-depth consideration of the question in this thesis.

After a presentation of the data stream approach in section 3.2.1, the concept of events will be discussed in section 3.2. Events are used to model the captured network traffic. Section 3.3 explains the representation of quality of service through QoS base dimensions used to provide unified QoS information at a common interface, and the concept of data context, which is used to weight the fidelity of the collected data. Finally, the collector, correlator and analysator will be presented in a more in-depths discussion of their functionality in sections 3.5.1 through 3.5.3.

3.2 Events - a unified description of network activity

The input interface of the correlator needs a unified data structure usable for the common representation of SAP interaction captures as diverse as streams of IP datagrams, a HTTP transaction or multimedia traffic over ATM. The data captured by the collector is very protocol-specific and contains a lot of superfluous information which is of no use to the analysator. The collector is not only responsible for capturing the data stream of network communication, but also for providing the correlator with a refined view on that data stream.

3.2.1 The data stream approach

The concept of data stream used here is presented in [29]. The type of communication considered here is a stream of data between two edge systems in a communication network - that is, a system connected to the edge of a network - or a stream of data inside a system, at different levels of the protocol stack. What happens between the communication endpoints is of no interest: the network, or the stack layers through which the data stream flows, are considered as a black box. The properties of the data stream are dependent both of the layer and the protocol. At the lowest layer, a data stream is a stream of electromagnetic impulses, at layer two it is a stream of, for example, Ethernet frames; at layer 3 we see IP packets or, when considering ATM traffic, cells, byte streams or frames, depending on the AAL chosen.

3.2.2 Events

Before the concept of events in the context of the QoS measurement architecture is discussed, a quick overview of the OSI reference model is used to clarify the terminology used.

The OSI reference model is used to model a layered network architecture. Each of the seven layers defines a different level of abstraction, performing a well-defined function with clear layer boundaries so that information flow across the interfaces is minimized, and functionality is clearly encapsulated inside the layer. According to [33], three interfaces can be identified in the OSI model: the system interface, which defines the boundaries between transit systems and end systems; the service interface, which describes the communication between layers in functional terms and identifies the service primitives used; and finally the protocol interface, which defines the instances which 2 a service defined by the service interface across the systems given by the system interface. Figure 3.3 details the three interfaces.

The data collectors operate either across the service interface or across the protocol interface, as seen in 3.1. In the first case, this means the data collector is used to observe the interaction of different layers of the protocol stack on one machine, thus needing a specially instrumented protocol stack.

In the second case, the measurement is performed directly at the service access point, and only a specific layer is considered. A typical example of that type of measurement is the use of the *tcpdump* ([3]) utility for capturing network traffic. The latter measurement is not performed across the system interface, as the focus lays on the interaction of specific protocols across a network.

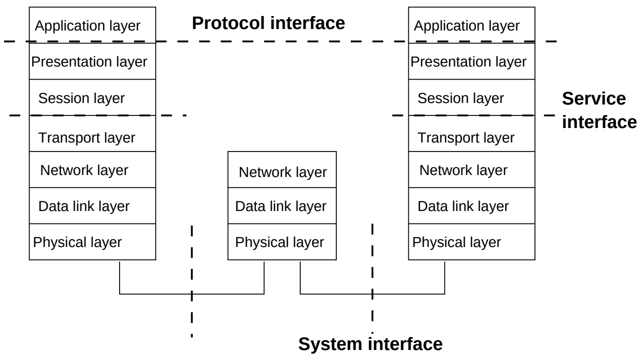


Figure 3.3: The OSI protocol stack

It is quite straightforward to associate an event type with every set of service primitives used to describe a specific protocol when measuring across the service interface or the protocol interface. Thus, measuring across the service interface means identifying and capturing the SDUs[1] encountered, while measuring across the protocol interface means identifying and capturing the PDUs[2].

All those events can be derived from a common abstract event class, as outlined in figure 3.4. They all share the following attributes:

- a timestamp, to indicate the time at which the event was collected
- a unique ID identifying the event

The ID is necessary because the timestamp can not serve for IDing. There are two reasons for that: in a distributed environment, where clocks are not necessarly synchronized, simultaneous events could have different timestamps; on the other hand, simultaneous events with similar timestamp would become indistinguishable without ID. The ID can be computed, it can be a simple count, or it can be extracted from the PDU itself. In a TCP event, the TCP sequence number could serve as an ID.

A special type of event, the timer event, is also introduced. This event consists only of a timestamp and a unique ID, and is used to have a reference time line for the collector. By having the collector generate a timer event at fixed intervals, it is possible for the correlator to detect how old the events of a non-timer event stream are without needing synchronized clocks.

Furthermore, non-timer events have also the following attributes:

- the name and type of the service primitive responsible for the generation of this event
- parameters passed along with the service primitive

[1]Service Data Unit
[2]Protocol Data Unit

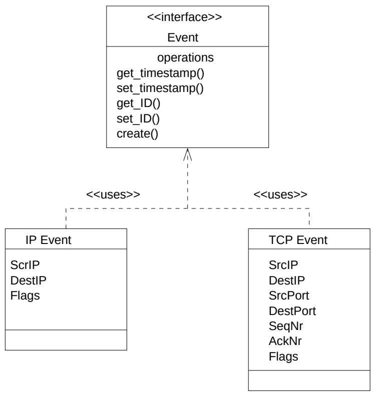


Figure 3.4: Event classes

So, what can intuitively be conceived as an event? It could be anything from the arrival of an IP datagram (C.DATA.indication) over the sending of an HTTP get request (A.ASSOCIATE.request) to the arrival of a video frame in the player software. If the event is generated at a layer where present procotol mechanisms already provide a unique ID, the matter of identifying the event is easily solved be 4 the ID. For example, if the event stream is generated only on layer 4, and the data stream intercepted is a TCP stream, then a unique ID for each event can be easily generated using TCP sequence numbers.

The layer at which events are generated depends heavily on the viewpoint one wants to take to assess the quality of a service. Consider for example a user surfing the world wide web. If the focus of the analysis is to be on the quality of service of the network, the data stream would best be captured at layer 3 and 4 of the OSI stack, using a TCP/IP event type. On the other hand, if the quality of the HTTP service is the main point of interest, one would chose a HTTP event type, analyzing the data at level 7.

This event approach provides a unified view on data streams and thus, activities at the SAP. It is easily extensible: adding an event type for a new protocol simply means another implementation of the abstract event class. Having such a unified view on data streams means that further processing of the events can be done using unified methods, thus fulfilling the requirement of having a generic framework.

3.3 Modeling quality of service

Quality of service is, as discussed in chapter 2, a very vague and high-level concept. While user satisfaction might be a desirable definition of service quality, this is not a concept which can be fed ”as is” into a measurement architecture. User-oriented quality of service is often defined with very vague terms, and the subjective perception of a service's quality can greatly differ from person to person. Translating this high-level representation onto a more technical view of QoS serves two goals: having a more technical view of QoS makes it easier to find a suitable mapping onto measurable network parameters; and this mapping establishes the QoS parameters as a unified interface hiding underlying network heterogeneity. This set of QoS parameters will be called QoS base dimensions.

3.3.1 Requirements

While the mapping of QoS parameters into network parameters is often left to the implementation, here a careful modeling of QoS parameters is required in order to keep the architecture generic. The following requirements have been identified as 2 necessary to obtain a suitable QoS model, and will be discussed in more detail:

- A non-ambiguous mapping onto protocol-specific measurands is needed.
- QoS should be network-centric rather than user-centric, in the sense that the focus lies on technological aspects of service quality.
- Measurands should be quantifiable.
- The completeness of this representation needs to be discussed (that is, whether all aspects of service quality can be covered by the QoS representation used here)

Non-ambiguous mapping

The mapping of high-level QoS characteristics onto lower-level QoS characteristics is discussed in [21] and [14]. [21] proposes a QoS architecture providing a unified framework in which user level QoS (e.g. picture quality, sound fidelity) is mapped down onto lower level QoS. [14] states:

> QoS mapping performs the function of automatic translation between representations of QoS at different system levels (i.e., operating system, transport layer, network) and thus relieves the user from the necessity of thinking in terms of lower level specifications. For example, the transport level QoS specification may express flow requirements in terms of QoS commitment, average and peak bandwidth, jitter, loss and delay constraints - all related to transport packets.

Why is the non-ambiguity of the mapping important? One of the goals of this measurement architecture is to ensure a certain comparability between the service qualities measured, without having to consider the (hidden) details of the network. If a mapping between QoS parameters and measurands is done at random, the QoS representation might possibly not be generic anymore, as the same QoS parameters are mapped onto measurands which are too different to be comparable, or which are even totally unrelated.

For example, consider "error rate" as a possible QoS parameter. This could be mapped onto the packet fragmentation rate in TCP/IP networks and the cell loss rates in ATM networks. While each mapping seems valid for itself, they introduce a big semantic difference in the interpretation of "error rate", depending on the underlying network. The data loss in an ATM network and the conservation of the data stream structure in a TCP/IP network are two 1 different things, while cell loss and packet loss are somewhat comparable. Had packet loss rate been chosen as the gauge for the QoS parameter "error rate" in IP networks, both measurands would have stayed comparable.

This problem is inherent to the concept of mapping QoS parameters onto measurands and thus not easily avoidable. A careful mapping will eliminate most problems, but non-ambiguousness cannot be guaranteed.

Finally, a careful mapping is central to having a generic QoS measurement architecture, as it is used to hide the protocol-specific details of the measurands behind the unified interface of the set of QoS base dimensions. Also, it is this mapping which ensures the comparability of the QoS information gained, regardless of the underlying network technology.

Network-centric QoS

The next requirement was the network-centricity of the QoS representation chosen. The emphasis lies on network-centric aspects of QoS because user-centric aspects are difficult to grasp and quantify. Consumer satisfaction is subjective and not easily measured. On the other hand, bandwidth, throughput, network congestion etc. can be measured and monitored and give a good indication of the current QoS.

Quantifiable measurands

This aspect is also found in the requirement of quantifiable measurands. While "equipment manufacturer" might be a very important aspect of service quality for a specific user having had a bad experience with the reliability of certain manufacturer's equipment, and while this is certainly a technical and low-level QoS parameter, it is not quantifiable and thus unsuitable for further statistical analysis. This QoS measurement architecture focuses on statistical analysis, so that quantifiable measurands are needed.

Completeness

Finally, completeness means the QoS base dimensions should cover enough aspects of high-level QoS (in the sense of "user satisfaction") to be considered an adequate modeling. It is not within the scope of this work to present a proof of the completeness. Instead, I will argument that the representation chosen is a suitable approximation and can be used in this context.

After a suitable set of QoS base dimensions has been chosen, a non-ambiguous mapping of those base dimensions onto protocol-specific measurands will be presented, and the validity and completeness of this approach discussed.

3.3.2 QoS base dimensions

As stated in [14]:

> In characterizing the QoS of activities, it is necessary to identify dimensions along which QoS can be measured and quantified.

The QoS base dimensions with their mapping onto protocol-specific measurands are characteristics describing service quality; that is, they are the dimensions along which service quality can be quantified. The ISO QoS standard X.641 ([35]) lists QoS characteristics "of general importance to communications and processing" and classifies them in eight characteristic groups:

- time-related
- coherence
- capacity-related
- integrity-related
- safety-related
- security-related
- reliability-related
- other

Security- and safety-related QoS characteristics will not be considered here. It is difficult to find quantifiable measurands for those characteristics, so their monitoring is not a task this QoS measurement architecture is suited for. For example, the QoS characteristic "security" might be an agreement between a service provider and a customer to make use of certain technologies in order to assure a secure data transfer; for example to use only virtual private networks with IPSec encryption and to avoid any traffic passing unencrypted over the public 3. There is no level of compliance to be measured here - either the service provider honors the customer's request, or he does not, and is in breach of contract.

Coherence denotes the completition of certain tasks (data production, data transmission, data consumption) within a given time interval and can thus be considered as a special case of timeliness. A further discussion of this topic can be found in [25]. The category "other" only covers precedence, "the relative importance of an object, or the urgency assigned to an event", and will thus not be considered further, as this type of QoS is seldom encountered and can adequately be described using time- and capacity-related characteristics.

[17] lists three categories in the overview of technology-based QoS characteristics:

- Timeliness
- Bandwidth
- Reliability

This fits quite well with the remaining four categories from [35]: time-related, capacity-related, integrity-related and reliability-related characteristics. I thus identify the following four QoS base dimensions:

- Reliability - the reliability of the service offered by the layer at which the event stream is captured. This can be the residual error rate, the resilience for connection-oriented services or the service availability on layer seven.
- Capacity - throughput or bandwidth between the end systems
- Integrity - how much are the data / datagrams affected by the transport (ignoring errors, that is considering only syntactic changes bearing no semantic significance)
- Timeliness - timing characteristics

In order to ease the mapping onto protocol-specific measurands, table 3.1 details the specific meaning of those QoS base dimension at different layers of the OSI stack. The session and presentation layers are not considered, as they are of little importance outside the OSI model.

	Reliability	**Capacity**	**Integrity**	**Timeliness**
Level 7	Service availability	Application level data rate	Data is untampered	Response time
Level 4	Residual error rate or resilience	Throughput	Sequencing errors	Msg transmission delay
Level 3	Residual error rate	Throughput (pps)	Fragmentation	Msg transmission delay
Level 2	Resilience	Throughput (kbps)	Transmission error	Frame transmission delay

Table 3.1: QoS base dimensions

Tables 3.2, 3.3 and 3.4 detail three exemplary mappings of the QoS base dimensions onto protocol-specific measurands, for a HTTP connection over a TCP/IP network, video / audio streaming with RTP over a UDP/IP network, and ATM network. The detailed choice of the measurands is left to the implementation, but should stay consistent with the approach proposed here. The measurands listed in 3.2 will be used for the prototypical implementation in this thesis.

It should be noted that the ATM example is not very satisfactory, but, to quote [49]:

	Reliability	**Capacity**	**Integrity**	**Timeliness**
HTTP	Availability of WWW-Server	Application level data rate	Web page is complete	Server response time
TCP	Packet loss	Throughput	Sequencing errors	–
IP	Packet loss	Throughput	Packet fragmentation	One-way delay

Table 3.2: Mapping of QoS base dimensions onto a HTTP connection over a TCP/IP network

	Reliability	**Capacity**	**Integrity**	**Timeliness**
Video conference (7)	Frame loss	Frames per second	Frame errors	Coherence
ATM adaption layer (4)	Cell loss	Service bit rate	Misinserted cells	–
ATM layer (2-3)	VC resilience	VC bit rate	n.a.	Delay

Table 3.3: Mapping of QoS base dimensions onto ATM measurands

The layers of the ATM model do not map onto the OSI layers especially well, which leads to ambiguities.

3.3.3 Discussion

The set of QoS base dimensions chosen 2 heavily based on the QoS categories discussed in the ISO QoS standard X.641, it should be fairly complete and cover most relevant aspects of service quality. Notable exceptions here are safety- and security-related QoS characteristics, which need other monitoring mechanisms than those discussed within the scope of this QoS measurement architecture. The QoS base dimensions chosen provide a fairly complete cover-

	Reliability	**Capacity**	**Integrity**	**Timeliness**
Video application	Video outage	fps	Frame loss?	Smooth video image etc.
RTP	Packet loss	session bandwidth	Sequencing (with timestamps, seq. nr)	–
UDP	–	Throughput	–	One-way delay
IP	Packet loss	Throughput	Packet fragmentation	One-way delay

Table 3.4: Mapping of QoS base dimensions onto RTP over UDP

age of service quality, and can safely be used as a base to derivate user-level QoS.

Finally, the QoS representation chosen here fulfills the requirements that were outlined in 3.3.1. The QoS base dimensions are obviously network-centric rather than user-centric, in the sense that they relate directly to network-centric measurands. A non-ambiguous mapping onto network-level measurands is possible, and an exemplary mapping has been proposed. The base dimensions are easily mappable onto quantifiable measurands; there are no abstract dimensions like "user satisfaction" which would have to be discussed and further refined.

