UNIVERSITETET I OSLO
Institutt for informatikk

Hierarchical Layered Multimedia Streaming using TFRC and TCP Separated by a Proxy

Masteroppgave

Steffen Fiksdal

19. oktober 2003

Contents

1 Introduction **9**
- 1.1 The big picture . . . 9
- 1.2 The problem . . . 11
- 1.3 Description of this thesis . . . 14
- 1.4 Is TCP friendliness the only solution? . . . 16
- 1.5 The vision . . . 18
- 1.6 The motivation . . . 18
- 1.7 Composition and methodology view . . . 19
- 1.8 Acknowledgements . . . 20

2 Protocol details - General overview **21**
- 2.1 Chapter overview . . . 21
- 2.2 Real Time Streaming Protocol (RTSP) . . . 22
- 2.3 Session Description Protocol (SDP) . . . 23
- 2.4 Real Time Protocol (RTP) . . . 26
- 2.5 TCP Friendly Rate Control (TFRC) . . . 27
- 2.6 Transport Control Protocol (TCP) . . . 30
  - 2.6.1 General problems with streaming multimedia data over TCP . . . 30
  - 2.6.2 Different dominant TCP implementations and their features 31
  - 2.6.3 Example of congestion control in a Tahoe TCP implementation . . . 32
- 2.7 Datagram Congestion Control Protocol (DCCP) . . . 33

3 Protocol details - Research specific overview **35**
- 3.1 Chapter overview . . . 35
- 3.2 Real Time Streaming Protocol (RTSP) . . . 35
- 3.3 Session Description Protocol (SDP) . . . 36
- 3.4 Real Time Protocol (RTP) . . . 36
  - 3.4.1 The RTP packet format and Komssys specific features . . 38
  - 3.4.2 The RTCP packet format and Komssys specific features . 40
- 3.5 Transport Friendly Rate Control (TFRC) . . . 42
- 3.6 Transport Control Protocol (TCP) . . . 44
  - 3.6.1 The stream oriented TCP versus message based transport protocols. . . . 46

4 Implementation changes in Komssys **48**
- 4.1 General description of Komssys . . . 48
- 4.2 Extended the TFRC implementation . . . 50
  - 4.2.1 TFRC congestion control support between proxy and origin server . . . 51
- 4.3 Stream RTP over TCP . . . 51
  - 4.3.1 TCP connection negotiation . . . 52
  - 4.3.2 Main implementation changes in the RTP sender application 53
  - 4.3.3 Main implementation changes in the RTP receiver application . . . 54
- 4.4 Create RTP/UDP to RTP/TCP filter in proxy . . . 56

5 Experiment preparations **60**
- 5.1 Emulation of TFRC backbone transfers 60
- 5.2 Emulation of TCP access network transfers 65
- 5.3 Discussion of a combined setup with a pure reflection proxy . . . 66

6 Main emulations **69**
- 6.1 Introduction 69
- 6.2 Emulation testbed 69
- 6.3 Emulation parameters 69
  - 6.3.1 Round trip time 70
  - 6.3.2 Bandwidth 70
  - 6.3.3 Router queue capacity 71
  - 6.3.4 TCP flavor 71
  - 6.3.5 Background and access network traffic load 72
- 6.4 Structure of the emulation presentations 74
- 6.5 No cross traffic in the backbone 74
  - 6.5.1 No cross traffic in backbone, 0% packet drop in access network 74
  - 6.5.2 No cross traffic in backbone, 2% packet drop in access network 78
  - 6.5.3 No cross traffic in backbone, 5% packet drop in access network 78
- 6.6 Medium cross traffic on the backbone link 82
  - 6.6.1 Medium cross traffic in backbone, 0% packet drop in access network 82
  - 6.6.2 Medium cross traffic in backbone, 2% packet drop in access network 83
  - 6.6.3 Medium cross traffic in backbone, 5% packet drop in access network 83
- 6.7 High cross traffic on the backbone link 83
  - 6.7.1 High cross traffic in backbone, 0% packet drop in access network 83
  - 6.7.2 High cross traffic in backbone, 2% packet drop in access network 84
  - 6.7.3 High cross traffic in backbone, 5% packet drop in access network 84
- 6.8 Blocking TCP 84
  - 6.8.1 No cross traffic in backbone, blocking TCP in access network 86
  - 6.8.2 Medium cross traffic in backbone, blocking TCP in access network 87
  - 6.8.3 High cross traffic in backbone, blocking TCP in access network 88
  - 6.8.4 Blocking TCP results - a deeper look 88
- 6.9 Result matrix - No blocking and blocking TCP tests 92

