Department of Teleinformatics

MSc. Thesis Project Report

The Effect of Combining Network and Server QoS Parameters on End-to-End Performance

Work Conducted at the IBM Zurich Research Laboratory

Enikő Fey

Stockholm, December 22, 2000

Contents

Abstract

Application hosting is becoming a popular business. However Application Service Providers (ASPs) need to keep up with the increasing pace of the market. This implies that they have to provide infrastructure to an increasing number of clients, and at the same time give QoS guarantees to these clients. One solution for ASPs to both guarantee a certain service level (QoS) for their clients and keep expanding would be to have so many resources as to be able to provide more than the maximum aggregate need of their clients. This may turn out to be an expensive or even an impossible solution, hence sharing infrastructure between clients and offering some means of resource reservation, and using charging to insure that clients only reserve the resources they need, is an alternative.

However, it is not an easy problem to solve, particularly if the procedure for adding new clients is to be automated and the resources dynamically allocated. The ICorpMaker framework being developed at the IBM Zurich Research Laboratory offers a solution to the above named problems. In the ICorpMaker framework dynamic resource allocation is achieved by letting clients modify the amount of resources allocated to them in a simple manner, requesting more or less resources than their current allotment.

The difficulty in achieving the end-to-end performance the client desires, lies in the fact that it is not certain how modifying resource allocation at the network respectively server level will combine and affect the end-to-end performance experienced by the end users of the service. The aim of this thesis project was to study the correlation between different network and server QoS parameters and the resulting end-to-end performance by making measurements. The results obtained from these measurements give an answer to the question of how to change the network and server resource allocations, when a client's application does not perform in a satisfactory way and hence the client requests additional resources. Certain optimizations for the resource (re)allocation were also suggested based on these results.

Acknowledgements

Special thanks to those who helped me most: to my advisors at the IBM Zurich Research Lab, Sean Rooney and at KTH, Professor Gerald Q. Maguire Jr.; to my fiancee Thomas Stuedeli for his support in everything; to Christian Rohner for the significant help I got from him and to Raimund Brandt who also helped me get through smaller or bigger obstacles on my way.

Further thanks to the members of the research group at IBM: Anthony Bussani, Metin Feridun, Christian Hoertenagl, Jessica Kornblum, Jens Krause, and Stefan Pleisch.

Chapter 1

Introduction

1.1 Background

The exponential growth of the Internet is being fed by the fact that more and more companies use it for commercial purposes. Internet is becoming not only the means of communication via electronic mail, presenting information via web sites and alike, but also a place for "real business". However, setting up, maintaining and dynamically expanding a company's Internet-related infrastructure is not only a difficult but also a very costly task. In this regard, application hosting is an emerging market opportunity. Rather than concentrating on managing their Internet presence, companies should be concentrating on what their key businesses are. Application Service Providers (ASPs), companies taking care of other companies' IT needs, offer the solution. Briefly, we can define an ASP as a company that hosts the commercial activity (related to Internet services) of other companies by offering them the necessary infrastructure. The infrastructure typically consists of a number of servers and a network connecting them together, and to the ASP's edge routers.

As the market for application hosting expands, large ASPs will need to find a way to automate both the process of dynamically modifying the resources allocated to certain client companies, as well as the process of adding new clients. Another important issue for ASPs is to be able to satisfy the needs and wishes of their clients, with respect to the resources available to them, so that each client will be assigned enough resources in order for them to offer their services to their users.

The difficulty in satisfying client needs lies not only in the complexity of allocating the necessary amount of physical server- and network-resources at a given moment, but also in the fact that it is very difficult to understand what client demands are and what service level will bring them satisfaction. It is only the client that really knows what level of performance is satisfactory for his applications, but it would be unreasonable to request clients to specify exactly what kind of resources and how much of each they need. This has to be hidden inside the ASP, without the client needing to bother about it. The ideal solution is thus to delegate the responsibility of allocating resources to the clients, but without requiring them to know the details of the underlying infrastructure.

A potential solution is offered by the ICorpMaker [44], a dynamic framework for ASPs, being implemented by researchers at the IBM Zurich Research Laboratory.

1.2 ICorpMaker - A Short Introduction and Concept Definitions

The main idea behind the ICorpMaker is the following: one allocates a certain amount of resources to each client as they sign up for the service, guaranteeing an initial, coarse grained service level (QoS) chosen by the client. After the initial sign-up and resource allocation, clients are able to adjust the resources currently allocated to them using simple to use client control software. By *clients* we mean the companies that are content owners purchasing infrastructure from an ASP, while *users* are the population that then avails itself of the client's hosted content [44].

The resources allocated for each client consist of both network and server resources, as well as proxies. The allocation of each involves a separate process, but the two must be coupled together and coordinated in such a way that they yield an infrastructure satisfying the client's needs and can give certain performance guarantees. Having possibly different underlying technologies at layer 2 of the Internet suit, one needs to find a suitable abstraction that allows the separation of the "QoS guarantee" concept from the underlying technology. In the ICorpMaker, the abstraction used is that of virtual networks (VNs). A VN is a logical partition (subset) of the physical network that is assigned to a certain client, and appears to the client as if it was a dedicated physical network.

A similar abstraction is needed also for the server resources, ICorpMaker calls these *instant servers* (ISs). *Instant server chunks* (ISCs) are defined as some logical partition of a physical server, while an IS is defined as a set of ISCs. If a physical server has no partitioning mechanism, then it is considered to be a single ISC [44]. Server partitioning is a means towards efficient server resource usage, since allocating one physical server for each application that the ASP hosts on behalf of its clients would be quite a wasteful strategy. Thus, a client's applications are run on one or multiple ISCs. At the access point to the ASP network an IS proxy determines which ISC should carry out a given end-user request.

Now that we have defined the abstractions used at the network respectively at the server level, we can define the concept of an *Instant Corporation* (ICorp): the ISCs allocated to a client together with the access points of the ASP network and the virtual network interconnecting them forms an ICorp. The ICorpMaker is the framework for dynamically creating ICorps for the ASPs clients, allocating resources from the ASP's infrastructure [44].

1.3 Problem Statement

As stated above, at ICorp creation the clients are initially allocated a certain amount of resources guaranteeing a certain coarse grained QoS level. However, it is not certain that the allocated resources offer a satisfactory end-to-end performance for the client, as the notion of "client satisfaction" is extremely abstract, and difficult to map to resource allocation parameters. Also, even if we know the exact amount and nature of the low-level server and network resources allocated to a service, it is not certain what is the end-to-end performance experienced by the end users.

In the ICorpMaker framework, clients can request more resources using a simple API if their applications do not perform in a satisfactory way. However, the same problem exists for these new requests as for the initialization: it is not certain that allocating more physical resources has the desired effect on the end-to-end performance of the application. For example, increasing server resources for a certain application does not improve the end-to-end performance that the users experience, if there is not enough bandwidth to carry the additional traffic. Correspondingly, increasing bandwidth alone does not improve end-to-end performance if the respective server was already overloaded.

The aim of this thesis project was to carry out measurements in order to study the correlation between changes in network and server resource allocation parameters, and their combined effect on the resulting end-to-end performance. This implies that an example application for testing had to be found, along with a relevant way of end-to-end performance analysis for that application. In order to conduct the measurements, simple but still adequate network and server resource allocation setups were needed. Apart from general conclusions concerning the correlation between the different resource allocation parameters, results directly applicable for ICorpMaker were expected. Specifically, this consisted of answers to the question of how to change the resource allocation parameters for a client, when his application did not perform in a satisfactory way and hence he requested more resources.

1.4 Structure of the Report

The rest of the report is organized in the following way: in the next chapter, we take a look at the concepts and protocols that are relevant in the field of network resource allocation and guaranteed quality of service. Next, some products and research projects that can be used for logical server partitioning (even if not necessarily designed for that purpose, e.g. VMWare) are presented, as well as some possible charging models. After that, the ICorpMaker architecture is presented in more detail, building on the previously presented concepts, protocols, and products, and showing how they interact and are used in the ICorpMaker. The application chosen for the measurements was video streaming, using RealNetwork's software suit consisting of RealServer, RealPlayer and RealProducer. Therefore, a certain knowledge and understanding of these tools is needed. The second last section of the chapter discusses the most relevant issues in this respect. Finally, a few related projects are presented in the last section.

The following chapter, Chapter 3 begins with a survey of the work done in the field of performance analysis, concentrating on available tools. In the same chapter, the method that was used for performance analysis in this project is presented, preceded by a detailed motivation of the choices and description of the learning process that led to those choices.

Chapter 4 presents in detail the experimental setup, consisting of both network and server level QoS setups. Similarly as in Chapter 3, also here the motivation of the choices and the way that led to them are described. Thereafter, a summary of the setup and the different measurement scenarios is given. The chapter concludes with a brief discussion of how our setups correspond to real-life situations.

Chapter 5 consists of the evaluation of the results, presenting the analysis that followed the measurements. Different issues are discussed in this chapter - first, in what way the different network and server QoS levels of our setup interacted. Thereafter some ideas of how to use the results in the ICorpMaker project are presented.

Chapter 6 concludes with a summary of the most important results obtained, while the final chapter describes possible future extensions of the work. Finally, appendices and references follow.

Chapter 2

Background

In the beginning of this chapter, a short review of the concepts related to Quality of Service in the Internet of today is given. As the commercial demand for supporting QoS is constantly increasing, new concepts e.g. Service Level Agreements (SLAs) are emerging and becoming important. Along with other concepts, such as a formal definition of QoS, policies, and virtual networks, SLAs are also presented. Different approaches for network QoS provisioning such as ATM, Integrated Services, Differentiated Services, and the MPLS label switching technology are also discussed, to give an overview of existing architectures in the field of network QoS. The emphasis is on those aspects that are important in the context of ICorpMaker. After the review of solutions for networks, we take a short look at solutions for server partitioning, this being the approach used for offering "server QoS" in the context of this project. As charging for the resources that offer the required QoS is a very important issue, one solution for charging for network usage, and another charging solution for server resources are presented. After this overview the ICorpMaker architecture and its current implementation are presented in detail, coupling together the concepts and technologies presented previously into a coherent picture. Next, RealNetworks' software suite consisting of RealProducer, RealServer and RealPlayer is presented. As being the example application chosen for the measurements, an understanding of them is of significant importance. Finally, a short but not exhaustive presentation of related work is given.

2.1 QoS Related Concepts in the Context of this Thesis Project

2.1.1 Quality of Service (QoS)

Quality of Service (QoS) is one of the key concepts in the context of this thesis project. In the literature, several slightly different definitions of the concept are presented, but the main idea behind them is the same. In [7], QoS is defined as the "general ability of systems to differentiate between communications traffic in order to provide different levels of service" respectively as the term referring to "the performance seen by an end user or an Internet application across a network or the Internet". In [58], a more abstract definition, specific for network QoS is given, that is, QoS comprises those properties of the network that directly contribute to the satisfaction of the users, relative to the network's performance. Similar to [7], [19] defines QoS as the classification of packets in certain classes or flows that are to be treated in a differentiated way compared to other packets in the network elements.

In general, QoS aims to make the unpredictable, "best-effort" service of the Internet predictable, moreover, to give some guarantees in advance concerning the performance to be experienced by the end users. There are a variety of parameters that can be used as measures for this performance, such as bandwidth, packet delay, packet loss rates, load on the servers, and the experienced quality of different applications by the end users (e.g. quality of real-time video

presentation). It is important though to differentiate between the QoS related to the performance experienced within the network and the QoS related to the performance and resource allocation strategy of the servers the applications are running on.

There has been a lot of work concentrating on QoS issues related to the network, but this alone can not guarantee a certain satisfactory level of service for the end user, or, more explicitly said, a certain end-to-end performance. Both network level and server level QoS have to be taken into account in order to be able to give these guarantees. This thesis concentrates on this issue: to analyze how the two notions of QoS combine and effect the end-to-end performance.

Giving performance guarantees at the network and server levels requires the allocation of corresponding resources in the network elements respectively the servers. Note that in the context of this thesis project the terms "QoS" and "resource allocation" are considered equivalent.

2.1.2 Policies

Offering QoS guarantees implies that some users or some application data will get better service than others. One needs to make sure that this differentiated treatment is enforced in practice. Also, giving a better service level implies a higher price for the service, and one has to make sure that only those users or applications who are entitled to it get a better service level. Thus there is a need for some rules specifying who is allowed to do what, which requests should be satisfied and which are to be rejected. A *policy* is one or more rules that describe the actions to be taken when some specific conditions exist or some specific events occur. These policy rules i.e. the conditions and the respective actions to be taken, must be unambiguous and verifiable [19].

For example, if the rules state that real-time video traffic has a higher priority than data transfer (e.g. FTP packets), and the network is congested, the packets belonging to a data transfer should be dropped and not the ones belonging to the video stream. Other rules may state that customers paying more for some service than others are entitled to better service, so when competing for resources, these customers should be prioritized over others.

2.1.3 Service Level Agreements (SLAs)

SLA is a term that can actually be used in any situation where a provider-customer relationship exists. The SLA provides a way of defining the service required by the customers, it is a specification of what the customer is willing to accept and what the provider guarantees to give. It also defines what QoS level is to be provided, and the performance levels that the provider must achieve [58]. For ASPs, SLAs are becoming increasingly important, as the means of specifying to their clients the leased infrastructure and the performance it has to achieve.

The contents of an SLA typically includes at least the following: the type and nature of the service to be provided, the expected performance level of the service, the process for monitoring and reporting the service level, and the charges for the provided service [58]. In the ICorpMaker infrastructure some of these points are not explicitly defined - no complex SLA is needed between the ASP and its clients. As the clients can dynamically adjust their resource allocation, until they consider it to be satisfactory, no explicit specification of the performance level has to be given, only an initial service level specified. Also, no monitoring and reporting of the service level from the side of the ASP is needed, as the clients themselves have to monitor and decide whether the service level is satisfactory. However, charging models are important to include, and also a way of notifying the clients about the changing prices for the service as they modify the service level. Note that it is also up to the client to monitor the performance level after the modifications.

2.1.4 Virtual Private Networks (VPNs) and Virtual Networks (VNs)

The concept of a Virtual Private Network (VPN) is a little bit ambiguous, though it is quite widespread. There is a lack of general agreement on the definition and scope of VPNs and there are a wide variety of solutions that are all described by the term VPN. In [14], a general definition applicable in most situations is given: "VPN is defined as the emulation of a *private* Wide Area

Network (WAN) facility using IP facilities". That is, the many types of VPNs are all grouped into one concept. VPNs are modeled as connectivity objects in the context of [14], and the following requirements are placed on any VPN implementation:

- opaque packet transport: i.e. the traffic carried within a VPN should have no relation to other traffic on the IP backbone, and the addressing may also be unrelated to that of the IP backbone. The addresses used on the client's network may be non-unique, private IP addresses.
- data security: secure packet transport services have to be provided, this is the responsibility of the service provider.
- QoS guarantees, such as bandwidth or latency guarantees.
- tunneling mechanism: based on the first two of the above stated requirements, one can state that VPNs must be implemented through some form of tunneling mechanism.

The rest of the paper named above discusses the different types of VPNs and the respective requirements, proposing implementation mechanisms.

In essence, a VPN has to appear to the clients as if it was a *dedicated physical network*. In order to achieve this, differentiation between traffic crossing the network elements has to be done, respectively some policy has to be enforced. Also, clients have to be protected from interferance with and by other clients' activities.

However, most often people couple the concept of VPNs to that of security, that is, the securing of traffic in transit between trusted locations by means of secure protocols as SSL, IPSec [21], etc. In the scope of this thesis and the research it builds on, the central concept is not security, but rather quality of service. With the aim of avoiding confusion, a different term "virtual networks" (VNs) will be used in this thesis.

More explicitly, within the context of this project a VN is defined as some logical partition of a physical network (i.e. a subset of the network resources) that gives the clients the same conditions as if they were using a dedicated physical network with a well defined capacity and behaviour. Also, at the network nodes traffic belonging to a certain VN has to be distinguishable from other traffic and treated according to some specific policy [42]. In order to be able to use a high level of abstraction where minimum assumptions are made about the underlying technology, in ICorpMaker a control plane has been introduced where the mapping of the abstract VN concepts to different protocol layers is coordinated [44], [42]. This yields a coherent framework that allows the abstraction of VNs to be independent of the underlying technology and/or protocols used to support it, i.e. virtual circuits in ATM, Ethernet VLANs, MPLS tunnels, Differentiated Services. In the next section, we take a look at these technologies and protocols.

In the ICorp architecture, a slightly modified version Resource reSerVation Protocol (RSVP) is used as a signaling protocol between the elements in the control plane. For enforcing the network policy necessary for maintaining the VNs, COPS (Common Open Policy Service Protocol) is suggested as a candidate protocol in the ICorp architecture, however in the test network at ZRL (Zuerich Research Laboratory) it is not used at the moment. RSVP and COPS are also presented in the next section.

2.2 Relevant Technologies and Protocols Addressing Network QoS Issues

2.2.1 ATM - A Short Reminder

Asynchronous Transfer Mode (ATM) is a switching technology that utilizes fixed-length packets (called cells) to carry different types of traffic. ATM is connection oriented, that is, a connection has to be established between the communication endpoints before any exchange of data can be

done. ATM connections have two different levels: virtual circuits (VCs) and virtual paths (VPs). A virtual channel connection (or virtual circuit) is a connection between two endpoints that communicate with each other. A VC is a basic unit that carries a single stream. A set of multiple VCs that travel along the same path between the same endpoints are bundled together into a virtual path, yielding a virtual path connection (VPC); see Figure 2.1 for a scheme demonstrating the concept.

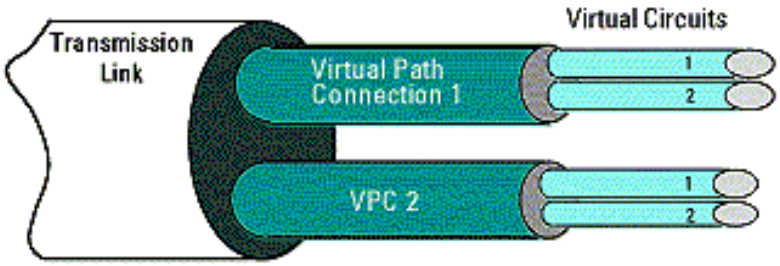

Figure 2.1: A VPC bundles together a set of VCs traveling along the same path [9]

Quality of Service is integrated into ATM – a specific service class is coupled with each VC, three of which support QoS guarantees. The service classes are the following:

- constant bit rate (CBR) – typical for audio and video traffic. Is characterized by a fixed peak cell rate (PCR) at which cells can be injected into the connection.
- real time variable bit rate (rt-VBR) – for example interactive compressed video applications use it. Data can be sent in to the channel at an average sustained cell rate (SCR) with bursts allowed up to PCR. There is also an upper bound on the cell delays and delay variation.
- non-real time VBR (nrt-VBR) - can be used for multimedia e-mail. It is similar to rt-VBR, but some characteristics differ, e.g. greater delay is allowed, but the cell loss rate is typically lower. Also, larger buffers can be used.
- available bit rate (ABR) - typically used for data traffic. A minimum cell rate (MCR) is assured for connections belonging to this class, but if they can send in cells at higher rate, they are provided with this information.
- unspecified bit rate (UBR) - provides the best-effort service we are used to from the IP world.

For CBR, VBR and ABR traffic, QoS guarantees are given, in form of bounds on some performance metrics such as cell loss rates, cell transfer delays, cell delay variation, etc.

Signaling is an important issue in the context of establishing connections, while traffic control is important for enforcing the policies regarding QoS. In ATM, an out of band signaling system is used, through the use of dedicated signaling virtual channels. In traffic control, there are a few important issues to be named. *Connection admission control* refers to the actions to be taken when a new connection is requested. If the available network resources are not enough for establishing a VC/VP connection with the required QoS and without affecting the QoS of the already established connections, the establishment of a new connection is refused. *Usage parameter control* has as it's main purpose the protection of network resources from malicious as well as unintentional misbehavior which could possibly affect the QoS of the existing connections. *Priority control* makes sure that the network respects the priorities specified in the cell header (by the cell loss priority bit) in case of congestion, i.e. when some cells need to be dropped. *Congestion control* is aimed to try reducing the network overload by taking some specific actions.

A short overview and good introduction to ATM is given in [58], and [60].

2.2.2 Ethernet VLANs

From the short description above, we see that ATM is a technology that has the possibility of virtual network creation – using a set of virtual channels with a specified QoS, virtual networks can be created. However, in the ICorpMaker architecture, it is important that VNs can be created in heterogeneous networks. Ethernet virtual LANs (VLANs) are a means to create VNs in networks with Ethernet as underlying technology.

One way to look at traditional LANs is to define them as a broadcast domain, i.e. a LAN is a network that contains a collection of hosts that will receive the broadcast messages sent by an other host on the same LAN. The physical connections determine which hosts of the network will actually belong to the same LAN. In contrast, a virtual LAN is a network that contains a set of hosts that are coupled together in order to form a logical group, depending on some management policy or communication patterns. The initial aim of VLANs was to limit broadcast traffic, such that only those hosts that are interested (i.e. logically belong to the same group of hosts) get relevant broadcast messages. One way to prevent broadcast traffic from traveling across the network is to use a router – these devices do not forward broadcast traffic. The use of VLANs reduces the number of routers needed, since VLANs create broadcast domains using switches instead of routers [57].

The Ethernet switch is thus enhanced with additional intelligence and functionality, so that it determines which frames should be forwarded between local segments belonging to certain VLANs. Many modern Ethernet switches support VLANs, and there are a number of ways to define and specify which hosts should be associated with the same VLAN. For example, rules can be based on the IP address of the host or the IP subnet the host belongs to, the MAC address of the host, the port of the switch the host is attached to, the protocol type field in the layer 2 header, higher layer protocols the host is using etc.

It is necessary to have a mechanism to indicate which frames traveling across the network belong to which VLANs, so that switches output the frames only on those ports that belong to the same VLAN as the incoming frame does. This is achieved by adding a tag header to the frame. The tag can also include a priority field, thus integrating QoS into VLANs is also an issue of increasing interest.

Further information about Ethernet VLANs is to be found in [57] and [45], whereas [18] is a draft standard defining the architecture that Ethernet switches supporting VLANs should conform to so that switches from different vendors can interoperate.

2.2.3 IntServ and RSVP

The concept of Integrated Services (IntServ) arose due to the need to support both real-time traffic and traditional, non-real-time traffic in the Internet. The problem addressed by the IntServ model is to provide support to real-time and time-sensitive traffic by means of integrating it with other traffic, such that the real-time traffic will benefit of QoS guarantees without significant modifications needed to be done to the devices supporting it. The solution is to make resource reservations in the devices along the path the packets will travel, using a signaling protocol - the Resource reSerVation Protocol (RSVP).

A few concepts need to be introduced and/or clarified when we look more closely at IntServ and RSVP operation. The concept of a *flow* is important, as reservation of resources in network devices are always made for flows. A flow is defined [7] as a stream of packets traveling hop-by-hop through the network from a specific (i.e. the originating) application on a specific host to a specific (i.e. the receiving) application on the destination host or hosts. More generally, a flow identifies a set of packets which should be treated in the same way. To each flow, a certain QoS is coupled. Each device needs to maintain information about each flow that traverses it. A *classifier* has the responsibility of determining to which flow each incoming packet belongs to, thus the device is able to treat the packet in accordance with the QoS specified for the flow that it belongs to. In each device (i.e. router), a *scheduler* determines the order in which incoming packets are processed and/or transmitted over an outgoing link. Scheduling is most often implemented with queues, so

that incoming packets are placed in different outgoing queues, depending on some well-defined policy. Often, multiple FIFO queues corresponding to certain priority classes are maintained, so that packets in the highest priority queue are processed and transmitted first, followed by the lower priority ones, in order.

Applications that wish to reserve network resources communicate their requirements to the network devices using the RSVP signaling protocol. RSVP is a simplex protocol, that is, resource reservations are made only in one direction, namely in the direction that the information flows. The sender, i.e. the application that is going to generate the data that the receivers are interested to get, is responsible for starting the communication by sending a PATH message to the receiver. The function of this message is to inform all the network devices along the path where the data will travel about the data flow that the sender will generate. The PATH message contains a traffic specification, with parameters describing the data flow that will be generated. These parameters include peak data rate in bytes per second, maximum datagram size, etc. The PATH message travels all the way to the receiver, recording all the hops it traversed. It also records characteristics of each hop in an advertisement specification, that is updated by each router on the path.

As soon as the receiver gets the PATH message, it sends a RESV (reservation) message, which travels along the reverse path recorded in the PATH message. Each RESV message carries a flow specification, which contains a reservation specification besides the traffic specification, that states what is the data rate the receiver would like to reserve along the path.

So the resource reservation is actually initiated by the sender, while the establishment of the resource reservation for each flow is initiated by the receiver. The network devices accept new resource reservations if they have enough resources, otherwise they refuse it. The admission control procedure depends not only on whether there are enough resources, but also on specified policies.

It is important to note that the route the PATH (and therefore the RESV) message takes is not influenced by the RSVP protocol, but is decided by the routing protocols that are running in the respective network. If the packets get rerouted, these messages will travel along the new path. PATH and RESV messages are periodically sent, routers that did not get any PATH or RESV message for a certain amount of time remove all state information about this flow and the respective reservations. Because of this behavior, RSVP is called a *soft-state* protocol – the state in the routers simply times out if it is not explicitly refreshed. However, state and reservations can explicitly be removed by the sender or the receiver by the use of tear-down messages.

In IntServ, two classes of service are defined: *guaranteed service* and *controlled load service*. The former guarantees the maximum delay that the packets will experience, based on information added to the PATH message's advertisement specification by each hop. The latter gives no explicit guarantees concerning delay or bandwidth, but simulates a best-effort quality of service not affected by excessive load. This is achieved by setting aside a specific amount of bandwidth for applications that request the service. Traffic that is not classified into one of the above named two classes, i.e. data from applications that did not use signaling to establish a state for themselves in the network, gets the traditional best-effort service.

In the context of the ICorpMaker, RSVP is important as a signaling protocol. Rather than building a new protocol for the purposes of the ICorpMaker, a modified version of RSVP is used for VN creation. This modified version of RSVP is named RSVP-ICorp in the context of ICorpMaker. For VN creation, resource reservation is essential; in Section 2.5 a description of how RSVP-ICorp is used to propagate control messages between both layer 2 and layer 3 network elements that participate in the given VN, can be found.

2.2.4 Differentiated Services (DiffServ)

Another way of differentiating between traffic and providing different traffic classes with different service is the proposed DiffServ architecture. As opposed to the IntServ model that uses signaling to establish a state in the routers in order for them to treat packets belonging to different flows differently, in DiffServ it is the packets themselves that are explicitly marked. The ease and simplicity of DiffServ lies in two factors: first, the fact that it uses the ToS (Type of Service) field in the IP header for marking the packets - in fact, this was the initial idea behind having a ToS

field in the IP header. Only the first 6 of 8 bits are used by DiffServ. Second, as opposed to IntServ, DiffServ is not and end-to-end protocol.

The architecture has at it's base the separation between core routers and access routers. A DiffServ domain is made up of the core routers (and the network connecting them together), and the access routers. Access routers are on the edge of the domain and connect it with customer networks and/or other ISP networks. Core routers are only connected to each other and the access routers, but not to routers from outside the domain. These core routers treat the packets in a differentiated way, depending on the value of the 6 bits in the ToS field. It is the duty of the access routers to fill in these bits with the right values according to some well defined policy. That is, sophisticated classification, marking, policing and shaping operations are needed at the domain boundaries, but inside the domain a relatively simple scheme is used. See Figure 2.2 for a scheme of the DiffServ architecture.

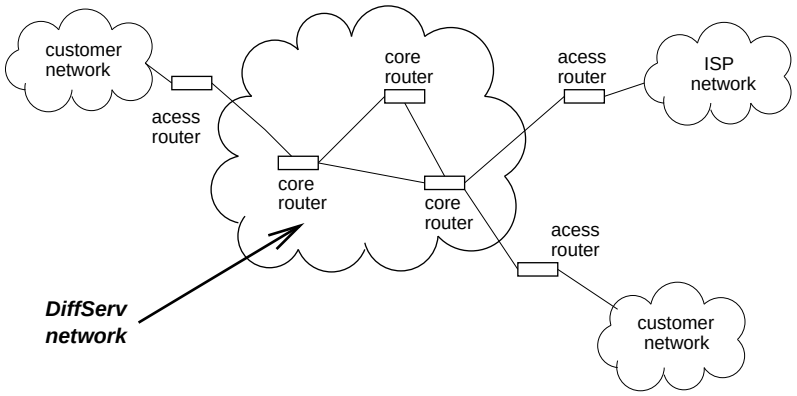

Figure 2.2: The DiffServ architecture

An access router needs to perform classification of packets – this is done based on the field values in the IP header (e.g. IP source and destination addresses, transport protocol, source and destination ports). Based on defined (policy) rules, each packet will be marked by setting the appropriate ToS bits, indicating the per-hop forwarding behavior on nodes in the core network. Access routers can also perform shaping of the traffic, letting only a specified amount of traffic belonging to the different service classes enter the DiffServ domain. Also, metering (measuring traffic properties) can be performed in the access routers. Core routers identify the incoming packets and the per-hop behavior (PHB) they indicate, and process these packets accordingly. As part of this differentiated treatment, different priority-based scheduling strategies for the packets belonging to different PHB classes can be used.

At the access routers, classification of packets is done based on certain policy rules, as stated above. Configuration of the access routers according to the actual policy rules in DiffServ is achieved by using a centralized model. In each DiffServ domain, a centralized policy server is maintained, providing a consistent policy configuration of all the access routers. The core routers can also access the policy server to obtain configuration information that will allow them to map the different PHBs to the appropriate scheduling behavior.

Based on the traffic differentiation performed by DiffServ, i.e. the different priority classes result in the appropriate treatment of the packets, assurances about the performance of the aggregate traffic can be given, but no end-to-end QoS is provided for a certain application flow from a certain source to a certain destination.

A short introduction to DiffServ is given in [58], while the Internet RFC 2475 [2] gives a detailed description.

2.2.5 MultiProtocol Label Switching (MPLS)

Traditional layer 3 routing implies that each router along the path a packet travels, will process a packet independently of the other routers' processing, and it will make a routing decision depending on the routing information that is currently available to it. Label switching is a technique that makes the processing done by the routers simpler by means of applying a label to each packet. Each router possesses a table telling the next hop based on the label that the packet has, and also a new label that has to be added to the packet (by switching it with the one that the packet has when it arrives) in order for the next router to be able to perform this switching procedure. In [46] and [3], the concept of a *Forwarding Equivalence Class* (FEC) is defined, as being the group of IP packets which are forwarded in the same manner, that is, over the same paths, with the same forwarding treatment. Based on the previous definition, a *label* in MPLS is defined as a short, fixed length value which is used to identify a FEC.