The QoS representation using a set of four QoS base dimensions based on the QoS categories of the ISO QoS standard X.641 provides a unified representation of service quality, and thus ensures the comparability of measurement results. This approach was chosen to meet the requirement of having a generic measurement architecture with unified methods and unified views, as using QoS base dimensions makes it easy to define generic interfaces.

3.4 Context of the measurement process

To ensure the usability of the collected data, the measurement process should be accurate and lossless. As this goal is not reachable due to inherent limitations of the technology, it becomes necessary to provide mechanisms to control and quantify the error level introduced by the measurement process itself, and to record the conditions under which the event capture was realized. Furthermore, the analysis of the event stream in the correlator and analysator might also introduce an error factor and thus falsify the data. This information shall be call context:

> The interrelated conditions in which something exists or occurs. ([43])

This is a problem commonly encountered with any kind of measurement. For example, [7], which details a one-way delay metric for IPPM, states:

> The calibration and context in which the metric is measured MUST be carefully considered, and SHOULD always be reported along with metric results. [...] Any additional information that could be useful in interpreting applications of the metrics should also be reported.

Many different factors can contribute to errors during the measurement process and have to be identified accordingly. Also, information implicitly present in the data stream can get lost during measurement, for example:

- If the incoming data rate is higher than the rate at which the data can be processed, the event stream produced will not reflect the nature of the incoming data stream, as part of the data stream are simply not considered.

- If the sampling interval is very coarse, the arrival distribution of the data stream gets lost. This can mean a loss of important QoS information. I.e., if the data stream is sampled only every 15 minute, a temporary service outage of less than 15 minutes will not be recorded.
- If not enough data points are collected, the correlator might have to interpolate and thus make assumptions about the data stream that are not necessarily correct.

The goal is to preserve this implicit information and to make it available to the data analysator and, accordingly, to the end user. It should also be possible for the data analysator to specify error level requirements, boundary conditions for the data collection process etc. to the collector and correlator, so that both can adapt the measurement and analysis process.

3.4.1 The measurement context

To obtain a representation of context sufficient for the implementation, a bottom-up approach was chosen to identify the most important aspects of the measurement process which count as context. A more thorough discussion of context should, of course, feature a top-down analysis of the subject.

The contextual information can be roughly classified into two categories: context of the measurement itself, and error context. Error context gives information about the reliability of the measurement process, while measurement context designates contextual information about the event stream and the conditions under which the data was captured.

Furthermore, it is important to carefully discriminate between the distinct contextes at different stages of the measurement process. Most of the measurement context has to be captured in the collector, as this is where the measurement process happens. Once generated, the event streams remain rather static, as the correlator and analysator operate on the event streams without changing them. The error context, on the other hand, is strongly present in the correlator and analysator, as this is where the analytical part of the measurement process happen and interpolation, rounding etc. might introduce a certain error. Furthermore, the error context is cumulative and can only be fully evaluated in the analysator. An error factor previously introduced by the correlator will still be present in the analysator and thus has to be considered there too.

After identifying measurands that constitute measurement and error context, their occurrence in the measurement process will be discussed. The following context information can be denoted as measurement context:

- The sampling scheme used (systematic, stratified random, simple random etc.) and details of its implementation (i.e. for systematic sampling, where every *kth* sample is 2 processed: how big is k?) For a discussion of sampling schemes and their application to network traffic characterization, see also [19].

- The buffer size used in the correlator and analysator to correlate events and process the correlated event stream - this determines how far back into the past events are 2 considered, and how much averaging is 2 done. This, as well as the sampling buffer size, is relevant as the processes considered are not time-homogeneous. For example, using a buffer size of 24 hours to measure bandwidth usage will hide the usage-peaks that typically occur around noon and in the afternoon.
- Freshness - the time gap between data capture and result output. Of course, this is only relevant when doing analysis in real-time. When using a pre-recorded event-stream, freshness is irrelevant. A low freshness here means the time gap was small, a high freshness means the time gap was large.

Error context comprises the following:

- The influence of the measurement process on the system performance
- The reliability of the measurement (i.e., is a significant data pool used? does the analysis process interpolate? how much approximation is performed?)

This list is, of course, not exhaustive, but it covers the main aspects of measurement context. Context as it is considered here is in part event-stream-specific[3], and in part event-specific. Event-stream-specific means that some context information is valid for the whole stream, i.e. the sampling scheme used. Discussing the sampling scheme for one single event makes little sense, as the sampling scheme becomes important only when considering a set of events. Also, a sampling scheme switch affects the whole event stream, and it cannot be changed for each event, as this would not make much sense. Freshness, on the other hand, is a typical example of event-specific context. Each event has a certain freshness. If the collector process happened to be swapped out when interaction at the SAP called for event generation, then the freshness value of the next event will be high. Freshness can be individually changed. For example, the correlator process might request a lower freshness for each event generated by the collector until the lowest possible freshness has been reached.

3.4.2 Backpropagation of the data context

Clearly there is a strong interdependency of the different contextes at different levels of the measurement process. For example, a high error rate during the data collection process will be difficult to compensate later in the correlator or analysator. The time taken by the collector and correlator to process raw data into an event stream will influence the freshness of the QoS data provided by the analysator.

This dependency also exists in the other direction: to ensure a higher freshness of the QoS data, it is necessary to lower the time taken by the data

[3]As will be shown in chapter 5, context as considered in the prototypical implementation is, due to technical limitations, collector-specific

collection and correlation process - possibly by choosing a slightly bigger sampling interval, thus reducing the system load and enabling higher throughput. By giving the user appropriate control of the context, the behavior of the measurement software can be adapted to the user's need: quicker response or more accurate data, a broad overview of network behavior for the last hours or very fine-grained measurement etc. This property of the measurement architecture is named "context backpropagation".

3.4.3 Conclusion

With the concept of contextual information, it is possible to capture the conditions under which a measurement occurred, and use it during the event stream processing. Two types of context where identified: measurement context, which covers details of the measurement process itself, such as event freshness and sampling scheme; and error context, which covers the errors introduced by the correlation and analysis process. Finally, the concept of context backpropagation forms a feed-back control loop to balance the user's requirement with the measurement's efficiency.

3.5 Architectural details

In this section, the details of the three main points of the architectural framework, the collector, correlator and analysator, will be discussed.

3.5.1 Data collector

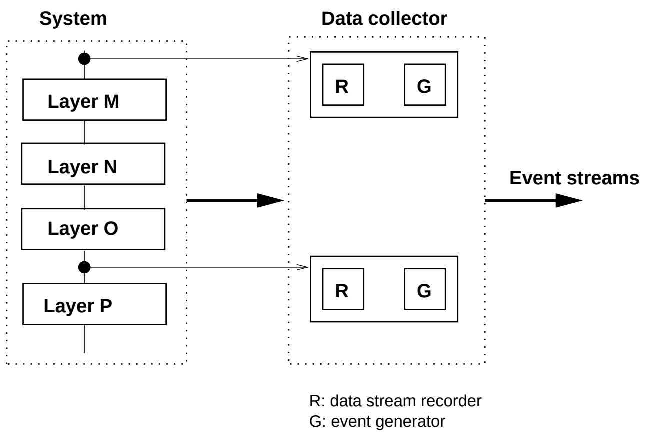


Figure 3.5: Overview of the data collector

The data collector, which was already presented in a broad overview in section 3.1, is responsible for the recording of network traffic and the extraction of events. The collector records n data streams simultaneously and produces as many event streams as there are data streams. Generating less event streams than there are data streams would mean that some arbitrary correlation of the data streams has already taken place, and that information was lost. This is different from a sampling of the incoming data stream, as the information loss through sampling is due to a consistent choosing operation on one data stream, since the sampling is a well-defined operation which can later be considered in the correlation and analysis phases.

For each system, the collector can be thought of as consisting of as many recorder / event generator boxes as there are layer instances at which the data stream is to be captured. In figure 3.5, the protocol stack of the system consists of four layers, with only two recorder / event generator boxes. The upper one records the service primitives generated and received by layer M, the lower one is responsible for layer P. Both generate an event stream. The term "data collector" is used to designate the set of recorder / event generator boxes associated with one system. When considering more than one system, each system has its own data collector, which may consist of one or more recorder / event generator boxes.

Let's consider a real-world example. To 2 the service quality of a HTTP connection between two hosts, each system will be outfitted with a data collector. Each data collector will in turn consist of two recorder / event generator boxes, one operating on layer 7 to collect data pertaining to the HTTP traffic itself, one operating on layer 4 to collect data which will later be used to generate information about the quality of service of the network connection.

Figure 3.6 gives a detailed view of the recorder / event generator boxes. The recorder simply gathers all the service primitives sent or received by the layer, and timestamps each of these. It filters out unwanted traffic which is not to be considered further (i.e., one might want to 2 only one TCP connection) and passes the captured primitives on to the event generator. The event generator is then responsible for stripping the service primitives of whatever superfluous information they contain (this could be, for example, the payload or header fields that are not used in the correlation or analysis process), and generates an event with a unique ID for each service primitives. The event streams thus generated are then passed on to the data correlator. By timestamping the events, the arrival distribution of the service primitives is well documented and can be used as a base for the computation of data context.

To keep the data collector generic, it is necessary to separate the knowledge about protocol mechanisms needed for the event capture and generation from the collector itself. By using a suitable formal language for the specification of protocol mechanisms, as proposed in 3.1, the data collector could be kept generic with unified interfaces, but easily adaptable to any kind of underlying network.

Of course, the number of data streams which can be simultaneously captured is not limited by the measurement architecture presented here. Still, a careful choice of the number of data streams and the layer at which they

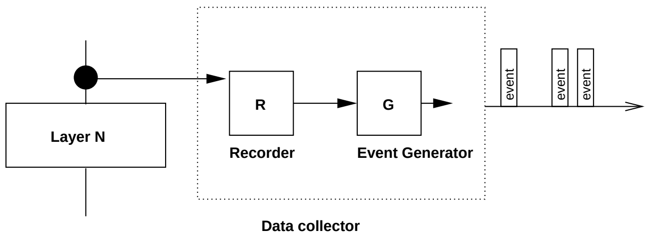


Figure 3.6: Detailed view of the data collector

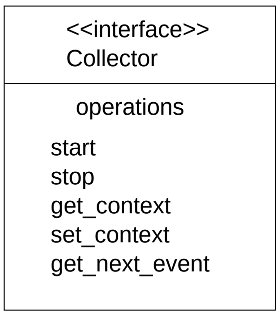


Figure 3.7: Collector interface definition

will be captured has to be made when using this architectural principle in an actual implementation, as the event stream presented at the collector / correlator interface affects the quality of service information which can later be computed from it. For example, the capture of only one data stream at only one point in the network will yield information about jitter, but not about packet loss. When planing to correlate event streams across a system interface, it is also advisable to generate the event stream on the same layer in each system.

Figure 3.7 shows the interface definition for the collector in UML notation. *get_context* and *set_context* are used by the correlator to obtain information about the collector context and update the collector's context to the desired value. *start* and *stop* start or stop a given capture, and *get_next_event* returns the next event produced.

3.5.2 Data correlator

The data correlator takes a number of streams as input and tries to establish a mapping on those streams; that is, the correlator identifies corresponding events in the streams. These could be, for example, the ICMP echo reply corresponding to an ICMP echo request which was sent.

Correlation

The correlation process itself is one of the more complex and interesting aspects in this measurement architecture. The goal of the correlation is to

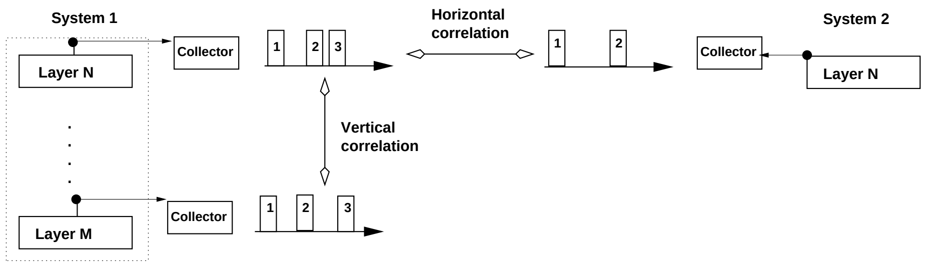


Figure 3.8: Vertical and horizontal correlation

find matching n-tuples of events in a set of n event streams. A matching n-tuple of events can be determinated by a well-defined relationship between events. When the correlation is done across the system interface, two or more events are matched if they were all raised by the same PDU across the network. This is called a horizontal correlation. The events represent snapshots of one PDU at different moments in time. When the correlation is done across the service interface, two or more events are matched if they represent the same SDU at different layers of the OSI stack. This is called a vertical correlation. The n-tuple of events can here be thought of as a chain reaction spreading through the stack and caused by the arrival of a SDU at layer 1.

One can also consider the semantic relationship between events. Consider for example the three-way-handshake which is used to establish a TCP connection. System A sends a SYN packet to system B, which responds with an ACK, causing system A to send an ACK to system B too. These three events (SYN, ACK, ACK), though not temporally or causally related, form an event triple which is "meaningful" to the transport layer, although the three events do not bear any obvious relationship to the network layer. Thus, this form of correlation is called "semantic correlation".

How is the correlation performed? This is no trivially answered question. There are different possibilities to perform the correlation, but as we will see, this is not always easy to do - especially across the service interface. The easiest way to find matching event n-tuples would be to compare payloads. As the application data unit travels through the OSI stack, each layer adds its PCI. If the original payload is still present in the event stream, it should be possible to compare those. This approach is immediately defeated if the payload is changed in the stack, for example by encryption or fragmentation. Since this approach is so easily defeated, it will not be considered further.

The next idea would be to fit each event with a unique ID, which should of course be the same for matching events, and through this ID recombine the matching n-tuples. If the protocols considered already contain some sort of sequence number, this approach is easily implemented when performing the correlation across the protocol interface. For example, if the traffic of a TCP/IP connection is captured, one might take the TCP sequence number to uniquely identify the events.

Tagging events

When correlation is attempted across the service interface, such protocol-specific IDs can not be used as they are not propagated through the entire protocol stack. Any protocol-specific information will be lost as soon as the processing for this stack layer has been done. This is an obvious drawback of the layer encapsulation of the OSI protocol stack. Instead, when attempting to correlate events across the service interface, it is necessary to "tag" every PDU through the stack without disrupting the processing operations. This approach also has its complications.

Since an untampered operation of the protocol stack is necessary to obtain meaningful measurements, any extention of the stack to include some kind of tagging layer is out of the question. Another drawback is that only a protocol stack instrumented to understand the extra tagging layer could serve as a communication endpoint. This 1 limits the range of experiments which could be performed using such a setup.

Another idea would be to implement a mechanism to keep track of the service primitives exchanged and help the collector boxes synchronize their event streams by counting events. Why isn't it sufficient to simply count service primitives? There are occurrences where a layer might receive an input and ignore it - for example, if the ATM layer receives malformed cells, those will not be relayed to a higher layer. A layer might also send two or more service primitives for one it received - this happens, for example, when the IP layer decides that a fragmentation of the received TCP packet is needed. In the reverse, when packet reassembly is performed, the network layer (here the IP layer) receives more service primitives as input as it sends to the transport layer (here the TCP layer). In all those cases, any simple counting scheme would soon become heavily desynchronized.

Figure 3.9 presents a mechanism for keeping a synchronized track of events over a multilayered protocol stack. The top part of the figure shows the timelines for three layers of the protocol stack and for two collectors. Collector *n-1* monitors the activity between layer *n-1* and layer *n*; collector *n* monitors the activity between layer *n* and layer *n+1*. Every time a collector sees the exchange of a service primitive, it sends a tick to the collector next above it. When a layer sends out more service primitives than it received, a backpropagation of the ticks is necessary. The lower part of this figure shows the event streams generated by the two collectors. If the collector monitored as many service primitives as it received ticks, then each tick corresponds to an event. When extra ticks have to be propagated, the collector generates an synchronization event - a placeholder which keeps the count even.

Four possible situations are presented in figure 3.9. Situation 1 is the most trivial. One service primitive is sent from layer *n-1* to layer *n*, and one from layer *n* to layer *n+1*. Each collector generates one event. In situation 2, layer *n* receives only one service primitive, but sends two. Collector *n* has to backpropagate a tick. Collector *n* generates two events, collector *n-1* generates one event and one placeholder. Situation 3 shows what happens when a

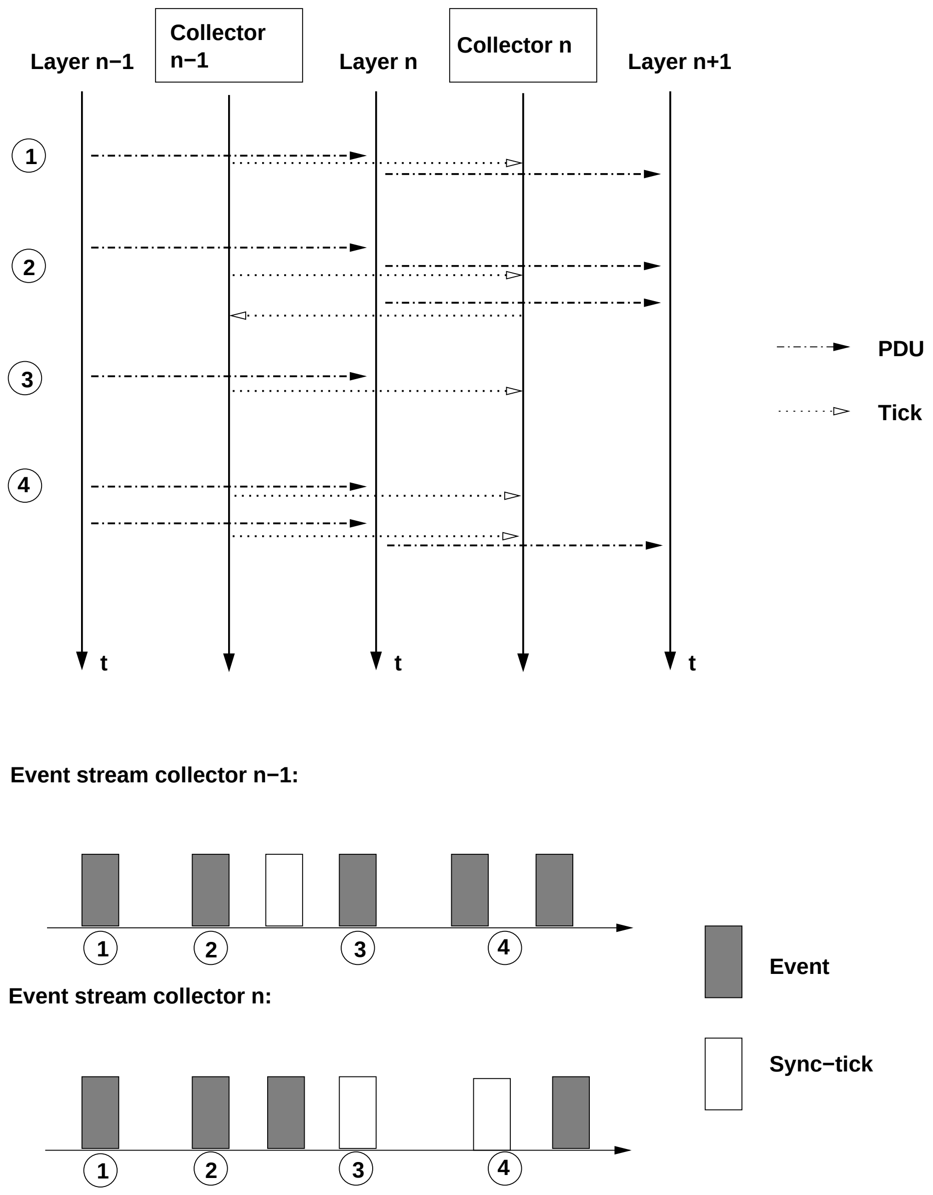


Figure 3.9: Correlation across the service boundary with an event counter

malformed packet is dropped by a layer. Collector *n-1* generates an event, collector *n* only has a placeholder. Finally, in situation 4, layer *n* receives two service primitives but only sends one. This can be seen as a special case of situation 2 - packet loss - as there is no possibility to determine whether a PDU was rejected, or packet reassembly was performed, without help of some semantic analysis. Here, collector *n-1* generates two events while collector *n* has one event and a placeholder. As can be easily seen, both event streams are still perfectly synchronized - the total number of events and placeholders is the same for both collectors. If the events are timestamped with enough precision, it would even be possible to determine the influence of the protocol stack on temporal characteristics of the data stream: the delay and jitter induced, for example.

While this concept might, in theory, seem feasible, its implementation is rather problematic:

- It works fine for a synchronous protocol stack, but fails in asynchronous protocol stacks, as a lower layer can have started the processing of a new PDU before all higher layer have finished their processing. Since there are no synchronous protocol stack implementations, the concept is not useful outside a theoretical consideration.
- Using hard real-time systems, it could be possible to achieve a satisfying level of reliability, as timing boundaries could be specified for the event processing, thus guaranteeing an event was generated at most *n* ms after the PDU arrival, and the event streams remains globally in correct ordering.

As a consequence, the prototype implementation presented in chapter 4 will focus on event correlation across the protocol interface and leave aside correlation across the service boundary.

Conclusion

Event correlation is used to find a mapping between a number of event streams. Three types of correlation have been identified: vertical correlation, when considering an event stream captured across the service interface; horizontal correlation, when considering an event stream captured across the protocol interface; and finally semantic correlation to detect protocol mechanisms like a TCP handshake.

The problem of tagging events - that is, of giving them a unique ID - was discussed. When related events from different event streams share IDs, the correlation is very easy. For example, the TCP sequence numbers can be used to correlate two captures of the same TCP connection. When the protocol considered features unique IDs for its datagramms, those can be used for tagging the events. Otherwise, IDs have to be generated, which leads to severe synchronization problems when capturing multiple event streams in a protocol stack. An alternative to IDing is payload comparison.

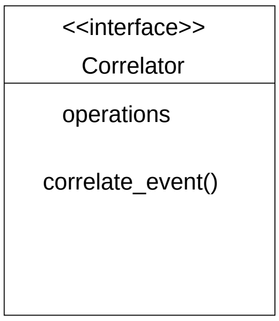


Figure 3.10: Correlator interface definition

3.5.3 Data analysator

Statistical analysis

Once a suitable event correlation has been found, an analysis of the correlated event streams has to be performed to determine the value of the QoS base dimensions:

- Reliability
- Capacity
- Integrity
- Timeliness

Reliability

Reliability denotes the reliability of the service offered at the specific network layer. At the application layer, this is the service availability: how probable is it the end user will be able to use the offered service at a given timepoint? At the transport layer, the meaning depends on the type of service offered. When a connection-oriented service is offered, the reliability denotes the resilience of the connection, that is, the probability that the connection will be dropped prematurely. For a connection-less service, it is the residual error rate: the ratio of incorrectly received and undetected cells, frames or packets to the total number received.

The non-availability of a service or the premature closing of the connection can be detected using semantic correlation on the event streams. Here, the correlation should be successfull only when the interaction between the client and the service provider is successful itself; that is, as long as the attempt to use the service did not fail, the correlation itself should also be successful. A detailed knowledge of the inner workings of the concerned protocols is, of course, necessary to distinguish between a premature connection closing and a normal one, or between a successful service usage and an unsuccessful one.

For a connection-less service, horizontal correlation is sufficient to detect incorrectly received cells, frames or packets when measuring the protocol interface. If the events do not possess a unique ID determinated by protocol mechanisms (for example, a sequence number), it might be necessary to compare payloads in order to obtain a meaningful correlation.

Capacity

Capacity is the throughput or bandwidth between two end systems. The capacity measured here is not the total capacity, but rather the capacity used by the protocol at the considered layer. The available bandwidth of a connection depends on two things: the underlying capacity of the network path, and the amount of other traffic competing on this path (see [16]). The easiest way to determine capacity is simply by counting the number of events per second on a single event stream. By correlating the event stream vertically with a timer event stream, it becomes possible to evaluate the event streams on a continuous timeline without synchronized clocks. Using the timeline, it is possible to detect gaps in the transmission that would otherwise go unnoticed: without synchronized clocks, the correlator can not know how old the last event in a given event stream is and whether a big time difference between the event timestamp and the correlator time is due to very bad clock synchronization or to a gap in the transmission.

Integrity

Integrity denotes how much the data stream was affected by the transmission. At layer 7, this means the payload was untampered. Here, a horizontal correlation between event streams using payload comparison is necessary to detect changes in the payload. At layer 4, the definition of integrity depends strongly on the definition and feature-set of the protocol considered. If UDP is the protocol considered at layer 4, the QoS base dimension integrity becomes meaningless, as UDP provides an unreliable ("best-effort"), unordered delivery service. Using TCP, on the other hand, integrity designs the correctly sequenced arrival of all packets. When considering a protocol that uses sequence numbers, sequencing errors can easily be determined by comparing the event succession with the sequence number succession. No correlation is necessary.

Timeliness

Finally, timeliness denotes the delay encountered for frame / message transmission or the service response time when considering the application layer. When measuring across the protocol interface, a horizontal correlation of event streams is needed; but since the clocks of the different collectors are not necessarily synchronized (see 5.3.2), it might not be possible to compute the absolute transmission delay. In this case, the average deviation or the variance of the transmission delay can be computed. When measuring across the service interface, a vertical correlation of the event streams is sufficient, and since there are no problems with clock synchronization, the transmission delay can be directly computed.

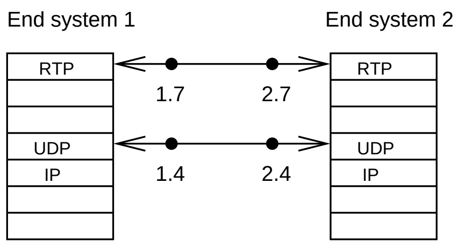


Figure 3.11: Analysis of a video transmission between two hosts

3.5.4 Further analysis?

The data analysator provides a rather technical and very detailed view on service quality. The information is broken down according to the four QoS base dimensions and to the protocol stack layer at which the data stream was captured. Here, a second level of correlation can be used to perform an additional analysis on the data. Also, a more concise representation of QoS might be needed for end-users. Only a broad overview of these ideas will be presented, as they are outside the scope of this QoS measurement framework.

Second correlation

The QoS data provided by the data analysator is, as stated previously, layer-specific. Consider a video transmission with RTP over a UDP/IP network (fig. 3.11). Four event streams are generated: the RTP event stream before and after transport over the network (labeled 1.7 and 2.7) and the UDP event stream before and after transport (labeled 1.4 and 2.4). Since the measurement took place across the protocol interface, the correlation will consider event streams 1.4 and 2.4, and event streams 1.7 and 2.7 respectively.

A second correlation process could now take place to find relations between the QoS information of layer four and layer seven and 2 the repercussions of lower layer QoS changes on higher layers.

Why is it important to differentiate between this correlation process and the one that was considered earlier? The correlation process taking place during the measurement is a correlation of event streams: it basically matches events against each other. This correlation process, on the other hand, takes place on statistical data. It is important to note this distinction: despite the similar names, the correlation process proposed here is entirely different of the one taking place on the event streams.

User-centric QoS

The QoS base dimensions considered so far are rather technical. User-centric QoS, on the other hand, typically deals with very vague and human concepts. A further analysis step might be used to further refine the QoS representation.

End users typically seek a simple representation of quality of service: the relevant question is whether the service provider keeps his contractual obligations regarding service quality. The end user wants to know if the product delivered (be it connectivity, application serving etc.) corresponds to the one he paid for - and has probably little need for all the detailed technical information that was gathered during the QoS measurement process.

The lack of a common, standard definition of high-level QoS makes it difficult to treat this question in depth, so only a broad overview of possible approaches to the problem of providing QoS information to an end-user will be given.

There needs to be an appropriate representation of user satisfaction which can be mapped onto the QoS base dimensions in order to assess whether the QoS encountered satisfies the user's expectations. The QoS base dimensions alone are not sufficient to judge whether a given service quality should be considered "good" or "bad". What satisfies one user's expectations (for example, a low latency, high throughput connection-oriented service) might be entirely unsatisfying to another user. The AAL service classes are a good example different service types with potentially conflicting goals regarding QoS.

If a service level agreement specifying the service quality in technical terms (that is, offering guarantees such as keeping the packet loss under a certain limit etc.), and when a mapping from those guarantees onto the QoS base dimensions specified here can be found, it is sufficient to monitor the base dimensions and notify the user when the service quality does not satisfy the SLA anymore.

It is also possible to define a metric for user satisfaction (see, for example, [48], which was already discussed in section 2.2.1) and map it onto the QoS base dimensions. An additional problem that has to be considered here is the interdependency of the QoS base dimensions: for example, if the packet loss increases, but the one-way-delay decreases, the effects might cancel each other out, thus giving no impression of subjective degradation of the service quality to the end user.

It should also be noted that the QoS measurement framework presented here does not consider all aspects of service quality that are important to many end users: for example, the security and safety of a connection. This limitation was discussed in 3.3.2.

3.6 Discussion

After this in-depth discussion of the proposed QoS measurement framework, I will now evaluate this proposal in reference to the requirements that were listed in section 2.3.1. Those were:

1. Use a generic, high-level and complete QoS definition and define a suitable mapping onto procotol-specific measurands

2. Hide underlying network heterogeneity by using a generic modeling of data streams
3. Provide unified views and interfaces
4. Use context parameters to control the conditions under which the measurement takes place

The four QoS base dimensions identified provide a high-level, generic QoS definition. Since those are based on the ISO QoS standard X.641, the QoS representation is fairly complete and can be regarded as generic. An exemplary mapping onto network-specific measurands for TCP/IP networks, multimedia applications over RTP and video conferencing over ATM networks was proposed, showing that the architecture is not limited to packet-switched networks. Mappings onto other network technologies should be fairly straightforward and, as long as care is given to a non-ambiguous mapping, the measurement results stay comparable, thus keeping the architecture generic.

The event paradigm gives an abstract description of data streams. By defining suitable event classes inheriting the event class proposed, this generic concept can be used to describe any kind of data streams. Having one common model hides the heterogeneity of underlying networks and enables the use of unified methods. The concept of event correlation is also very generic.

The three blocks of the architecture, the collector, correlator and analysator, permit a modular implementation. Using externally defined rules for event generation, correlation and analysis keeps the interfaces and methods generic, as protocol-specific information is not part of the measurement framework, but easily exchangeable and modular.

This proposal of a QoS measurement architecture clearly meets all the requirements stated in section 2.3.1. It is generic, not limited to a particular type of network, and provides unified views, hiding the underlying heterogeneity.

The limitations of this approach should also be noted. The definition of service quality much of this framework is based on is very network-centric and technical. The reasons for this limitation have been discussed in 3.3.2; basically, the QoS base dimensions have to be restricted to quantifiable measurands in order to be of any use in such a measurement framework. Still, it is a drawback that this measurement architecture offers no provision for the handling of, for example, security-related QoS characteristics. Also, the concept of QoS proposed ignores functional QoS: that is, it cannot evaluate whether the service provided the functionality desired by the user. For example, there is no way to distinguish between a meaningful video transmission and white noise using the four QoS base dimensions discussed previously.

Certain ideas that have been proposed and discussed will be very difficult to implement. Correlating across the service interface is problematic, as has been discussed in section 3.5.2; even the measurement process itself is non-trivial in this case, as a live protocol stack has to be instrumented in such a fashion that accurate results can be obtained without disrupting normal operation. Such a setup also carries very extensive timing problems, but a very precise and timely measurement is needed to get usable results.