Similar to DiffServ, with its core domain and core routers and access routers, in MPLS we also talk of a core domain or network, composed of MPLS-aware devices. The edge routers have to perform the initial assignment of labels to the incoming packets that are going to cross the domain. This assignment is done essentially based on the packet's network layer header. Core routers need not do any analysis of the network layer header, as the already assigned label is used to identify the next hop. The labels can be used not only to identify the next hop (i.e., the FEC), but also to define a class of service for the packet. Thus a label may represent the combination of an FEC and a class of service [46]. In some cases, the label is part of an encapsulation header that is specifically defined for this purpose, while in other cases, the label may be integrated into the existing data link header.

Once they have been assigned a label, the packets will follow a virtual path inside the MPLS domain, called a *Label Switched Path* (LSP). That is, there is a unique mapping between each label and the corresponding LSP. There exist signaling protocols that perform label distribution inside the domain, such that each router involved in an LSP will know which label the incoming packets will have and which label it has to apply to the packets itself, i.e. what label swap it will have to perform.

MPLS offers an efficient way of explicit routing (i.e. source routing) through the labels that are assigned to the packets. This ability makes it especially useful for QoS routing, that is, routing in which the choice of a route for a particular stream is made in accordance with the QoS required for that stream [46]. This is achieved by mapping the required QoS level to an appropriate label, which will yield a route for which the requested QoS guarantees can be met.

In the ICorpMaker infrastructure MPLS labeling is a candidate for use within virtual networks - in network nodes, the traffic belonging to a certain VN needs to be distinguished from other traffic, as well as treated according to some policy. Thus each packet belonging to a certain VN has to have a label which allows each network node to treat the packets accordingly.

2.2.6 The Common Open Policy Service Protocol (COPS)

As we have seen, many actions/decisions that are to be taken by the network elements depend on some predefined policy. It is therefore important, that all the network elements are aware of the actual policies and are able to consistently enforce them. COPS offers a simple client-server, query-response model for supporting policies [6]. Policy information is exchanged between a policy server (Policy Decision Point or PDP) and its clients (Policy Enforcement Points or PEPs). The model does not make any assumptions about the policy server, but is based solely on the server returning decisions in response to policy requests. The protocol uses TCP as it's transport protocol, ensuring reliable exchange of messages between the policy clients and server. Also, message level security for authentication, replay protection, and message integrity is provided.

Recall that in RSVP, a router will accept a new resource reservation if it has enough resources, and if the resource reservation conforms with the policy applied in the network (e.g. this user is allowed to make this reservation in the current context). Thus, an example of a policy client or PEP could be an RSVP router. The PDP might be a network server directly controlled by the

network administrator who enters the policy statements.

PEPs will issue a request to a PDP when some specific event occurs; the context of each request will correspond to the event triggering it. COPS identifies 3 types of events: the arrival of an incoming message, allocation of local resources, and the forwarding of an outgoing message. Each of these events may require different decisions to be made. The PDP will respond with an appropriate decision, that will determine subsequently how the PEP will perform. PEPs can also send configuration requests to the PDP, which will respond with configuration data. The PEP will perform the configurations indicated in the response message, and will report back to the PDP. The server may later update or remove the configuration data by issuing new decision messages.

Internet RFC 2748 [6] provides a complete description of the protocol, further details can be found there.

2.2.7 Network Element Control Protocol (NECP)

NECP [4] is a protocol primarily dealing with server load balancing, it is a lightweight protocol for signaling between servers and the network elements that forward traffic to them. It provides methods for network elements to learn about the capabilities and the availability of the servers, and hints as to which flows can and can not be serviced. This allows network elements to perform load balancing across a farm of servers [4]. NECP is an application layer protocol, using TCP as its transport protocol.

In the context of the ICorpMaker, NECP does not play an important role, as signaling between the different elements in an ICorp is done with RSVP. However, it demonstrates the need for collaboration between servers and networks in order to optimize overall system performance.

2.3 Some Commercially Available Solutions for Servers

Until now, we have looked at protocols and technologies that were important in the context of network-level quality of service, specifically network resource allocation. In this section, we look at two comercially available solutions that allow some means of server resource allocation or server partitioning. Note that the term of "server partitioning" refers to logically dividing a single server into a number of independent servers, by allocating a certain amount of resources to each server partition.

2.3.1 Ensim's ServerXChange

In Ensim's ServerXChange Technical White Paper [8] a few challenges regarding server partitioning are stated, and the solution offered by their product is described. These challenges are, however, generally valid for server partitioning. As previously stated, for application service providers (ASPs) it is a cheaper and more effective solution to divide the resources of a single server (server partitioning) among several clients than to provide each client or each application with a separate server. The challenges or problems with doing that are the following [8]:

performance isolation : the presence of one customer should not affect the quality of service seen by the others. Thus if one customer's service becomes heavy loaded, the other customers should experience unchanged performance which conforms to their respective guaranteed QoS.

functional isolation : some applications assume that only one instance of that application is running on that host. When several instances are running simultaneously, independent control of each application instance has to be provided.

fault isolation : if several clients share the same server and one client's application crashes, this should not affect the other customers' applications. That is, customers should be protected from each other's faults.

address isolation : when intranet services are hosted on a shared server, it is not excluded that two or more clients will be using overlapping private address spaces. Thus isolation between the different customers' address spaces is needed.

Ensim developed a service deployment platform addressing the above named challenges. Their solution is based on three components, which are briefly described below.

By enhancing the operating system with a thin software layer, ServerXChange makes the partitioning of a shared server into several Private Servers (PSs) possible. This means that each PS will be allocated a certain amount of the server's hardware resources, such as CPU, memory, etc. This provides for the different sorts of isolation between the customers. The second component is a management console that makes monitoring, configuration and control of the PSs running on different machines possible. It also provides methods for configuring services on a PS. The third component is monitoring the state and the resources consumed by each PS. ServerXChange administrators can configure this component to trigger alarms when specified events occur. In addition, the ServerXChange platform offers the possibility of running application binaries in an unmodified form.

2.3.2 VMWare

VMWare is an emulation solution that allows running multiple operating systems at the same time, which appear as multiple virtual computers on a single PC. That is, from the user's viewpoint, a VMWare instance is almost like a regular computer with its own hardware resources (e.g. parallel and serial ports, disks, memory, network card, etc.) running a certain operating system and a set of applications. But VMWare is in fact just an application run within the operating system of the machine, from the viewpoint of that operating system.

The operating system that is initially installed on the machine is called the *host operating system*, while the one started as an application is called the *guest operating system*. At the moment, VMWare can be installed on hosts running Linux, Windows NT, and Windows 2000 as host operating systems. The installation is just like any other application's installation. However, during this installation configuration of the virtual machine is needed (including networking parameters, e.g. IP addresses). For the guest OS, a wide spectrum is provided, among these are Windows 95, Windows98, Windows 2000, Windows NT, DOS, NetWare, FreeBSD, and different Linux flavors [56], [59]. Figure 2.3 shows VMWare for Windows NT running Linux and Windows 2000 simultaneously.

Because VMWare runs as an application, that is, it appears as a regular process within the host OS, it is handled as any other process running within the host OS, as concerns resource sharing and allocation. In particular, in Unix host operating systems, a priority is assigned to each process (thus also to the VMWare instance running the guest OS), influencing the amount of CPU that the process will get. Also, the guest OS being a process, it can do no harm to any process belonging to the host OS if the guest OS crashes. These are quite nice features and are especially important in the context of the ICorpMaker.

There are two virtual machine configuration issues that we take a closer look at, memory usage and networking. For other issues, consult [56] and [59]. VMWare allows users to set the amount of physical host memory reserved for virtual machines. However, this amount tightly depends on the applications that are to be run in the virtual machine, on other virtual machines (and their respective memory usage) that will be running on the same physical machine, and on the other applications that are going to run on the same machine at the same time. To ensure a good overall performance of all applications running on the physical host, VMWare enforces a limit on the total amount of memory that can be used by virtual machines. Also, users can configure a parameter specifying the amount of memory that is reserved for all running virtual machines. This is in order to improve virtual machine performance. The amount of reserved memory used for a particular virtual machine varies in time, as the virtual machine runs - if multiple virtual machines run simultaneously, they will work together to manage the reserved memory [59]. Memory usage configuration along with the priority assigned to the process running the guest OS are interesting

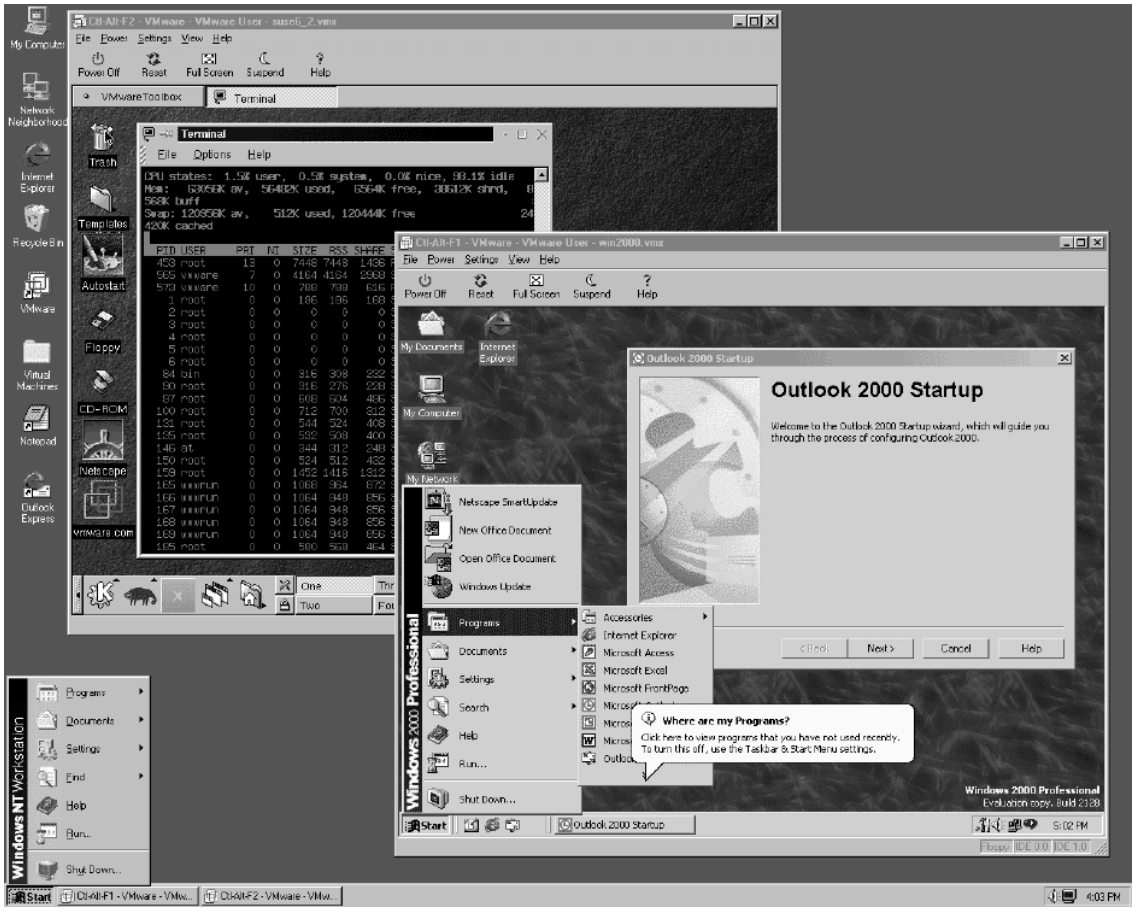

Figure 2.3: VMWare for Windows NT running Linux and Windows 2000 simultaneously [10]

issues when VMWare is used as a means of server partitioning - as we will see below, ICorpMaker uses VMWare for this purpose.

Each virtual machine can also have its independent network configuration, having its own IP and MAC address. There are four different types of networking offered by VMWare: no networking, host-only networking, bridged networking, and custom networking. In the first case, when no networking is used, the virtual machine is run isolated from the host operating system and any other (virtual) machines. If host-only networking is used, the virtual machine will be able to communicate with the host OS and other virtual machines running on the same host. Bridged networking implies that the virtual machine is bridged to the physical network by the host OS, which is acting as a bridge on behalf of the virtual machine. Custom networking is a cover for all configurations other than described above [59]. Networking is an important issue for functional and address isolation in the context of server partitioning.

In the ICorpMaker architecture, VMWare virtual machines are used to implement the Instant Server Chunks (ISCs). Though VMWare is not explicitly a server partitioning solution, it provides the necessary performance, functional, fault, and address isolation that is needed in the context of the ICorpMaker architecture.

2.4 Charging Models

The business of ASPs is of course not just to provide infrastructure for the commercial activities of their clients, but also to charge them - potentially, this could be based on the resources they are using. There are no standard ways of charging at present, but there is a lot of research being conducted in this area. In this section, we take a brief look at two charging models, one for network and one for server resources, these could be candidate models for use in the ICorpMaker.

2.4.1 Microsoft's Congestion Pricing for Congestion Avoidance

The main idea behind Microsoft's project is that if users are charged accordingly if their packets cause congestion, one can avoid or reduce congestion in the network [22]. The architecture builds on the concept of Explicit Congestion Notification (ECN) [40] that has been proposed as a means of indicating congestion instead of the standard mechanism of dropping packets used by TCP. In ECN, active queue management mechanisms detect congestion before some queue in a router overflows, and provide an indication of this congestion to the end nodes by setting a certain bit in the packet headers. The end nodes are thus expected to modify (i.e. decrease) their traffic rates.

In the model proposed by Microsoft, users would be charged based on the number of their packets that caused congestion, i.e. those packets that are marked by the ECN procedure. Also, they would have to pay fixed costs related to the infrastructure they are using. In the context of the ICorpMaker, this model can be used in order to prevent clients from requesting unreasonably much resources, which would affect the QoS seen by the other clients which compete for the same resources.

2.4.2 Xenoserver - Charging for Server Resources

The Xenoserver project [41] is mainly concerned with the issue of executing untrusted code (i.e. code supplied by an untrusted user), but also considers QoS issues and economic aspects. This latter includes charging for the resources used by the applications that are executed, as well as finding an economical and efficient way of placing the different services and data relative to each other in the network. Clearly, if entities that communicate with each other are located close to each other, this will both increase performance and decrease cost. We are however interested only in the charging issues related to resources in this section.

Xenoserver proposes a model where all resources are scheduled or allocated and respectively charged for separately. These include:

.

- CPU cycles spent executing an application
- CPU scheduling guarantees
- guaranteed response latency to external events (e.g. incoming messages)
- context switches caused by an application
- packets and bytes received and transmitted
- disk space rental charges
- disk block read and write usage

The disk device driver schedules the activity of the disk head itself, and accounts for usage by the clients. The filesystem server also accounts for operations on the metadata [41].

The above is interesting mainly to get an idea of what resources and operations charging models could include. What is even more interesting, but marginal in the context of this thesis project, is how all this is implemented. Xenoserver builds on an operating system called Nemesis, which has been designed to enable applications to receive quality of service guarantees for all the resources they require, enabling support for high quality multimedia and other real-time applications [27].

2.5 ICorpMaker Architecture - Detailed Description

As stated in Chapter 1 of this report, ICorpMaker offers a solution for large ASPs to create automatic procedures for adding more clients and allocating them the necessary resources, as well as a means for clients to dynamically adjust the amount of resources that are assigned to them. Changes in the resource allocation are to be made by the clients, but without requiring them to have an exact knowledge of the low-level resources that they are assigned. This aim is to be achieved by offering a simple API to the clients, where they can increase or decrease the network or sever resources allocated to them, and be notified about the different charges these modifications imply.

2.5.1 Network and Server Resource Allocation in the ICorpMaker

As an ASP leases infrastructure consisting of both network and server resources to its clients, the allocation of both of these resources has to be made in such a way that they give a satisfactory performance to the clients and the end users. Also, the allocation of these different resources has to be made in a coordinated way in order to offer a manageable infrastructure.

Network resource allocation can be achieved in different ways, as we have seen in the previous chapter. ATM virtual circuits can give a well defined resource guarantee to a certain flow of data, in IntServ it is also possible to precisely define the resources that will be reserved in the routers for each flow of data. In DiffServ, priority classes specified the different service levels that the flows get. It is not only in layer 3 devices where the possibility of resource reservation or traffic classification exists, but even in layer 2 devices, as we have seen in Ethernet VLANs with QoS support. Though networks are heterogeneous and the means of assuring QoS on them is heterogeneous as well, the net result of a resource allocation is the same, e.g. a certain amount of bandwidth between a set of hosts. The abstraction used in ICorpMaker to hide these various techniques that offer a guaranteed service is that of virtual networks. As explained in the previous chapter, a virtual network is a logical partition of the physical network that offers the clients the same conditions as if they were using a dedicated physical network, preventing them also from interference with the other clients. However, at the time of creation of the virtual network it is important to know what the underlying technology is, so that the desired resource reservations are possible to make. Thus there is a distinct control plane where the mapping between the abstract notion of a VN and the actual underlying technology is made.

The actual mapping is achieved by having different layers of abstractions within the control plane and message passing between each layer. So for example a general VN creation message on an ATM switch will be transformed into a general VP/VC description and passed down to a VN ATM adaptor layer. Here the necessary transformations are done and a corresponding description that is meaningful for the actual switching technology is passed down to the switch, where the appropriate reservations are committed. A similar procedure would be performed for an Ethernet switch, only the mapping would result in the corresponding VLAN creation message that the switch would make.

Server resource allocation is done in a similar way in terms of using an abstraction in order to hide the different solutions used to implement the actual server partitioning. The abstraction used here is that of Instant Servers (ISs). ISs are a set of Instant Server Chunks (ISCs), which in their turn are some logical partition of a physical server. As we have seen, there are a lot of different types of server resources that have to be allocated i.e. partitioned between the different ISCs that are running on the same server. Just as in the case of virtual networks, it is important that the ISCs offer the required service guarantees as well as the necessary isolation of the clients from each other. We have also seen two possibilities for server partitioning, but it is more difficult to deal with this issue than with network bandwidth partitioning. This is partly due to the fact that there are far more types of server resources than network resources (i.e. resources of network devices and the links connecting them) and partly to the fact that far less work has been done in this area until now.

As an ICorp (Instant Corporation) is created, a number of ISCs are assigned to it, as well as a

virtual network interconnecting the ISCs and the access points of the ASP network that the users connect to. At each access point, an IS Proxy is run in order to decide which ISC will have to carry out the users' incoming requests. This IS Proxy appears to the end user as a single physical server. Figure 2.4 shows an ASP network with three servers interconnected with two access points and a user by a virtual network, here we see that the VN is a partition of the physical network and an ISC is a partition of a physical server, as well as are the IS Proxies running on the edge routers (constituting the access points).

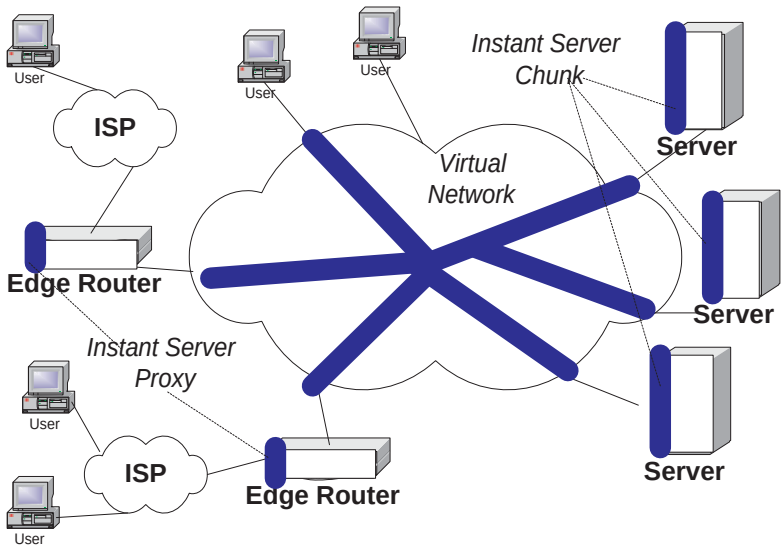

Figure 2.4: A sample ICorp infrastructure [43]

As packets belonging to different ICorps (i.e. different virtual networks) need to be distinguished in some way, some sort of labeling has to be used. A choice is to be made between using an implicit label, such as an IP address, or an explicit label, such as an MPLS label. In the ICorpMaker, implicit labeling was chosen, as it requires no modification in the forwarding functions or the data formats. Each ICorp is designated a private subnet, that is, each network element participating in the ICorp gets a new IP address belonging to the private subnet of the respective ICorp assigned to it. That is, a particular network device or server will possibly have a number of different IP addresses assigned to it, one for each ICorp that the device is participating in. This is done in order to achieve a high degree of separation between the different ICorps and to allow easy distinguishing between packets belonging to these ICorps at a certain network element.

2.5.2 Communication Patterns in the ICorpMaker

Using a simple web interface, clients communicate with an entity called a *gatekeeper*. At ICorp creation, clients can choose from a predefined set of resource allocation patterns, and also specify the set of applications that are to be run on their behalf, yielding an ICorpDescription. The gatekeeper uses this information gained from the client and the information that it has about the current state of the ASP's infrastructure to select a set of the physical resources (i.e. network and server) and create a so-called ICorpContract that describes the resources that are to be allocated to the new ICorp.

As mentioned above, there is a control plane where the mapping between the abstract notion of a VN and the actual implementation of the resource allocation is made. The control plane is made up of the set of ICorpControllers running on each of the configurable network devices, where the ICorpController is the software controller managing these network devices as described above.

This control plane is also responsible for propagating the resource reservation messages containing the ICorpContract to the network elements, this is done using a modified version of RSVP, called RSVP-ICorp in the context of ICorpMaker.

Thus, as part of the ICorpController, an RSVP-ICorp daemon is running. The gatekeeper chooses one of the end-points (i.e. access routers and servers that participate in this ICorp) to initiate an RSVP-ICorp path message, which is propagated to all the other end-points. The path message contains the ICorpContract. When receiving an RSVP-ICorp path message, the recipient network device's ICorpController checks if it is possible to commit the resource reservation request carried in the ICorpContract, and if it is possible to do that, it actually commits the reservation. After that, it forwards the message to the next device specified in the ICorpContract. The RSVP-ICorp messages are propagated not only to layer 3 devices but to layer 2 devices as well, RSVP-ICorp daemons are running on both types of devices. An ICorpController can also add state information to the ICorpContract, which may be useful for the next hop. When a server's ICorpController receives the path message and this server is specified in the ICorpContract, if it is able to support the specified ICorp, it creates a corresponding ISC and replies with an RSVP-ICorp reservation message. The reservation message eventually arrives back to the initiating end-point. As a result of the above described procedure, a tree-shaped VN is created, with the root being the initiating end-point. However, each end-point knows of all the other participating end-points, as this information was contained in the ICorpContract. Therefore, if an end-point has knowledge of a better route between itself and other end-points, it can send path messages on those routes thus establishing a more efficient VN topology.

A soft-state model is used, meaning that the reservations time out after a while and they have to be renewed periodically in order to maintain the ICorp infrastructure. Dynamic changes in the ICorp infrastructure are also made by sending RSVP-ICorp path and reservation messages between the different elements in the current ICorp. When there is no possibility to commit the reservations asked for (e.g. increased bandwidth, more processing power) because of resource shortage in the elements currently belonging to an ICorp, the whole ICorp infrastructure is reconfigured by adding more ISCs, changing the location of ISCs, adding new access points and alike. It is again the gatekeeper which decides what the new infrastructure has to look like. Checkpointing of applications before reconfiguration, process migration and alike are not yet included in the ICorpMaker infrastructure, but at later stages these will certainly be a part of it.

2.5.3 The Role of Clients in ICorpMaker

The aim of the ICorpMaker is to offer clients an infrastructure so that their applications perform satisfactory. But it is not easy to map the very abstract and ambiguous notion of the clients' satisfaction to the actual resources that are allocated to the ICorp belonging to them. Their level of satisfaction depends on many factors that the ASP can't possibly know about. Thus, the best solution is to delegate to the clients the responsibility of adjusting their resource allocations so that their applications perform as they expect. In ICorpMaker, the aim is to offer clients a simple API where they can increase or decrease their resource allocation.

It is up to the clients to do monitoring of their applications' performance, thus the clients will know when they need more resources or when they can release some. Clients are not supposed to have exact knowledge of the ASP's infrastructure, but rather to be offered a possibility to simply ask for "more", if their applications don't perform in satisfactory way. After they do a new request, it is again up to the gatekeeper entity to decide exactly what actions are to be taken.

Each time the clients attempt to modify their resource allocation, they have to be informed about the new charges. Charging is thus a means for the ASP to limit the amount of resources allocated to the clients – they need to weight how much they are willing to pay for more service.

2.5.4 The Current ICorp Implementation

In the ZRL (Zurich Research Lab) the test network consists of two Ethernet islands interconnected via ATM. The Ethernet islands consisting of a few switches each are connected to the ATM network

by AIX workstations acting as routers. Attached to the Ethernet switches are a collection of servers. At ICorp creation, the Ethernet switches create a VLAN for the corresponding IP private subnet. The routers update their routing tables correspondingly, as well as the ATM switches choose an appropriate virtual circuit with the required class of service. Controllers are run on the Ethernet switches but not on the ATM switches.

ISCs are implemented using VMWare, an ISC corresponds to a guest Linux OS started when receiving an RSVP-ICorp path message for ICorp creation. The Linux boot process is modified so that an ISC controller daemon is executed before the login shell is started. This daemon starts the binaries of the application specified by the client. The guest OS is started with a low priority and a small virtual disk. The priority of the guest OS can be increased by the ICorpController if it's CPU usage exceeds a certain threshold value for a given time period.

2.6 RealNetworks' Software

As already stated in Section 1.4 on page 4, the chosen application for the measurements was video streaming. This was due to the fact that video streaming, namely the video streaming software suite offered by RealNetworks has been used for testing and demo purposes in the ICorpMaker. It is to hope that the obtained results are not dependent of this software, but rather generally valid for video streaming applications. However a good understanding of the tools is needed.

This section briefly presents the most relevant issues about RealNetworks' software. I used RealServer as server software, RealPlayer as client software, and RealProducer for producing the video clips used for the tests.

2.6.1 RealProducer

RealProducer is a tool that converts standard audio and video into streaming media clips. It is possible to convert audio and video files of different formats, to record from media devices such as video cameras, or to broadcast and stream live content.

The term *streaming* was already used in the context of the report, but has not yet been thoroughly explained. Basically, the term describes the way data is sent. Traditionally, the whole data file (i.e. the clip) had to be sent to the user's machine, and only after all of the data arrived, the playout of the clip could begin. With streaming, users can view the clip almost instantly. At the beginning, an initial amount of information is sent, that is buffered and contributes to a smooth playback even in case of minor variations in the network bandwidth. After the data rate peak of the buffering period, information is sent at a constant rate.

RealProducer makes the creation of streaming media for different target audiences possible, in what concern the bandwidth requirements. There are six different target audience groups, that one can select and as a result, the produced clips will be adopted to a corresponding set of bandwidths. The "SureStream" technology makes it possible to create a single clip that is aimed at multiple target audiences, so that the clip will automatically switch to a lower bandwidth during poor network conditions [36]. This is denoted as a *downshift* in the context of the report. The different bandwith levels that a clip is recorded at correspond to the different *encoding levels* of the clip, that is, every encoding is done for a specific bitrate. The higher the resolution a stream has, the higher the bandwidth it needs to get the information through to the end user. For example, when selecting as target audience "DSL / Cable Modem" and "Corporate LAN", the clip is encoded for four different bandwith levels, namely 220, 150, 112, and 90 Kbps. The 220 Kbps encoding level corresponds to the best picture and sound quality, while the other encoding levels correspond to lower qualities. If during the playback the network conditions degrade, a downshift to a lower encoding level takes place.

There are several parameters that can be set or adjusted, when producing a clip with RealProducer, these include:

- recording source (e.g. existing file, media device)

- recorded file type (e.g. multi-rate SureStream, single-rate stream)
- audio format (e.g. voice only, voice with background music, music, stereo music)
- video quality (e.g. normal motion video, smoothest motion video, sharpest image video, slide show)

There is special support for live broadcasts as well, these are also produced using the SureStream technology. Also worth mentioning is that the files produced with RealProducer get the extension (filetype) `.rm`, that stands for "RealMedia".

2.6.2 RealServer

RealServer is the software used to stream the clips to the clients. Files that were produced with RealProducer as well as a few other media file types (e.g. `.avi`) can be streamed with RealServer. It is possible to request the playout of both live content (produced as "live broadcast" with RealProducer) and stored material.

The protocol used for the exchange of streaming information is the Real Time Streaming Protocol (RTSP) [48], this protocol makes it possible to exchange messages such as "play", "stop", "fast forward", etc. In Section 3.2.3 on page 33 a more detailed description of the RTSP protocol along with a description of other protocols used between RealServer and RealPlayer is given.

There are a few parameters that can be set for RealServer. These are contained in a configuration file. Under Unix, RealServer is started from the command line, where a configuration file name must be given as an argument. The parameters specified in the configuration file include:

- important path information (e.g. log files)
- port numbers (e.g. RTSP port, HTTP port, monitoring ports, administration port)
- different passwords used in conjunction with RealServer, and other authentication parameters
- logging parameters
- MIME types
- caching parameters

These parameters and a few more can either be changed directly in the configuration file, or alternatively, through an administration web interface. Detailed documentaion about RealServer can be found in [34].

2.6.3 RealPlayer

RealPlayer [35] is the corresponding client software for RealServer, and the third product from RealNetworks [38] that was involved in this thesis project. The most important issues about RealPlayer in the context of this project are some parameters that can be set in RealPlayer (i.e. connection speed, transport protocol) and the statistics shown in the "Statistics Window".