Chapter 4 will present the UML modeling for the QoS measurement and monitoring architecture, and prepare the ground for the discussion of the implementation aspects in chapter 5 and the proof-of-concept implementation in chapter 6.

Chapter 4

Software model

4.1 Interaction

Figure 4.1 shows the typical interaction of a user with the system. When the user starts a measurement, the correlator starts all necessary captures at the data collectors, which in turn send the event streams corresponding to those captures. After a pre-buffering phase, the correlator begins sending a correlated event stream to the analysator, which outputs the QoS base dimensions to the user. The data collection, correlation and analysis process are concurrent.

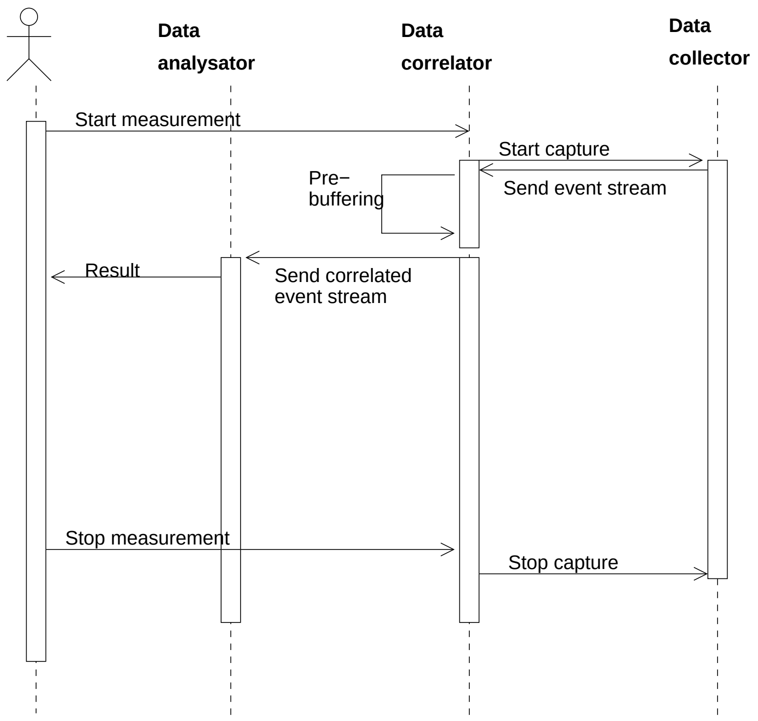


Figure 4.1: Sequence diagram

4.2 Software deployment

Figure 4.2 shows the deployment of the software on different nodes: the measurement point, which does the packet capture, and the analysis computer, responsible for correlation and analysis of the event stream. The number of data collector nodes is not limited, but there is only one correlator / analysator node which handles all the event streams generated.

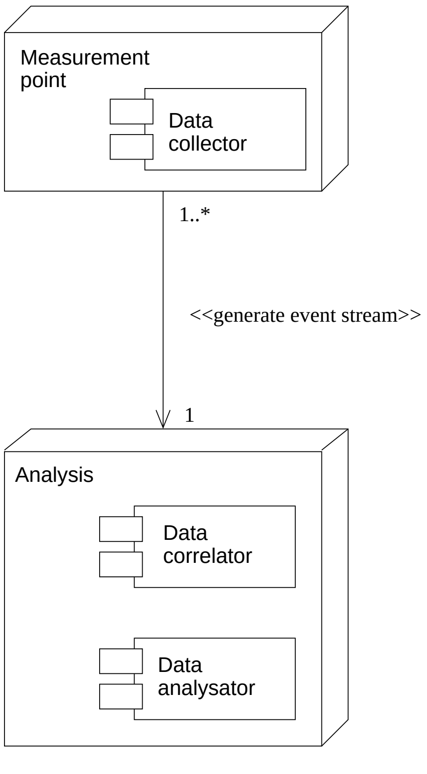


Figure 4.2: Deployment diagram

4.3 Events and context

Figure 4.3 details the concept of event streams. An event stream is a composition of events. Context is modeled as an association class.

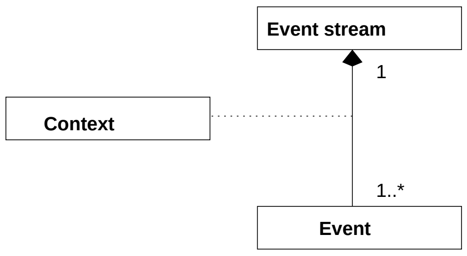


Figure 4.3: Event streams

Figure 4.4 shows the event classes that were discussed in section 3.2. The interface defines all the externally accessible operations: creating a new event, and getting / setting its timestamp and ID attribute. The IPEvent and TCPEvent class, which implement the operations defined by the interface class, have a number of additional protocol-specific attributes which are needed by the correlation process.

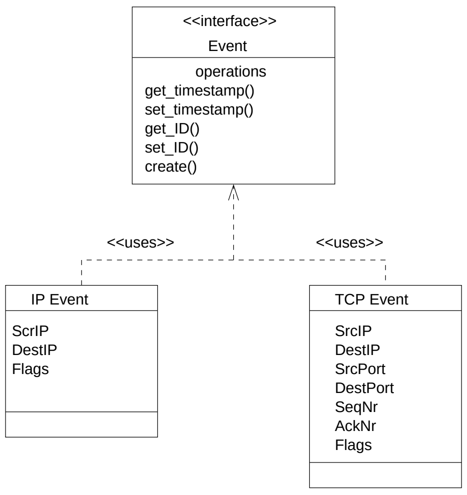


Figure 4.4: Event classes

Figure 4.5 shows the context class. The CPU load, the current error estimation, the freshness (total processing time) and buffer sizes of the analysator and correlator (*sampling_buf_size*) are attributes of the context. Two methods, *get_context* and *set_context* can be used to interact with the context object.

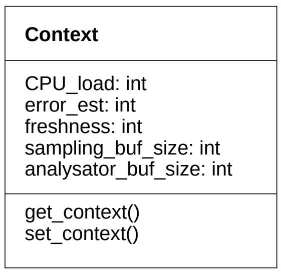


Figure 4.5: Context

4.4 Collector, correlator and analysator

Figure 4.6 shows the class diagramm and the interaction of the collector, correlator and analysator. The correlator is an active class; as detailed in figure 4.1, it starts and stops the captures of the collector and provides a correlated event stream to the analysator.

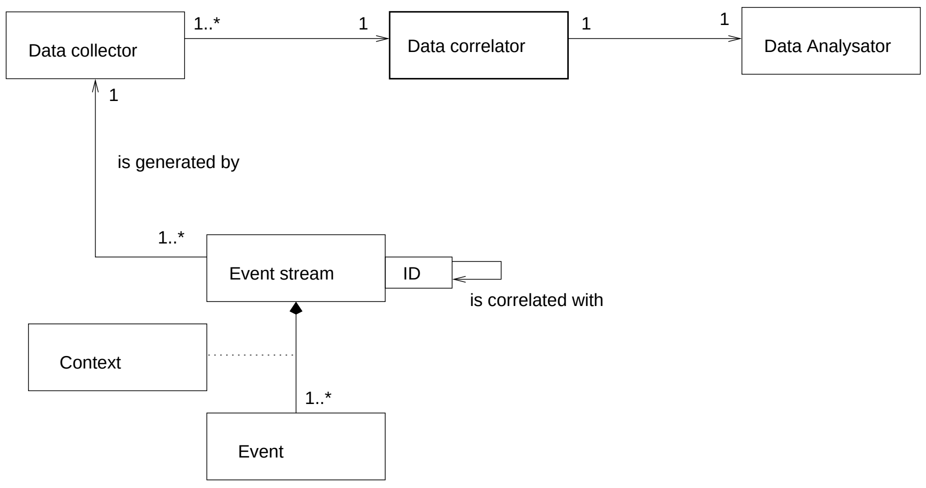


Figure 4.6: Class diagramm

Figure 4.7 shows an instantiated object diagramm of the class diagramm previously shown. Two collectors, Coll01 and Coll02, produce each an event stream. which are correlated. There are only one instance of a correlator and an analysator.

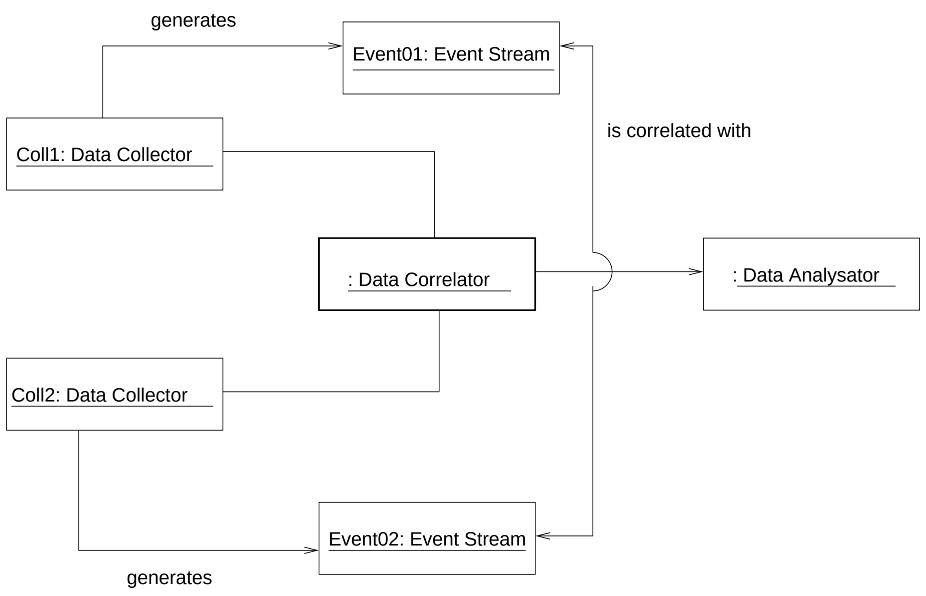


Figure 4.7: Object diagramm

Chapter 5

Implementation aspects

After presenting the architectural framework in chapter 3 and its realization as a UML software model in chapter 4, this chapter will discuss the details of the implementation. Section 5.2 discusses the different possibilities of the experimental setup. Section 5.3 discusses the technical constraints likely to be encountered during the implementation: timing considerations, limitations of the implementation, the problem of calibration etc. Finally, section 5.4 presents a prototypical implementation with reduced feature set that was written to assess the feasibility of certain architectural ideas and find problematic issues that had been overlooked.

5.1 General considerations

In section 2.3.1, a number of requirements for the QoS measurement and monitoring architecture and its implementation were identified. The following requirements are relevant to the implementation:

- The collector and the correlator / analysator should be implemented as separate programs in such a way that they can run on separate computers. Otherwise, analyzing data streams from separate measurement stations would not be possible.

- A serialization of non-correlated event streams should be possible in order to permit a delayed traffic analysis

- Both the collector and the correlator / analysator should be modular, maybe even plugin-capable, to facilitate the handling of new protocols. Adding a new event type and rules for the correlation / analysis of that event type should be as easy and straightforward as possible.

- A long latency between the measurement and the actual result output is not problematic, but the delay should be quantifiable.

- A good error approximation is preferable to a low error rate without error approximation.

- The impact on system performance should be minimized and, if possible, quantifiable.

5.2 Experimental setup

5.2.1 Introduction

Before the software implementation can be started, the experimental setup to be used during measurement has to be clarified. This covers questions such as: how many measurement points shall be used, how many streams shall be captured, will an active or a passive measurement be used etc. The measurement framework does not specify those points: an arbitrary number of streams can be captured and, as long as a correlation rule set exists, the event streams generated from those data streams can be correlated and analyzed. Also, the measurement method is not specified: any kind of interaction between network entities can serve as a starting point for further analysis - whether this interaction is passively observed or actively injected traffic is irrelevant from the viewpoint of the measurement framework.

While the flexibility of the measurement framework might make such questions look trivial and irrelevant, they are indeed quite fundamental and should be examined in depth before the measurement framework is implemented. After clarifying some of the terms and concepts used here, I will discuss the details of the experimental setup.

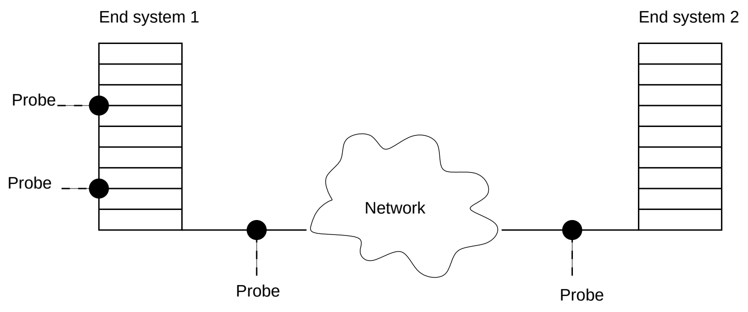


Figure 5.1: Different measurement points

As discussed in chapter 3, the measurement can take place across the protocol interface or across the service interface. The first case refers to a situation where a (possibly dedicated) machine is placed on the network between the endpoints of a communication and records the traffic; the latter implies the use of a specially instrumented protocol stack in the operating system's kernel on the target machine, enabling the user to generate an event stream based on the actual data exchanged between layers of the protocol stack. Figure 5.1 details those possibilities.

For reasons of practicability, only the former case will be considered here. A measurement point designates a machine which intercepts the traffic at a

	-	+
One measurement point	Capture of one stream for each traffic direction adds server response latency times	Easy to set up, collector and correlator on one computer
Two measurement points	Correlator does not run on same computer as collector. Latency between event stream generation and analysis.	Two data streams, more information available through data analysis

Table 5.1: Comparison between a one-point and a two-point measurement setup

	-	+
Passive measurement	Very small choice of analyzable traffic when using with one meas. point	No falsifying of observations by traffic injection
Active measurement	Packet injection falsifies observation	Very customizable. Widely used.

Table 5.2: Comparison between passive and active measurement

designated point in the network. Of course, more than one event stream can be generated at the measurement point. Possible placements of measurement points in the networks and the implications will be discussed next. Since the proof-of-concept implementation will operate on IP networks, the following considerations will be limited to unicast traffic in IP networks.

5.2.2 Comparing the different approaches

The first decision to take is the numbers of measurement points to arrange in the network. Using one measurement point has the clear advantage of an easy set-up: there is no need to place another measurement point in a possibly geographically remote location to arrange for another capture. Also, the correlator and analysator can run on the same machine, thus reducing networking overhead and providing the user with possibly fresher data.

The drawback is a certain limitation concerning the type of traffic that can be captured and, later, analyzed. A two-point-setup enables the user to capture the same traffic before and after its trip through the network and thus make a "snapshot" of the same data at different moments in time. A one-point-setup capturing two streams (the incoming and outgoing data streams, for example) will never catch the same packet twice. Here, vertical correlation (as discussed in chapter 3.5.2) will probably be used, in contrast to the much simpler horizonal correlation which can be used with two-point-setups when capturing packets that are uniquely IDed. Also, the latency of the server's (or client's) reaction time might falsify the results and has to be taken into account.

In the literature, one-point-setups are often used with active measurement techniques: that is, a special packet stream is injected into the network to serve as an active probe. One example can be found in [24], where specially prepared ICMP packets are used as probe stream. Finding correlation rules for that type of event streams is straightforward, as obviously every ICMP echo request should be paired to it's corresponding ICMP echo reply.

Active measurement can be problematic because it has the potential to falsify the measurement results by putting undue load on the network. Thus, when using active measurement, special care has to be taken to ensure the injected packet stream does not disrupt normal network operation. [23] proposes a mechanism for the calibration of active delay measurement systems by a passive measurement system. Also, a specially crafted traffic stream has to be designed in order to be able to use active measurement. Simple probe packets like ICMP echo requests and replies do not always provide a complex enough probe stream to gather the required data for analysis. Using a badly designed probe stream can be very limiting, and by doing passive measurement, one can easily circumvent this difficulty.