Under the menu point "Preferences - Connection", the speed of the end user's network connection has to be specified. The RealServer will send the clips at a corresponding encoding level. Both a normal bandwidth and a maximum bandwidth level have to be specified. In "Preferences - Transport", the transport protocol that is used must be given. Checking the check-box "Automatically select best transport" basically always results in the use of RTSP for the exchange of streaming information (e.g. "play", "stop"), the use of RDT for the data transport (i.e. video and audio packets), and UDP as transport protocol. When selecting "Use specified transports" and "RTSP settings", one can specify if and when TCP or UDP are used as transport protocols. Though the use of UDP is most common for video streams, especially for live broadcasts, TCP

is often used when a firewall is to be traversed. UDP is better suited to such applications due to it's lack of an acknowledgment mechanism - most important is not that all packets arrive to the player, but rather that those which arrive, come in time to be played out. In the context of this thesis project, it is also UDP that is most relevant as transport protocol. All the results are based on measurements done with UDP, though some tests with TCP were also conducted.

An even more important issue about RealPlayer is the "Statistics Window". There are several types of statistics shown in this window, the most interesting information for us was the current encoding level of the received clip and the current bandwith, i.e. data rate at which information arrives to the RealPlayer. The information showed in this window was very important for the performance analysis that I did, as is described in the next chapter.

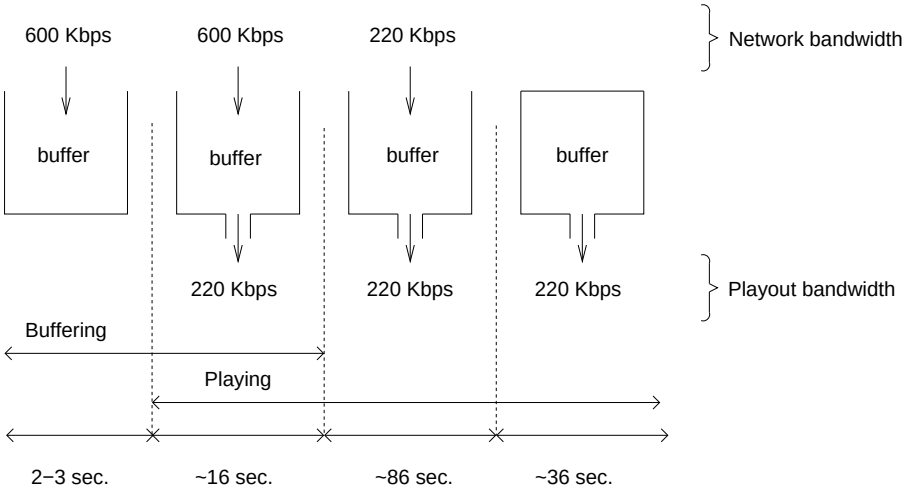

Figure 2.5: The buffering performed by RealPlayer, in case of a normal play at the 220 Kbps encoding level, and UDP as transport protocol. After a short buffering-only time, the playback begins. The buffering stops after about 16 more seconds, whereas at the end of the playback all data comes from the buffer.

A quite important issue concerning RealPlayer is the buffering, demonstrated on Figure 2.5. There is a buffering period at the beginning of a playback, during which the data rate sent by RealServer is quite high (about 600 Kbps [1]). Playback begins after about 2-3 seconds, and the buffering continues for about 16 seconds in the normal case, for the clip I used for my tests. It is due to this buffering, that even in cases when the effective data rate that arrives to RealPlayer is not enough to keep an encoding level of let's say 220 Kbps, the playback can still be done at this level, until the buffer is emptied. Also, it is due to this buffer that the flow of data stopped at about 1:42 minutes (102 seconds), for a clip that was 2:18 minutes (138 seconds) long. This is because the buffer size is a bit more than one megabyte. We will see later that this buffering property is a very important one, as it significantly affects the results obtained.

Another concept that needs to be explained more in detail is "downshift". As stated in conjunction with RealProducer, using the SureStream technology, a single file is created that is encoded at different levels (e.g. 220, 150, 112, and 90 Kbps). An automatical shift to a lower bandwidth occurs, when the arriving data rate is not high enough for the respective encoding level. This is denoted *downshift*. What the user sees, is that the picture quality changes, the motion is not as smooth as before, and the encoding level and the incoming data rate as shown in the Statistics Window are also less. Downshifts can occur from a high encoding level to the next lower encoding

[1] This is true when UDP is used as transport protocol. A similar buffering is done in the case when TCP is the used, but the examples here refer to measurements done with UDP.

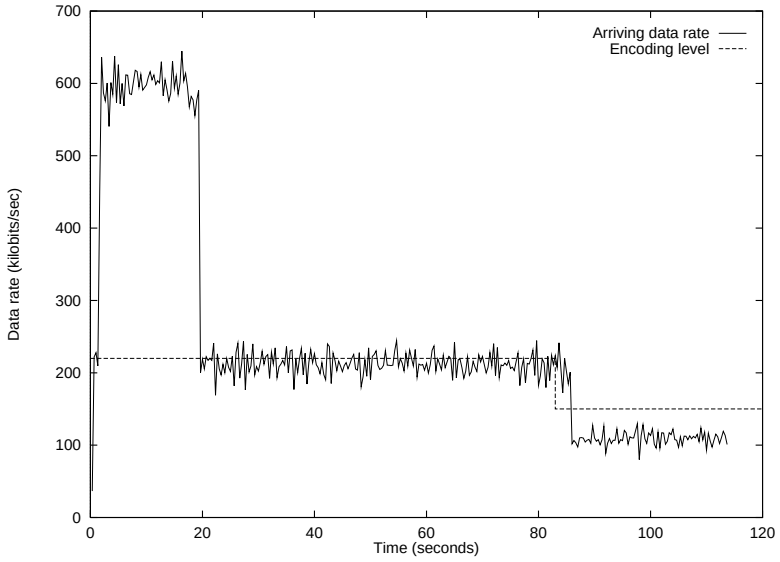

Figure 2.6: Data rate versus time plot, showing the arriving data rate and the corresponding encoding level in the case of a playback, when UDP is used as transport protocol. A single downshift from the 220 Kbps encoding level to the 150 Kbps encoding level occured at 83 seconds.

level, or directly to an even lower level, i.e. from 150 Kbps to 90 Kbps directly, without using the 112 Kbps encoding level. Similarly, *upshifts* can also occur in the case when the arriving data rate is high enough for the transmission of a higher encoding level than the current one. That is, if the current encoding level was 112 Kbps, but the data was arriving at a rate of 200 Kbps, an upshift to the 150 Kbps encoding could follow. An example of a dowhshift is shown on Figure 2.6, here a downshift from the 220 Kbps encoding level to the 150 Kbps encoding level occured at 83 seconds. The fact that the data rate decreased slightly later than the downshift occurred, as well as the fact that the arriving data rate after the downshift was smaller than 150 Kbps are both due to the buffering performed by RealPlayer.

2.7 Related Work

In this chapter, we have so far looked at different issues that are necessary as a background for understanding the rest of the work and the report. This final section presents a few projects that are related to the work done in the context of ICorpMaker and to my thesis project.

2.7.1 Traffic Engineering for Quality of Service in the Internet, at Large Scale (Tequila)

Tequila [51] is a project within the Information Society Technologies (IST) Programme, a major theme of research and technological development within the European Union [17]. The Tequila project started January 1st 2000 and runs until June 2002. The overall objective of the project is "to study, specify, implement and validate a set of service definition and traffic engineering tools to obtain quantitative end-to-end Quality of Service guarantees through careful planning, dimensioning and dynamic control of scaleable and simple qualitative traffic management techniques within the Internet" [51].

In the context of the project, the term Service Level Specification (SLS) is used with the same meaning as we defined Service Level Agreements (SLAs) in the context of this report. This concept

seems to play an important role in the Tequila project, as among it's key objectives we find [51]:

- "study the issues behind, develop architectures for, and propose algorithms and protocols to enable: negotiation, monitoring and enforcement of Service Level Specifications between service providers and their customers and between peer providers in the Internet."
- "develop a functional model of co-operating components, related algorithms and mechanisms to offer a complete solution for intra-domain traffic engineering to meet contracted SLSs in a cost effective manner."
- "develop a scalable approach, architecture and set of protocols for interdomain SLS negotiation and QoS-based routing to enforce end-to-end quality across the Internet."

The project's expected results include, among others, a validated framework for the provisioning/definition of end-to-end QoS through the Internet, a validated framework for end-to-end QoS monitoring and validated intra- and inter-domain traffic engineering methods, tools and algorithms deployable in DiffServ capable IP networks.

As opposed to the ICorpMaker, where the aim is to provide QoS guarantees independent of the underlying techologies, it seems that in the context of the Tequila project the DiffServ architecture plays a major role. Also, the main objectives of the two projects are related but not overlapping. The ICorpMaker aims at dynamic and automated ICorp creation where resource allocation can dynamically be changed in order to offer a satisfying QoS for the end users, whereas Tequila is neither concerned with dynamic changes in resource allocation nor with automation of the process of actually allocating the resources.

2.7.2 Adaptive Resource Control for QoS Using an IP-based Layered Architecture (Aquila)

Aquila is also a project within the IST Programme of the European Commission within the Fifth Framework Programme, just as Tequila. It also focuses on the QoS architecture of the Internet, and there is a high probability that the two projects will have a lot of information, technology, and knowledge exchange. According to the project's abstract, "Aquila defines, evaluates, and implements an enhanced architecture for QoS in the Internet. Existing approaches e.g. Differentiated Services, Integrated Services and label switching technologies will be exploited and significantly enhanced, contributing to international standardisation" [1].

The Aquila project has more similarities with the ICorpMaker, as one of it's objectives is to enable dynamic end-to-end QoS provisioning in IP networks for QoS sensitive applications. Some other objectives, like the development of an architecture with dynamic resource and admission control and the development of distributed QoS measurement infrastructure, are also issues that are relevant in the context of the ICoprMaker architecture.

Similar to Tequila (and opposed to ICorpMaker), the Aquila project also takes the DiffServ architecture as a starting point. That is, extensions of DiffServ will be made in order to avoid statically fixed pre-allocation of resources (i.e. enable dynamic adaptation of resource allocation). This will be achieved by adding a new layer (RCL - Resource Control Layer) above the DiffServ network to provide dynamic access to QoS services. The RCL has three tasks: to monitor and control the resources in the network, to control access to the network by performing policy control and admission control, and to offer an interface of this QoS infrastructure to the applications. The idea of adding a control layer where the resource allocation is to be controlled is also one that appears in the ICorpMaker infrastructure as well. However, the major difference is that in the latter project, this layer controls (e.g. control messages are exchanged between) both layer 2 and layer 3 devices.

Apart from the proposed QoS architecture, built on DiffServ enhanced with the RCL layer, Aquila focuses on other issues as well, namely on QoS traffic studies and engineering, QoS aware applications, and user services and distributed QoS measurements.

Chapter 3

Survey of Performance Analysis Tools and Methods

According to the problem statement in section 1.3 on page 3, the project required the setting up of different scenarios with different resource allocation patterns and carrying out end-to-end performance measurements for each of the scenarios. It follows that an appropriate performance measurement (analysis) tool was needed for the test application.

The first section of this chapter presents a few selected tools that I found during my search for available performance analysis tools. The tools presented were either interesting as candidates for performance analysis and monitoring in the project, or are representative of tools that actually are available. I reviewed a lot more tools than presented here, among others were some tools for monitoring the operating system behavior, or monitoring systems based on performance assertion checking. However, I considered these other tools to be irrelevant in the context of this project.

As none of the tools I found were appropriate to directly be used for performance analysis in this project, I had to find my own way around the problem and develop an own performance analysis method. As this involved a long process of learning and trying out different possibilities and ideas, the whole process is described in the second section of this chapter.

3.1 Search Results - Presentation of a Few Selected Tools

I conducted my search using Google [11], and I also looked on the web sites of the leading companies in networking and that of the universities that are most active in research in this area. I concentrated on three issues: end-to-end performance analysis, performance monitoring of servers and network. I also examined a handfull of articles published in [29] with the subject "Network Traffic Measurements and Experiments".

The first three tools presented below are comercially available products. I present these three from among the many becuase they use three different methods for performance analysis and monitoring. There are however other products available that use similar principles. The other three presented tools are not comercially exploited. I present them because I found them interesting as well as considered using them directly in the early phase of the project.

3.1.1 Grid Performance Working Group

[13] is the performance working group of the Grid Forum [12]: "the Grid Forum is a community-initiated forum of individual researchers and practitioners working on distributed computing, or *grid* technologies. Grid Forum focuses on the promotion and development of Grid technologies and applications via the development and documentation of best practices, implementation guidelines, and standards with an emphasis on rough consensus and running code". The Performance Working Group [13] aims to develop a general format to interchange data between different performance

monitoring and analysis tools, however the first steps towards this goal consist of establishing a taxonomy of existing tools.

An example tool found there is **NetLogger** [16], a toolkit for distributed system performance analysis, that monitors the behavior of all elements of the application-to-application communication path. That is, it combines network, host and application-level monitoring in order to provide a complete view of an entire system. The NetLogger toolkit consists of three components: an API, a set of event log collection and sorting tools, as well as a tool for visualizing and analysis of the log files. For the log files, a common log format, proposed as an IETF draft standard, is used. To produce event logs, applications are augmented with calls to the NetLogger API at critical points in the code. In such a way, each time a selected event occurs, a record is written to a log file. Logging is automatically done by sending the log entries to a chosen host and port, where a daemon writes them to a file on the local disc. Log entries can also be generated by system event monitoring services. The collected log event data is presented via a graphical visualization tool. Currently, applications written in Java, C, C++, Perl, Python, and Fortran can be augmented with calls to the NetLogger API. The software is downloadable from the NetLogger homepage [28], documentation is also available from the same page. This is an interesting tool, however it was not applicable in my case as I did not have access to the test applications' source code.

3.1.2 Lucent Networkcare: VitalSuite

Lucent offers a set of tools for monitoring and performance analysis of applications, networks, business processes etc. inside an entire enterprise. The tools are gathered into the VitalSuite [55] product family. There are four main parts in the product family, respectively:

VitalNet - is the tool for network performance monitoring and management. It gathers and aggregates performance data from routers, switches, WAN links etc.

VitalAgent - provides end-to-end performance analysis, as seen from the end user's perspective

VitalAnalysis - uses data collected by VitalAgent to provide summarized reports about network and application performance

VitalHelp - is a fault detecting and troubleshooting tool, faults and errors are reported by VitalAgent

According to the whitepaper [54], VitalAgent aims to measure performance as seen by the end user, and locate the possible performance bottlenecks along the end-to-end path. It is based on TCP session flow analysis, the measurements are done at the end-user side, outliers (i.e. measurement data that have a significant deviation relative to other measured data) are filtered out in order to gain a more realistic view. Also, many parameters are characterized based on their variability. Session flow analysis is done based on TCP packet inter arrival times, retransmissions and alike. The data obtained is "transaction time measurements (with breakdown of transaction time into client, network and server components); congestion magnitude; the bottleneck link speed from the server to the client; end-to-end packet loss; server throughput estimation; path length (number of hops); service provider domain identification". [54] gives a more or less detailed description of the calculation methods. Also, possibilities to traceroute and to display captured packets exist, they are however not as powerful as other tools (e.g. tcpdump, traceroute).

The VitalAgent software is downloadable from the homepage [55]. After some testing, I found that the information provided by it was not exact enough. Also, as I concentrated on UDP streams in the project, this tool was not appropriate.

3.1.3 DeskTalk: Trend

DeskTalk's network performance analysis and reporting solution, TREND [52] is being integrated into Nortel Network's products, however it still exists as an independent product. Being one of

the many which use SNMP and RMON for their performance data collection, it is worthwhile to take a look at it. The product is built up in a client-server fashion, it basically consists of three main building blocks:

- collectors, which are responsible for polling devices and storing data in a relational database
- servers, which are storing and processing the collected data
- clients, which provide (web) interfaces for manipulating data and displaying different reports based on statistical analysis

The software aims to provide network managers with a solution for managing service levels, optimizing the current network and the use of it's resources and planning future growth. In other words, it makes it easier to predict long term trends in the network as well as to identify existing problems [52].

Trend uses SNMP for retrieving MIB data from various devices supporting SNMP. However, data can also be collected using non-SNMP mechanisms, through extracting proprietary management information from proprietary management systems and importing the data to Trend. RMON data conforming to the standards for RMON1 and RMON2 is also collected. The data is stored in a relational database, which makes it easier to retrieve data from various applications different than a TrendClient. Statistics are done for hourly, daily, weekly and monthly tables. However, you can specify how long data should be kept in the database. As what concerns data reporting, there are many different reports that are provided by Trend, such as "Hot Spots", "Executive Summary", "Forecast", "Capacity Planning", "Top Ten", "Quick View" for routers, LAN, WAN, system etc. showing graphs for parameters like availability, traffic volume, bandwidth utilization, network response time, grade of service, CPU, memory and buffers utilization, errors, etc. (see the demos on the homepage [52]).

3.1.4 Luca Deri's `ntop`

Another tool for network monitoring is `ntop` [30], written by Luca Deri. Similar to the Unix comand `top` (where it also get it's name from) `ntop` is aimed at tracking and displaying the top users of the network. Just as `tcpdump` (and `tcptrace` for that matter, a tool presented below), `ntop` is based on the `libpcap` packet capture library. `ntop` shows the current and historical network usage by displaying the list of hosts that the sniffer has seen communicating with each other since it was started. The hosts are ordered according to the volume of the traffic they are generating respectively receiving. Much of the displayed information is split into incoming and outgoing traffic, respectively sorted by protocol. There are two user interfaces available, a web interface (which I used) as well as a command line version, `intop` (interactive `ntop`) which is a network shell based on the `ntop` engine.

In more detail, the information displayed by `ntop` includes:

Data received / sent, listed for each host:

- for all protocols that `ntop` recognizes (e.g. TCP, UDP, ICMP, ARP/RARP, IGMP, etc.) the volume of traffic in Kb (or KB, MB if that's appropriate) respectively in percentage of the total traffic volume seen
- for all IP protocols that `ntop` recognizes (e.g. FTP, HTTP, DNS, Telnet, SNMP, NFS, X11, and other IP protocols) also in Kb respectively in percentage of the total IP traffic volume seen
- the actual, average and peak throughput in Kb/s and Pkts/s
- net flows

Statistics with multicast traffic, graphical representation of global traffic distribution, information about each host, throughput statistics of last hour, day and month, domain statistics as well as information about plugins.

IP traffic distribution such as volume of traffic remote-to-local, local-to-remote, local-to-local, distribution of IP traffic on protocols, list of identified servers and clients respectively the ports they are using, information about TCP sessions and routers – however these are not very detailed.

`ntop` is freeware, so I have downloaded it and found it useful. I was going to use this tool for network monitoring after the changes in the network resource allocation patterns, however I did not use the tool during the measurements. It would have been interesting to do network monitoring and in this way observe the effect of other traffic on my measurements, however the limited time that I disposed of did not allow me to do so.

3.1.5 Bruce A. Mah's `pchar`

`pchar` [32] is a re-implementation of the `pathchar` utility, [31] written by Van Jacobson. Both programs attempt to characterize the bandwidth, latency, and loss of links along an end-to-end path in a correct manner. In fact, the aim of these tools is to recognize which is the bottleneck link on an end-to-end path, but they at the same time measure and calculate the above named parameters. The `pchar` utility is developed for both IPv4 and IPv6 networks. It operates in the following way: initially, it sends a set of pings to check if the host on the other end is alive. If these tries succeed, `pchar` goes on with the actual measurements. First, it generates a set of random packet sizes to test, where the test sizes go from an initial "increment" value (that can be specified as a command line option to pchar, default is 32) to the maximum multiple of "increment" that fits into the specified MTU value (default is 1500 bytes). Then, it sends UDP packets of the generated sizes, with increasing TTL and for a certain number of tries for each hop, and analyses ICMP messages generated by the intermediate routers respectively the target host. By measuring the response time for packets of different sizes, `pchar` can estimate the bandwidth and fixed round-trip delay along the path. The response times are measured by determining the minimum response times for each packet size, and then performing a simple linear regression fit to these response time minima. This is done in order to eliminate outliers, that is, to isolate jitter caused by network queuing.

An example of `pchar` output follows:

```
pchar to {undisplayed IP address and hostname} using UDP/IPv4
Packet size increments by 32 to 1500
46 test(s) per repetition
32 repetition(s) per hop
 0: {undisplayed IP address and hostname}

    Partial loss:      0 / 1472 (0\%)
    Partial char:      rtt = 0.344704 ms, (b = 0.000913 ms/B), r2 = 0.999508
                       stddev rtt = 0.002296, stddev b = 0.000003
    Partial queueing:  avg = 0.000027 ms (29 bytes)
    Hop char:          rtt = 0.344704 ms, bw = 8760.351508 Kbps
    Hop queueing:      avg = 0.000027 ms (29 bytes)
 1: {undisplayed IP address and hostname}

    Partial loss:      759 / 1472 (51\%)
    Partial char:      rtt = 1.423452 ms, (b = 0.001189 ms/B), r2 = 0.991379
                       stddev rtt = 0.012572, stddev b = 0.000017
    Partial queueing:  avg = 0.005771 ms (20836 bytes)
    Hop char:          rtt = 1.078748 ms, bw = 28981.840270 Kbps
    Hop queueing:      avg = 0.005744 ms (20807 bytes)
 2: {undisplayed IP address and hostname}
```

```
Path length:       2 hops
Path char:         rtt = 1.423452 ms, r2 = 0.991379
Path bottleneck:   8760.351508 Kbps
Path pipe:         1558 bytes
Path queueing:     average = 0.005771 ms (20836 bytes)
```

The tool can be downloaded from the page [32]. Just as the previous tool, `pchar` could have been of good use to check how the network behaves when resource reservation changes are done. I did not use it during the measurements due to the already named time limitation.

3.1.6 Shawn Ostermann's `tcptrace`

Being a `tcpdump` dump file analysis tool, `tcptrace` [50] reads output dump files of `tcpdump`, `snoop`, and a few other packet capturing programs. Based on information extracted from these dump files, `tcptrace` displays information about each connection that it has seen. There are several output formats that one can use, depending on the amount and detail of the needed information, the output formats range from very short and simple to rather detailed. There is also a possibility of displaying some graphs, such as the time sequence graph, throughput and round trip times graph.

A sample output follows:

```
Running file 'outfile.dmp'
Using 'pcap' version of tcpdump
Trace file size: 50266 bytes
566 packets seen, 560 TCP packets traced
elapsed wallclock time: 0:00:00.011557, 48974 pkts/sec analyzed
trace file elapsed time: 0:00:43.236603
        first packet:  Wed Aug  9 13:48:11.563221 2000
        last packet:   Wed Aug  9 13:48:54.799825 2000
TCP connection info:
31 TCP connections traced:

...

================================
TCP connection 5:
        host i:        {undisplayed IP address}:4014
        host j:        {undisplayed IP address}:3333
        complete conn: yes
        first packet:  Wed Aug  9 13:48:15.180981 2000
        last packet:   Wed Aug  9 13:48:15.473800 2000
        elapsed time:  0:00:00.292819
        total packets: 23
        filename:      outfile.dmp
   i->j:                                j->i:
     total packets:            10           total packets:            13
     ack pkts sent:             9           ack pkts sent:            13
     pure acks sent:            7           pure acks sent:            2
     unique bytes sent:       371           unique bytes sent:     11540
     actual data pkts:          1           actual data pkts:         10
     actual data bytes:       371           actual data bytes:     11540
     rexmt data pkts:           0           rexmt data pkts:           0
     rexmt data bytes:          0           rexmt data bytes:          0
```

```
     outoforder pkts:           0          outoforder pkts:           0
     pushed data pkts:          1          pushed data pkts:         10
     SYN/FIN pkts sent:       1/1          SYN/FIN pkts sent:       1/1
     mss requested:        1460 bytes      mss requested:        1460 bytes
     max segm size:         371 bytes      max segm size:        1460 bytes
     min segm size:         371 bytes      min segm size:          16 bytes
     avg segm size:         370 bytes      avg segm size:        1153 bytes
     max win adv:          8760 bytes      max win adv:         32120 bytes
     min win adv:          6560 bytes      min win adv:         31749 bytes
     zero win adv:            0 times      zero win adv:            0 times
     avg win adv:          9328 bytes      avg win adv:         32091 bytes
     initial window:        371 bytes      initial window:        880 bytes
     initial window:          1 pkts       initial window:          2 pkts
     ttl stream length:     371 bytes      ttl stream length:   11540 bytes
     missed data:             0 bytes      missed data:             0 bytes
     truncated data:        345 bytes      truncated data:      11290 bytes
     truncated packets:       1 pkts       truncated packets:       9 pkts
     data xmit time:      0.000 secs       data xmit time:      0.013 secs
     idletime max:        212.0 ms         idletime max:        267.3 ms
     throughput:           1267 Bps        throughput:          39410 Bps
================================

...
```

Detailed analysis possibilities are only offered for TCP connections registered by the sniffer (e.g. `tcpdump`), some minimal information is provided for UDP traffic as well. However, the kind of information provided was not very useful for the sake of my analysis, especially as the work concentrated on UDP traffic.

3.2 Methods Used for Performance Analysis in the Project

3.2.1 The Initial Idea

After doing the above survey, the conclusion was that there is no tool available that could be directly used to analyse the performance of RealPlayer, rather, there are a few tools that could have been used for monitoring the network and analyzing the changes that different resource allocation patterns cause.

That is, the need to develop an own, dedicated method for performance analysis arised. The initial idea was the following: capture the packets that are received by the machine that RealPlayer runs on during the playback of the clip. Produce graphs that show the received data rate (in kilobytes per second and packets per second) versus time (in seconds), respectively packet size versus time. By looking at the quality of the video as played out by RealPlayer (e.g. is it continuous, how distorted is it, etc. – note that these measures are quite subjective) and how good the audio is, establish a set of graphs that correspond to different quality levels. These graphs will be the input to the analysis.

As opposed to the video stream, where one can only get more or less subjective information about the quality, one can get more precise information about the audio stream with RealPlayer. RealPlayer sends back statistics which are stored in a log file maintained by RealServer, these include: the total number of packets received, total number of packets arriving out of order, and number of missing packets. This information gives a good measure of the audio stream quality. However no such information is recorded about the video stream, thus an other method to gain information about its quality was needed.

So the initial idea was to capture the packets coming in to the machine where RealPlayer runs, and by looking at the video and checking the audio log files, to establish a set of data rate versus time graphs that correspond to, let's say, a performance level of 100, 98, 95, 93, 90, 85, 80 and 75 percent. That is, to subjectively establish when a playback was perfect (100%), when it was almost perfect (98%) and so on, and to have a corresponding graph based on the arriving data. Then, use only this set of graphs to decide upon the performance level the RealPlayer had, when doing the measurements.

The next step was to find a way to establish these graphs. This task consists of two components: starting from a file that contains `tcpdump` output, captured on the machine running RealPlayer, produce the graphs that plot the data rate versus time, and the packet size versus time. The other component is to produce different scenarios, where RealPlayer performs in different ways, yielding the different graphs needed.

The first issue was solved in the following way: a Perl script was written which reads the file containing textual `tcpdump` output, and filters out the rows that contain information about the data sent from RealServer to RealPlayer. This is easily done by applying a filter expression that looks for UDP packets sent to the port 6970. This was namely the port that is used by RealServer and RealPlayer for the data transmission in the case of a single data stream (i.e. one clip played at a time) as given in RealServer's configuration file. Next, the amount of data that was sent during a specified time interval is summed up, and together with the respective time interval is written to a file, which is then plotted using the gnuplot program. The same can also be done for TCP packets, by just applying a different filter expression. The time interval that the summing is done for can be specified as an integer via a command line argument to the Perl script. This integer will be interpreted as the fraction of a second that the calculation are done for, so for example if it has the value 3, the Perl script will use one third of a second as the time interval for the calculations. The default value that I used was three.

For the measurements, several runs were done for the same setup in order to get a more general result, and a statistical mean for the different runs with the same setup was calculated. This was done using a Matlab script that reads those files produced by the Perl script, that correspond to the same setup, and calculates the mean value of the data rate (kilobits/second) for each time interval.

Another Perl script reads the same `tcpdump` output files, and does some other processing of the data. This results in a packet distribution graph, that is, a graph where for each packet, it's size and the time when it was captured are plotted. The two axes are packet size and time. This graph allows the distribution of the packets and their respective sizes to be seen. Another graph is also produced, where for each packet size that was seen, the packets with that size are counted (a.k.a. histogram). This was interesting because one could observe that for each clip, packets of a certain size occured very often and for other packet sizes, not. A more detailed discussion how the different packet sizes relate to the contents of each packet follows in the next section.

In order to produce the different graphs that correspond to the different performance levels, the idea was to simulate a controlled level of network congestion. This was achieved by throwing away outgoing packets on the machine where RealServer was running. This was done by modifying a loadable kernel module written by another member of the research group. For a detailed description of the loadable kernel module, refer to subsection 4.1.1 on page 40. The module can be dynamically loaded into the kernel, and sits between the TCP/IP stack and the device driver. The result is that packets sent to a certain destination IP address and a certain port, are filtered and every n-th one of these packets are not actually sent further, but rather the buffer holding them is freed and a NULL is returned. Thus, these packets are never transmitted and consequently never received by the RealPlayer. RealPlayer reacted very strongly if each 5-th packet coming from the RealServer was dropped (i.e. there was a loss rate of 20%), the quality was still very poor if each 10-th packet was dropped (10% loss rate), although if each 15-th packet was dropped, the audio stream was practically not affected, but rather only the picture. The correctness of the kernel module code was also checked by ping-ing a certain host and looking at the ICMP sequence numbers. It could be clearly seen that, for example, if each 5-th packet was dropped, there was no output for that sequence number as well as 20% packet loss was reported at the end of the ping

session.

3.2.2 Refinements of the Idea

In the solution presented above, the packets were dropped in the kernel module regardless of their content. A refinement would be to actually look into the contents of the packets and drop selectively. In order to be able to decide what packets should be dropped in this case, a short theory of video encoding is needed.

So first, a few definitions. *Encoding* is the process of reading a stream of input pictures or audio samples and producing a valid coded bitstream. Accordingly, *decoding* is the process of reading a coded bitstream and producing decoded pictures or audio. Basically, in our case it is the RealProducer software that does the encoding, and it is RealPlayer that does the decoding.

Coded video data consists of an ordered set of video bitstreams, called layers. If there is only one layer, the coded video data is called a non-scalable video bitstream. If there are two layers or more, the coded video data is called a scalable hierarchy. That is, in the case of layered encoding, the bitstream is structured in two or more layers, where the first layer is a stand-alone base layer, and the following layers are enhancement layers. A picture belonging to the base layer can be decoded independently of other pictures, as it contains all information needed to display the picture. A picture belonging to an enhancement layer uses information from pictures belonging to lower layers. In the H.262 standard [20] three different types of pictures are defined, as follows:

I-pictures or *intra-coded pictures* are pictures that are coded using information only from themselves

P-pictures or *predictive-coded pictures* are coded using motion compensated prediction from past reference pictures, that is, they indicate something like "put out here the same pixels as in the previous picture" or "put out there the same pixels as in the previous picture, translated to the right with a certain amount". P-pictures can have references to previous I- or P-pictures.

B-pictures or *bidirectionally predictive coded pictures* are coded using motion compensated prediction from both past and future reference pictures. B-pictures can have references to previous or following I- or P-pictures.

If layered encoding is used, a certain pattern consisting of a combination of I-, P-, and B-pictures is most often used, e.g. a repetitive pattern like I-P-B-B-P-B-B. Then, the first P-picture will have references to the preceding I-picture, the next P-picture to the first P-picture, and the first two B-pictures will have references to both the first and the following P-pictures. This concept is illustrated on Figure 3.1. That means, that if a B-picture is lost, only the information contained in itself is lost. If a P-picture is lost, then the information contained in it and in all the other P- or B-pictures referring to it is also lost. If an I-picture is lost, all information contained in the following pictures up to the next I-picture is lost.

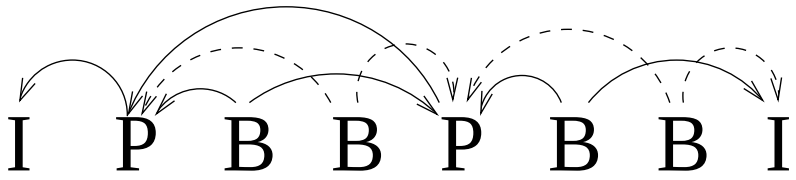

Figure 3.1: A pattern consisting of I-, P-, and B-pictures. The allowed references are depicted with the arrows.

Also, it is common to have one picture sent over the network in one packet, but it can also happen that a single picture is carried in multiple packets, or multiple pictures are carried in one

packet. In most cases, i.e. if there is not very much motion in the video, and each packet carries only one picture, then the packet sizes will correspond to the types of the pictures, that is, an I-picture will always have a bigger size, a P-picture a smaller size, and a B-picture an even smaller size. Of course in the case of much motion or a large change in the picture the predictively coded pictures can result in bigger packets, as they need to indicate the large amount of changes.

So in the case of selective dropping, one should look into the packets and establish which kind of picture they contain, and drop according to that. To establish a 100% good playback (and thus a corresponding graph), one should not drop any packets. To get a slightly poorer picture quality, and the next graph, some of the B-pictures should be dropped. To get an even poorer picture quality, P-pictures should be dropped, and so on. However, this "philosophy" is not always true, as the players used today have sophisticated ways of masking the losses, that is, they try to recover from the loss of pictures in different ways, so that the human eye does not recognize these losses.

That is, one needs to find a way to look into the packets and identify the picture type and drop according to a defined policy. As stated above, most often there is a one-to-one mapping between the packet size and the picture type, and a repetitive pattern, so this information could also be used: instead of looking into the packets, one could drop based on the sizes, after establishing the pattern. However, for the clips that I have looked as produced with RealProducer, no such pattern could be identified by looking at packet sizes, and no periodicies were found with fast Fourier transform (FFT) neither. These files were produced from a DVD or from a live camera. Also, in the documentation for RealProducer [36] no statement about (multi)layered encoding occurs. I also contacted RealNetworks Technical Support with this question, and as they later confirmed, no layered encoding is used. However, I only got this confirmation after I did some more "research", as described below.

Thus unfortunately a very simple way of identifying which packets contain which type of pictures is not applicable in our case. That is, one needs to look into the packets themselves in order to find out what information is carried in the packets. In order to do that, one needs to know what protocols are used between the RealServer and the RealPlayer for communication and data transport.

3.2.3 Protocols Used for Communication and Data Transport

There are several different communication channels between a media server and a media player, as is the case between RealServer and RealPlayer. The first channel contains coordination information between the two ends, such as when does the user press the "play", "stop" or "pause" buttons, which data transport protocols are supported by the player respectively server, which underlying protocols are to be used (i.e. UDP or TCP), and whether unicast or multicast is used. That is, this channel is a delivery control connection. For this communication, the most common protocol that is used is RTSP (Real Time Streaming Protocol) [48], an application-level protocol.

Another communication channel between the server and the player is that of the data stream(s) sent from the server to the client, that is, the actual video and audio data. For this, the most common protocol used is RTP (Real-Time Transport Protocol) [49], which is an Internet standard. However the RealNetworks software does not use this protocol but rather a proprietary one, called RDT (Real Data Transport), although RealNetworks [38] claims that RTP can also be used.

On the third communication channel between the server and the player, control information about the data delivery is exchanged, such as amount of lost and late packets, etc. For RTP, an embedded protocol called RTCP (Real Time Control Protocol) was developed [49], the two protocols use adjacent port numbers: RTP uses an even port number, usually starting from 6970 and the corresponding RTCP stream uses the next higher (odd) port number. In the case of RDT, a single stream is used; the control messages are sent on the same channel as the data and are identified by a different RDT header (based on my own experiments). The ports used are those specified in the RealServer configuration file, also 6970 in our case.

So what I wanted to find out, was if information is contained in the data packets that tells which kind of pictures they are carrying. For that, one first have to know what protocol is used for the data transport. As this is negotiated between the player and the server on the RTSP channel,

one needs only to look at these messages to find out what data transport protocol is used. One straightforward solution to look inside the RTSP packets was to use an RTSP proxy (initially developed for firewalls) in debug mode, where all the messages exchanged are written out to the console. RTSP proxy source code is provided on the RealNetworks firewall support homepage [37], hence I used this software. By examining the output according to the RTSP RFC [48], a lot of information about the player's and server's behaviour was learned.

Basic RTSP Operation

There are a number of different messages exchanged between the server and the client before the actual sending of the data can begin. These correspond to different methods supported by the server. The basic methods that should be supported by all servers are OPTIONS, DESCRIBE, SETUP, PLAY, and TEARDOWN. The OPTIONS message is used by the client to learn about the methods that the server supports. Usually, the communication begins with this message being sent by the client. The server responds with a list of supported methods. The DESCRIBE message is used to retrieve information about a particular presentation, where a presentation can consist of one or more data streams (e.g. audio, video). The response from the server contains a description of the media content, most often using the SDP (Session Description Protocol) format. These first two messages are optional, but they are very useful. The SETUP message is used to negotiate the transport parameters, that is, the data transport protocol (e.g. RTP, RDT), the transport layer protocol (e.g. TCP, UDP) and the respective port numbers, that will be used for the transmission of the requested media file. The client can specify a number of different data transport protocols, in the order of preference - the server will chose the first protocol from this list, that it supports. The server responds to the SETUP message with the chosen transport settings and a unique session number that will identify the session. After receiving a valid reply from the server (i.e. no error messages), the client can send a PLAY message, specifying the media file and the session ID that the server sent in the previous message. After this, the server begins to send the media data.

The RTSP proxy sits between the player and the server, and relays messages between these two. Here I was only interested in the RTSP proxy to output all RTSP communication between the client, the server, and the proxy, additional information about the proxy's other functionalities can be found in [37].

Below is an example of RTSP messages exchanged between RealPlayer and RealServer, with (for the sake of demonstration) irrelevant lines omitted:

```
rtspd[14514]: RTSP Proxy Reference Implementation Version 1.0.0.0
...

---------- C->S ----------
OPTIONS rtsp://server:554 RTSP/1.0
CSeq: 1

---------- S->C ----------
RTSP/1.0 200 OK
CSeq: 1
Public: OPTIONS, DESCRIBE, ANNOUNCE, SETUP, GET\_PARAMETER,
SET\_PARAMETER, TEARDOWN

---------- C->S ----------
DESCRIBE rtsp://server:554/luzi.rm RTSP/1.0
CSeq: 2
Accept: application/sdp
Bandwidth: 10485800

---------- S->C ----------
```

```
<description in the SDP format>

---------- C->S ----------
SETUP rtsp://server:554/luzi.rm/streamid=0 RTSP/1.0
CSeq: 3
Transport: x-real-rdt/udp;client\_port=6970;mode=play,x-pn-tng/udp;
client\_port=6970;mode=play,rtp/avp;unicast;client\_port=6970-6971;
mode=play

---------- S->C ----------
RTSP/1.0 200 OK
CSeq: 3
Session: 9466-2
Transport: x-real-rdt/udp;client\_port=6970;server\_port=1147

---------- C->P ----------
SETUP rtsp://server:554/luzi.rm/streamid=1 RTSP/1.0
CSeq: 4
Transport: x-real-rdt/udp;client\_port=6970;mode=play
Session: 9466-2

---------- S->C ----------
RTSP/1.0 200 OK
CSeq: 4
Session: 9466-2
Transport: x-real-rdt/udp;client\_port=6970;server\_port=1149

---------- C->S ----------
PLAY rtsp://server:554/luzi.rm RTSP/1.0
CSeq: 5
Session: 9466-2

---------- S->C ----------
RTSP/1.0 200 OK
CSeq: 5
```

Now that we have seen how RTSP works, we proceed towards our goal of finding out which packets contain which kind of pictures. The idea was, that if a standard protocol such as RTP is used for the data transport, this information should be available. So a substantial amount of work was put into trying to make RealPlayer and RealServer use RTP as their data transport protocol, instead of the proprietary RDT. However, as I found out that no layered encoding was used, the goal of having RTP as a data transport protocol changed. In RTCP interesting information about the quality of the delivered media data is given, that is, losses, delays, and alike. Thus the information provided in the RTSP client reports could have directly been used as performance measure for RealPlayer! Having this in mind, I went on trying to force RealPlayer and RealServer to use RTP.

3.2.4 Forcing RTP as Data Transport Protocol

As explained above, it seemed to be a good idea to have RTP as the data transport protocol, as the embedded control protocol, RTCP will supply useful information about the quality of the data stream arriving to the player. After some research in the jungle of the RealNetworks' documentation and help pages [38], and some contact with their Technical Support Staff, it seemed that there is an easy way to configure RealPlayer to specify RTP as it's data transport protocol

of choice. Namely, one has to create a key called "UseRTP" in a special directory in the Windows registry, and set this key to the value 1. However, this solution works only for Windows and only for G2 versions of the RealPlayer and RealServer. After making these changes and inspecting the RTSP messages exchanged by RealPlayer and RealServer (with the help of the RTSP proxy) I concluded that RTP was not used, but rather the proprietary RDT protocol. That is, RealPlayer always specified RDT in the first place in the Transport field of the SETUP message, so RealServer chose this protocol for the data transport.

A way around this is to modify the RTSP proxy in such a way, that RTP is the only protocol in the Transport field. That is, the modified proxy does not only relay messages between the player and the server, but modifies the SETUP message sent from the player. I did this modification, but the server responded with an "Unsupported Transport" message. The reason is that RTP can not be used in conjunction with RealMedia (.rm) files.

Despite several tests with different .avi files and the modified proxy, the behaviour of RealPlayer was not satisfactory. After the clip began to play, the transmission was suddenly broken and the clip playout stopped. The same happened for different .avi files, however the time that the clip could be played differed from clip to clip. I concluded after a number of tests that this was not a valid solution. After some more contact with RealNetworks' Technical Support it was clear that I could not force RealPlayer and RealServer to use RTP for the data transport. However, during my search for documentation and other users' experiences in mailing list archives and with Google [11], I found an interesting paper [15] describing an edge caching strategy that researchers at AT&T had implemented, using RealPlayer and RealServer as their chosen applications. They wrote an RTSP proxy as well, and forced RealPlayer and RealServer to use RTP in the same way that I tried. Instead of using the RTSP proxy supplied by RealNetworks, they wrote their own, as the edge caching was also implemented inside the RTSP proxy in their project. Some of my experiences with RealNetworks' software was confirmed in this paper, such as the only way to force the use of RTP is by modifying the Transport field in the SETUP message, however RTP is never used with RealMedia and some other file types. The paper gives also a nice presentation of RTSP operation and their edge caching implementation.

After concluding that RTP and RealPlayer / RealServer won't work together, I looked for other video streaming software that uses RTP. One such package is *vic* [53], an MBONE tool for video conferencing. This tool uses RTP, but is not suitable for my setup. *Vic* was developed for live video cameras, that is, there is no server or server software. Thus, for my purposes i.e. for different sever QoS setups, *vic* is not the solution. The other software that I looked at was Apple's QucikTime player [33] (available only for Windows) and the DarwinStreamingServer [5] provided by the same company, available also for Linux. For my purposes it was not only important to have a server software, but also to have a direct way to modify some of the resources allocated to it (e.g. CPU by changing priorities). As this pair satisfied both conditions, I installed both in the test network. I wanted to use the RTSP proxy to look at the RTSP messages exchanged, but the proxy that I had could not handle the RTSP traffic generated by the QuickTime software. But by looking at the port number pairs in use (6970, 6971 and sometimes an additional pair, 6972 and 6973) , it was clear that RTP is used. Now I needed a way to look into the RTCP packets. I found the freeware tools *rtpmon* and *rtpdump*. Both claim to report information extracted from RTCP reports, so I installed the tools. However, after several tests to make them work I concluded that they were developed to be used in conjunction with *vic*, and my limited project time did not allow me to further explore these tools and modify them for my purposes. Rather, I decided to concentrate on a simple, not necessarily 100% correct method for performance analysis, a solution similar to the prsented initial performance analysis idea.

3.2.5 The Final Method for Performance Analysis

In this chapter, we have gone through the different tools, methods and ideas that could be used for performance analysis. We have seen that due to the limitations of the environment a simple but still adequate method had to be used. The decision was to use a method similar to the initial idea presented in Subsection 3.2.1 on page 30, with some modifications.

In Subection 2.6.3 about RealPlayer, the different encoding levels at which a clip can be played, and the downshifts that occur when changing to a lower encoding level, were presented. The final decision was to base the performance analysis on the encoding levels and downshifts occuring during the playback of a clip. This is in fact quite a relevant method: though no detailed analysis of the (control) information exchanged between the player and the server is done nor analysis of the packets' contents, but as there is a feedback mechanism between the player and the server, each time a downshift occurs it means that the performance was not good enough. To be more explicit, suppose that you are watching a video at the 220 Kbps encoding level. Network congestion occurs, so not all the packets get to the player, or they get there late. Eventually, a downshift occurs. If one would have analysed the control information exchanged between the player and the server, one would have seen that the player requested a lower encoding level by reporting some parameters (e.g. losses, late packets, experienced current bandwidth) back to the server. One would have had more details available about why the downshift occured, but the fact that it occured and the information about the new encoding level tell enough.

Looking at the encoding levels and the downshifts is similar to looking at data rate versus time graphs (i.e. the initial idea) in that the encoding levels are reflected in the actual data rate. There is a very close relationship between these two, the encoding level and the arriving data rate are in most cases basically equal. The clip can be played at a certain encoding level only if the arriving data rate plus the data available from the buffer is sufficient for the playback at that encoding level. For example, if the arriving data rate (as reflected in the graphs) is at 200 Kbps, the clip can be played at the 220 Kbps encoding level only as long as there is enough data in the buffer. Thereafter, a downshift to the 150 Kbps encoding level occurs. Also, the arriving data rate is larger then the encoding level at which the clip is played only when buffering is done, as long as the buffer is not full.

The idea was to give a grade to each setup, so that it reflects the encoding levels at which the clip was played, that is, the number of downshifts that occured during the playback, and the time duration under which the clip was played at the respective encoding level. It was also important to assign the grade such that it reflected the observed picture quality of the playback - it is not certain, that the picture quality was always good even if a certain encoding level was kept e.g. sometimes the picture was bad even though the RealPlayer kept the 90 Kbps encoding level. Note that the terms "good" and "bad" refer to my own subjective perception in this context.

The grading method used was the following: I assigned a certain grade, or weight, to each encoding level, that the clip could be played at (i.e. the 220, 150, 112, and 90 Kbps encoding levels). As to reflect the picture quality as well, I assigned a weight to the playback when the picture was bad, too. I knew exactly for each clip, when and how long it was played at the different encoding levels, and also when and how long the picture quality was bad. So the formula used for calculating a grade was to calculate how long the clip was played at each encoding level, in percents of the total time. Then, I multiplied this number with the weight assigned to that encoding level or picture quality, and summed up the values. This gave a grade for each individual playback. I then calculated the mean value of the grades for the measurements done for each setup, and this mean value was the grade assigned to the setup as a whole.

That is, the following formula gived the grade for the individual measurements, where a clip encoded at the 220, 150, 112, and 90 Kbps levels was used:

$$\begin{aligned} grade \;=\; & weight_{220} * (time_{220}/time_{total}) + weight_{150} * (time_{150}/time_{total}) + \\ & + weight_{112} * (time_{112}/time_{total}) + weight_{90} * (time_{90}/time_{total}) + \\ & + weight_{badpicture} * (time_{badpicture}/time_{total}) \end{aligned}$$

Here, the total time is of course the sum of the other times, i.e. $time_{total} = time_{220} + time_{150} + time_{112} + time_{90} + time_{badpicture}$. Also note that the time when the picture quality was bad is substracted from the time of the encoding level at which the clip was played at, that is, if during a 40-seconds long 90 Kbps encoding level period the picture was bad for 10 seconds, $time_{90} = 30$

and $time_{badpicture} = 10$. In this way, I obtained a percentual proportion (relationship) between the grade and the duration of the playback at the different encoding levels.

In Section 5.1, where the results and their evaluation is presented, a motivation of how the weight for each encoding level was chosen is given. In the same section, it is also presented how the grading method is extended to reflect the last encoding level at which a clip was played. As the above decisions were taken during the evaluation, they are considered to belong to Chapter 5 rather than here.

Though the graphs were not directly used for the performance analysis, they were used for the evaluation of the results and for a better understanding of what happened during the individual playbacks.

Chapter 4

The Experimental Setup and Scenarios

As explained in the "Problem Statement" Section on page 3, the aim of the thesis project was to find correlations between the server and network resource reservation patterns and to study their combined effect on the end-to-end performance. That is, we have a triangle composed by "performance analysis", "network QoS" and "server QoS" symbolizing the dimensions of the project, as shown in Figure 4.1. Note that the terms "QoS" and "resource reservation pattern" are considered equivalent in the context of this report.

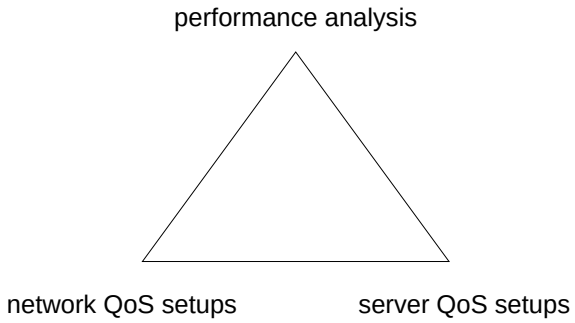

Figure 4.1: Dimensions of the thesis project.

In the previous chapter, we examined in detail the tools, ideas and methods that concern performance analysis, that is, the top of the triangle. In the next two sections, we look at the other two issues constituting the base of the triangle, that is, the network QoS and the server QoS setups. As in the case of the performance analysis method, the solutions were not straightforward but rather emerged from experiment. First, the different ideas and possible solutions are presented, finally both sections conclude with the presentation of the chosen setups. The following section summarizes the chosen setup and exact scenarios, based on the description from the previous sections. The final section discusses the relationship between the chosen model (consisting of network and server setups) and possible real-life situations.

4.1 Network Level QoS Issues

In the beginning of this report, we have already looked at relevant technologies and protocols addressing network QoS issues. From the technologies and methods presented in 2.2 beginning on

page 7, the only two that could be applied in this project are the allocation of different VC-s with different traffic classes at the ATM switch in the test network (at the moment, the ICorpMaker demo uses this technique) or the introduction of DiffServ in the network. The latter is under testing now, that is, at the time when I was doing my work it was not yet appliable. However by the end of my stay things evolved significantly.

Two different ideas developed under the course of the work. The first was to try to simulate a controlled network connection between the server and the client, that is, to control exactly when and if the packets are sent between the server to the player. This would mean that one can control the amount and kind of losses and delays on this link. This can be done by writing a loadable kernel module that modifies a certain function of the device driver. The following subsection presents the work I have done in this area. However this solution proved not to be the best one, and instead I used a traffic shaper. Traffic shapers are readily available software tools that "shape" (i.e. limit) the traffic flowing through a certain machine. Two traffic shapers are presented in Subsection 4.1.2, along with the motivation of why I chosed the latter one.

4.1.1 Simulation of a Controlled Network Connection

As stated above, the method of using loadable kernel modules was chosen for the simulation of the controlled network connection. The concept of a loadable kernel module was briefly presented in 3.2.1 on page 30. In order to understand the work I did in this area, a more detailed presentation of these issues are needed. Let us therefore look more closely at what loadable kernel modules really are, and what my kernel module did.

Dynamically loadable kernel modules are a useful feature of Linux. The Linux kernel uses a monolithic architecture with file systems, device drivers, and other pieces statically linked into the kernel image to be used at boot time. As opposite to this static structure, the dynamic kernel modules make it possible to write portions of the kernel as separate objects that can be loaded and unloaded on a running system [61].

According to [61], "a kernel module is simply an object file containing routines and/or data to load into a running kernel. When loaded, the module code resides in the kernel's address space and executes entirely within the context of the kernel. Technically, a module can be any set of routines, with the one restriction that two functions, init_module() and cleanup_module(), must be provided. The first is executed once the module is loaded, and the second, before the module is unloaded from the kernel. Of course, programmers must also observe all of the precautions and conventions used by kernel-level code when writing modules".

A kernel module can dynamically be loaded to the kernel by using the *insmod* command, along with the object file, and unloaded similarly with the *rmmod* command.

The kernel module that I modified was a device driver module. That is, it modifies the behaviour of the network device driver. Whenever the kernel needs to transmit a data packet, it calls the `hard_start_xmit` method to put the data on an outgoing queue. Each packet handled by the kernel is contained in a socket buffer structure, `struct sk_buff`, defined in `<linux/skbuff.h>` . The socket buffer passed to the method `hard_start_xmit` contains the physical packet in it's data field, including the transmission-level headers. The headers can be extracted from the packet by knowing at which offset a certain header begins. In such a way, the IP header and the transport header (TCP or UDP) can easily be extracted from a packet. As soon as one has these headers, one can use the functions that act on them in order to extract information such as source and destination IP addresses, source and destination TCP or UDP ports. I needed this information in order to filter out those packets that belong to the connection that I wanted to control.

In fact the driver module "overloads" the `hard_start_xmit` function. The network driver is modified such that it calls this modified function instead of the original one. At the end of the modified function, the original function has to be called in order to achieve the actual packet transmission. As each and every packet "flows through" this kernel module, it is possible to exactly define how these packets should be handled.

The easiest way to simulate packet losses is to simply free the memory where the socket buffer resides and ignore calling the original `hard_start_xmit` function at the end of the modified one.

By filtering on the source and destination IP addresses and the source and destination ports, it is easy to define which packets should be dropped (i.e. not transmitted). It is also trivial to have a counter as a global variable and drop, let's say, each 5th or 10th packet. So the idea was to use such a technique to simulate packet losses. Further, as the data contained in the packet is also accessible, it is possible to drop based on information contained in the packets. Not only losses, but delays can also be simulated in the kernel module, by simply delaying the transmission of packets. However, here a problem arose: I managed to delay the transmission only by doing busy waiting. I used the `udelay` function, which uses a software loop to delay execution for the required number of miliseconds. The problem is, that during this time the whole kernel is blocked, as the wait occurs in kernel mode. This causes some problems in the case when the machine that the kernel module runs on has to perform some important task. In the test network, as the kernel module ran on the same machine as the RealServer[1], these busy waiting periods had a huge effect on the RealPlayer's performance, as the server was just totally blocked for quite long time periods (in the order of 100 milliseconds for each packet!). This is a problem also because if the machine is blocked, the traffic sent in the other direction can not get through neither.

In conclusion, this initial solution was not adequate. One idea would be to have the kernel module run on an intermediate machine between the RealServer and the RealPlayer, but as in the meantime a better solution - that of traffic shapers - came into discussion, hence I did not try out this later idea.

You find more information about kernel modules and device drivers in Linux in [61] respectively in [47]. Also, a lot of information can be obtained from the manual pages.

4.1.2 Traffic Shapers

The next idea was to use traffic shapers. These are basically tools that limit the bandwidth of the traffic flowing through the machine where the shaper is installed. Also, the bandwidth of individual connections can be limited with traffic shapers. The advantage is that shapers are readily available tools, and they offer the grade of control over network connections that I needed. They are also easy to configure, and able to offer a large enough set of QoS setups for our purposes.

There are two kind of approaches to traffic shaping: you can either shape incoming traffic or shape outgoing traffic. Though the second approach seems more intuitive in some way, it is not all the time that one only wants to shape the outgoing traffic. However, the outgoing traffic will of course also be shaped implicitly, when the incoming traffic is shaped. For my purposes, it did not matter where and how the shaping was done, as the shaper ran on an intermediate node between the server and the client.

I have looked at both kinds of shapers, let us take them one after the other.

Traffic Shaping with `rshaper`

The `rshaper` tool is written by the same person as the book [47], Alessandro Rubini. As I got some inspiration, tips, and description of different functions from this book, it was not difficult to work through the sourcecode of the shaper and understand what it does. Basically, it modifies the standard `netif_rx` packet receiving function in a similar way as I described above for the function `hard_start_xmit`. That is, the network driver is modified so that it calls the shaper's receive function instead of the original one. In the shaper, the transmission of the packet is delayed accoring to the expected data flow. As one does not have an unlimited queue where the packets can wait, the length of the queue can also be specified directly from the command line, otherwise the default value of 5 seconds is used. In version 1.06 of the tool, an internal limitation to the queue size was done in order to prevent memory saturation. The amount of pending data is limited by a macro, which can be modified at compile time and defaults to 500 KBytes. The shaper code

[1] Also, it is worthwhile mentioning that I also experimented with modifications to the packet reception method, that is, `netif_rx`. Here the idea was to do the filtering, dropping etc. on the client side (the machine where RealPlayer is running). As the driver kernel module sits between the network interface and the TCP/IP protocol suite, packets can be dropped / delayed there so that this is invisible to the higher layers.

has to be compiled and loaded to the running kernel, after which the bandwidth associated with the host can be limited with the command:

```
rshaperctl <hostname> <bandwidth value> <queue size>
```

where the *hostname* defines the host for which the incoming traffic is to be shaped, the *bandwidth value* specifies the allowed bandwidth in bytes per second, and the *queue size* defines the length of the queue in seconds. Using the above command with a bandwith value of zero and no queue size will remove the shaper associated with the specified host.

The loading and unloading of the shaper kernel module is done as described in Subsection 4.1.1, with the *insmod* and *rmmod* commands. However one needs a kernel patch in order for the shaper to function.

Though this shaper would have been straightforward to use, and the ideas behind it were also easy to understand, it proved not to be the ideal one for my test network. This was due to a kernel version conflict between the shaper (written for kernel 2.0 and 2.2) and the other testing activities conducted in the lab, which involved another version of the kernel (2.4). Also the fact that a kernel patch would be needed in order for the shaper to function with the Ethernet device of the machine was a negative point, especially as no patch was readily available for the device I had.

The shaper can be downloaded from [39], the Readme file gives a short overview of the shaper's functionality, and the source code speaks for itself.

The Linux Traffic Shaper

An alternative to the `rshaper` is the Linux traffic shaper, this tool shapes the outgoing traffic. To use the traffic shaper facility of Linux, one needs an appropriately configured kernel and the `shapecfg` utility. With Red Hat Linux 5.2 the first step is easily done, one needs only to `make config` and set the CONFIG_SHAPER parameter. The installation of the `shapecfg` utility is straightforward as well. The ease of installation as compared to the shaper presented in the previous paragraph was a major factor in the decision of chosing this shaper.

Although little documentation is available for the shaper, configuration was easy to do. Before we look at that, let's take a short look at the ideas behind this shaper.

According to the Linux kernel documentation [23], the traffic shaper is a virtual network device. In fact, with the shaper you limit the outgoing data volume for this virtual device. The shaping of certain traffic can be achieved by routing this traffic through the virtual device. The actual transmission is however done by the physical interface that the shaper is attached to.

The traffic shaper must be built as a kernel module, that is, one needs to use the *insmod* command to load the `shaper.o` module into the kernel. Thereafter, the shaper device (i.e. the virtual network device) has to be attached to a physical interface, after which the maximum speed (throughput) supported by the virtual device can be configured. The shaper device needs to be configured for networking just as any physical device, that is, with the `ifconfig` command. As to achieve shaping of the outgoing traffic, the routing tables have to be updated as well, so that the traffic flows through the shaper and not directly through the physical interface. As I was to execute the above steps multiple times, I used a short shell script to perform all the above steps:

```
#!/bin/bash

insmod /usr/src/linux/drivers/net/shaper.o
/sbin/shapecfg attach shaper0 eth0
/sbin/shapecfg speed shaper0 256000
ifconfig shaper0 <IP_address> netmask <netmask> broadcast <broadcast> up
route add -host <dest_host> dev shaper0
```

It is also important that the shaper device gets the same IP address as the device that it is attached to. One needs to remember to add the route so that the traffic flowing to the desired host will traverse the virtual device.

After the execution of the shell script, the virtual device `shaper0` will be seen as a normal network interface by the system, with the limitation of 256 kilobits per second that can flow through this interface. According to the documentation [24], the usable range is from 9600 to 256000 bits per second. According to [25], higher speeds work well (i.e. the metering is accurate), but the traffic is bursty. In my experince, the shaper worked pretty well at values ranging from 300 to 500 kilobits per second.

To disable the shaper, one needs to bring down the virtual interface, and to remove the module:

```
ifconfig shaper0 down
rmmod shaper
```

After deciding to use this shaper, I did some tests to check if it correctly shapes the traffic flowing through it. The first tests I did with FTP: I set the shaper speed to different values e.g. 64000, 256000, 800000 bits per second and started an FTP transfer for a big file. The throughput values reported by FTP were quite close to the values that I set the shaper at, that is, 7.7, 35, and 140 kilobytes per second, that correspond to 63078, 286720, and 1146880 bits per second. As FTP does not have a very accurate throughput measurement method, I additionally checked the shaper in two other ways.