[39] proposes a very interesting implementation of a passive measurement technique which shows how implicit attributes of a network stream can be used for measurement. By examining the behavior of loss pairs - that is, a pair of packets which travel the same path and close together in time such that exactly one of the packets is loss - the authors show that they can discover network properties such as packet dropping behaviors of routers on the network path. This shows that passive measurement techniques should not be underestimated, and can offer valuable insights.

The proof-of-concept implementation will use a passive measurement technique and one or two measurement points, as well as the capture of an arbitrary number of streams, thus combining the advantages of a non-invasive, no-impact passive measurement with the powerful correlation possibilities of a larger choice of streams. Tables 5.1 and 5.2 provide a summary of the advantages and drawbacks of one- and two-point-setups and of active versus passive measurement techniques.

5.3 Technical constraints

5.3.1 Sampling methodology

Using a full trace for traffic analysis can be very time- and resource-consuming, especially when considering real-world networks where users and applications cause a lot of traffic. The solution to this problem is to use only a fraction of the network traffic, that is, use a suitable sampling methodology to record only chosen parts of the network traffic. Of course, using the full packet population for analysis yields better and more exact results, but each instance of sampling uses resources such as CPU time, buffer space etc. It is necessary here to weigh the sampling frequency against the accuracy requirements.

In [19], different sampling methodologies were evaluated in the context of network traffic analysis. Both timer- and packet-driven sampling methodologies were considered. The former make use of a timer to trigger the selection of a packet: when the timer expires, the next packet to arrive is selected. Packet-driven sampling methodologies, on the other hand, use a packet counter to trigger the selection of a packet. Furthermore, three sampling algorithms were considered: systematic (take the first member out of n buckets), stratified random (take a random member out of n buckets) or simple random (take n random members out of the whole set).

Time-triggered techniques were found to perform not as well as packet-triggered ones. The performance differences within each class (time- or packet-based techniques) were found to be small, contrary to what one might have expected. Random sampling is less efficient than stratified random or systematic sampling only when considering populations with a linear trend. Also, the exactitude of the results quickly declines with bucket sizes over 1024 packets.

5.3.2 Computer clocks

As stated in [47],

Measurements of time lie at the heart of many Internet metrics.

It is important to consider the technical constraints imposed upon any computer-based measurement setup by the errors and uncertainties of imperfect clocks. The following issues regarding computer clock precision have to be considered:

- Offset: A clock's offset at a particular moment is defined as the difference between the time reported by the clock and the "true" time, defined by UTC[1]. The offset can be subject to jitter, due for example to time-keeping algorithms of the kernel trying to "catch-up" the clock too fast, to hardware problems or to the time-keeping algorithms of NTP. See [22] for a practical examination of clock offset variation.

- Skew: The frequency difference (first derivative of its offset with respect to UTC) between the clock and UTC. Real clocks show some variation in skew: the second derivative of the clock's offset to UTC is non-zero. This is defined as the clock's drift. A quartz crystal clock typically has a drift rate of 10^{-6}, a high-precision quartz clock has a drift of 10^{-7} to 10^{-8}.

The Network Time Protocol cannot be used for clock synchronization in this context, as its goal is to provide accurate timekeeping over long time scales, i.e. minutes to days, while precise measurements call for short-term

[1]Universal Time Coordinated; uniform atomic time system kept very closely to UT2, that is, the universal time deduced directly from observations of stars, and corrected for polar motion and seasonal variation in the earth's rotation rate

accuracy. NTP clients use fairly complicated algorithms to slowly adjust the local computer's time to NTP time, in order not to hurt the assumption that timestamps always increase monotonously over time.

Using a radio controlled precision clock (which, themselves, are not without problems - see [44]) for timekeeping is not a practical option for the experimental setup discussed here, as it is both expensive and, as will be shown, superfluous. Instead of relying on absolute time and time synchronization across all measurement points, it is possible to either measure time only at one point (which is trivial to do in a one-point-setup as discussed in section 5.2.2) or measure only relative time differences.

In [18], it is discussed how measurements targeting not absolute time values but fluctuations of values (here, the variation in, rather than the absolute value of, one-way-delays) can ignore the problem of tight clock synchronization. Furthermore, even using radio-controlled precision clocks does not solve some of the other timing problems that are likely to come up when doing computer-based measurements: the system load, system tuning parameters (for example, the use of optimization switch in the compilation of the operating system kernel) etc. will influence the accuracy of any measurement.

For the implementation purposes of this thesis, the measurement setup focuses on relative time differences, not absolute time values; and the skew and drift of computer clocks are ignored, as the implementation proposed serves only as a proof of concept to demonstrate the practicability of the architectural framework. Any real-world implementation should, of course, pay attention to this problem.

5.3.3 Libpcap interface

The BSD packet filter provides a facility for user-level packet capture. Network monitors normally run as user-level processes, thus making a copy of network packets from kernel to user space necessary. This copying is time- as well as resource-consuming. In order to minimize this overhead, a kernel agent called ”packet filter” can be used to discard unwanted packets early on, as long as they are still in kernel space. This way, the application or network monitor is furnished only with the part of the traffic capture it considers relevant. Figure 5.2 gives a schematic view of the BSD packet filter. [42] discusses the architecture of the BSD packet filter in more detail.

Today, implementations of the BSD packet filter have been ported to almost every flavor of *NIX. A more user-friendly interface to the BPF can be found in the pcap library ([4]). The pcap library provides functionality to capture packets at network interface level, and to execute a predefined function call for every packet captured. Using the filter syntax known from the *tcpdump* utility ([3]), packet filters can be compiled and set, so that the BPF can discard unwanted traffic.

To capture packets using the pcap library, first a packet capture descriptor

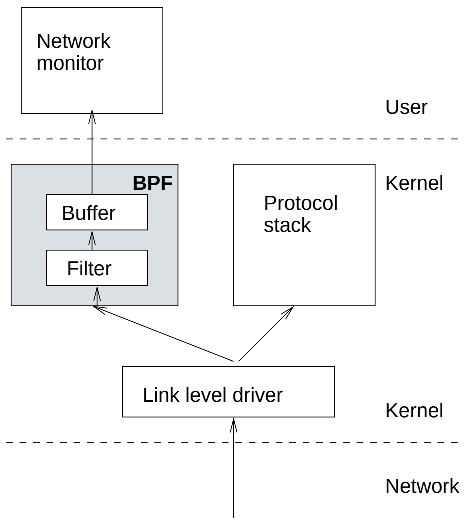


Figure 5.2: The BSD packet filter

has to be obtained using, for example, the *pcap_open_live* function. This function takes a network device (i.e., *eth0*), the maximum number of bytes to capture from each packet, a read timeout and a flag indicating whether the network device should be put into promiscuous mode as arguments and returns a packet capture descriptor.

After initializing the capture device, capture filters using the *tcpdump* syntax can be compiled, and information about the configuration of the network interface can be gained using *pcap_lookupnet*. The function *pcap_loop* keeps reading packets from the interface until an error occurs. This function takes a callback function as an argument. The callback function is executed every time a packet is read. This function takes a pointer to a user-specified structure (so that the user can pass arguments), a pointer to the *pcap_pkthdr* structure (which contains information such as the capture time) and a pointer to the packet data.

The BSD packet filter and the pcap library are used as part of the data collector implementation, as their functionality is more than sufficient here, and a reimplementation would be not only unnecessary, but also need a massive programming effort.

5.3.4 Calibration

Assessing the error level introduced by the measurement process is not an easy task. Measuring the application's resource usage or the degradation of the measurement computer's activity do not give information about the level of precision which can be guaranteed by the measurement point. In order to get detailed information about the error level, a calibration process can be used.

The idea behind this is to use a generated packet stream with well-defined characteristics as a calibration device. Ideally, the calibration stream would be injected into the network at such a point that it suffers as little modification as possible before it is captured by the measurement station. By comparing the original characteristics of the stream to those of the captured streams, the error level introduced by the measurement process can be more accurately estimated. A very simple calibration stream could be, for example, a *ping*.

Characteristics that can be compared and give an indication of the error level introduced by the measurement process include, for example, the arrival distribution of the packets compared to the distribution of the generated events. A large discrepancy here might indicate efficiency problems.

5.3.5 Other technical limitations

Further technical limitations to the implementation have to be considered. Not all ideas presented in chapter 3 are practical and easy to implement. Any measurement to be done across the service interface of the OSI stack requires an operating system's protocol stack to be modified in a way that measurements can easily be obtained. Consider for example the TCP/IP protocol stack of a typical Unix system like FreeBSD. The kernel source tree (including the networking source) can be found in [5]. While not monolithical in design, the sheer complexity of the source code makes it clear that modifying the kernel in order to accommodate a measurement mechanism is no small task.

Even considering a specially instrumented protocol stack, the problem of clock skew remains. When considering timing aspects of such minute events as the passing of a service primitive between different layers of the protocol stack, a clock skew of a few milliseconds - which, as was discussed in section 5.3.2, is to be expected with modern computer hardware - is totally unacceptable and falsifies the results. Considering those practical problems, the proof-of-concept implementation presented with this thesis concerns itself only with measurement across the protocol interface.

The problem of instrumentation techniques is currently being actively researched. [32] presents an architecture for the automated management instrumentation of component-based applications, underlying the importance of such techniques especially in the field of QoS monitoring. The Open H323 project ([6]), which aims to create a full featured, interoperable, Open Source implementation of the ITU H.323 teleconferencing protocol ([36]), also takes this problems into consideration and offers detailed resources for monitoring.

Measuring across the protocol interface is also only an approximation, as the measurement happens at the service access point, i.e., just after the network card. The captured network trace has to be decoded in parallel in the measurement software, but no information is available about the inner workings of the operating system's protocol stack. Thus, the generated event streams can not be used as a base for correlation across the service interface, especially when the correlation is done in order to analyze timing-related phenomena.

A further question of interest is whether the use of a real-time operating system and a measurement software obeying to hard real-time constraints carries any advantages. Some of the correlation problems discussed in 3.5.2 might be averted. If the event generation is subjected to timing limits, and an event has to be generated within a certain timeframe of its corresponding packet or primitive capture, then two or more event streams could be kept tightly synchronized, thus easing the correlation process. A complete discussion of this problem is outside the scope of this thesis, but the approach might be promising.

5.4 Prototype implementation

To evaluate the feasibility of an implementation of the measurement framework presented and prepare the ground for the proof-of-concept implementation (see 6), a prototype with a minimal feature set was implemented. Perl was chosen as programming language because of its suitability as a rapid prototyping language and because Perl's data manipulation features make it an ideal choice for event stream correlation and analysis.

The prototype was to have the following feature set:

- Support of IP and timer event streams
- One collector, one correlator / analysator
- Collection of maximum two event streams (one IP stream, one timer stream)
- Measurement of bandwidth by counting packets
- Context realized as: percentage of captured packets, packet processing time

Figure 5.3 details the experimental setup that was chosen for the prototype.

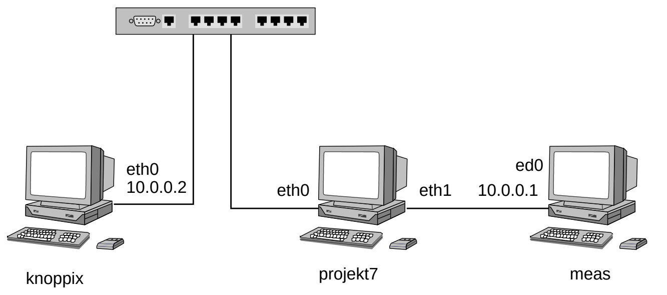


Figure 5.3: Experimental setup for the prototype

To produce test traffic, the measurement station *meas* was used to ping *knoppix* with variable packet intervals. Using the *ping* utility had the added benefit that the generated packet stream had a well-defined bandwidth usage, as the number of packets to send per second can be specified on the command line interface. The computer *projekt7* serves as a bridge between both networks. It runs a software written by Boris Lohner which disturbs the flow of traffic as a WAN would do: it offers the capability to simulate packet loss, delays, delay jitter and packet duplicates with a configurable interface. More details about the WAN emulator can be found in [40]. Both the collector and the correlator / analysator were run on *meas*.

5.4.1 Prototype software

Two Perl programs were written to implement the prototype: collector.pl and correlator.pl. To enable a distributed use and test this concept, they were both given the capability to communicate using network sockets. In this scenario, collector.pl acts as the server, correlator.pl as the client. The functionality and realization of both programs will now be discussed in more detail.

collector.pl

In this scenario, the program collector.pl acts as a server. It accepts one control connection and as many data connections as the client opens - each data connection being equivalent to one event stream. The context is global to the collector instance, which means that all event streams from one collector will have one common context. This is due to the nature of the implementation. Since the data collector runs on one machine, the context has to be global to avoid conflict. If, for example, the user requires freshness from one stream and completeness from the other, the collector process would have to satisfy two conflicting goals: that of minimizing the packet processing time, for example by using suitable sampling methods, and that of delivering a complete trace. Furthermore, since all event streams are generated from one network trace, i.e. one instance of the pcap driver, every context switch means that all captures have to be restarted.

After some initialization at startup, the program opens a socket and waits for connections. Once a control connection has been accepted, the pcap interface is initialized, and the client can open a data connection. For every event stream requested by the client, a child process is forked, starts a recording loop on the pcap interface, and instantiates a callback function of the correct type. For every packet that comes across the interface, the callback function is triggered. After performing some context checks (i.e., do some sampling and test whether the maximum processing time is likely to be exceeded), the packet is decoded and an appropriate event is generated. Whenever a change in context is requested by the client, all current event streams are re-initialized.

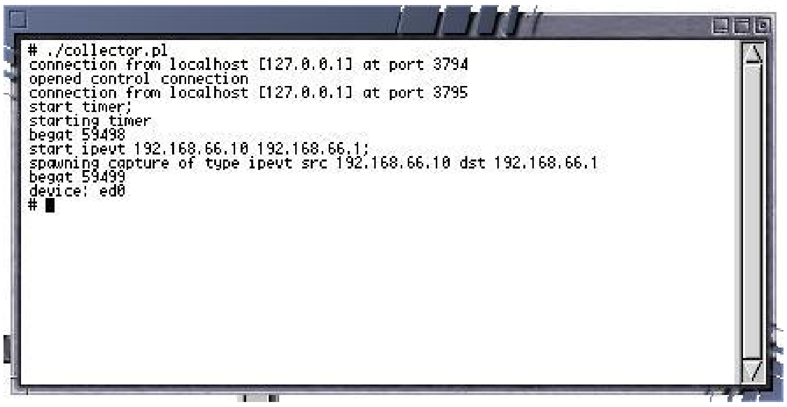
correlator.pl

The *correlator.pl* program contains both the correlator and the analysator functionality. As the functionality of correlating event streams and later analyzing the correlated stream come together quite naturally, no effort was made to separate the functionality into two distinct programs.

The QoS to be measured here was capacity, in the form of packets per second. The measurement was done using a timer- and a packet-based sampling. In the first form, the bandwidth usage was measured by counting the number of packets in sampling bins of five seconds length, marking the beginning and end of a bin by timer events; in the second form, a buffer of 5 packets was used to determine the time elapsed between the first and the fifth packet. Both results where then converted to packets per seconds. Due to a methodical error, the results of the timer-based approach were wrong and, thus, not comparable to the results of the packet-based approach.

The details of the correlator / analysator implementation will now be detailed. After some initialization at startup, the program connects to the collector server and requests a timer event stream and an IP event stream. A buffer size of 10 events is used for correlation. The program starts with filling the IP event buffer with 5 events before any correlation and analysis is done. After the prebuffering phase, the time- and the packet-based bandwidth computation is performed and printed on the standard output.

Figure 5.4 shows a screenshot of the prototype collector program. Figure 5.5 and 5.6 show the collector program. "pps1" is the bandwidth computed with a packet-based sampling, "pps2" the bandwidth computer with a timer-based sampling. In both screenshots, the traffic measured consisted of a simple ping.