There is another tool that can be used for throughput measurements, called `tcpspray`. The idea behind it is very simple: it just opens a TCP connection to the `discard` or `echo` service of the specified machine, and sends data. It then measures how long it took to send the data. You can specify the block size, that is, the size of the buffer that it will send data from, and the number of blocks to be sent, as well as the inter-buffer transmission delay. In this way, you can influence how and how much data will be sent. The program just opens a socket at the specified location to the specified port (echo or discard) and writes data to the socket as long as the already written data is less than the amount given by number of blocks times block size. A `gettimeofday()` is executed both at the beginning and at the end of the transaction, so the throughput can be calculated as the data amount sent, divided by the time it takes to send. If the echo service is used, that is, the connection is duplex (data gets sent in both directions), separate throughput measurements are done for the send and the receive.

So I did some tests using `tcpspray` as well, with the same values for the shaper, i.e. 64000, 256000 and 800000 bits per second. Here, I got higher throughput values reported by `tcpspray` than the shaper was set at, respectively 68812, 286720, and 1146880 bits per second.

I also did experiments with RealServer and RealPlayer, that is, run the shaper on a machine inbetween these two. As I have the Perl script, that calculates throughput very exactly based on the tcpdump output, I could see exactly how accurate the shaper is. I did tests with a number of different files that were encoded at different levels, with files of different length, and with both UDP and TCP as transport protocols. Basically, I almost always got slightly higher throughput than I set the shaper to. For example, when setting it to 300 kilobits per seconds, it actually let through traffic up to 350 kilobits per second. However, at lower values such as 180 kilobits per second the shaping was quite adequate.

Due to problems that I experienced with TCP in conjunction with my server QoS setups, I later decided to only use UDP as transport protocol for the measurements (I first planned to do tests with both). In the rest of this section, the statements refer to UDP even if they do not explicitly express this.

The file that I used for the measurements was encoded at four levels, that is, at 220, 150, 112, and 90 kilobits per second. When using UDP as transport protocol, a large amount of buffering, approximately for a total of 19 seconds, is performed at the beginning of the playback, when about 600 kilobits per second data are sent from the server to the client. The buffering performed by RealPlayer was described in Subsection 2.6.3 in detail. The idea was to have the following, abstractly defined setups:

1. normal case, without any shaping

2. a scenario with the shaper set to a speed that alone does not affect the playback, but possibly combined with different server QoS levels will give a different behaviour from the previous case
3. the shaper set to a speed that slightly affects the playback
4. the shaper set to a speed just above the highest encoding level
5. the shaper set to a speed just below the highest encoding level
6. the shaper set to a speed just above the next highest encoding level
7. try to go on with this idea of "just above" and "just below" as long as it works

After some experimenting with the chosen file, I established that for shaper speeds below 180 kilobits per second the RealPlayer most often could not even start the playback, as the rate of data flowing to it during the initial buffering period was too little. For case number 2, a shaper speed of 500 kilobits per seconds was chosen. For the next case, a limit of 300 kilobits per second proved to be a good choice, as it did not affect either the picture quality or the encoding level at which the clip was played, but did cause dataflow different from the first case. Specifically, instead of having an initial buffering period of about 16 seconds, after which data is sent at the same rate as the encoding level of the playback, with the shaper set at 300 kilobits per second, data was sent at a rate of about 270 kilobits per second until the buffers got filled up. This took about one minute, whereas the clip is 2:18 minutes long. For the cases 4, 5, and 6 the values of the shaper's speed were chosen to be 250, 200 and 180 kilobits per second. Lower values were not relevant, as explained above. Figure 4.2 summerizes the chosen shaper speeds (cursive lines) relative to the different encoding levels (discursive lines).

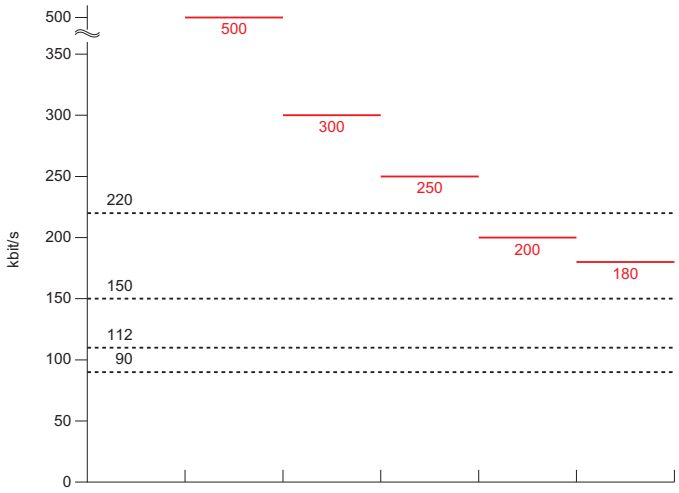

Figure 4.2: The chosen shaper speeds, relative to the encoding levels.

In Figure 4.3 example plots of kilobits per second versus time for different shaper setups are presented. The top curve corresponds to the normal behaviour. As mentioned above, a buffering period of about 16-20 seconds occurs at the beginning, after which the transmission is about at the level of the nominal encoding value, that is, 220 kilobits per seconds. In the second case, where the shaper is set to a speed of 300 kilobits per second, at the beginning one can observe a steep up and down trend. This is the typical behaviour of RealPlayer, when the traffic shaper has a speed limitation that cuts down the data rate at the beginning of buffering. This is due to negotiations between RealPlayer and RealServer, or better said, a backoff of RealServer when

RealPlayer reports high losses. The third and last graph, that is, the bottom curve shows the corresponding data rate versus time values for the case when the shaper is set to the speed of 200 kilobits per second. Here we can see the interaction between the shaper speed and the different encoding values that the clip has - at the beginning, RealPlayer expects a 220 kilobits per second stream. As less than this data rate is received, it shifts down to 112 Kbps encoding at the very beginning. However, as more than that much data flows to it, it soon shifts up to the 150 Kbps encoding level; having data in the buffer as well, a slightly lower data flow level is enough to keep this encoding level. The peak around the 35th second is also due to a "try" to do an upshift, while the peak at about the 90th second results in an upshift to the 220 Kbps encoding. Due to the shaper, such a data flow cannot be kept up with, so an additional downshift occurs at the end.

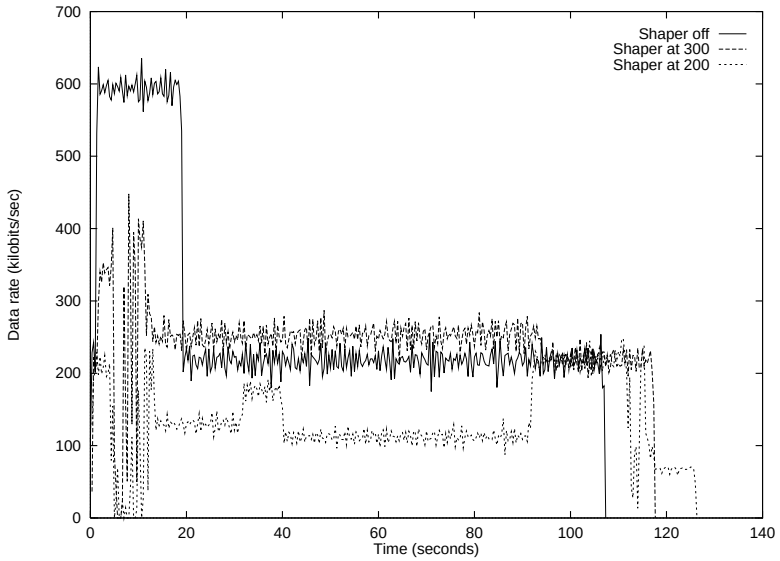

Figure 4.3: Data rate versus time, with the shaper off (top curve), at 300 Kbps (middle) and 200 Kbps (bottom curve) speeds

4.2 Server Level QoS Setups

As we have seen in the previous section, there are six different setups for the network. It would be preferable to have about the same number of setups for the server as well, in order to have a balanced problem space.

Recall from Sections 1.2 on page 2, and 2.5 on page 17, that in the ICorpMaker architecture, there are one or more instant server chunks (ISCs), which together with the virual network that interconnects them and the entry points (proxies) constitute an ICorp, an instant corporation. It is the resources allocated to each ISC we are concerned with, when talking about "server resource allocation", "server QoS levels", etc. However, as there was only one application (RealServer) running inside the ISC in my tests, the resources allocated to the ISC roughly corresponded to the resources allocated to the test application.

In order to have different server QoS setups, one needs a means of varying some resources that are allocated to the ISC and the tested application. In the current ICorpMaker implementation, each ISC is running as a VMWare instance on a server machine. This VMWare instance runs as a regular process inside the host operating system, with a certain priority of the process. As the process is a regular one, it is straightforward to change it's priority with the `renice` Unix command. In this way, it is possible to influence the resources (i.e. CPU) allocated to the server

chunk when it is already running. As under normal Linux it is not possible to directly allocate a certain percentage of the CPU to each process, using different priority values while varying the load on the machine is a solution for the different server QoS setups. Also, memory and other resources for the VMWare instance can be adjusted[2], but these configurations have to be made before starting VMWare. It is preferable for the purposes of ICorpMaker to have a way of changing server resources while the server chunk is already running, and it is also by means of priority changes that the server resources are changed in the ICorpMaker demo. According to the above, my choice was to implement the server resource allocation setups solely by adjusting priorities of certain processes.

It was not adequate to simply give the VMWare instance a low priority, when basically no other processes were running on the same physical server. Rather, a certain process or processes were needed that caused "CPU congestion" (heavy load on the CPU) on the physical machine. I also wanted to have a solution that is more or less controllable. The idea that is used in the ICorpMaker demo is to start a process that puts a heavy load on the CPU and to change the priorities of this process and the VMWare process. Using the same idea, one could change the processes' priority so as to have the desired server QoS levels. The program used for the "CPU congestion" in the ICorpMaker demo is a prime number calculating program written in Java. Thus the first tests that I conducted were using these processes.

Using the `tcpspray` tool to generate traffic from inside the VMWare instance, and capturing the packets at the same target machine where later the RealPlayer was run, I did some experiments in order to see what happens when the priorities are changed. By running the prime number program on the host OS and changing the priority of the VMWare instance, at the target host I noted how far apart in time the traffic bursts occured and how their length changed when I changed the VMWare priority. That is, by changing the priorities, it was possible to influence how often and how long a process inside the VMWare instance ran.

Therefore the setup I used was a RealServer inside the VMWare virtual machine, and a single RealPlayer running on a remote machine connecting to it. I did a number of tests with different combinations of the priority value for the prime number calculating, "congesting" Java process and the VMWare process, by assigning negative priorities to the Java process and leaving VMWare at the normal zero priority (-20 is the highest priority level, and +19 the lowest). My observation was, that in a certain range of negative priorities for the Java process, no reaction was seen at the RealPlayer side. After decreasing the priority again, quite abrupt changes were seen at RealPlayer. That is, at a priority value of -11 for the Java process, normal play was experienced without any downshifts. But at a value of -12, several downshifts were experienced and at -13 the playback eventually stopped before the clip ended. Such abrupt changes from one performance level to an other are not what I wanted - as stated above, I wanted to have at about 5-6 different setups. That is, I wanted to have a higher granularity. I made several tests with different priority level combinations, such as negative priorities for the Java process and positive ones for VMWare, as well as zero priority for the Java process and positive values for VMWare. These however did not change the behaviour, that is, there still was an abrupt change from good playout to a bad one, so that at most three setups could have been done using this arrangement.

The prime number process proved to be too CPU intensive in this setup, so another solution had to be found. As the RealServer itself does not need very much CPU (0.1-0.8% CPU usage is displayed by `top`, at about each 10 to 20 seconds), it was not adequate to try to find a way to ignore VMWare and have the server setups be only running on a single OS. Running the prime number process at a high priority and a RealServer inside the same OS did not affect the playback at the RealPlayer at all. What I really needed, was a way to limit the resources that VMWare gets, but not as much as with the prime number program and preferably in a more controllable way. The fact that VMWare runs as a process inside it's host OS means that the *whole* virtual machine - that is, it's network interface and disk as well - runs only when the VMWare process

[2] It has to be mentioned that the VMWare instance I used for the measurements was configured with 32 Megabytes of memory, and bridged networking, details about these issues were presented in Subsection 2.3.2. I did some measurements with larger memory size as well, but this did not affect the experienced behaviour of the RealPlayer client.

runs in the host OS.

Another idea was to have several other RealServers run on the guest OS, limiting in such a way the CPU that the VMWare gets. This proved not to be a good solution, at the beginning it seemed to work well only because the directory of RealServer on the host OS was mounted from an other machine with NFS, and this other machine was heavy loaded at the time I did my test measurements. It was quite straightforward, that when several RealServers ran locally on the host OS, even together they did not use much CPU, and as the other resources were fixed for the VMWare, this idea did not work out.

It was getting clear that I needed a method to control more exactly how often the VMWare process runs, and for how long. The idea of using computations in order to produce "CPU congestion" was good, but needed to be applied in another way. It would have been nice to have a program that did some computations, and after a time it went to sleep for some time period, during which time the other processes could run. This proved to be a usable and implementable idea; to make sure that the process that did the computations and went to sleep from time to time got a lot of CPU as long as it ran, an additional trick was needed.

Under Linux, there are several scheduling algorithms, or scheduling policies, that are used. Recall that the scheduler is the kernel part that decides which runnable process will be executed by the CPU next. The scheduling algorithms describe how this decision is taken. The Linux scheduler offers three different scheduling algorithms (policies, as they are called in the Linux man pages), one for normal processes and two for real-time applications. In this context, "real-time" does not refer to a true real-time scheduling strategy, e.g. the process with the "first deadline" is first run, but rather strikes the fact that processes running with one of the above named "real-time scheduling policies" will always have higher priority than the normal processes. The three policies are SCHED_OTHER, SCHED_FIFO, and SCHED_RR. The second two are the real-time policies. They differ from each other in the way they handle processes with the same priority. With SCHED_FIFO, a first-in first-out queue is maintained for each priority level. This scheduling does not use time slicing, that is, a process runs until either it is blocked by an I/O request, it is preempted by a higher priority process, or it calls a certain function (`sched_yield`). As opposite to SCHED_FIFO, SCHED_RR uses time slices and a round-robin mechanism, that is, each process is run for a time slice (a maximum time quantum). As soon as a process uses up it's whole time slice, it is put at the end of the list for it's priority. In this way, it is not possible for a process to take all the CPU cycles and make it impossible for other processes to run, whereas this danger exists with SCHED_FIFO [26].

In the Linux manual page for the function `sched_setscheduler` [26] it is also recommended that one runs a shell scheduled under a higher priority than a tested application that runs with SCHED_FIFO or SCHED_RR, as a non-blocking endless loop in such an application can block all other processes with lower priority forever. That would practically mean that the machine is dead, a situation that one has to prevent.

So the idea was to write a small C program, where a PID is given as argument and as a result, the process with the given PID will get the SCHED_RR scheduling policy and the highest possible priority level assigned to it. The program is included below:

```
#include <sched.h>
#include <pthread.h>

main(int argc, char* argv[]) {
  long my_pid;
  struct sched_param param;
  int max_prio;

  my_pid = atoi(argv[1]);
  printf("my_pid: %d \n", my_pid);

  if (sched_getparam(my_pid, &param) != 0)
```

```
    perror("sched_getparam()");

  max_prio = sched_get_priority_max(SCHED_RR);
  param.sched_priority = max_prio;

  if (sched_setscheduler(my_pid, SCHED_RR, &param) == -1)
    perror("sched_setscheduler()");

  printf("Scheduler: %d ", sched_getscheduler(my_pid));
  printf("  (OTHER/FIFO/RR  %d/%d/%d) \n", SCHED_OTHER, SCHED_FIFO, SCHED_RR);
};
```

As mentioned above, it is a useful idea to apply this program not only to a process that is used for testing, but also a shell, in this way preventing the machine from blocking. If one applies it to both processes, they are scheduled in a round-robin fashion, one after the other, so there is no need to assign a higher priority to the shell than to the Java program that did the computations (i.e. the "congesting" process).

The Java program is a small and simple one. It executes in an endless loop a for-loop of length x, inside which some computations are done. In the endless loop, after the for-loop, the executing thread is put to sleep for a time t (expressed in milliseconds), where t is a command line argument passed to the program when starting it. The result is that the computations are done, taking some milliseconds, after which the process goes to sleep, for the specified amount of milliseconds. As only a shell is running with the same high priority, there is a round-robin swap between these two processes. However, most of the time the shell is inactive, so basically only the Java process is running, and as long as it sleeps, all the other processes can be run, e.g. VMWare.

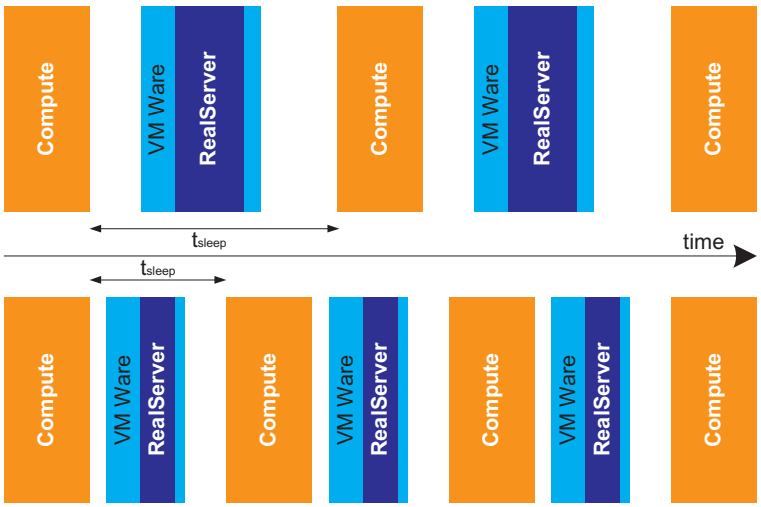

Figure 4.4: With a fixed amount of computations, the time during which RealServer runs decreases if one decreases the time for which the Java process sleeps between the computations.

In this way, I could control much better (i.e. at a finer level of granularity) when and for how long time VMWare could run, and therefore the RealServer inside the virtual machine. It was possible to tune both paramters x and t, that is, the amount of computations and the length of the sleep. I have conducted some tests for x values of 500, 1000, 1500, 2500, 3000, 3500, and 4000, and for t values of 40, 50, 60, 75, 80, 90, 100, and 120 milliseconds (but not in all combinations!), when I used UDP as transport protocol. Figure 4.4 demonstrates schematically what happens when one has a fixed amount of computations, but decreases the length of the sleep time.

I also did some experiments using TCP as transport protocol, however I only did this after experimenting a lot with UDP. When using TCP as transport protocol in the same setup as before, RealPlayer and RealServer reacted very strongly (i.e. very bad playback) as soon as there was a shell on which the above presented C program was applied. As I considered measurements with TCP as a possible future work, I decided at this stage to drop this idea totally. Consequently, all the measurements were done using UDP as the transport protocol. In the rest of the report, this will not be explicitly stated any longer.

Now that I had a working setup set, I only had to decide upon the different server setups. The idea in looking for a suitable set of server QoS setups was similar to when establishing the network QoS setups, that is, I wanted to have the following scenarios, as seen from the RealPlayer's side:

1. a setup with normal play, without any influence from the server
2. a setup with some server QoS involved, but without affecting the playback at the RealPlayer
3. a server QoS setup that slightly affects the playback, that is, zero or one downshifts are experienced
4. a setup where two downshifts are experienced during the playback
5. a setup with three downshifts
6. possibly one additional setup, with even poorer performance level

The decision on having this set of scenarios followed after some experimenting, as stated above, on page 48. Basically, it was almost impossible to try enough combinations to get exactly the desired scenarios enumerated above. In particular, it was almost impossible to find a setup where a downshift to the 112 Kbps encoding level would occur - in the majority of the cases, the downshift from the 150 Kbps encoding level was directly to the 90 Kbps encoding level.

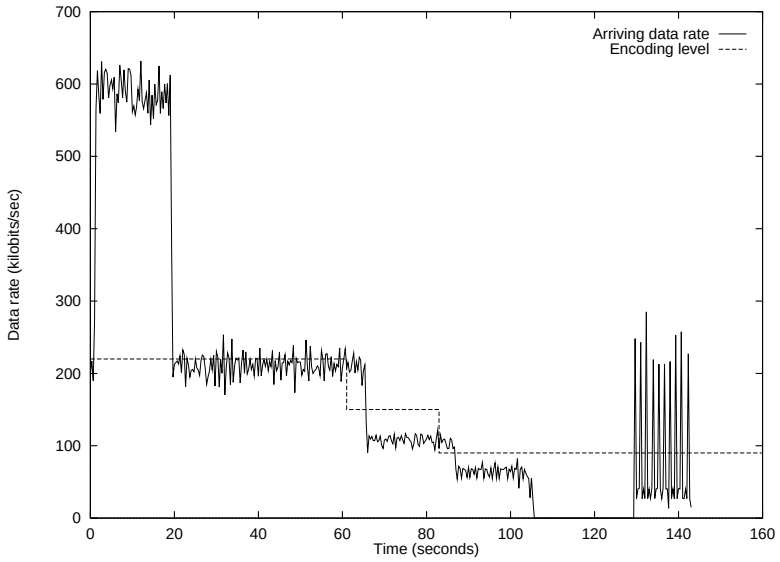

Figure 4.5: Data rate versus time, with t=60. The dotted line shows the corresponding encoding level of the playback.

But after some trial from the above named range, I decided on 5 setups, with the parameter values x=4000 and t=60, 80, 90, 120, and the first setup with no congesting Java process running, but rather only the shell at the high priority level. Here, the setup with t=60 corresponds to

the worst level, as the Java process sleeps for only 60 milliseconds. Hence there are less then 60 milliseconds left for the other processes to run, which results in a poorer performance for the RealServer that runs inside the VMWare as in the case when the Java process sleeps 120 milliseconds between two sets of computations.

The behaviour observed for the setup with t=60 (corresponding to the 6th case), was downshifts until the 90 Kbps encoding level, a drop to zero and then some fluctuations in the transmitted data rate, as shown in Figure 4.5. As stated above, the downshift from 150 Kbps encoding level to 112 Kbps encoding level almost never occured, rather, the downshift was directly to the 90 Kbps encoding level. In the case of the playback shown in Figure 4.5 it was also the case, so there were in fact only two downshifts to a lower encoding level, along with a zero rate period in the sent data rate and some fluctuations at the end, which caused still pictures or bad quality playback in many cases. The zero rate period is probably due to RealPlayer and RealServer's adaptation strategy. They probably suppose that a temporary congestion caused the decrease in the arriving data rate at RealPlayer, and they wait for a certain time and thereafter try to find out the actual bandwidth again.

What we see in Figure 4.5, can be explained in the following way. At the beginning, the playout buffer is filled with data, this is why the playback is at the 220 Kbps encoding level even though the data rate arriving to the machine is slightly lower than 220 kilobits per second. This lower data rate is due to the server QoS setup, that is, the data rate that RealServer wants to send is at 220 Kbps, but as it does not get to run often enough to send at this rate, the actual data rate is lower. After a time, RealPlayer senses this and sends some feedback message to the RealServer, asking for a lower encoding level. The server shifts down to a lower encoding level, and sends at the lower data rate, however it does not get to run as often as to actually send at the nominal data rate. That is, less than 150 kilobits per second arrive to the RealPlayer. This results in a request to downshift after a certain amount of time. We can observe this behaviour in Figure 4.5. Additionaly, after experimenting with RealPlayer and RealServer some, I observed that at the beginning of a playback, RealServer gets scheduled more often than at later stages of the playback. This also affects the consecutive downshifts. However, I can not prove this "scheduling-policy theory".

For t=80, a similar behaviour occured, but the downshift to the 90 Kbps encoding level occured at the end of the data transmission, at about 1:30 minutes out of the 1:50 during which data was sent, thereafter the playback was done from the buffer, at the 90 Kbps encoding level. For the setup with t=90, one downshift to the 150 Kbps encoding level occured, for about 40% of the conducted measurements, while with t=120, no downshifts occured.

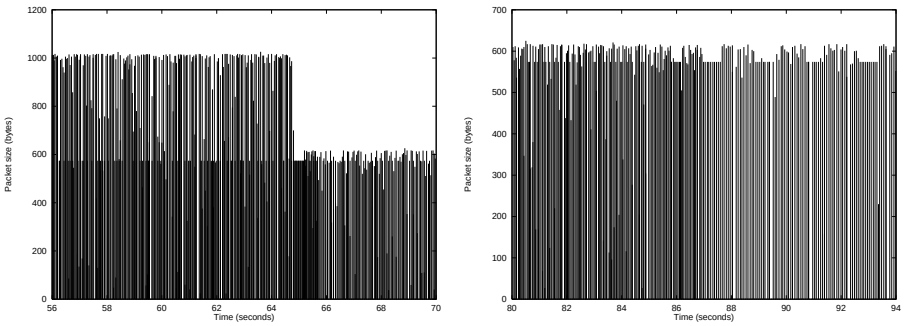

Figure 4.6: A downshift from the 220 Kbps encoding level to the 150 Kbps encoding level involves a change in the packet sizes. A downshift from the 150 Kbps encoding level to the 90 Kbps encoding level involves a change in the rate at which packets arrive.

An issue related to the downshifts, that needs to be briefly discussed is what actually happens

when a downshift occurs. What the end user can see is a change in the encoding level to a lower value, expressed in kilobits per second. The lower data rate is due to two factors: a decrease in the packet size and a decrease in the rate at which packets arrive. I observed the first phenomena (i.e. decrease in packet size) in the case of a downshift from the 220 Kbps encoding level to the 150 Kbps encoding level, and the second one (i.e. decrease in the rate at which packets arrive) in the case of a downshift from the 150 Kbps encoding level to the 90 Kbps encoding level. The packet size versus time plots shown on Figure 4.6 demonstrate the two phenomena. In fact, the data that these two figures present comes from the same recording session as the data on Figure 4.5, recall that the downshift to the 150 Kbps encoding level occured after 61 seconds and the one to the 90 Kbps encoding level occured after 83 seconds. Note that the packets with the size of 574 bytes, that occur most often, are the packets containing the audio and not the video data (statement based on own experience, from clips that I produced with video only).

4.3 The Exact Measurement Setup and Scenarios

In Sections 4.1 and 4.2 above, the choices for the measurement scenarios regarding network and server QoS were motivated and presented. In this section, we briefly summarize by presenting the exact setup and scenarios.

The setup used was as following: on one machine, I had Linux as host OS and Linux as guest OS inside VMWare. Inside the VMWare, a RealServer was running, being basically the only running process. On the host OS, besides the VMWare a few processes were running:

- a Java process that did some computations and then went to sleep, the amount of computations and the length of the sleep are expressed in the variables x and t respectively. This process was set to the SCHED_RR policy and the maximum priority
- a shell that was also set to the SCHED_RR policy and the maximum priority, so as to have an emergency terminal to control the system
- the Unix *top* command, in order to follow how the processes behaved
- an Emacs instance, to make changes to the configuration as necessary

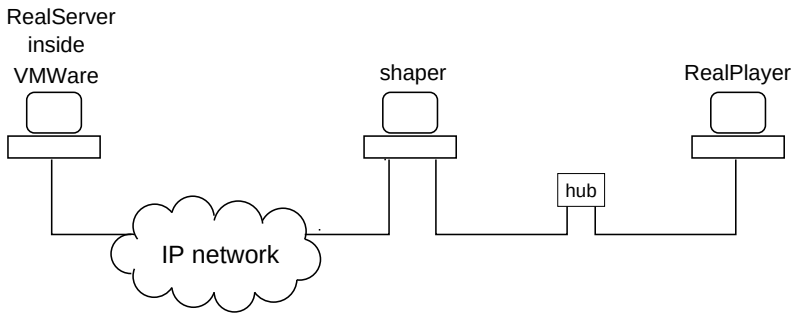

Figure 4.7: The setup used, with VMWare and the RealServer on a machine at one end, RealPlayer at the other end, and a shaper inbetween.

The RealPlayer was run on a machine that was 5 hops away, this machine was on the same subnet as the shaper. In order for the traffic addressed to RealPlayer to flow through the shaper, the routing tables were modified on the router of this subnet. Between this subnet and the server machine, a couple of switches and routers were sitting, these were however irrelevant for my tests

as I could not control them, they are shown in the Figure as "IP network". The setup is shown on Figure 4.7.

As stated in the previous two sections, there were six different scenarios for the network QoS and five for the server QoS, yielding a total of 30 different combinations and thus scenarios. On the network side, the setups consisted of different shaper speeds, namely 500 Kbps, 300 Kbps, 250 Kbps, 200 Kbps, and 180 Kbps. The sixth setup was when the shaper was off. On the server side, the amount of computations done in the congesting Java program was the same in each case, i.e. x=4000, while the amount of sleep time was different, that is, there were four different sleep values at t=120, t=90, t=80, and t=60. The fifth setup here was when the Java program was not running. There were 5 measurements done for each scenario, so that I could look at some statistical mean values and get a general impression, as well as analyse the individual measurements.

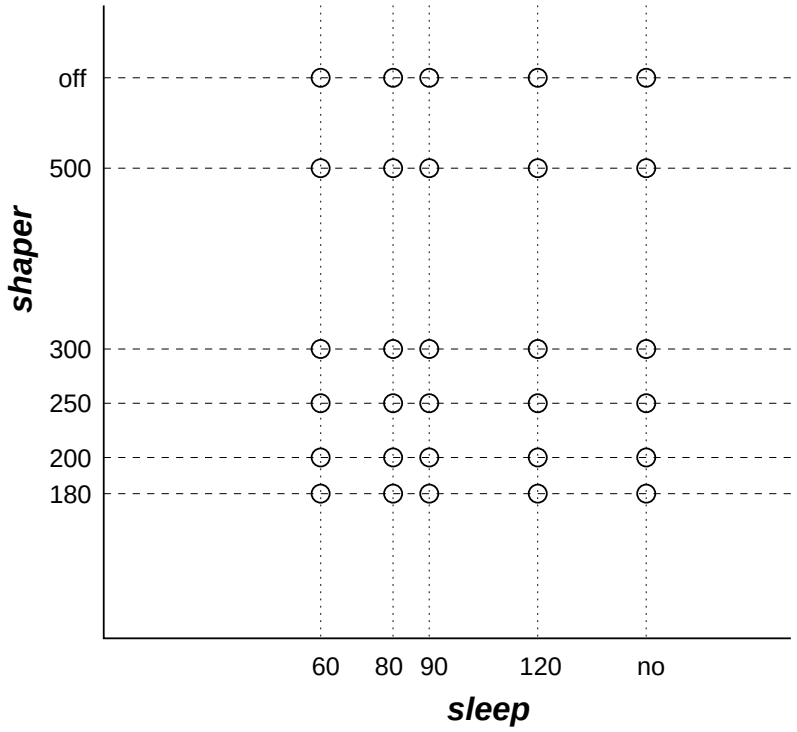

Figure 4.8: The thirty different measurement setups, marked as circles on the figure.