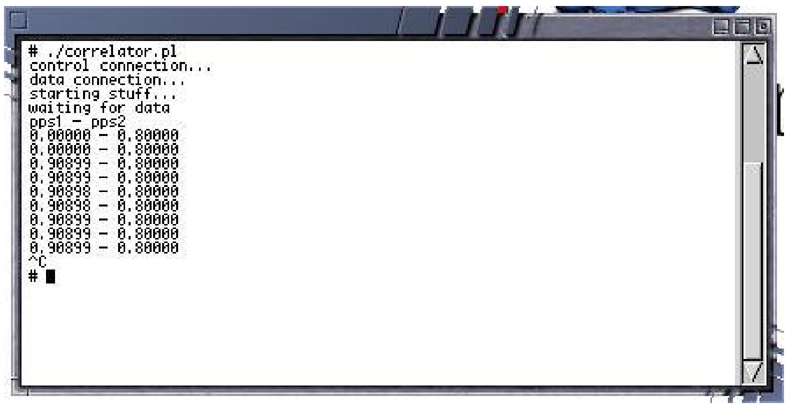


```
# ./collector.pl
connection from localhost [127.0.0.1] at port 3794
opened control connection
connection from localhost [127.0.0.1] at port 3795
start timer;
starting timer
begat 59498
start ipevt 192.168.66.10 192.168.66.1;
spawning capture of type ipevt src 192.168.66.10 dst 192.168.66.1
begat 59499
device: ed0
# 
```

Figure 5.4: Screenshot of the prototype collector

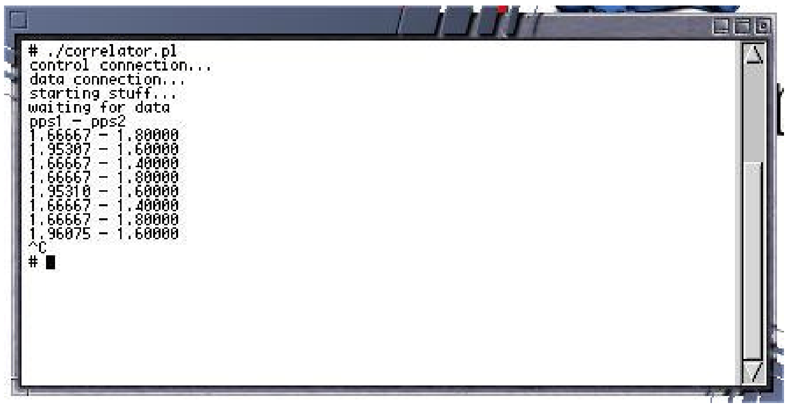
5.4.2 Conclusion

The prototype software clearly showed that an implementation of the QoS measurement architecture proposed in this thesis is feasible. It also helped to clear up the concept of context when viewed from the implementation side and prepare the ground for the proof-of-concept implementation. The



```
# ./correlator.pl
control connection...
data connection...
starting stuff...
waiting for data
pps1 - pps2
0.00000 - 0.80000
0.00000 - 0.80000
0.90899 - 0.80000
0.90899 - 0.80000
0.90898 - 0.80000
0.90898 - 0.80000
0.90899 - 0.80000
0.90899 - 0.80000
0.90899 - 0.80000
^C
# █
```

Figure 5.5: Screenshot of the prototype correlator - 1 packet / second ping



```
# ./correlator.pl
control connection...
data connection...
starting stuff...
waiting for data
pps1 - pps2
1.66667 - 1.80000
1.95307 - 1.60000
1.66667 - 1.40000
1.66667 - 1.80000
1.95310 - 1.60000
1.66667 - 1.40000
1.66667 - 1.80000
1.96075 - 1.60000
^C
# █
```

Figure 5.6: Screenshot of the prototype correlator - 2 packets / second ping

proof-of-concept implementation will have a very system- and implementation-oriented concept of context, which covers things such as CPU load, event processing time, buffer size etc. That context here is collector-specific and cannot be broken down to be event-stream-specific is due to the specifics of the implementation. This was discussed in section 5.4.1. The prototype implementation also helped to gain a better definition of event streams. A question which has yet to be answered and will be considered more carefully for the proof-of-concept implementation is whether the correlator process can correlate smoothly over context switches, or whether the correlation has to be restarted whenever the collector process restarts the event streams.

5.5 Conclusion

In this chapter, varied aspects of the implementation and the experimental setup were discussed. Different kinds of measurement setups were presented: one-point-setups, which are very easy to realise but quite limiting for the traffic analysis and two-point-setups, which are more difficult to realize as both endpoints of a connection have to be instrumented with a data collector, but offer a wider range of subsequent traffic analysis possibilities through the capture of two trace files. The advantages and drawbacks of active and passive measurement techniques were discussed. Active measurement techniques involve the injection of a probe packet stream into the network and can thus possibly influence network performance and falsify measurement results.

Technical constraints of the implementation were discussed. Since computer clocks are very inaccurate and difficult to synchronize, it is necessary to take that into consideration and use analysis techniques that do not rely on absolute time. The possible influence of the correlator / analysator and the collector program on system performance also have to be taken into account. A special calibration process can be used to estimate the error introduced.

After having discussion the fundamentals of the implementation work in this chapter, chapter 6 will present the proof-of-concept implementation and discuss the code and algorithms used.

Chapter 6

Proof of concept implementation

After the prototype model presented in chapter 5, this chapter will discuss the proof-of-concept implementation. This implementation covers a larger feature set and shows that the QoS measurement framework proposed can be implemented and that the implementation fulfills the requirements outlined in section 2.3.1. The proof-of-concept implementation supports IP, TCP, UDP, ICMP, FTP and timer event types. Horizontal and semantic correlation illustrate the concept of event correlation discussed in section 3.5.2. The analysator can process IP, TCP and ICMP event types, and features an exemplary mapping of protocol-specific measurands onto QoS base dimensions. Both the collector and the correlator / analysator programs show how the core program can be kept generic by a modular approach, cleanly encapsulating the protocol-specific code.

The client-server model, which had proved itself useful in the prototype model, was reused for the proof-of-concept implementation. Also, the proof-of-concept implementation was realized in Perl and PerlTk, as the Perl language is very powerful for rapid software prototyping and for string evaluation and supports complex data structures. This features were very helpful for the correlation and analysis processes.

The collector and correlator / analysator implementation will now be discussed with special attention to the event trace format used by the collector and the details of the correlation and analysis process.

6.1 Collector

coll-poc.pl is the data collector. It acts as an event stream server and waits for the client (the data correlator) to open a TCP/IP connection to the server and start the measurement.

The event generation functions are cleanly separated from the core code. The file *events.pl* contains event generation functions for IP, TCP, ICMP, UDP and FTP event streams. The pcap library is used for packet capture, as detailed in section 5.3.3. The choice of event types was motivated by the availability of the Perl NetPacket package which provides a very easy-to-use interface for parsing the libpcap output into Perl objects, and supports IP, TCP, UDP, ARP, and ICMP. Other protocols have to be parsed by hand or using other libraries to produce events.

The core code is mainly protocol-independent and generic. Setting the appropriate pcap filters for the capture has not been encapsulated into a separate function, but this would be rather straightforward. Adding support for new event types to the core code is very easy: a new event generation function and an appropriate pcap filter have to be added.

6.1.1 Event trace format

To simplify the usage of the collector and the implementation, a simple ASCII format was used for the event streams. All event streams have a few common characteristics: a single character indicating the event type, a stream number to differentiate simultaneous streams of the same event type, a sequential counter providing a unique ID, and a timestamp indicating when the capture occurred. This timestamp is taken from the *pcap_pkthdr* structure, not from the packet header itself and denotes the time of capture at the service access point. All event types except the timer event also feature the packet checksum, which is not used during further processing but included as an example.

The TCP event stream format is the following: stream type, stream number, sequential counter, processing time in the collector, capture timestamp, source IP:source port, destination IP:destination port, sequence number, acknowledgment number, checksum. Here a trace file example:

```
t 0 1 0000000000.009308 1034956430.222131 192.168.66.1:22 192.168.66.10:2032 1161751207 582185694 24441
t 0 2 0000000000.005549 1034956430.242704 192.168.66.1:22 192.168.66.10:2032 1161751208 582185694 40744
t 0 3 0000000000.005278 1034956430.251550 192.168.66.1:22 192.168.66.10:2032 1161751262 582185716 20061
t 0 4 0000000000.011882 1034956430.344180 192.168.66.1:22 192.168.66.10:2032 1161751814 582186180 34537
t 0 5 0000000000.011915 1034956430.444118 192.168.66.1:22 192.168.66.10:2032 1161751814 582186196 34501
t 0 6 0000000000.010918 1034956432.695591 192.168.66.1:22 192.168.66.10:2032 1161751814 582186196 50250
t 0 7 0000000000.012380 1034956432.973722 192.168.66.1:22 192.168.66.10:2032 1161752094 582186468 33443
t 0 8 0000000000.009834 1034956436.126960 192.168.66.1:22 192.168.66.10:2032 1161752094 582186468 31577
```

The IP event stream format is the following: stream type, stream number, sequential counter, processing time in the collector, capture timestamp, source IP, destination IP, header flags, checksum. Here an example:

```
i 0 1 0000000000.011267 1034956304.595531 192.168.66.1 192.168.66.10 2 46494
i 0 2 0000000000.007164 1034956304.615260 192.168.66.1 192.168.66.10 2 46445
i 0 3 0000000000.008387 1034956304.624136 192.168.66.1 192.168.66.10 2 45946
i 0 4 0000000000.011444 1034956304.720855 192.168.66.1 192.168.66.10 2 46497
i 0 5 0000000000.001633 1034956304.820808 192.168.66.1 192.168.66.10 2 46496
i 0 6 0000000000.010155 1034956307.112412 192.168.66.1 192.168.66.10 2 46215
i 0 7 0000000000.001671 1034956307.380706 192.168.66.1 192.168.66.10 2 46494
i 0 8 0000000000.007013 1034956310.575713 192.168.66.1 192.168.66.10 2 45725
i 0 9 0000000000.010326 1034956310.672157 192.168.66.1 192.168.66.10 2 46476
i 0 10 0000000000.001796 1034956310.900692 192.168.66.1 192.168.66.10 2 46491
i 0 11 0000000000.004970 1034956310.947449 192.168.66.1 192.168.66.10 2 46442
```

The ICMP event stream format is the following: stream type, stream number, sequential counter, processing time in the collector, capture timestamp, source IP, destination IP, ICMP type, ICMP code, ICMP sequence number, checksum. The example features two different traces. The first capture (sequence numbers 6 to 11) was a capture of ICMP echo request, which can clearly be seen from the ICMP type which is 0. The second capture (sequence numbers 1 to 6) was a capture of ICMP echo replies, where the ICMP type is 8:

```
p 0 6 0000000000.009004 1034956358.805915 192.168.66.1 192.168.66.10 0 0 0000 65345
p 0 7 0000000000.012140 1034956359.812671 192.168.66.1 192.168.66.10 0 0 0100 9256
p 0 8 0000000000.012110 1034956360.822718 192.168.66.1 192.168.66.10 0 0 0200 59904
p 0 9 0000000000.012097 1034956361.832751 192.168.66.1 192.168.66.10 0 0 0300 54233
p 0 10 0000000000.012060 1034956362.842789 192.168.66.1 192.168.66.10 0 0 0400 50098
p 0 11 0000000000.012064 1034956363.852830 192.168.66.1 192.168.66.10 0 0 0500 41867
p 0 1 0000000000.014830 1034956386.795612 192.168.66.10 192.168.66.1 8 0 0000 5991
p 0 2 0000000000.008062 1034956387.802177 192.168.66.10 192.168.66.1 8 0 0100 23117
p 0 3 0000000000.052036 1034956388.812179 192.168.66.10 192.168.66.1 8 0 0200 17702
p 0 4 0000000000.024002 1034956389.832215 192.168.66.10 192.168.66.1 8 0 0300 1496
p 0 5 0000000000.004688 1034956390.842225 192.168.66.10 192.168.66.1 8 0 0400 60848
p 0 6 0000000000.012623 1034956391.852232 192.168.66.10 192.168.66.1 8 0 0500 54409
```

Finally, the timer event stream, which is the simplest so far, features a stream type, stream number, sequential count and a timestamp. Timer events are used to indicate the time flow on a remote server, and have a reference time for the analysis of packets produced by this server. For example, to count the number of packet arrivals in the last 10 minutes, one has to know what "the last 10 minutes" meant in absolute time for the remote server.

```
h 1 0 1034956462.356046
h 1 1 1034956463.363309
h 1 2 1034956464.373256
h 1 3 1034956465.383262
h 1 4 1034956466.403276
h 1 5 1034956467.423348
h 1 6 1034956468.433316
h 1 7 1034956469.453321
```

Of course, a real event stream will be a mix of all those event types. When the event streams are then parsed by the correlator, the stream type and stream number are used to distinguish the individual event streams. While the ASCII format has the advantage of being easily readable to a human operator, a binary format would certainly be called for when dealing with large traces, to reduce trace file size. Furthermore, using a remote procedure call architecture might also be an interesting approach, as this would eliminate the need for event parsing in the correlator, and support the distributed implementation of this framework.

6.1.2 Code

The core program itself is rather simple, as the functionality lies mostly in the event generation functions. The main routine is a *while* loop which waits for user input on the control channel. The following commands are understood by the collector:

- *quit* - quit the collector
- *context* - context switch
- *start* - start measurement
- *stop* - stop measurement
- *stats* - output capture statistics

```
 1 while(!$exitflag){
 2  if(/^quit/i){
 3    ...
 4  } elsif(/^context (...) (\d*)/i){
 5    ...
 6  } elsif(/^start (.*_evt) (\d*?\.\d*?\.\d*?\.\d*?) (\d*?\.\d*?\.\d*?\.\d*?);/i){
 7    spawn(\&data_capture, $1, $2, $3);
 8  } elsif(/^stop (\d*)/i){
 9    ...
10   } elsif(/^stats/i){
11    ...
12   }
13 }
```

When the *start* command is received, the *spawn* function is called, taking as arguments the callback function *data_capture*, the event type and the source and destination IP addresses. Event type source and destination are parsed from user input, as can be seen in the regular expression from line 6. The *spawn* function takes care of internal statistics (how many captures are running, what arguments were they called with etc.) and then executes *data_capture*, or changes the coderef to the timer event generation function when a timer event stream has been requested. In this case the *data_capture*, which initializes a pcap capture, does not have to be called.

```
sub data_capture {
 my ($nr, $event, $src, $dst) = @_;
 [...]
 # Initialise capture device
 my $dev = $user_dev || Net::Pcap::lookupdev(\$errbuf);

 # snaplen 1024 byte, promisc = 1, timeout 10ms
 my $p = Net::Pcap::open_live($dev, 1024, 1, 10, \$errbuf);
 [...]
 my $myf = "dst $dst and src $src and not port $port";
 if($event =~ /tcp/){
   $myf .= " and tcp";
 } elsif($event =~ /icmp/){
   $myf .= " and icmp";
 }
 [...]
 if(Net::Pcap::compile($p, \$ft,  $myf, 0, $net) != 0){
   print STDERR "error compiling filter\n";
 }
 if(Net::Pcap::setfilter($p, $ft) != 0){
```

```
21    print STDERR "error setting filter\n";
22  }
23  [...]
24  Net::Pcap::loop($p, -1, \&event, $usr);
25
26  while(1){
27    # do nothing
28  }
29
30  return 1;
31 }
```

The *data_capture* function opens a pcap capture device and prepares a capture filter which makes sure that the internal traffic between the collector and its client is not captured, and that only the traffic between the source and destination IP address is captured. Furthermore, the protocol to be captured can be specified. This is clearly protocol-specific code in the core program, but it is easy to encapsulate it so that the core program remains generic.

Filtering the traffic between two IP-addresses is done for several reasons: the measurement framework proposed concerns itself with end-to-end QoS, so only the interaction between two communication endpoints is considered. Furthermore, the less specific the event streams are, the more complicated the correlation process is as it becomes necessary to differentiate between all the endpoints involved in this communication. Since the implementation presented here serves as a proof-of-concept, it is preferable to keep the code simple and self-explanatory.

In line 24, the pcap capture loop is started with the callback function *event* as an argument. It serves as a wrapper function for the actual event function to be called. The *$usr* structure is used to pass the event type.

```
sub ip_evt {
  my ($usr, $hdr, $pkt) = @_;

  my $ip_obj = NetPacket::IP->decode(NetPacket::Ethernet::strip($pkt));

  my ($t1, $mt1) = gettimeofday;
  print DATA_OUT "i $usr->{'nr'} $cnt " .
        sprintf("%010d.%06d", ($t1 - $hdr->{tv_sec}), ($mt1 - $hdr->{tv_usec})) .
        " $hdr->{tv_sec}." . sprintf("%06d",$hdr->{tv_usec}) .
        " $ip_obj->{src_ip} $ip_obj->{dest_ip} $ip_obj->{flags} $ip_obj->{cksum}\n";

}
```

The event generation functions are in a separate file. Here, the *ip_evt* function will be discussed as an example. This function takes the *$usr* structure, the pcap header *$hdr* and the packet structure *$pkt* as arguments. The packet is parsed using the NetPacket::Ethernet and NetPacket::IP modules. Then, the event is output to the data channel using the output format that was discussed in section 6.1.1. The NetPacket::ICMP module had to be

	Reliability	Capacity	Integrity	Timeliness
ICMP	Every echo req has echo reply	PPS	–	RTT
TCP	Packet loss	PPS	Sequencing errors	–
IP	–	PPS	–	–

Table 6.1: Measurands and QoS base dimensions

patched, since it did not parse the complete ICMP header. A patched version of the NetPacket::ICMP module is included in the source file directory under *cecile/ICMP.pm* .

6.2 Correlator - Analysator

In this implementation, the correlator and analysator were realized in one program. There is no reason to separate the functionality, as no advantage is to be gained by having the correlator and analysator run on separate machines. The correlator-analysator features a user-friendly GUI and a text output to the console. It can be used interactively to see the analysis in real-time or the console output can be captured to a file.

Again, the core program is generic. The protocol-specific functions, that is the event stream correlation and analysis, are separate from the core code. The event parsing has not been modularized, but this would be easy to do.

6.2.1 Measurands

Table 6.1 illustrates the mapping from QoS base dimensions onto protocol-specific measurands that was chosen for the proof-of-concept implementation. The table is not exhaustive. The calculation of certain base dimensions was omitted to keep the implementation simple and the code mostly self-explanatory. A comparison with table 3.2 shows that the mapping discussed in chapter 3.3.2 is usable for implementation purposes.

6.2.2 Code

The code for the GUI setup will not be discussed in this section, as it is straightforward and does not contain any aspects relevant to the correlation or analysis of event streams. The following code shows how the event stream capture and processing are scheduled. *fileevent* is a Tk function which calls the callback function it gets as third argument every time there is new data waiting to be read on the socket. *repeat* is a Tk function which executes a callback function some milliseconds later. The time interval to wait between execution is given as first argument. Using the *repeat* and *fileevent* functions at the same time introduces a certain level of concurrency.

```
foreach(1..$C){
 $M->fileevent("DATA".$C, 'readable', \&${"read".$_});
}

$co_id = $M->repeat(($freshness || 500), \&correlate);
```

When the *read* function is called, it reads the next line from the data socket and parses the event stream into a Perl data structure. The data structure used is a buffer of fixed length. When the collector process starts delivering data, the correlator fills up the buffer until the maximum length is reached. After that, the buffer length is kept constant by pushing out the eldest event every time a new event arrives. Furthermore, every event stream has its own data structure: there is no mix between streams from different collectors, or between different streams from the same collector.

The *correlate* function, which is called at fixed time intervals, first calls the event-type specific correlation functions and then, accordingly, the analysis functions. I will now discuss one correlation function and one analysis function in detail to show how correlation and analysis are realized here.

```
sub icmp_correl_semantic {

  if($pi1 < 1){ return; }

  for(my $h = 0; $h <= $pi1; $h++){
    if($p_1->[3]->[$h]->{'c_c'} ne ""){
      next;
    }
    for(my $k = 0; $k <= $pi1; $k++){
      if( ( $p_1->[3]->[$h]->{'seq'} eq $p_1->[4]->[$k]->{'seq'} )
       && ( $p_1->[3]->[$h]->{'seq'} ne "" ) ){
        $p_1->[3]->[$h]->{'c_c'} = $p_1->[4]->[$k]->{'cnt'};
        $p_1->[4]->[$k]->{'c_c'} = $p_1->[3]->[$h]->{'cnt'};
      }
    }
  }

  return;
}
```

icmp_correl_semantic tries to find a semantic correlation between two ICMP event streams; that is, given two ICMP event streams recorded by the same collector machine, it tries to find the corresponding ICMP echo reply to each ICMP echo request. Since the collector uses a libpcap filter for each event stream, one stream contains only the ICMP echo requests and the other only the ICMP echo replies. The *icmp_correl_semantic* function iterates through both event streams and, if a correlation is found - that is, if the ICMP echo request sequence number of one event corresponds to the ICMP echo reply sequence number of an event in the other stream, a pointer from one event to each other is set. The event stream with the pointers forms the correlated event stream.

```
sub icmp_rel {

  $icmp_rel_c++;
  my $b = $icmp_rel_c % $MC;

  for(my $h = $pi1; $h > 0; $h--){
    if($p_1->[3]->[$h]->{'c_c'} ne ""){
      for(my $k = $pi1; $k > 0; $k--){
        if($p_1->[3]->[$h]->{'c_c'} eq $p_1->[4]->[$k]->{'cnt'}){
          dot("rel", $b, 20, "P");
          return;
        }
      }
    } else {
      next;
    }
  }

  return;
}
```

icmp_rel is the function to measure the reliability of ICMP. The analysis is very simple: an iteration cycles through the correlated event stream, beginning with the oldest event. If the event has a pointer set to an event in another stream, clearly the echo request was followed by an echo reply. Of course, should the correlation fail, a false negative is be displayed.

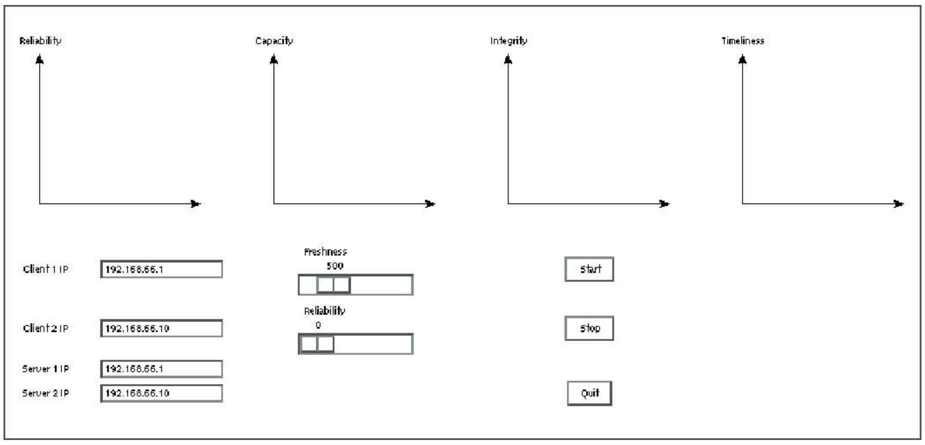
6.2.3 Context

The only type of context available in the proof-of-concept implementation is freshness. Here, freshness is not used to represent the total processing time, but rather the time interval to wait before each successive call of the correlator function. The more frequent the correlation is, the ”fresher” the results are.

The context switch is executed by canceling the last *repeat* call and executing a new one, with a different interval. The correlation does not happen over context switches, as the correlation is not a time-continuous process.

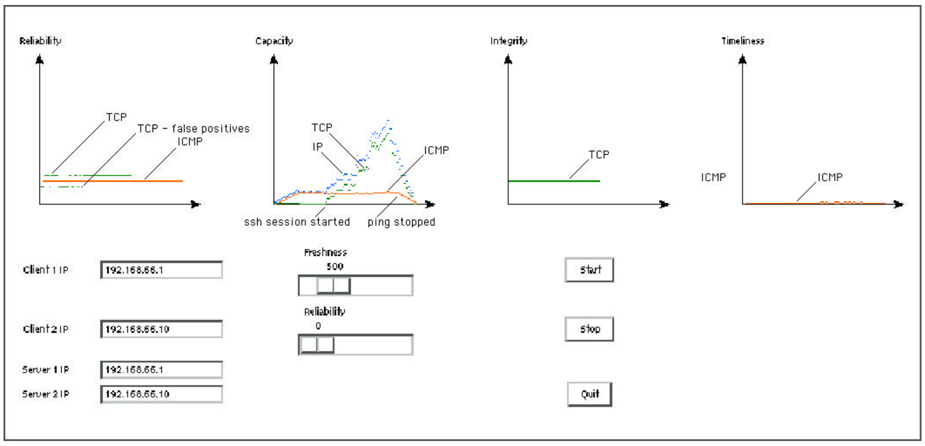
6.3 Application screenshots

This section presents some screenshots from the proof-of-concept implementation. The measurements were executed using collector servers on two computers, and one correlator / analysator. A *ping* and an *ssh*-session were used to produce test traffic.



Client 1 IP 192.168.66.1
Client 2 IP 192.168.66.10
Server 1 IP 192.168.66.1
Server 2 IP 192.168.66.10
Freshness 500
Reliability 0

Figure 6.1: Analysator screenshot



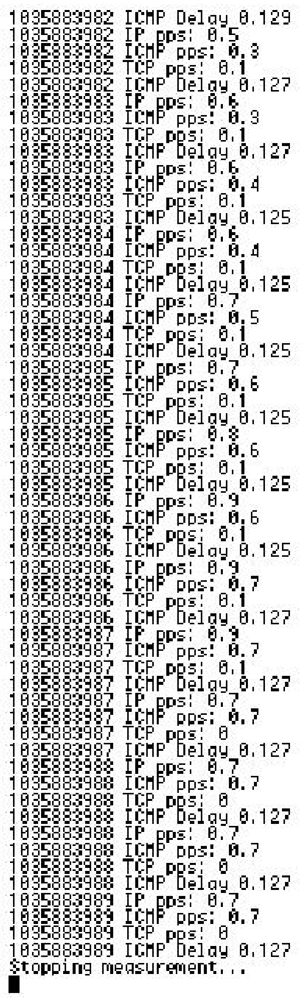
Client 1 IP 192.168.66.1
Client 2 IP 192.168.66.10
Server 1 IP 192.168.66.1
Server 2 IP 192.168.66.10
Freshness 500
Reliability 0

Figure 6.2: Analysator screenshot during measurement

Figure 6.1 shows a screenshot of the correlator / analysator at startup. In the lower left-hand-side, the collector server IP addresses and the IP addresses of the two clients between which the traffic is captured are displayed.

Figure 6.2 shows a screenshot of the correlator / analysator during a measurement. The display of the QoS base dimensions is color-coded: blue for IP events, yellow for ICMP events and green for TCP events. The "reliability" display shows many false negatives for the TCP packet loss; this is due to errors in the correlation. The "capacity" display shows first the capture of a *ping* (yellow line) and then some additional TCP traffic. It is quite visible how the total IP traffic is the sum of the ICMP and the TCP traffic. The "integrity" display shows TCP sequencing errors (none happened during that capture). The "timeliness" display shows the round-trip-delay of the ICMP packets.

Figure 6.3 shows a screenshot of the analysator output to the standard output during a measurement. The leftmost column is a timestamp. The



```
1035883982 ICMP Delay 0.129
1035883982 IP pps: 0.5
1035883982 ICMP pps: 0.3
1035883982 TCP pps: 0.1
1035883982 ICMP Delay 0.127
1035883983 IP pps: 0.6
1035883983 ICMP pps: 0.3
1035883983 TCP pps: 0.1
1035883983 ICMP Delay 0.127
1035883983 IP pps: 0.6
1035883983 ICMP pps: 0.4
1035883983 TCP pps: 0.1
1035883983 ICMP Delay 0.125
1035883984 IP pps: 0.6
1035883984 ICMP pps: 0.4
1035883984 TCP pps: 0.1
1035883984 ICMP Delay 0.125
1035883984 IP pps: 0.7
1035883984 ICMP pps: 0.5
1035883984 TCP pps: 0.1
1035883984 ICMP Delay 0.125
1035883985 IP pps: 0.7
1035883985 ICMP pps: 0.6
1035883985 TCP pps: 0.1
1035883985 ICMP Delay 0.125
1035883985 IP pps: 0.8
1035883985 ICMP pps: 0.6
1035883985 TCP pps: 0.1
1035883985 ICMP Delay 0.125
1035883986 IP pps: 0.9
1035883986 ICMP pps: 0.6
1035883986 TCP pps: 0.1
1035883986 ICMP Delay 0.125
1035883986 IP pps: 0.9
1035883986 ICMP pps: 0.7
1035883986 TCP pps: 0.1
1035883986 ICMP Delay 0.127
1035883987 IP pps: 0.9
1035883987 ICMP pps: 0.7
1035883987 TCP pps: 0.1
1035883987 ICMP Delay 0.127
1035883987 IP pps: 0.7
1035883987 ICMP pps: 0.7
1035883987 TCP pps: 0
1035883987 ICMP Delay 0.127
1035883988 IP pps: 0.7
1035883988 ICMP pps: 0.7
1035883988 TCP pps: 0
1035883988 ICMP Delay 0.127
1035883988 IP pps: 0.7
1035883988 ICMP pps: 0.7
1035883988 TCP pps: 0
1035883988 ICMP Delay 0.127
1035883989 IP pps: 0.7
1035883989 ICMP pps: 0.7
1035883989 TCP pps: 0
1035883989 ICMP Delay 0.127
Stopping measurement...
```

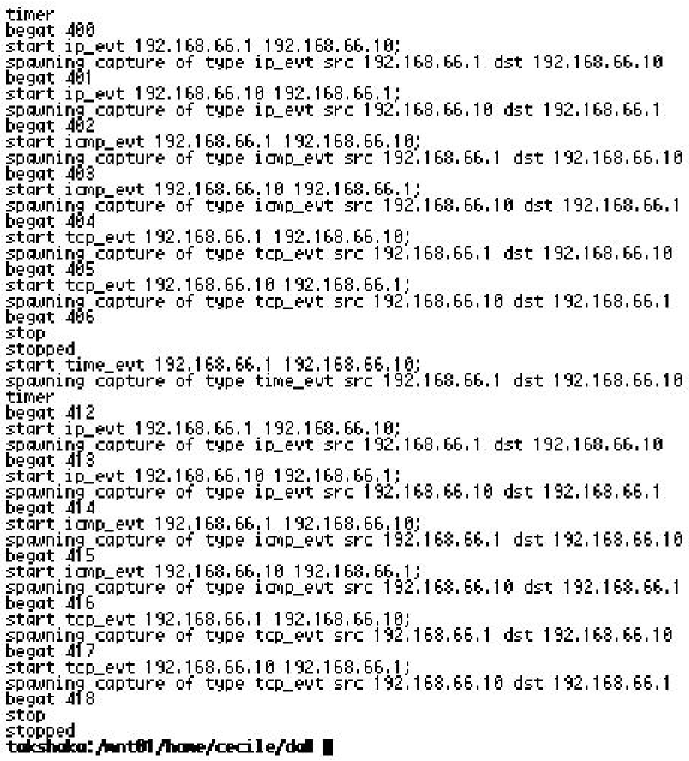
Figure 6.3: Analysator output during measurement

measurement was executed with a freshness value of 500ms. Accordingly, the trace file shows how each base dimension is computed twice per second.