The measurements were conducted in the following way: for a certain server QoS setup, each of the six network setups were used, in the order: shaper off, shaper at 180 Kbps, 200 Kbps, 250 Kbps, 300 Kbps, and 500 Kbps. As the influence from other traffic in the network was not totally negligable, this manner of doing the measurements allowed me to have a kind of "check" at the end, to see if everything is still working the same - the setups with the shaper off and with the shaper at speed 500 Kbps are almost equivalent. Figure 4.8 presents the thirty different setups graphically.

The clip that was used for the measurements was produced with RealProducer, and has four different encoding levels: 220 Kbps, 150 Kbps, 112 Kbps, and 90 Kbps. Based on experiments with different clips, I concluded that the amount of motion in the clip did not play any role for the data rate or the downshifts, so I produced a clip with varying amount of motion. A length of about two minutes proved to be long enough to experience all the downshifts that eventually would occur, thus there was no need to use a longer clip.

4.4 Justification of the Model

Now that we have seen the exact measurement setup and scenarios, there is a question that one must ask: how does our model correspond to reality? As in the case of every model, ours is a simplified picture of the reality.

On the network side, the model matches best the IntServ concept from the network QoS solutions presented in Chapter 2. The similarities consist of the following:

- from the beginning, there is a certain amount of resources (bandwidth) reserved for the video stream
- ours is also a simplex model, that is, the resource reservation affects only one direction of the traffic flow, namely from the sender to the receiver i.e. the direction in which the information flows.

Note that the last item does not imply any problem, as there is very little data flowing in the other direction.

However there is no virtual network in place nor a complete isolation of our traffic from other traffic in the network, as the ICorpMaker model requires. This fact also influences the results obtained. These are negative properties of our model. However, there is complete traffic isolation (i.e. no other traffic) between the shaper and the machine where RealPlayer runs, this turns out to be important for the correctness of the results. The network QoS is modeled via bandwidth limitation, this corresponds well to real-life situations (e.g. a virtual network with guaranteed bandwidth).

On the server side, the model is not complete either. There are numerous server resources that can be of significant importance to how an application performs, some of them were named in Subsection 2.4.2 on page 16. From the resources named there, in our model one can only influence the amount of CPU cycles that the server partition (i.e. VMWare), and thus RealServer gets, but there is no possibility to specify it exactly. Also we do not know exactly what percentage of the total CPU they get in the case of the different setups. But, as described in Section 4.2, I based the setups on the behaviour of the client application. In this way we do not need to care about exact numbers, because it is the experienced performance levels[3] that define our server QoS levels.

It is also questionable whether the CPU is the right resource to limit in the case of the chosen application. RealServer and video streaming applications in general are not CPU bound processes, but there is one fact that makes our model adequate. Namely, there are two aspects of the server QoS: the resources that the server application gets, and the resources that the server partition (ISC) gets. Limiting the second one implicitly influences the first one. So by limiting the CPU resources that the VMWare process gets, we limit the amount and duration of time that the whole virtual machine (i.e. the server partition) runs. In this way, we have quite a realistic *server partitioning* model. That is, even if perhaps it is not most relevant to limit the CPU for the chosen application, it *is* adequate to limit the CPU of the server partition that the application runs inside.

The above being said, we can conclude that although our setup is only a simplified model of reality, it is adequate enough in order for our measurements and results to be quite realistic.

[3] as according to our performance measure

Chapter 5

The Measurement Results and their Evaluation

This chapter begins with the presentation and the discussion of the measurement results. Following this, an evaluation of the results, based on two different ideas is presented. The first idea is the same as the initial motivation for the thesis project, namely to answer the question of which of the server and network resources we need to allocate differently, if the ASP's client asks for more resources. The other idea is that of maximizing performance for a given cost, when the costs of network and server resources are fixed and given as a simple, linear function.

5.1 The Measurement Results and Discussion

In the previous chapter, the motivation for the network and server QoS setups and the setups themselves were presented. As we have seen, there were thirty different setups, for each of them 5 measurements were conducted. The thirty different setups were presented in the previous chapter, a scheme of the different setups was given in Figure 4.8 on page 52, in the previous chapter.

The measurement results are presented in the following way: in Appendix A, the data rate versus time mean value graphs for the five different measurements of each setup can be found. Note that all the graphs were truncated at 400 Kbps (y-axis), in all cases where the portion above this line was not empty, it shows a peak of 600 Kbps (similar to that shown on Figure 4.6). In the present section, the grading method described in Section 3.2.5 on page 36 is extended, and the decisions are motivated. The final gades according to the chosen grading method can be found in Appendix C. The most interesting results are discussed in the present section as well, the evaluation is based on the the final grades, the data rate versus time mean value graphs shown in Appendix A, and individual measurement results.

Recall from Section 3.2.5, that a grade was given to each indvidual playback, based on the encoding levels and picture quality observed during the playback, and the time duration under which the clip was played at the respective level. As the total time of the clip was 138 seconds, we have the following formula giving the grade:

$$\begin{aligned} grade \;=\; & weight_{220} * (time_{220}/138) + weight_{150} * (time_{150}/138) + \\ & + weight_{112} * (time_{112}/138) + weight_{90} * (time_{90}/138) + \\ & + weight_{badpicture} * (time_{badpicture}/138) \end{aligned}$$

Then, the mean value of the grades for the 5 measurements of each setup was calculated, and this mean value gave the grade assigned to the setup as a whole. The next issue that had to be considered was how to assign the weights (or grades) to the different encoding levels. As this

was not straightforward to establish, I tried out different weight sets. One important issue was how to distribute the weights, i.e. is a playback at the 220 Kbps encoding level twice as good as a playback at the 112 Kbps encoding level, as experienced by the user watching the video (this having been myself during the project work). I considered the following four weight distributions (sets):

220	150	112	90	bad picture
5	4	3	2	0
10	9	7	4	2
10	7	6	3	1
10	8	7	5	2

Table 5.1: The different weights that I considered to assign to the different encoding levels respectively picture qualities.

In fact, it is the picture quality that is weighted, but as there is a close relationship between the picture quality and the encoding level of the playback, this kind of grading system seemed fair. The first row of Table 5.1 corresponds to the simplest weighting model, but it does not allow a weighting that is proportional with the encoding level. Also, zero weight is assigned for the class "bad picture", which is not a good choice: even though the picture is bad, the experience of the user is better than if there was no video played at all. That is, I decided not to assign a zero value to this class. The second row in the table shows a grading where the 220, 150, and 112 Kbps encoding levels are considered very good, but the 90 Kbps encoding level quite bad. Here, a scale from 1 to 10 rather then 1 to 5 is used, as this allows for a finer differentiation between the different encoding levels, and I considered this more realistic. The last two rows are variations of the second one, that is, they use the same 1 to 10 scale, but assign different weights to the different classes. Based on my subjective experience, I had the feeling that the last row corresponded best to the actual relationship between the qualities of the playback for the different encoding levels. Nevertheless, a more adequate motivation for the choice was needed.

sleep/shaper	180	200	250	300	500	off
60	2.07	2.69	3.76	3.92	3.61	3.64
80	2.54	3.1	3.64	4.41	4.3	3.97
90	2.38	2.94	4.01	4.63	4.55	4.65
120	3.46	3.81	4.84	4.96	5	5
no	3.48	3.5	4.73	5	5	5

Table 5.2: The grades with the 5, 4, 3, 2, 0 weighting scheme.

sleep/shaper	180	200	250	300	500	off
60	5.34	6.17	8.38	8.47	7.46	7.58
80	5.99	7.08	8.02	9.25	8.93	8.14
90	5.22	6.73	8.77	9.41	9.2	9.55
120	7.53	8.61	9.81	9.94	10	10
no	7.99	7.91	9.73	10	10	10

Table 5.3: The grades with the 10, 9, 7, 4, 2 weighting scheme.

In order to make a more adequate choice of a certain weight set, than just the subjective "feeling", I proceeded as follows. I calculated the grades for each setup as the mean value of the

sleep/shaper	180	200	250	300	500	off
60	4.14	5.03	7.08	7.51	6.73	6.87
80	4.77	5.64	6.59	8.56	8.16	7.5
90	3.94	5.36	7.32	9.08	8.88	9.17
120	6.5	6.83	9.56	9.93	10	10
no	6.29	6.35	9.21	10	10	10

Table 5.4: The grades with the 10, 7, 6, 3, 1 weighting scheme.

sleep/shaper	180	200	250	300	500	off
60	5.28	6.3	7.92	8.21	7.65	7.69
80	6.01	6.82	7.59	8.98	8.73	8.23
90	5.57	6.58	8.17	9.35	9.21	9.41
120	7.44	7.79	9.69	9.94	10	10
0	7.3	7.38	9.47	10	10	10

Table 5.5: The grades with the 10, 8, 7, 5, 2 weighting scheme.

grades for the respective five measurements (as given by formula on page 54), for the above weight sets. The resulting grades for the thirty different setups are shown in the Tables 5.2 to 5.5.

Graphical representation of the Tables 5.2 to 5.5 can be found in Appendix B.

In fact, the grades are quite similar in all the cases relative to each other, so it seemed that it did not really matter which weight set I chosed. However, there are some fine differences, and I had to decide upon the most appropriate weight set. The method I used was looking at the graphs presented in Appendix A, respectively my notes about each playback, and in such a way established which of the weight sets was most appropriate. With "appropriate" I mean the following: for each setup, the grade should say something about the encoding level and picture quality that the setup resulted in. That is, I considered for example that the playbacks for the setup with the sleep value t=60 and shaper speed at 300 Kbps corresponded as a whole quite well to a 150 Kbps encoding level playback, based on the mean value graph and my notes. Thus the grade assigned to this setup should be as near as possible to the weight assigned to the 150 Kbps encoding level, which is the case with the weight set 10, 8, 7, 5, 2 as shown in Figure B.4 in Appendix B. Observe that here, the weight (grade) 8 corresponds to the 150 Kbps encoding level, and the point for the setup is just above the value 8. Also, as we see in Figure B.2 , in the case of the weight set 10, 9, 7, 4, 2 the grade for this setup is nearer to the weight of the 112 Kbps encoding level (namely 7). That is, the weight set 10, 8, 7, 5, 2 is more appropriate than the 10, 9, 7, 4, 2 one, for this setup. Based on such considerations, by checking the graphs and my notes against the resulting grades for the setups and the different weight sets, I decided that the set 10, 8, 7, 5, 2 was most appropriate. That is, the grades shown in Table 5.5 are the final ones that I decided for.

Now, we have a grade assigned to each setup calculated as the mean value of the 5 grades, where each of the 5 grades was calculated using the formula on page 54, as shown in Figure 5.1 (A). However, these grades reflect only what happened during the 138 seconds of the test clip. So they are relevant only for a clip of this length. If we do an approximation and say, that this period is long enough for the encoding level "to settle" and we suppose, that if the clip would be longer, the encoding level at which it would be played is strongly affected by the last one we see, we can draw more general conclusions that are not limited to the 138-second time frame. Let us suppose that this is true, that is, the playback would continue at the encoding level that we last experienced. So it would be nice if the grade could reflect this encoding level as well. That is, we do an additional calculation, namely we assign to each playback a separate grade based on the last encoding level that was experienced. This grade is the same as the weight of the encoding level at which the clip

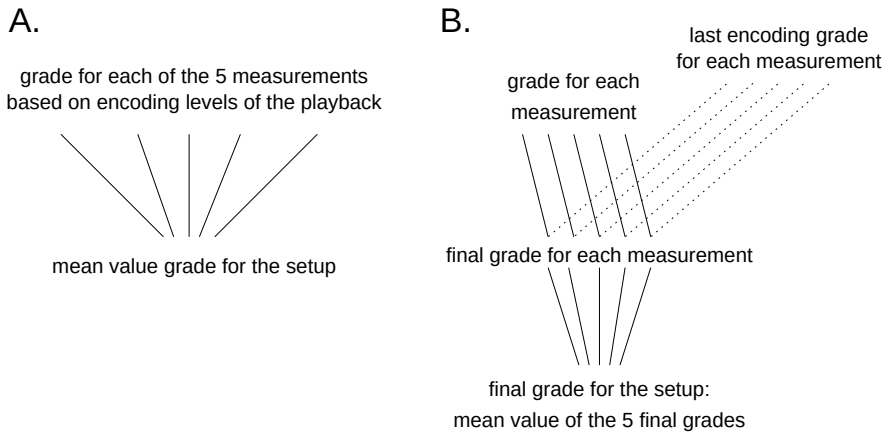

Figure 5.1: In the first scheme (A), the grade for a setup is the mean value of the five grades for the individual measurements, based on the formula on page 54. In the second scheme (B), the final grade for a measurement is the mean value of it's grade as given by the same formula as above, and it's "last encoding grade". The final grade for a setup is the mean value of the five final grades.

was played at the end. I.e. if a clip was played at the 112 Kbps encoding level at the *end* of the playback, the "last encoding grade" assigned to this measurement will be 7, which is the same as the weight corresponding to the 112 Kbps encoding level in our system. Now we calculate the mean value of the previously given grade and this "last encoding grade" for each measurement, let's call this value the "final grade for a measurement". The final grade assigned to each setup is then the mean value of the final grades of the five measurements for the respective setup. Figure 5.1 (B) shows how the final grade for a setup is obtained. The final grades resulting from this additional calculation are shown in Table 5.6.

sleep/shaper	180	200	250	300	500	off
60	5.34	6.25	7.66	7.6	6.32	6.34
80	5.9	6.41	7.09	8.49	7.26	6.61
90	5.28	6.19	7.68	8.67	8.4	9
120	7.72	7.89	9.84	9.97	10	10
no	7.55	7.39	9.73	10	10	10

Table 5.6: The final grades with the 10, 8, 7, 5, 2 weighting scheme.

Basically, these new grades do not say anything about the encoding level at which a, let's say, 30 minutes clip would be played when applying the respective server and network QoS levels. Rather, the idea behind it is to assign a significant weight to the last experienced encoding level - that is, to consider a playback that was good at the beginning and bad at the end to be much worse than a playback that was bad at the beginning and good at the end, or moderately good during the whole playback.

If we compare these new, final grades with those shown in Table 5.5, we see the following: for the server QoS setups with the sleep values t=60, 80, and 90 milliseconds, the final grades as shown in Table 5.6 are lower than the grades shown in Table 5.5, while for the best two server QoS levels (t=120 respectively no Java program running) the final grades are higher. These latter two correspond to the setups where the server QoS was so good that it hardly affected the playback, and the network QoS level was the limiting factor. From this we could draw the conclusion that,

within our model, the server QoS affects the encoding level at which the playback ends stronger than the network QoS does. Also, a bad server QoS will rather be reflected in the final encoding level, than in the quality of the playback as a whole - but this result is straightforward, as I have done the choices for the server QoS levels based on a similar criteria. Recall that the server QoS setups were establisched based on how many downshifts occured during the playback. That is, these results follow directly from the way the server QoS setups were chosen.

The biggest difference between the two tables are in the grades for the setups with sleep value t=60 and t=80, with the shaper off or at the 500 Kbps speed. This corner of the table is a notable one in any case. If we look at the rows of the tables, for each row (i.e. a certain server QoS setup) we see, that the grades tend to increase towards the right, that is, the grades increase as the shaper speed increases. This is exactly the behaviour that we expect: for a fixed server QoS, when the network QoS level gets better and better, the overall performance should also get better. However this increasing trend goes on only until the shaper speed value of 300 Kbps, thereafter the grades are lower again, for the server QoS setups with sleep values of t=60, 80, and 90. We can observe this regardless of the weighting system used, that is, on all of the Tables 5.2 to 5.6. Not surprisingly, the same behaviour is also reflected in the data rate versus time mean value grahs, see Appendix A. For example, look at Figures A.4 and A.5, showing the mean value graphs for the sleep value t=60 and shaper speed 300 respectively 500 Kbps.

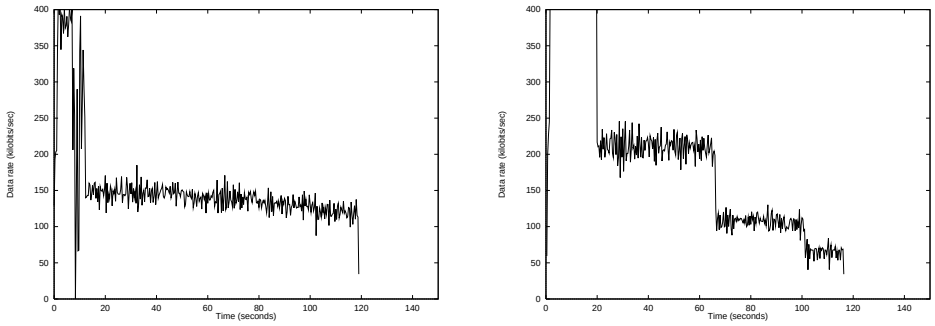

Figure 5.2: Data rate versus time graph for a single measurement for the setups with sleep value t=60 and the traffic shaper at 300 Kbps respectively 500 Kbps speed. Note that the graphs are truncated at 400 Kbps, as the unshown portion is irrelevant (the buffering implies a missing peak of 600 Kbps).

One individual measurement from the five for both setups are presented on Figure 5.2. First of all, one has to mention which encoding levels the clip was played at in these cases. For the measurement with the 300 Kbps shaper speed, the playback began at the 220 Kbps encoding level, a downshift to the 150 Kbps encoding level occured after eight seconds, and an additional downshift to the 112 Kbps encoding level occured after an other 37 seconds (at time 0:45). Thereafter, no more downshifts occured. In the case of the measurement with the 500 Kbps shaper speed, at the begininng there was a strong buffering period of 16 seconds, the clip was played at the 220 Kbps encoding level under this period, until a total of 61 seconds (time 1:01 in minutes). Then there was a downshift to the 150 Kbps encoding level. At time 1:26, that is, 25 seconds later, a downshift to the 90 Kbps encoding level occured, which was kept for the rest of the playback.

Let us try to understand what happened. In the case of the measurement with the shaper at 300 Kbps, at the beginning, the player sensed that the bandwith available for it was too little, so no buffering at 600 Kbps, as in the normal case, could be done. Eventually after some negotioations with the RealServer, the "best-fit" encoding level (150 Kbps) was found. This encoding level could be kept up at the beginning, as the initial peak made some buffering possible, that is, the buffer was not totally empty. However as the rate of data arriving to the RealPlayer was not enough for a play at 150 Kbps encoding level, and the buffer emptied, eventually a downshift occured. We can not see on the graph, when the downshift occured - the data rate is slightly decreasing all the

time, without any sudden drops. The fact that the data rate now was higher than the nominal encoding level, 112 Kbps, is due to the buffering performed by RealPlayer: a data rate that was bigger than the nominal encoding level was sent now, as long as the buffers were not filled up. That is, one could say that the behaviour experinced here was affected by the buffering and the feedback mechanism. For the measurement with the shaper at the 500 Kbps speed, we have a similar scenario as was presented in Section 4.2, with a detailed explanation on page 50. Shortly, the playback began with a buffering period, after which the server sended at the nominal encoding level. However, as it did not get to run often enough, it could only send at a slightly lower data rate than the nominal value; as a consequence, a downshift was requested by RealPlayer. The same thing happened in the case of the thereafter following encoding levels as well. That is, the feedback mechanism of RealPlayer combined with the sever QoS setup caused the above described behaviour, along with the buffering. We can also observe, that though the downshift from the 150 Kbps encoding level to the 90 Kbps encoding level occured at 86 seconds, the data rate goes down to 90 Kbps only at about 100 seconds, this was probably also due to buffering performed in these 14 seconds.

We have discussed about the two individual measurements; let us now make a comparison. It seems that what we have seen, is strongly affected by the buffering policy implemented by RealNetworks' software. The fact that the playback was actually better in the case of the 300 Kbps shaper speed setup is due to this buffering policy, as far as I can understand it. After the downshift to the 112 Kbps encoding level, data was sent at a higher rate than 112 Kbps, with the aim of filling up the buffers. That is, though the server did not get to run as often as in the case when we had no server QoS applied, the data rate that arrived to RealPlayer was high enough to keep up the 112 Kbps encoding level. The fact that there was one downshift from the 150 Kbps to the 112 Kbps encoding level is due to the initial peak, that made it possible to partly fill up the buffers.

As explained above, the tables show generally increasing grades both in the direction of increasing shaper speeds (from left to right, along each row) and in the direction of increasing sleep values (from top to bottom, along each column). An exception to this was the top right corner, which we have just examined. Some other, but basically unsignificant exceptions also occur. These are mostly due to influences from other traffic in the same network that I was experimenting with. However, another "hot corner" that deserves some words is the one with the shaper set at the 180 Kbps speed. With this network QoS setup, a lot of problems occured. Often, the RealPlayer could not even start playing, this is due to the strong back-off of the RealServer - as only about 30% of the traffic went through in the first seconds (180 Kbps instead of 600 Kbps), RealPlayer probably reported a huge loss to RealServer, after which RealServer decided not to send more data. After all, I managed in all cases (for all of the setups) to make at least 5 measurements with an acceptable play.

A general and quite important observation that we can make is, that having a very good QoS level for one of the server or network, and a very bad QoS level for the other (e.g. good server QoS and bad network Qos, respectively bad server QoS and good network QoS) resulted in a lower performance level than if both QoS levels were moderate. If we look at Tables 5.6 and 5.5, we see that the values in the middle are higher than in both top-right and bottom-left corners. I considered this to be the result of the following interactions:

- if the shaper is set to a low value, RealPlayer reports a low bandwidth back to RealServer, as a result of the negotiations a (more or less) optimal encoding level is found
- if the shaper is set at a fairly low value, no buffering can be done and thus later a higher data rate than the nominal encoding level is sent in order to fill the buffer, this is enough to keep up the current encoding level even if the server runs at a low QoS

Also, we can obseve that the server QoS on it's own has a stronger effect on the performance (grade) than the network QoS on it's own. This is due to the fact that I chose the server QoS levels "backwards" (starting from the performance levels, as expressed in downshifts, that I experienced) in such a way, that the lowest QoS level resulted in a very low performance. The network QoS

level choices were based on the encoding levels of the clip, but even with the lowest network QoS level (180 Kbps shaper speed) playing at the 150 Kbps encoding level is theoretically possible. Lower shaper speeds made an enjoyable play impossible.

5.2 Reallocating Resources

In the previous section, the motivation for the grading system's choice was presented, and the results of the measurements based on the data rate versus time graphs and the given grades were discussed. In this section, some possibilities of using the results in the ICorpMaker project are presented.

Recall from Chapters 1 and 2, that initially the ASP's clients get a certain amount of server and network resources allocated to their instant corporation. Thereafter, a dynamic solution of changing the resources allocated to them is offered to the clients. The clients monitor the performance of their applications, and if they are not satisfied with the current performance, they can change the resources allocated to their ICorp. The clients are however not required to have an exact knowledge about the amount of physical resources they posses, rather, they should be able to just request "more" resources, if their applications perform unsatisfactory, or "less" resources, if they consider that necessary. The aim of this thesis project work was to study how the two QoS levels, at the server and at the network level, combine, and to give an answer to the question of how we need to change the resources allocated to an ICorp when the client requests more resources.

The study I conducted was for the case of a video streaming application (RealServer, RealPlayer) and the results are also applicable only for this kind of applications, in particular for RealNetworks' software.

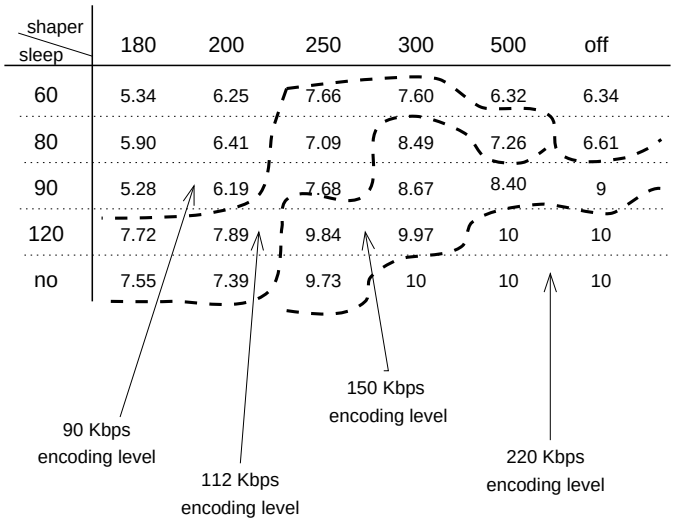

Figure 5.3: Encoding levels corresponding to the grades.

As presented in the previous section, the answer to the question "how to change the resource allocation" is not straightforward. For the purpose of this evaluation, I transformed this question to the following one: supposing that RealPlayer plays at a certain encoding level, how can you modify the server respectively network QoS levels so that the clip is played at the next higher encoding level. Let us base the evaluation on the grades obtained for the different setups in the previous section. We group the grades into four sets, according to which encoding level they correspond. That is, all values between 5 and 7 correspond to the 90 Kbps encoding level, all values between 7 and 8 correspond to the 112 Kbps encoding level, all values between 8 and 10

correspond to the 150 Kbps encoding level, and the value 10 to the 220 Kbps encoding level. Figure 5.3 shows the four obtained sets. We call such a table for a *resource allocation map*, the different server and network QoS levels being the coordinates, and the obtained grade or performance level the third dimension.

Now, the question of "how to modify the server and network resources" so that the playback is at the next encoding level, can be answered easily. We just have to locate where on the resource allocation map we are, that is, which are the current resource allocation coordinates (parameters). Then, we have to look for a setup with coordinates such that the encoding level corresponds to the next higher value, and change the resource allocations i.e. our position on the map, accordingly. We can chose this new position such that the changes in the resource allocation that we have to commit are minimal (the "nearest place" to our current position), or we can base our choice on some other criteria. Let us look at a simple example. Suppose that the server QoS value is the sleep value 60, the shaper is at 250 Kbps, and thus the playback corresponds to the 112 Kbps encoding level. On Figure 5.4 we see the encoding levels corresponding to each setup, with the first arrow pointing to where we currently are on the map. Now, to have a playback at the 150 Kbps encoding level, we just have to look at the map and recognize a setup for which the corresponding encoding level is 150 Kbps, and move there. The second arrow shows this on the resource allocation map presented on Figure 5.4. Here, we moved to the "nearest" position that satisfied the request, but other choices could also have been possible. The same procedure is applicable in the case when we want not only to move to the next higher encoding level, but the highest encoding level possible, i.e. 220 Kbps.

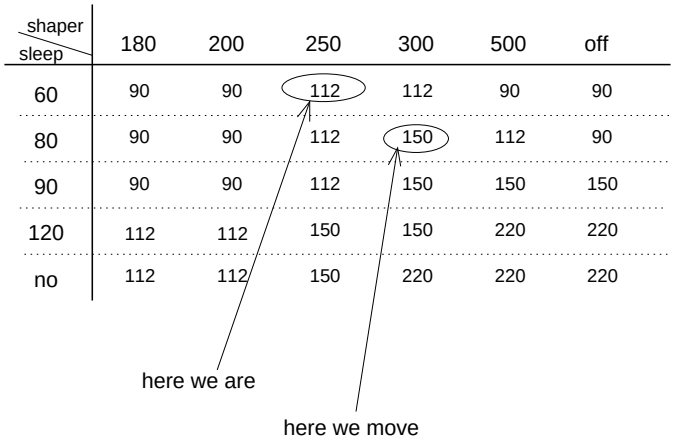

Figure 5.4: Resource allocation map showing the encoding levels corresponding to the different setups, with the arrows pointing to where we are and where do we have to move, if we want to increase the encoding level of the playback from 112 Kbps to 150 Kbps, as in the example on page 61.

There is another interesting way of using the results shown in the above maps. As we see, there are several setups that result in the same encoding level, e.g. the setup with sleep value t=120 and shaper speed of 250 Kbps respectively the setup with t=90 and the shaper at 500 Kbps. Notice that the shaper speed in the second case is twice as big as in the first case. Now, suppose that the ASP wants to do a kind of "load balancing" so that the servers and the network links are equally loaded - that is, suppose there are two clients that at the moment experience a playback at the 112 Kbps encoding level, and both ask for "more" resources, requesting that the playback would be at the 150 Kbps encoding level. It might be better from a "load balancing" viewpoint to satisfy these two request in such a way, that one client gets 500 Kbps bandwith and the other one 250 Kbps along with the corresponding server resources, than to satisfy both requests by giving

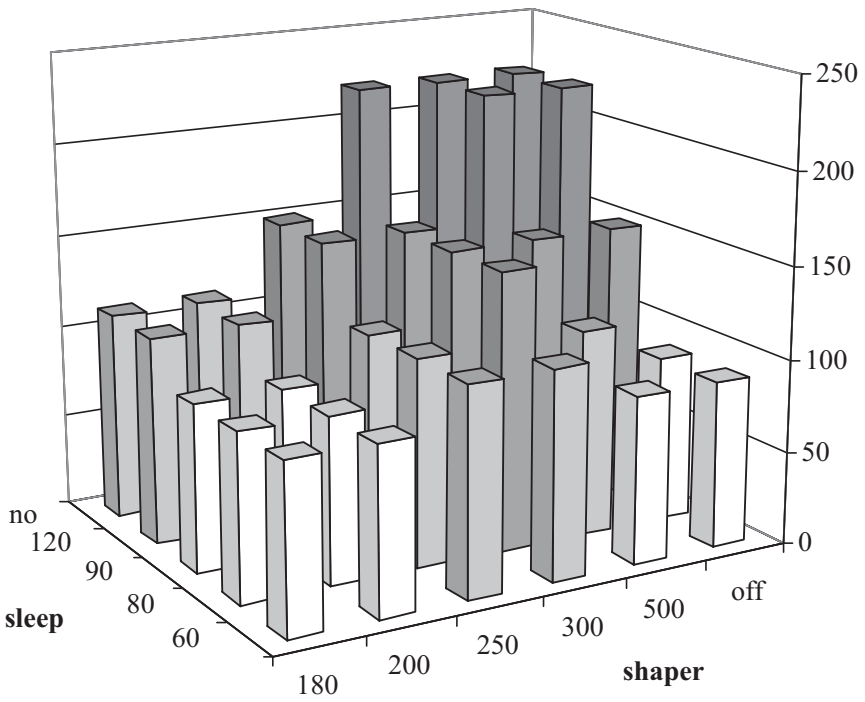

Figure 5.5: The 3D map of resource allocation, with the encoding levels for the different setups being the z-axis.

them both 500 Kbps bandwidth. Respectively, suppose that the ASP has only one server which is quite heavy loaded, but there is plenty of bandwith available. Then, when clients request "more" resources, one would try to move into a "better area" by allocating more bandwidth, but possibly no additional server resources. E.g. suppose a client has server QoS level with sleep value t=90, and the shaper at 180 Kbps speed. The playback is at the 90 Kbps encoding level. Now the client asks for more resources, that is, we have to allocate the resources so that the playback would be at the 112 Kbps encoding level. Then, rather than modifying the sleep value to t=120, which would put more load on the server, we increase the shaper speed to 250 Kbps.