Figure 6.4 shows a screenshot of one of the collector servers and shows how the different captures are started. The collector output to the standard output serves only diagnostic purposes.

6.4 Conclusion

The proof-of-concept implementation presented in this chapter has shown that the QoS measurement framework can be implemented in a very generic way. By using modular programming techniques, the core code is kept generic and can be reused with any type of protocol. The protocol-specific code is cleanly



```
timer
begat 400
start ip_evt 192.168.66.1 192.168.66.10;
spawning capture of type ip_evt src 192.168.66.1 dst 192.168.66.10
begat 401
start ip_evt 192.168.66.10 192.168.66.1;
spawning capture of type ip_evt src 192.168.66.10 dst 192.168.66.1
begat 402
start icmp_evt 192.168.66.1 192.168.66.10;
spawning capture of type icmp_evt src 192.168.66.1 dst 192.168.66.10
begat 403
start icmp_evt 192.168.66.10 192.168.66.1;
spawning capture of type icmp_evt src 192.168.66.10 dst 192.168.66.1
begat 404
start tcp_evt 192.168.66.1 192.168.66.10;
spawning capture of type tcp_evt src 192.168.66.1 dst 192.168.66.10
begat 405
start tcp_evt 192.168.66.10 192.168.66.1;
spawning capture of type tcp_evt src 192.168.66.10 dst 192.168.66.1
begat 406
stop
stopped
start time_evt 192.168.66.1 192.168.66.10;
spawning capture of type time_evt src 192.168.66.1 dst 192.168.66.10
timer
begat 412
start ip_evt 192.168.66.1 192.168.66.10;
spawning capture of type ip_evt src 192.168.66.1 dst 192.168.66.10
begat 413
start ip_evt 192.168.66.10 192.168.66.1;
spawning capture of type ip_evt src 192.168.66.10 dst 192.168.66.1
begat 414
start icmp_evt 192.168.66.1 192.168.66.10;
spawning capture of type icmp_evt src 192.168.66.1 dst 192.168.66.10
begat 415
start icmp_evt 192.168.66.10 192.168.66.1;
spawning capture of type icmp_evt src 192.168.66.10 dst 192.168.66.1
begat 416
start tcp_evt 192.168.66.1 192.168.66.10;
spawning capture of type tcp_evt src 192.168.66.1 dst 192.168.66.10
begat 417
start tcp_evt 192.168.66.10 192.168.66.1;
spawning capture of type tcp_evt src 192.168.66.10 dst 192.168.66.1
begat 418
stop
stopped
takshaka:/mnt01/home/cecile/dal █
```

Figure 6.4: Collector screenshot

encapsulated in separate functions and easy to reimplement and replace. Furthermore, by separating the collector from the correlator / analysator functionality, it is easy to reuse the collector for other tasks.

An interesting option for the implementation would be to use the IPPM framework to compute the QoS base dimensions, as this framework provides a comprehensive overview of metrics for IP networks. The IPPM framework defines a number of metrics that are relevant in the context of IP networks: for measuring connectivity (RFC2678 [41]), one-way-delay (RFC2679 [7]) etc. and was discussed in 2.2.2. Due to the modular composition of the implementation, adding new metrics is only a matter of changing the relevant correlation and analysis rules.

The collector and correlator exchange event streams in ASCII format and communicate using network sockets. This design decision was made to ease the implementation and make the debugging easier. Using remote procedure calls would make the event stream parsing that is done in the correlator unnecessary. Furthermore, the architecture could be implemented on a distributed platform like the Mobile Agent System Architecture (MASA) ([38]).

Chapter 7

Summary and conclusion

With today's networks becoming more and more service-oriented, a need for predictable and accountable service quality has arisen. Quantifying service quality is important not only to satisfy customer expectations, but also as a mean to describe traffic characteristics. The heterogeneity of today's networks and the lack of a unified QoS framework complicate the task of measuring and monitoring service quality. Existing approaches are too protocol-specific and lacking a generic framework.

This thesis proposes a generic QoS measurement and monitoring architecture, and presents a proof-of-concept implementation. The following requirements have been identified for the architecture: use a high-level and generic QoS definition instead of relying on technology-specific QoS concepts, hide the underlying network heterogeneity with unified views, and use unified methods and interfaces. Using a unified view on QoS is necessary to develop a generic QoS measurement architecture, as then the QoS concepts used are not limited to a specific networking technology and the architecture can adequately reflect the heterogeneity of today's networks. The implementation requirements are: keep the core code as generic as possible and use modular programming to make adding support for additional protocols as easy as possible. Furthermore, the implementation should also use unified views and unified methods to hide the underlying network heterogeneity from the end-user.

The QoS measurement and monitoring framework proposed consists of three conceptual blocks, the data collector, the data correlator and the data analysator. To model the interaction across the service or protocol interface, the concept of events is introduced using an abstract event class. Every protocol can then be modeled by constructing a suitable event class inherited from the abstract event class. The event stream, which is modeled as a composition of events, describes the interactions at the service access point.

Quality of service is modeled using four QoS base dimensions - reliability, capacity, timeliness and integrity - which were derived from the OSI QoS standard ([35]). Those base dimensions can be mapped onto protocol-specific measurands, but are very generic themselves.

The first of the three blocks, the data collector, captures the interaction at the service or protocol interface and generates a number of event streams, one for each protocol stack layer involved. Multiple instances of the data collector can be used concurrently to generate event streams. The data correlator takes those event streams and performs an event correlation to associate events from one stream with events from another stream. The correlation can be vertical (between protocol stack layers), horizontal (when correlating event streams which model the same data stream at different moments in time) or semantic (when correlating events according to protocol mechanisms, for example a TCP handshake).

Finally, the data analysator uses the correlated event streams to compute the QoS base dimensions, using, of course, the mapping onto network-specific measurands. Furthermore, the concept of context is introduced to model the interrelated conditions under which the measurement and analysis happen. Context can be captured during any step of the measurement, and is propagated to the data analysator, to provide the end user with context information. Furthermore, the end user can specify the measurement context (for example maximum error rates, sampling scheme to use etc.) which is then propagated back to the analysator, correlator and collector.

To determine how well and how easily the QoS measurement and monitoring architecture could be implemented, a prototypical implementation with basic feature set has been realized. This laid the groundwork for the proof-of-concept implementation which demonstrates the full possibilities of the QoS measurement architecture. The proof-of-concept implementation can be used in IP networks to generate event streams for IP, TCP and ICMP event types. Examples of horizontal and semantical correlation are included, and a mapping of the QoS base dimensions onto protocol-specific measurands is used to compute service quality. The proof-of-concept implementation is realized as a client-server application to be used as a distributed application, and features a graphical user interface for the analysator.

Both the measurement architecture and the implementation meet the requirements. The measurement architecture is generic, as its design is very abstract and hides network heterogeneity. The concept of QoS base dimensions provides a unified view on service quality by modeling service quality with abstract dimensions which are mapped onto protocol-specific measurands as needed. Events are derived from an abstract event class which defines a minimal set of attributes an event should have. Suitable event classes can then be defined for any protocol. Finally, the concept of event stream correlation is not protocol-specific. As long as a suitable mapping can be defined, event stream correlation is possible.

The core code of the proof-of-concept implementation is kept generic through modular programming. All the protocol-specific code - the event generation rules, the event correlation rules and the event stream analysis rules - are cleanly separated from the core program and can easily be replaced to support a different protocol set.

An interesting option for the implementation would be to use the IPPM framework to compute the QoS base dimensions, as this framework provides a comprehensive overview of metrics for IP networks. The IPPM metrics can be easily integrated into the implementation as only the relevant correlation and analysis rules have to be changed to accommodate another metric. Furthermore, since the three conceptual blocks of the measurement architecture can be implemented in a distributed fashion, it would be possible to use the MASA platform ([38]) as a foundation for the implementation.

Even though the proof-of-concept implementation consists of a very generic core code, any functional extension to support new protocols needs programming effort. Using a symbolic language to describe the protocol parameters to be captured, the correlation rules and the analysis rules would enable the end user to customize the QoS measuring tool very easily.

The set of four QoS base dimensions chosen is largely based on the OSI QoS standard, and has shown itself to be quite proficient through the theoretical as well as the more practical aspects of this thesis. It has yet to be seen whether the QoS model presented can be mapped onto a generic user-level QoS definition; but since the existing QoS frameworks lack a suitable definition of user-level QoS, this could not be assessed.

The concept of context is discussed using a practical bottom-up approach to identify a number of aspects relevant to the measurement architecture. Here, a more theoretical approach is needed. The concept of "context" has to be discussed in its relevance to measuring and monitoring. By identifying all parameters that contribute to measurement and error context and classifying them adequately, a suitable representation of measurement context can be gained. Furthermore, the interdependency of context parameters and the impact of that interdependency on context backpropagation have to be considered.

Finally, the discussion of event stream correlation has also raised a number of questions. Event stream correlation across the service interface is problematic, due to timing and synchronization considerations as well as technical details of the instrumentation of a protocol stack. It has yet to be determined how feasible an implementation of the proposed QoS measurement architecture is in this case. Finally, the types of correlation discussed were chosen with regard to their applicability and relevance. Other types of correlation might offer interesting insights and simplify the analysis.

Bibliography

[1] http://www.netsaint.org/.

[2] http://www.tinac.com.

[3] http://www.tcpdump.org/tcpdump_man.html.

[4] http://www.tcpdump.org/pcap3_man.html.

[5] FreeBSD Kernel source tree, http://minnie.tuhs.org/FreeBSD-srctree/FreeBSD.html.

[6] http://www.openh323.org/.

[7] G. Almes, S. Kalidindi, and M. Zekauskas. RFC 2679: A one-way delay metric for IPPM, 1999.

[8] G. Almes, S. Kalidindi, and M. Zekauskas. RFC 2681: A round-trip delay metric for IPPM, 1999.

[9] P. Almquist. RFC 1349: Type of service in the internet protocol suite, 1992.

[10] Uyless Black. ATM - foundation for broadband networks. Prentice-Hall, 1995.

[11] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. RFC 2475: An architecture for differentiated service., 1998.

[12] R. Braden. RFC 2205: Resource reservation protocol, 1997.

[13] R. Braden, D. Clark, and S. Shenker. RFC 1633: Integrated services in the internet architecture: an overview, 1994.

[14] A. Campbell. A quality of service architecture. Ph.D. Thesis, Computing Department Lancaster University, 1996.

[15] Andrew T. Campbell, Cristina Aurrecoechea, and Linda Hauw. A review of qos architectures — invited paper. In *Proceedings of the 4th International Workshop on Quality of Service (IWQoS)*, 1996.

[16] Robert Carter and Mark Crovella. Measuring bottleneck link speed in packet-switched networks. Technical Report 1996-006, 15, 1996.

[17] D. Chalmers et al. A survey of quality of service in mobile computing environments. IEEE Online communication Surveys, Vol.2, No.2, Second Quarter 1999, http://www.comsoc.org/pubs/surveys, 1999.

[18] Kimberly C. Claffy, George C. Polyzos, and Hans-Wener Braun. Measurement considerations for assessing unidirectional latencies. Technical report, 1993.

[19] Kimberly C. Claffy, George C. Polyzos, and Hans-Werner Braun. Application of sampling methodologies to network traffic characterization. In *SIGCOMM*, pages 194–203, 1993.

[20] Dave Clark. The quality management interface. Slides from IETF Meeting 31, Integrated service working group, 1994.

[21] A. Mauthe D. Hutchison and N. Yeadon. Quality of server architecture: monitoring and control of multimedia communications. IEEE Electronics and Communication Engineering Journal, 1997.

[22] Margaret A. Dietz, Carla Schlatter Ellis, and C. Frank Starmer. Clock instability and its effect on time intervals in performance studies. Technical Report Technical report DUKE–TR–1995–13, 1995.

[23] S. Donnelly. Passive calibration of an active measurement system, 2001.

[24] Andreas Fasbender. *Messung und Modellierung der Dienstgte paketvermittelnder Netze.* PhD thesis, RWTH Aachen, 1998.

[25] D. Ferrari. Client requirements for real-time communication services; RFC-1193. *Internet Request for Comments*, (1193), 1990.

[26] ATM Forum. ATM user-network interface specification, version 3.0, 1993.

[27] L. Fuente, M. Kawanishi, M. Wakano, T. Walles, and C. Aurrecoechea. Application of the TINA-C management architecture. In IFIP/IEEE International Symposium on Integrated Network Management, IV (ISINM'95), pages 424–435., 1995.

[28] K. Fukuda, N. Wakamiya, M. Murata, and H. Miyahara. Qos mapping between user's preference and bandwidth control for video transport. Proc. 5 th International Workshop on Quality of Service (IWQOS'97), Columbia University, New York, USA, Pages 291-302., 1997.

[29] M. Garschhammer and C. Neu. Using unified properties of data-flows as a generic basis to describe QoS parameters. Proceedings of the 9th workshop of the HP OpenView University Association, 2002.

[30] D. Grossman. RFC 3260: New terminology and clarifications for diffserv, 2002.

[31] IPPM Working Group. IPPM working group charter, http://www.ietf.org/html.charters/ippm-charter.html, 2002.

[32] R. Hauck. Architecture for an Automated Management Instrumentation of Component Based Applications. In *Proceedings of the 12th Annual IFIP/IEEE International Workshop on Distributed Systems: Operations & Management (DSOM 2001)*, pages 231–242, Nancy, France, October 2001. IFIP/IEEE, INRIA Press.

[33] Hegering, Abeck, and Neumair. Integriertes Management vernetzter Systeme. Konzepte, Architekturen und deren betrieblicher Einsatz. Dpunkt-Verlag, Heidelberg, 1999.

[34] ITU-T. General aspects of quality of service and network performance in digital networks. ITU-T Recommendation X.350, 1993.

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

[36] ITU-T. Packet based multimedia communications systems. ITU-T Recommendation X.323, 1998.

[37] M. Kara and M. Rahin. Comparison of service disciplines in ATM networks using a unified QoS metric. Proceedings of the 16th International Teletraffic Congress (ITC-16), volume 3B of Teletraffic Science and Engineering Series, pages 1137–1146, Edinburgh, UK. Elsevier Publishers., June 1999.

[38] B. Kempter. Entwurf eines Java/CORBA-basierten Mobilen Agenten. Master's thesis, August 1998.

[39] Jun Liu and Mark Crovella. Using loss pairs to discover network properties. In *ACM SIGCOMM Internet Measurement Workshop 2001*, San Francisco, CA, November 2001. ACM SIGCOMM.

[40] B. Lohner. Entwurf und Implementierung eines Leitungssimulators auf OSI-Schicht 2. Fopra, LMU, 2002.

[41] J. Mahdavi. RFC 2678: IPPM metrics for measuring connectivity, 1999. Obsoletes: 2498.

[42] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Winter*, pages 259–270, 1993.

[43] Webster Merriam. Merriam-webster online, 2002.

[44] David L. Mills. On the accuracy and stability of clocks synchronized by the network time protocol in the internet system. *Computer Communication Review*, 20(1):65–75, 1990.

[45] K. Nichols, S. Blake, F. Baker, and D. Black. RFC 2474: Definition of the differentiated services field (DS Field) in the IPv4 and IPv6 headers., 1998.

[46] V. Paxson. Towards a framework for defining internet performance metrics. Proc. INET '96, Montreal, 1996.

[47] V. Paxson, G. Almes, J. Mahdavi, and M. Mathis. RFC 2330: Framework for IP performance metrics, 1998.

[48] G. Quadros, E. Monteiro, and F. Boavida. A QoS metric for packet networks. Proceedings of SPIE International Symposium on Voice, Video, and Data Communications, Conference 3529A, Hynes Convention Center, Boston, Massachusetts, USA, 1-5 November 1998.

[49] A. Tanenbaum. Computer networks. Prentice Hall, 1996.

[50] TINA-C. Service architecture version: 2.0, 1995.