Figure 5.5 shows the resource allocation map with the encoding levels corresponding to the different setups as a 3D-image. The z-axis corresponds to the encoding level. For the ease of interpretation, the different encoding levels are shown with different colors in greyscale.

One last issue related to resource allocations has to be discussed here, even though it does not relate to the grading system or to how the server and network QoS combine; it is however considered to be an interesting result obtained from the experiments. As we have seen, at the beginning of a playback there is a buffering period, which in the case of a UDP stream results in a data rate at about 600 Kbps for about 20 seconds, in the case of our test clip. After this period, the data rate is at a constant value, e.g. 220 Kbps in the case of a normal playback. This means that the network bandwidth allocated to the respective stream has to be as big as to allow a 600 Kbps data rate for the first 20 seconds, after which period a bandwidth of about 250 Kbps is enough for a good quality playback. That is, the ASP can adjust the network resources allocated to a single stream accordingly. Also, if there are multiple requests coming in at the same time, the ASP can schedule the start of the playbacks in such a way, that as little as possible (or no) playbacks will have their buffering at the same time. That is, the ASP can decorrelate the peak required bandwidth of the different streams. In this way, a bandwidth of for example 1200 Kbps could be used for the transmission of 3 simultaneous streams instead of only two. Figure 5.6 shows

this later concept.

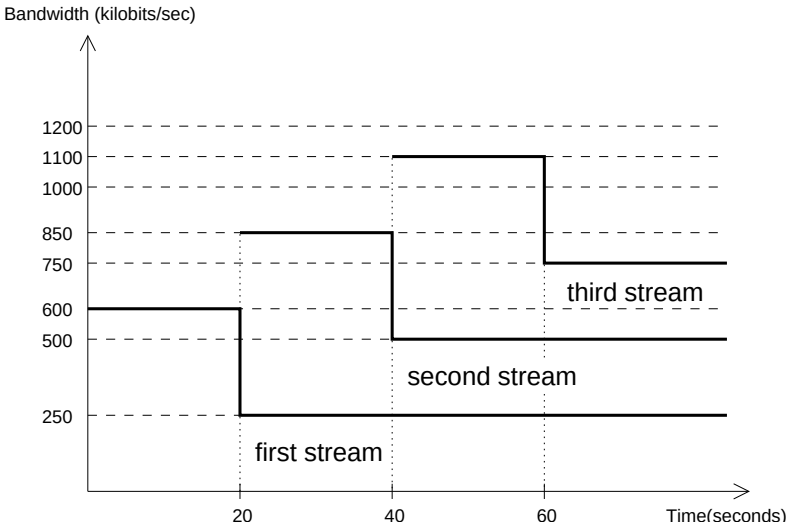

Figure 5.6: Total bandwith usage of 3 streams, if their starting time is sheduled such that the buffering period for the individual streams follow after each other.

5.3 Maximizing Performance for a Given Cost

An issue that is always important, when it comes to "real life", is cost. What the clients want, is to get the best possible service level or performance for the money they have. That is, we have to look at how one can maximize the performance for a given cost, given the data that we already have.

sleep value	60	80	90	120
cost	50	66	75	100

Table 5.7: The cost assigned to each server QoS setup.

Let us assign a linear cost to both the server and the network resources. From the five server and six network setups, we can only assign a cost to four respectively five setups, as the setups with no limitation (e.g. no Java program and shaper off) are not realistic in this case. Let us give the cost 100 to the best QoS levels for both server and network setups, and assign a proportional cost to the other setups. Then, we get the costs shown on Table 5.7 for the different server setups, respectively the costs shown on Table 5.8 for the different network setups.

shaper speed	180	200	250	300	500
cost	36	40	50	60	100

Table 5.8: The cost assigned to each network QoS setup.

Using this cost distribution, we can easily calculate the total cost for a certain server and network QoS setup, that is, for each of the 20 setups that are considered "valid" (i.e. have some limitations at both server and network level). Figure 5.7 shows a resource allocation map with the

encoding level	90	112	150	220
minumum cost	86	100	126	200
maximum cost	115	166	175	200
percentage	133%	166%	138%	100%

Table 5.9: Minimum and maximum costs that we get for each setup, as well as a value expressing the maximum value as percentage of the minimum.

cost for each setup, and the encoding level corresponding to the respective setup. With this scheme for assigning the costs, there are quite big variations between the minimum and the maximum cost values that we get for each encoding level. This is summarized in Table 5.9. The last row of the table shows the difference in percentage, i.e. the maximum value relative to the minimum value.

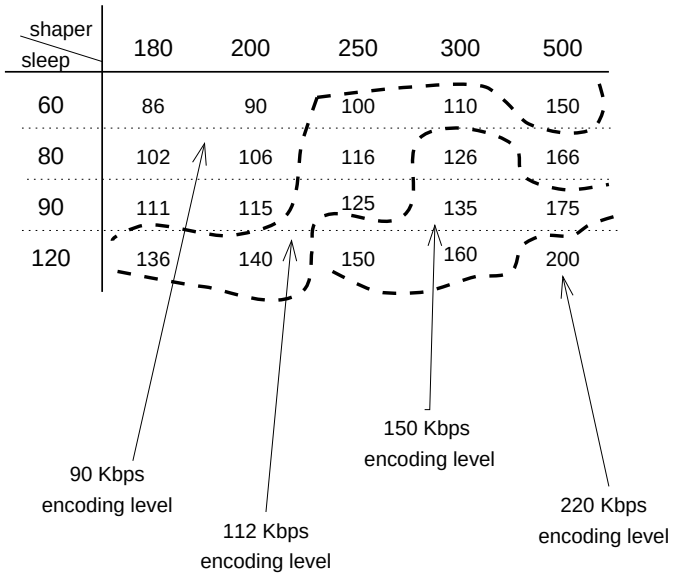

Figure 5.7: Resource allocation map with the cost for each setup and the encoding level corresponding to the respective setup.

Also, with this scheme it is straightforward to tell for a certain cost level, how the resources have to be allocated in order to get a maximum level of performance. That is, with a fixed cost value of 100 we can allocate the resources in such a way, that the encoding level will be at the possible maximum of 112 Kbps, by having a sleep value of 60 and the shaper at 250 Kbps.

In the above scheme, the term "cost" was used in a meaning similar to the "amount of money that a client would have to pay" for the resources. It also presented a very simple way of assigning the costs to the different resource allocation setups. However, the term "cost" is not necessarily directly coupled to the money payed by the client. Rather, costs are internal to the ASP and probably time variant, i.e. costs can reflect the importance of a given resource from other viewpoints. For example, the cost of the last 20% of the ASP's resources is significantly greater then that of the first 80%, while having used up 80% of the resources means for the ASP that he either will have to refuse requests from the clients or will have to extend his existing infrastructure. However, he should not charge the clients for this. Very sophisticated and complex charging systems can be developed based on different considerations, the ideas presented in this section gave only a very

simple and short introduction to this area and were used for motivation purposes only.

To summarize, we can say that the results obtained from the measurements are interesting on their own, as well as can be used directly in the context of the ICorpMaker for decisions and optimizations in the procedure of (re)allocating resources. The evaluation is certainly not exhaustive, a lot more issues could possibly be discussed and the results could possibly be used in a variety of other ways.

Chapter 6

Conclusions

From the work and results presented in this report, we can conclude that it is very difficult to predict or define a priori which resource reservation settings will yield which end-to-end performance, even in the case of a simple setup.

A generally valid result obtained from the measurements is that imbalance in the resource reservations always leads to a poor end result, that is "very bad" at one side combined with "very good" at the other side (e.g. sleep value 60 and no shaper, in our sever and network QoS setups) gives a worse end-to-end performance than moderate levels of both server and network QoS combined.

The other results obtained from the measurements are specific to the studied application. Nevertheless, the exploratory work preceding the measurements showed many noteworthy properties that might have general significance. For example it became clear that applications are not able to distinguish between network and server QoS limitations. In particular, RealServer always supposes that the problems arise from network congestion. Optimizations in this respect could allow for improvements. We tried to analyse application performance in a multidimensional QoS space, but it turned out to be hard to tune the QoS even for only one of the dimensions. This is especially true for server resources. The QoS classes (nature and amount of resources) are hard to establish, and probably have an other impact on different applications.

In the ICorpMaker the applications are isolated from each other by means of logical server and network partitioning, so it is in fact the application specific knowledge that the ASP needs. Having exact knowledge about the behaviour and requirements of the applications they host, ASPs can optimize the (re)allocation of resources. This implies that ASPs have to either first study the behaviour and QoS requirements of each application or have an a priori knowledge of it.

In this project, we have studied the behaviour of a video streaming application. We have concentrated on issues specific to ICorpMaker at the evaluation of the results. To simplify the evaluation, we have mapped the abstract notion of "user satisfaction" to the encoding levels at which a clip is played, this being our performance analysis measure. The obtained results, shown in the resource allocation maps presented in the previous chapter, give an answer to the initial question of how to modify the resource allocation for a client if he asks for "more" resources. Some optimization possibilities in the resource (re)allocation procedure were also pointed out. We have observed that the bandwidth for UDP video streams has to be large at the beginning of the playback so as to make the high data rate of the buffering period possible, but after this period the bandwidth can be reduced. That is, a correlation of different stream playback times leads to more effective bandwidth utilization. An other observation that can be used for optimizations was that a certain performance level can be obtained with different resource allocation setups. This allows the ASP to always use that server and network resource allocation tuple that is most convenient, according to some constraints (e.g. availability, cost).

A further contribution of the work was the exploration of the nature of the commercial video streaming application suite produced by RealNetworks. Numerous obstacles arose from the fact that the protocols used by this software are proprietary. However, this software was used for testing in the ICorpMaker project, hence an understanding of these protocols was important.

Chapter 7

Future work

The work and the results presented in this report were based on simple setups and simple performance analysis methods. The study was conducted for only one application, and one data stream. In all of these areas, that is, performance analysis, server and network QoS, the set of studied applications, and number of streams, a lot of additional work can and probably will be done.

In order to gain a more adequate understanding of the problem and to be able to give more adequate answers, better performance analysis methods of the studied applications are needed. In the case of video streaming applications, the next step would be to use software that uses RTP and RTCP, and base the performance analysis on RTCP message analysis. Measurement of other parameters, such as delays, jitter, etc. would also be needed for a better understanding of the application's behaviour. All in all, more sophisticated performance measurement and evaluation techniques are needed.

The presented work concentrated only on one single application. The next steps would be to study the behaviour of other video streaming applications, as well as the behaviour of other kinds of applications, such as web traffic, interactive games, etc. After studying the application in the case of one data stream, studies need to be conducted for multiple data streams that run in parallel as well - it is not certain, that using ten data streams, for example, will multiply the amount of resources needed by ten, quite the contrary.

The network QoS setups used were very simple, more advanced methods such as DiffServ or MPLS need to be explored as well. As the ICorpMaker project evolves, a more adequate way than that of using VMWare for server partitioning will be necessary. Distribution of the servers is a step that will also follow. As soon as a "complete ICorp implementation" is in place, new measurements and their evaluation need to be done. The studied dimensions - network QoS expressed in bandwidth and server QoS expressed in CPU cycles - are just two axes from the multidimensional problem space, others await exploration.

There are a lot of directions in which the work can continue, however the current project offers a basis that such future work can build on.

Bibliography

[1] Aquila - Adaptive Resource Control for QoS Using an IP-based Layered Architecture Website. WWW, May 2000. http://www-st.inf.tu-dresden.de/aquila.

[2] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Services. RFC Informational 2475, IETF, December 1998.

[3] R. Callon, P. Doolan, N. Feldman, A. Fredette, G. Shallow, and A. Viswanathan. A Framework for Multiprotocol Label Switching. Internet draft, IETF, September 1999.

[4] A. Cerpa, J. Elson, H. Beheshti, A. Chankhunthod, P. Danzig, R. Jalan, C. Neerdaels, T. Schroeder, and G. Tomlinson. NECP the Network Element Control Protocol. Internet draft, IETF, February 2000.

[5] Darwin Streaming Server. WWW, 2000. http://www.publicsource.apple.com/projects/streaming.

[6] D. Durham, J. Boyle, R. Cohen, S. Herzog, R. Rajan, and A. Sastry. The COPS (Common Open Policy Service) Protocol. RFC Standard 2748, IETF, January 2000.

[7] David Durham and Raj Yavatkar. *Inside the Internet's Resource reServation Protocol – Foundations for Quality of Service*. Wiley, 1999.

[8] Ensim Corporation. *ServerXChange - A Complete Service Deployment Platform*, October 1999. Technical White Paper.

[9] Figure: A VPC bundles together a set of VCs traveling along the same path (with permission of the IEC Web ProForum and Nortel). WWW, June 2000. http://www.iec.org/tutorials/atm_fund/topic03.html.

[10] Figure: VMWare for Windows NT running Linux and Windows 2000 simultaneously (with permission of VMWare, Inc.). WWW, June 2000. http://www.vmware.com/products/presentations/tour/tour_slide04.

[11] Google search site. WWW, July 2000. www.google.com.

[12] Grid Forum. WWW, August 2000. http://www.gridforum.org.

[13] Grid Performance Working Group. WWW, August 2000. http://www-didc.lbl.gov/GridPerf.

[14] B. Gleeson, A. Lin, J. Heinanen, G. Armitage, and A. Malis. A Framework for IP Based Virtual Private Networks. RFC Standard 2764, IETF, February 2000.

[15] Stephane Gruber, Jennifer Rexford, and Andrea Basso. Protocol Considerations for a Prefix-Caching Proxy for Multimedia Streams. Technical report, ATT Labs - Research, April 2000. http://www9.org/w9cdrom/349/349.html.

[16] D. Gunter, B. Tierney, B. Crowley, M. Holding, and J. Lee. NetLogger: A Toolkit for Distributed System Performance Analysis. *Proceedings of the IEEE Mascots 2000 Conference*, August 2000.

[17] Information Society Technologies Programme (IST) website. WWW, July 2000. http://www.cordis.lu/ist/home.html.

[18] IEEE. *Virtual Bridged Local Area Networks, P802.1Q/D1*, May 1997. IEEE Draft Standard.

[19] Stardust.com Inc. Introduction to QoS Policies. Technical report, Stardust.com, Inc., Campbell, California, USA, September 1999.

[20] ISO. *GENERIC CODING OF MOVING PICTURES AND ASSOCIATED AUDIO INFORMATION: VIDEO Recommendation ITU-T H.262 ISO/IEC 13818-2*, November 1994. Draft International Standard.

[21] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. RFC Standard 2401, IETF, November 1998.

[22] P. Key, D. McAuley, and P. Barham. Congestion Pricing for Congestion Avoidance. Technical report, Microsoft Research, February 1999.

[23] Linux kernel documentation. /usr/src/linux-2.4.0-test7/Documentation/Configure.help.

[24] Linux kernel documentation. /usr/src/linux-2.4.0-test7/Documentation/networking/shaper.txt.

[25] Linux Weekly News. WWW, 1998. http://lwn.net/1998/1119/shaper.html.

[26] Linux 2.2.11 Manual Page for the Functions sched_setscheduler, sched_getscheduler. Linux Man Page, August 1999.

[27] Nemesis Operating System. WWW, February 2000. http://nemesis.sourceforge.net.

[28] NetLogger. WWW, August 2000. http://www-didc.lbl.gov/NetLogger/.

[29] Network Traffic Measurements and Experiments, May 2000. IEEE Communications Magazine.

[30] Ntop. WWW, July 2000. http://www-serra.unipi.it/ ntop/ntop.html.

[31] Pathchar. WWW, November 1998. ftp://ftp.ee.lbl.gov/pathchar/.

[32] Pchar. WWW, June 2000. http://www.employees.org/ bmah/Software/pchar/.

[33] Quick Time. WWW, 2000. http://www.apple.com/quicktime/download/.

[34] Real Server Administration Guide, RealSystem G2, 1998. RealNetworks, Inc.

[35] Real Player Plus G2 Manual, 1999. RealNetworks, Inc.

[36] Real Producer Plus G2 User's Guide, Version 6.1, 1999. RealNetworks, Inc.

[37] Real Networks Firewall Support Page. WWW, January 2000. http://service.real.com/firewall/fdev.html.

[38] Real Networks Homepage. WWW, August 2000. http://www.real.com.

[39] rshaper: a Traffic Shaper for Linux. WWW, 2000. http://www.linux.org/apps/AppId_2030.html.

[40] K. Ramakrishnan and S. Floyd. A Proposal to add Explicit Congestion Notification (ECN) to IP. RFC Experimental 2748, IETF, January 1999.

[41] D. Reed, I. Pratt, P. Menage, S. Early, and N. Stratford. Xenoservers: Accountable Execution of Untrusted Programs. *IEEE Proceedings of Hot Topics in Operating Systems*, 1999.

[42] Sean Rooney. A Control Architecture for Lightweight Virtual Networks. *Proceedings of DSOM'2000*, December 2000.

[43] Sean Rooney. Figure: A sample ICorp infrastructure, September 2000. The ICorpMaker – A Dynamic Framework for Application Service Providers.

[44] Sean Rooney. The ICorpMaker – A Dynamic Framework for Application Service Providers. *Proceedings of IPOM'2000*, September 2000.

[45] Sean Rooney, Christian Hoertenagl, and Jens Krause. Automatic VLAN Creation Based on On-line Measurement. *ACM Computer Communication Review*, June 1999.

[46] Eric C. Rosen, Arun Viswanathan, and Ross Callon. Multiprotocol Label Switching Architecture. Internet draft, IETF, August 1999.

[47] Alessandro Rubini. *Linux Device Drivers.* O'Reilly, 1998.

[48] H. Schulzrinne, A.Rao, and R. Lanphier. Real Time Streaming Protocol (RTSP). RFC Standard 2326, IETF, April 1998.

[49] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC Standard 1889, IETF, January 1996.

[50] Tcptrace. WWW, September 1999. http://jarok.cs.ohiou.edu/software/tcptrace/tcptrace.html.

[51] Tequila - Traffic Engineering for Quality of Service in the Internet, at Large Scale Website. WWW, March/June 2000. http://www.ist-tequila.org.

[52] Trend. WWW, January 2000. http://www.desktalk.com.

[53] The Mbone Video Conferencing Tool vic. WWW, 1998. http://www.cs.columbia.edu/ yhwang/ftp/docs/mbone-app.htm#vic.

[54] VitalAgent. WWW, August 2000. http://www.lucent-networkcare.com/software/vitalsuite/vitalagent/demo/index.asp.

[55] VitalSuite. WWW, August 2000. http://www.lucent-networkcare.com/software/vitalsuite.

[56] VMWare Homepage. WWW, June 2000. http://www.vmware.com.

[57] Suba Varadarajan. Virtual Local Area Networks. WWW, February 1980. ftp://ftp.netlab.ohio-state.edu/pub/jain/courses/cis788-97/virtual_lans/index.htm.

[58] Dinesh Verma. *Supporting Service Level Agreements on IP Networks.* Macmillan Technology Series, 1999.

[59] VMWare. *Getting Started Guide for Linux*, June 2000. Technical Reference Notes.

[60] WebProForum. Asinchronous Transfer Mode Fundamentals Tutorial. WWW, June 2000. http://www.webproforum.com/atm_fund.

[61] Matt Welsh. Implementing Loadable Kernel Modules for Linux - Loading and unloading kernel modules on a running system. *Dr. Dobb's*, 1995. http://www.ddj.com/articles/1995/9505/9505a/9505a.htm.

Appendix A

Mean Data Rate Versus Time Graphs

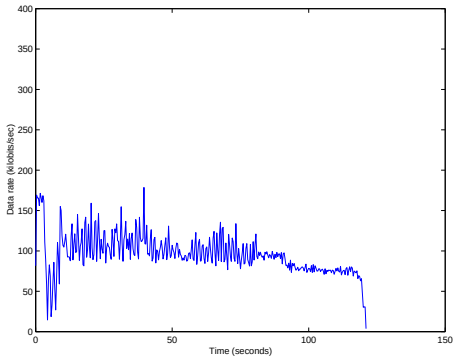

Figure A.1: Mean value graph for the 5 measurements for the setup with sleep value t=60 in the Java program, and the traffic shaper at speed 180 Kbps.

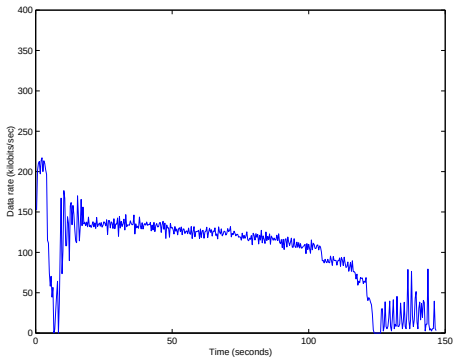

Figure A.2: Mean value graph for the 5 measurements for the setup with sleep value t=60 in the Java program, and the traffic shaper at speed 200 Kbps.

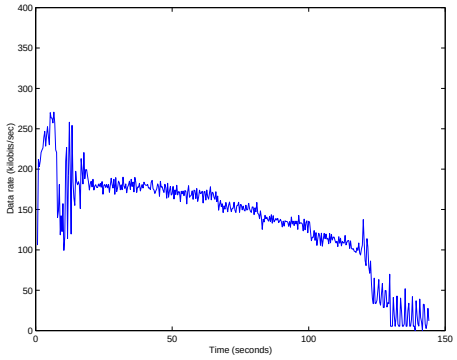

Figure A.3: Mean value graph for the 5 measurements for the setup with sleep value t=60 in the Java program, and the traffic shaper at speed 250 Kbps.

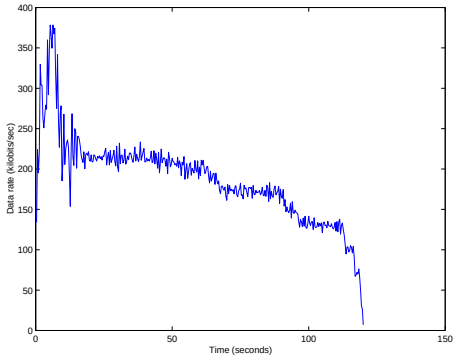

Figure A.4: Mean value graph for the 5 measurements for the setup with sleep value t=60 in the Java program, and the traffic shaper at speed 300 Kbps.

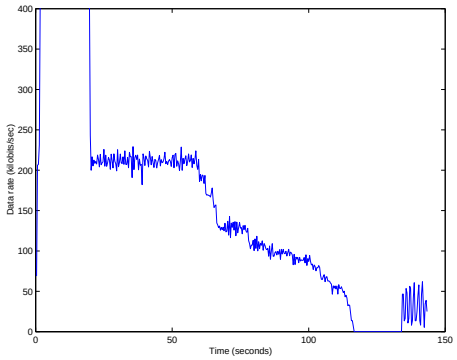

Figure A.5: Mean value graph for the 5 measurements for the setup with sleep value t=60 in the Java program, and the traffic shaper at speed 500 Kbps.

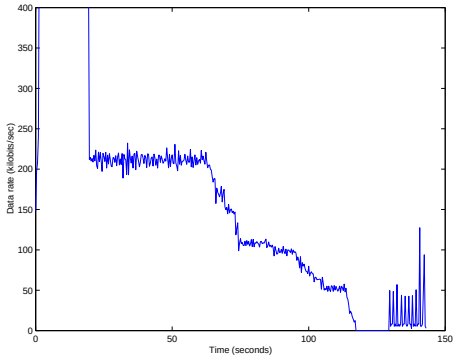

Figure A.6: Mean value graph for the 5 measurements for the setup with sleep value t=60 in the Java program, and the traffic shaper off.

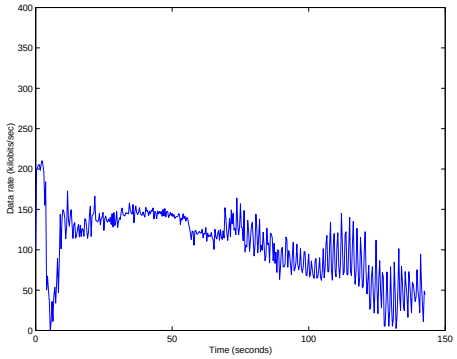

Figure A.7: Mean value graph for the 5 measurements for the setup with sleep value t=80 in the Java program, and the traffic shaper at speed 180 Kbps.

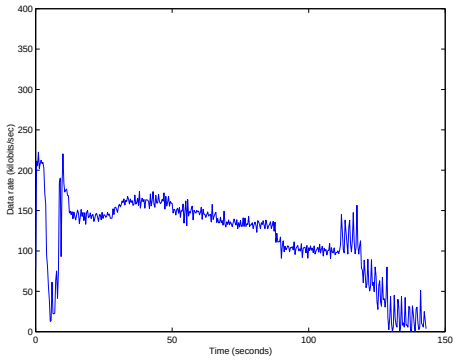

Figure A.8: Mean value graph for the 5 measurements for the setup with sleep value t=80 in the Java program, and the traffic shaper at speed 200 Kbps.

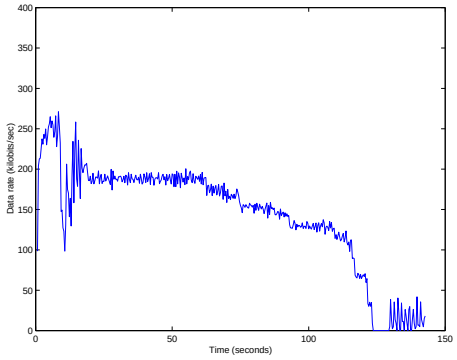

Figure A.9: Mean value graph for the 5 measurements for the setup with sleep value t=80 in the Java program, and the traffic shaper at speed 250 Kbps.

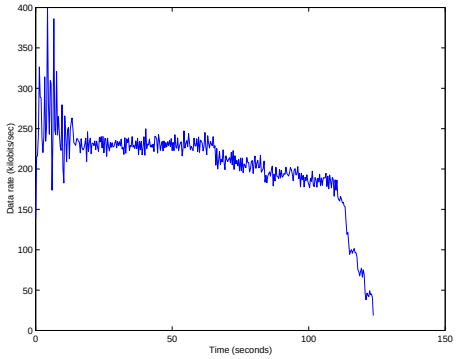

Figure A.10: Mean value graph for the 5 measurements for the setup with sleep value t=80 in the Java program, and the traffic shaper at speed 300 Kbps.

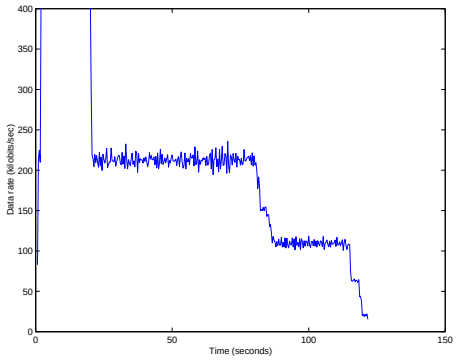

Figure A.11: Mean value graph for the 5 measurements for the setup with sleep value t=80 in the Java program, and the traffic shaper at speed 500 Kbps.

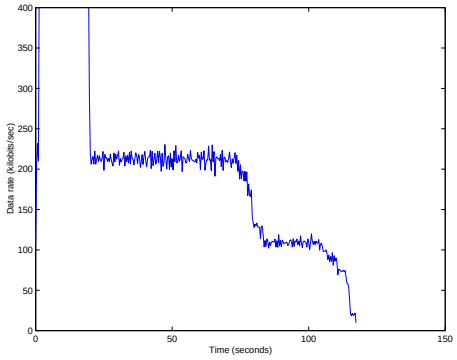

Figure A.12: Mean value graph for the 5 measurements for the setup with sleep value t=80 in the Java program, and the traffic shaper off.

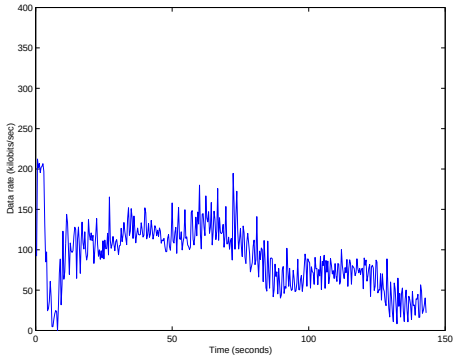

Figure A.13: Mean value graph for the 5 measurements for the setup with sleep value t=90 in the Java program, and the traffic shaper at speed 180 Kbps.

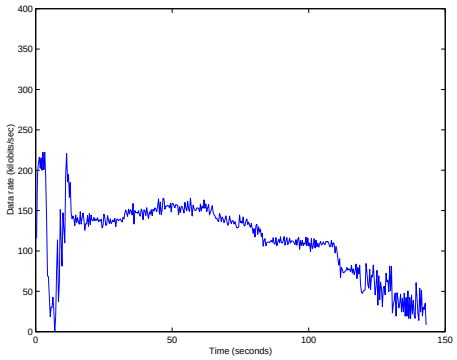

Figure A.14: Mean value graph for the 5 measurements for the setup with sleep value t=90 in the Java program, and the traffic shaper at speed 200 Kbps.

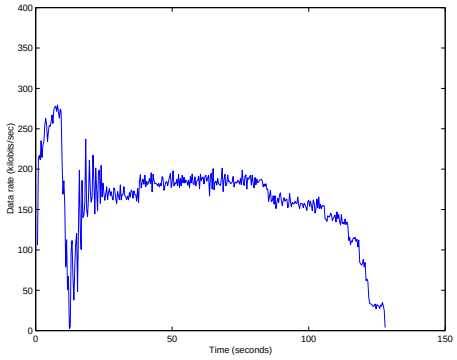

Figure A.15: Mean value graph for the 5 measurements for the setup with sleep value t=90 in the Java program, and the traffic shaper at speed 250 Kbps.

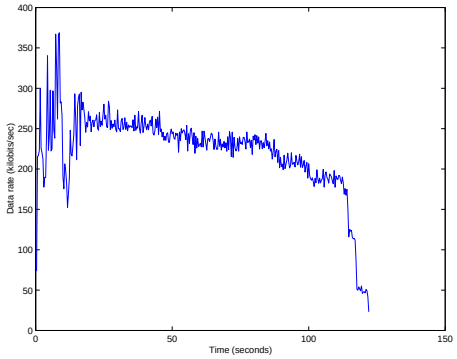

Figure A.16: Mean value graph for the 5 measurements for the setup with sleep value t=90 in the Java program, and the traffic shaper at speed 300 Kbps.

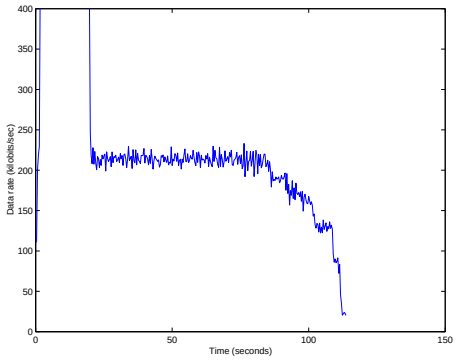

Figure A.17: Mean value graph for the 5 measurements for the setup with sleep value t=90 in the Java program, and the traffic shaper at speed 500 Kbps.

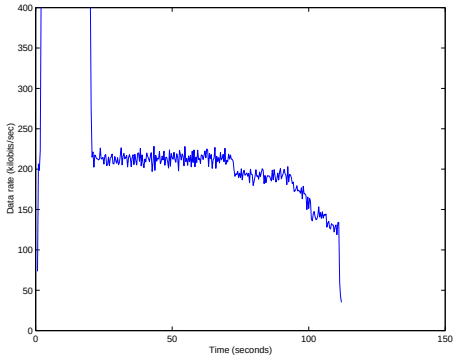

Figure A.18: Mean value graph for the 5 measurements for the setup with sleep value t=90 in the Java program, and the traffic shaper off.

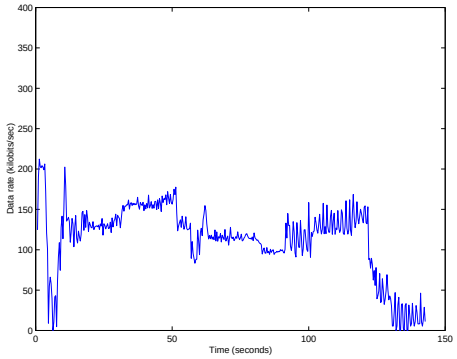

Figure A.19: Mean value graph for the 5 measurements for the setup with sleep value t=120 in the Java program, and the traffic shaper at speed 180 Kbps.

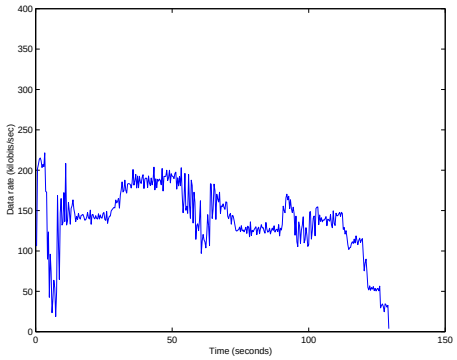

Figure A.20: Mean value graph for the 5 measurements for the setup with sleep value t=120 in the Java program, and the traffic shaper at speed 200 Kbps.

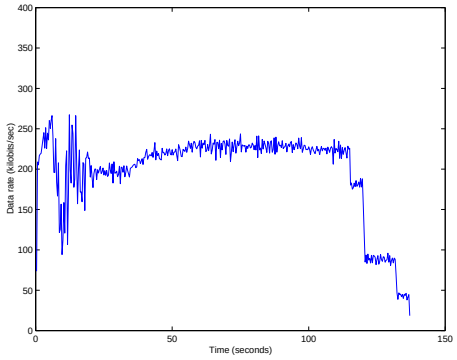

Figure A.21: Mean value graph for the 5 measurements for the setup with sleep value t=120 in the Java program, and the traffic shaper at speed 250 Kbps.

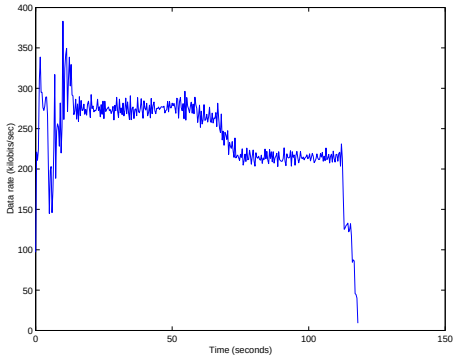

Figure A.22: Mean value graph for the 5 measurements for the setup with sleep value t=120 in the Java program, and the traffic shaper at speed 300 Kbps.

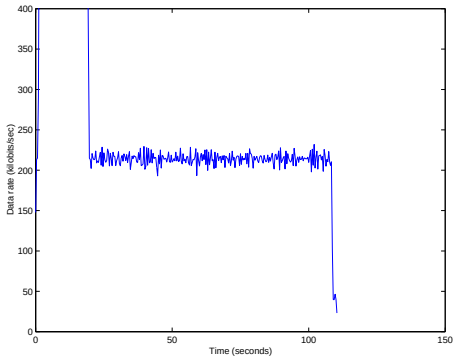

Figure A.23: Mean value graph for the 5 measurements for the setup with sleep value t=120 in the Java program, and the traffic shaper at speed 500 Kbps.

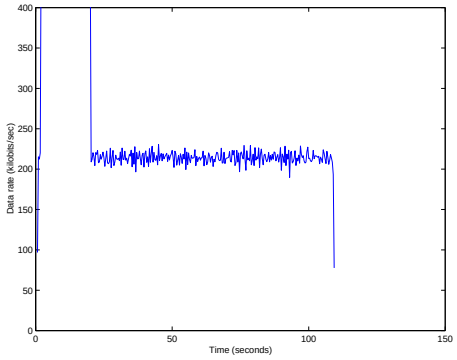

Figure A.24: Mean value graph for the 5 measurements for the setup with sleep value t=120 in the Java program, and the traffic shaper off.

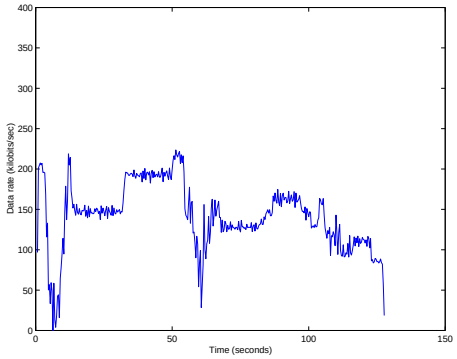

Figure A.25: Mean value graph for the 5 measurements for the setup with no Java program running, and the traffic shaper at speed 180 Kbps.

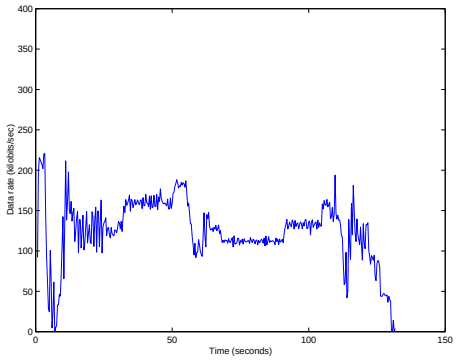

Figure A.26: Mean value graph for the 5 measurements for the setup with no Java program running, and the traffic shaper at speed 200 Kbps.

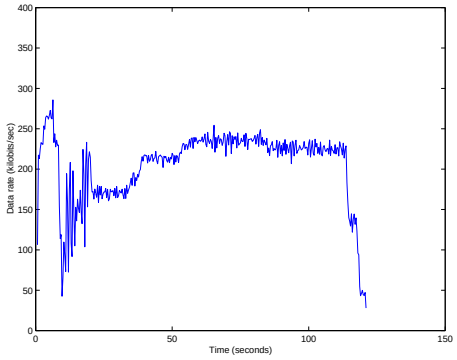

Figure A.27: Mean value graph for the 5 measurements for the setup with no Java program running, and the traffic shaper at speed 250 Kbps.

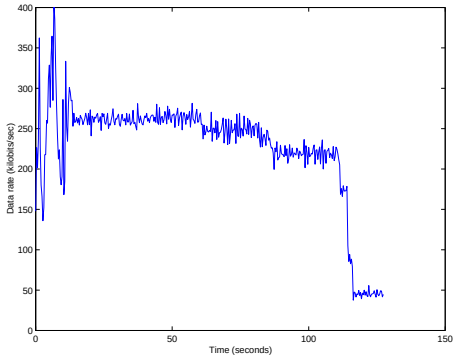

Figure A.28: Mean value graph for the 5 measurements for the setup with no Java program running, and the traffic shaper at speed 300 Kbps.

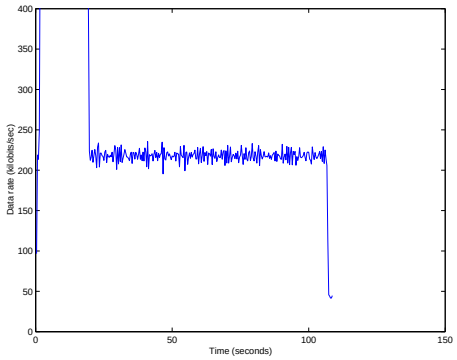

Figure A.29: Mean value graph for the 5 measurements for the setup with no Java program running, and the traffic shaper at speed 500 Kbps.

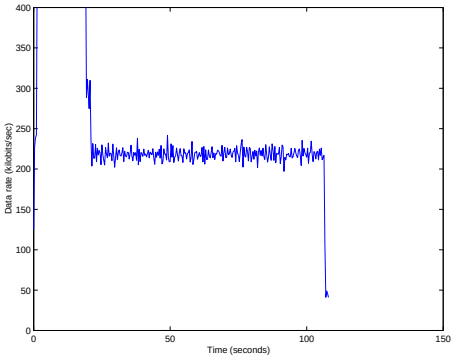

Figure A.30: Mean value graph for the 5 measurements for the setup with no Java program running, and the traffic shaper off.

Appendix B

Grades for Different Sets of Weights

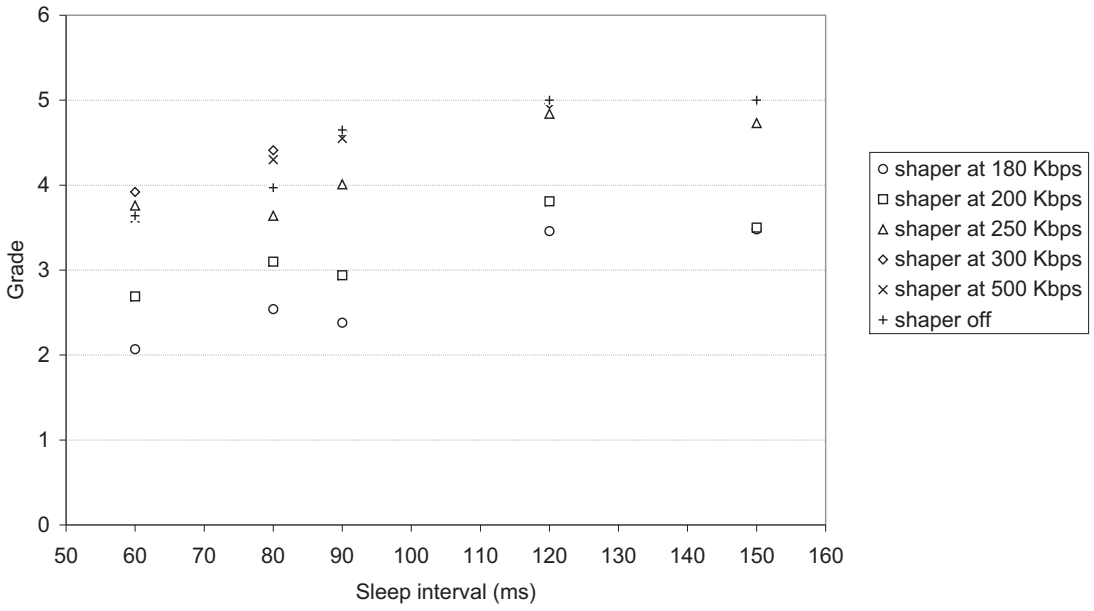

Figure B.1: Grades with the weight set 5, 4, 3, 2, 0. Note that the sleep value 150 stands for "no Java program".

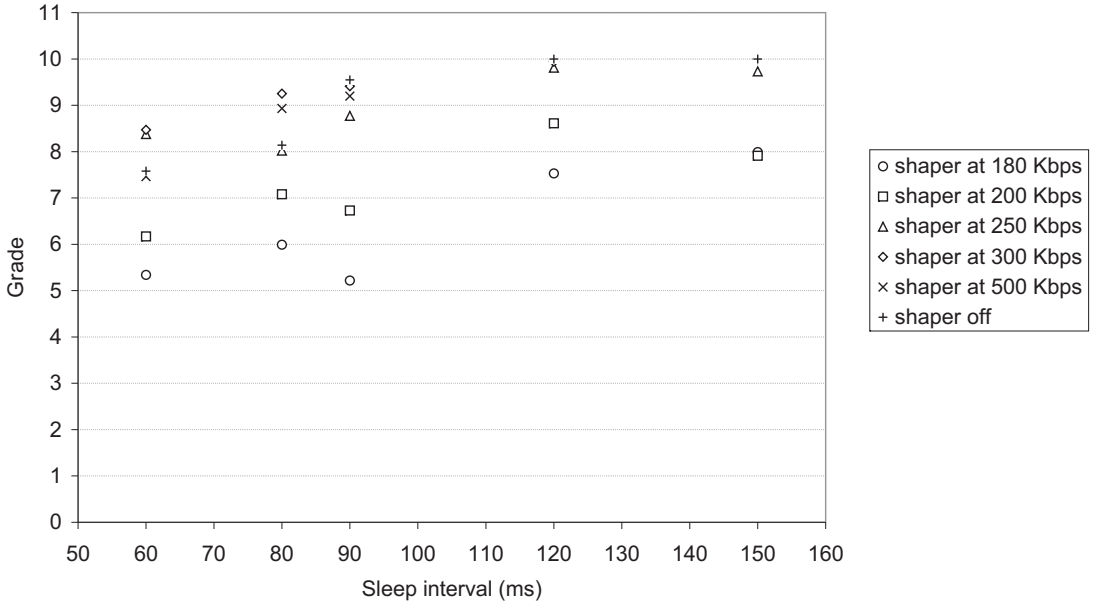

Figure B.2: Grades with the weight set 10, 9, 7, 4, 2. Note that the sleep value 150 stands for "no Java program".

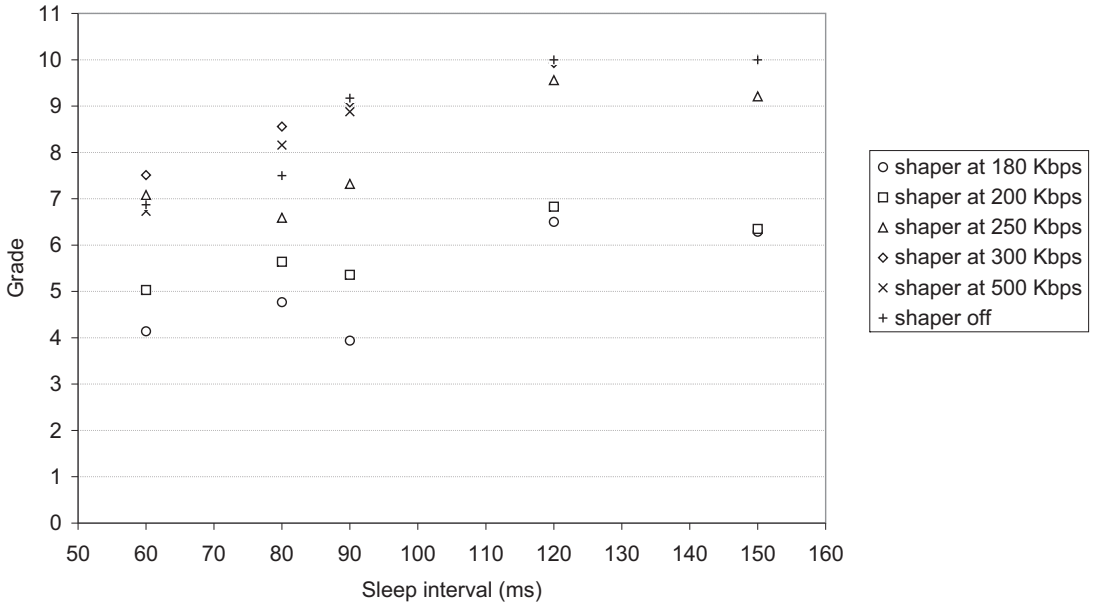

Figure B.3: Grades with the weight set 10, 7, 6, 3, 1. Note that the sleep value 150 stands for "no Java program".

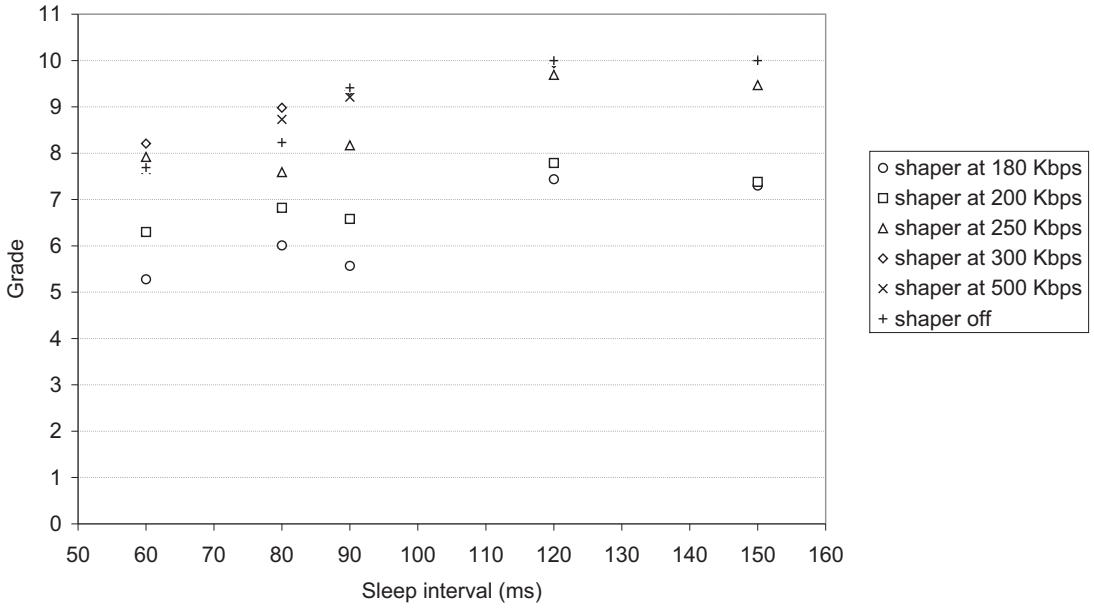

Figure B.4: Grades with the weight set 10, 8, 7, 5, 2. Note that the sleep value 150 stands for "no Java program".

Appendix C

Grades - Numerical Values

Below, the complete grading table for the different measurements and the corresponding setups are presented. The columns stand for:

- `sleep/shape` gives the sleep value and shaper speed for the respective measurement
- `E-220 time` to `E-90 time` give the time amount (in seconds) for which the clip was played at this encoding level
- `E-<90 time` gives the time amount for which the picture was bad
- `Grade` is the grade obtained for the measurement with the formula on page 54
- `Mean grade` is the mean value of the five obtained grades for the respective setup
- `Last-E grade` is the grade corresponding to the last encoding level at which the clip was played
- `Mean grade` is the mean value of `Grade` and `Last-E grade` for each measurement
- `Final grade` is the mean value of the five `mean grades` for the respective setup

The procedure of calculating the grades is graphically depictet on Figure 5.1 B. Recall that the weights assigned to each encoding level were 10, 8, 7, 5 for the 220, 150, 112, 90 Kbps encoding levels and 2 for playback with bad picture quality.

sleep/shape	E-220 time	E-150 time	E-112 time	E-90 time	E-<90 time	Grade	Mean grade	Last-E grade	Mean grade	Final grade
60/180	0	0	0	0	138	2.00		2	2.00	
60/180	0	0	91	47	0	6.32		5	5.66	
60/180	0	110	0	0	28	6.78		8	7.39	
60/180	0	0	30	48	60	4.13		5	4.57	
60/180	0	26	112	0	0	7.19	5.28	7	7.09	5.34
60/200	0	63	75	0	0	7.46		7	7.23	
60/200	0	0	138	0	0	7.00		7	7.00	
60/200	0	0	65	73	0	5.94		5	5.47	
60/200	0	37	83	0	18	6.62		7	6.81	
60/200	0	0	0	115	23	4.50	6.30	5	4.75	6.25
60/250	36	102	0	0	0	8.52		8	8.26	
60/250	48	26	64	0	0	8.23		7	7.62	

sleep/shape	E-220 time	E-150 time	E-112 time	E-90 time	E-<90 time	Grade	Mean grade	Last-E grade	Mean grade	Final grade
60/250	0	28	110	0	0	7.20		7	7.10	
60/250	10	128		0	0	8.14		8	8.07	
60/250	15	29	94	0	0	7.54	7.93	7	7.27	7.66
60/300	88	50	0	0	0	9.28		8	8.64	
60/300	59	55	24	0	0	8.68		8	8.34	
60/300	0	41	97	0	0	7.30		7	7.15	
60/300	80	26	0	27	5	8.36		5	6.68	
60/300	8	37	93	0	0	7.44	8.21	7	7.22	7.61
60/500	55	23	0	51	9	7.30		5	6.15	
60/500	57	15	7	50	9	7.30		5	6.15	
60/500	71	23	0	44	0	8.07		5	6.54	
60/500	61	29	0	48	0	7.84		5	6.42	
60/500	61	25	0	52	0	7.75	7.65	5	6.38	6.33
60/off	61	22	0	42	13	7.41		5	6.20	
60/off	65	25	0	48	0	7.90		5	6.45	
60/off	68	21	0	47	2	7.88		5	6.44	
60/off	63	27	0	48	0	7.87		5	6.43	
60/off	66	29	0	16	27	7.43	7.70	5	6.22	6.35
80/180	0	38	77	23	0	6.94		5	5.97	
80/180	0	0	0	100	38	4.17		5	4.59	
80/180	0	42	96	0	0	7.30		7	7.15	
80/180	0	0	96	0	42	5.48		7	6.24	
80/180	0	73	0	47	18	6.20	6.02	5	5.60	5.91
80/200	0	53	85	0	0	7.38		7	7.19	
80/200	0	138	0	0	0	8.00		8	8.00	
80/200	0	95	0	43	0	7.07		5	6.03	
80/200	0	15	39	50	34	5.15		5	5.08	
80/200	0	0	106	32	0	6.54	6.83	5	5.77	6.41
80/250	10	50	78	0	0	7.58		7	7.29	
80/250	2	79	0	57	0	6.79		5	5.89	
80/250	17	121	0	0	0	8.25		8	8.12	
80/250	10	68	0	44	16	6.49		5	5.75	
80/250	61	77	0	0	0	8.88	7.60	8	8.44	7.10
80/300	101	37	0	0	0	9.46		8	8.73	
80/300	56	40	42	0	0	8.51		7	7.75	
80/300	0	68	70	0	0	7.49		7	7.25	
80/300	102	36	0	0	0	9.48		8	8.74	
80/300	138	0	0	0	0	10.00	8.99	10	10.00	8.49
80/500	75	38	0	25	0	8.54		5	6.77	
80/500	75	35	0	28	0	8.48		5	6.74	
80/500	75	31	0	32	0	8.39		5	6.70	
80/500	79	59	0	0	0	9.14		7	8.07	
80/500	78	60	0	0	0	9.13	8.74	7	8.07	7.27
80/off	74	28	0	36	0	8.29		5	6.64	
80/off	73	31	0	34	0	8.32		5	6.66	
80/off	70	32	0	36	0	8.23		5	6.62	
80/off	77	23	0	38	0	8.29		5	6.64	
80/off	73	8	17	40	0	8.07	8.24	5	6.53	6.62

sleep/shape	E-220 time	E-150 time	E-112 time	E-90 time	E-<90 time	Grade	Mean grade	Last-E grade	Mean grade	Final grade
90/180	0	95	0	43	0	7.07		5	6.03	
90/180	0	0	0	118	20	4.57		5	4.78	
90/180	0	98	0	40	0	7.13		5	6.07	
90/180	0	0	0	118	20	4.57		5	4.78	
90/180	0	0	0	118	20	4.57	5.58	5	4.78	5.29
90/200	0	0	54	58	26	5.22		5	5.11	
90/200	0	13	55	49	21	5.62		5	5.31	
90/200	0	105	33	0	0	7.76		7	7.38	
90/200	0	40	98	0	0	7.29		7	7.14	
90/200	0	94	0	44	0	7.04	6.59	5	6.02	6.19
90/250	15	90	0	33	0	7.50		5	6.25	
90/250	10	117	11	0	0	8.07		8	8.03	
90/250	13	69	56	0	0	7.78		7	7.39	
90/250	10	128	0	0	0	8.14		8	8.07	
90/250	94	44	0	0	0	9.36	8.17	8	8.68	7.69
90/300	138	0	0	0	0	10.00		10	10.00	
90/300	82	31	25	0	0	9.01		7	8.00	
90/300	138	0	0	0	0	10.00		10	10.00	
90/300	100	38	0	0	0	9.45		8	8.72	
90/300	85	12	0	41	0	8.34	9.36	5	6.67	8.68
90/500	95	11	0	32	0	8.68		5	6.84	
90/500	138	0	0	0	0	10.00		10	10.00	
90/500	79	59	0	0	0	9.14		8	8.57	
90/500	90	0	0	48	0	8.26		5	6.63	
90/500	138	0	0	0	0	10.00	9.22	10	10.00	8.41
90/off	138	0	0	0	0	10.00		10	10.00	
90/off	138	0	0	0	0	10.00		10	10.00	
90/off	93	45	0	0	0	9.35		8	8.67	
90/off	66	4	68	0	0	8.46		7	7.73	
90/off	87	51	0	0	0	9.26	9.41	8	8.63	9.01
120/180	0	0	138	0	0	7.00		7	7.00	
120/180	85	43	0	0	10	8.80		10	9.40	
120/180	88	40	0	0	10	8.84		10	9.42	
120/180	0	0	0	128	10	4.78		5	4.89	
120/180	0	108	30	0	0	7.78	7.44	8	7.89	7.72
120/200	0	116	22	0	0	7.84		8	7.92	
120/200	6	127	0	0	5	7.87		8	7.93	
120/200	3	130	0	0	5	7.83		8	7.91	
120/200	7	103	23	0	5	7.72		8	7.86	
120/200	10	93	30	0	5	7.71	7.79	8	7.86	7.90
120/250	120	18	0	0	0	9.74		10	9.87	
120/250	95	43	0	0	0	9.38		10	9.69	
120/250	114	24	0	0	0	9.65		10	9.83	
120/250	133	0	0	0	5	9.71		10	9.86	
120/250	138	0	0	0	0	10.00	9.70	10	10.00	9.85
120/300	138	0	0	0	0	10.00		10	10.00	
120/300	138	0	0	0	0	10.00		10	10.00	
120/300	138	0	0	0	0	10.00		10	10.00	
120/300	133	0	0	0	5	9.71		10	9.86	

sleep/shape	E-220 time	E-150 time	E-112 time	E-90 time	E-<90 time	Grade	Mean grade	Last-E grade	Mean grade	Final grade
120/300	138	0	0	0	0	10.00	9.94	10	10.00	9.97
120/500	138	0	0	0	0	10.00		10	10.00	
120/500	138	0	0	0	0	10.00		10	10.00	
120/500	138	0	0	0	0	10.00		10	10.00	
120/500	138	0	0	0	0	10.00		10	10.00	
120/500	138	0	0	0	0	10.00	10.00	10	10.00	10.00
120/off	138	0	0	0	0	10.00		10	10.00	
120/off	138	0	0	0	0	10.00		10	10.00	
120/off	138	0	0	0	0	10.00		10	10.00	
120/off	138	0	0	0	0	10.00		10	10.00	
120/off	138	0	0	0	0	10.00	10.00	10	10.00	10.00
no/180	0	68	55	0	15	6.95		7	6.97	
no/180	6	117	0	0	15	7.43		8	7.72	
no/180	5	118	0	0	15	7.42		8	7.71	
no/180	7	116	0	0	15	7.45		8	7.72	
no/180	0	84	30	24	0	7.26	7.30	8	7.63	7.55
no/200	4	124	0	0	10	7.62		8	7.81	
no/200	0	77	61	0	0	7.56		7	7.28	
no/200	28	59	18	23	10	7.34		10	8.67	
no/200	3	80	45	0	10	7.28		7	7.14	
no/200	0	84	20	34	0	7.12	7.38	5	6.06	7.39
no/250	114	24	0	0	0	9.65		10	9.83	
no/250	97	41	0	0	0	9.41		10	9.70	
no/250	92	46	0	0	0	9.33		10	9.67	
no/250	114	24	0	0	0	9.65		10	9.83	
no/250	92	46	0	0	0	9.33	9.48	10	9.67	9.74
no/300	138	0	0	0	0	10.00		10	10.00	
no/300	138	0	0	0	0	10.00		10	10.00	
no/300	138	0	0	0	0	10.00		10	10.00	
no/300	138	0	0	0	0	10.00		10	10.00	
no/300	138	0	0	0	0	10.00	10.00	10	10.00	10.00
no/500	138	0	0	0	0	10.00		10	10.00	
no/500	138	0	0	0	0	10.00		10	10.00	
no/500	138	0	0	0	0	10.00		10	10.00	
no/500	138	0	0	0	0	10.00		10	10.00	
no/500	138	0	0	0	0	10.00	10.00	10	10.00	10.00
no/off	138	0	0	0	0	10.00		10	10.00	
no/off	138	0	0	0	0	10.00		10	10.00	
no/off	138	0	0	0	0	10.00		10	10.00	
no/off	138	0	0	0	0	10.00		10	10.00	
no/off	138	0	0	0	0	10.00	10.00	10	10.00	10.00