7 Tuning options **95**
- 7.1 Use of priority progress streaming in proxy 95
- 7.2 Minimize the TCP socket buffer in proxy for greater application level control 95

7.3 Introduce a jitter threshold in front of the application level TCP queue in proxy . . . 95
7.4 Degrade TFRC throughput during high TCP backpressure . . . 96
7.5 FACK modeled TFRC in backbone . . . 98
7.6 Drop TFRC badput in proxy . . . 98
7.7 Keep TCP session busy for improved throughput in access network 98

8 Conclusion **99**

A Reno, SACK and FACK TCP details **101**
A.1 Introduction . . . 101
A.2 Description of Reno, SACK and FACK TCP . . . 101
A.2.1 The earlier TCP implementations . . . 101
A.2.2 Reno TCP . . . 102
A.2.3 SACK TCP . . . 106
A.2.4 FACK TCP . . . 108
A.3 Simulation results using Reno, SACK and FACK . . . 109
A.3.1 Simulation setup . . . 109
A.3.2 Simulating one segment drop during slow start . . . 109
A.3.3 Simulating multiple segment drops during slow start . . . 111
A.3.4 Reno, SACK and FACK in a concurrent transfer with a traffic generator . . . 116
A.4 Conclusion . . . 123

List of Figures

1 Protocol placement compared to the TCP/IP model 21
2 RTP data interleaved with RTSP 24
3 Example of a SAP announcement packet with session description payload 25
4 TFRC throughput equation 29
5 TCP vs TFRC transmission rate 30
6 Tahoe TCP slow start congestion policy 32
7 RTSP session establishement 35
8 RTP and RTCP (TFRC reports) flow 37
9 RTP fixed header and extension header 39
10 RTP extension header containing TFRC and LC-RTP specific information 40
11 RTCP application defined packet in Komssys 41
12 Komssys overview 50
13 UDP and TCP application defined packets 53
14 Administration of not sent bytes after a `sendmsg()` call 55
15 Incoming RTP data packet 57
16 RTSP signaling with mixed transport protocol streaming through proxy 59
17 Throughput versus goodput using hierarchical layered streaming 61
18 TFRC emulation testbed 62
19 Hierarchical layered streaming using TFRC with DRD in router . 63
20 Hierarchical layered streaming using TFRC with tail dropping in router 63
21 TCP traffic throughput with TFRC and UDP streaming 65
22 TCP emulation testbed 66
23 Hierarchical layered streaming using TCP with DRD dropping in router 67
24 TFRC backbone and TCP access network throughput 68
25 Emulation testbed 69
26 Round trip times in the Internet backbone 70
27 Tg traffic generation script example 73
28 Emulation matrix - Non blocking TCP 75
29 Throughput: No traffic in backbone and no packet drop in access network 76
30 Jitter: No traffic in backbone and no packet drop in access network 77
31 Jitter sample: No traffic in backbone and no packet drop in access network 77
32 Statistics: No traffic in backbone and no packet drop in access network 78
33 Statistics: No traffic in backbone and 2% packet drop in access network 79
34 TCP application buffer size: No traffic in backbone and 2% packet drop in access network 79
35 Throughput: No traffic in backbone and 5% packet drop in access network 80
36 Jitter: No traffic in backbone and 5% packet drop in access network 80

37 Statistics: No traffic in backbone and 5% packet drop in access network . . . 81
38 TCP application buffer size: No traffic in backbone and 5% packet drop in access network . . . 82
39 Throughput: Medium traffic in backbone and no packet drop in access network . . . 82
40 Throughput: Medium traffic in backbone and 5% packet drop in access network . . . 83
41 Throughput: High traffic in backbone and 0% packet drop in access network . . . 84
42 Throughput: High traffic in backbone and 5% packet drop in access network . . . 85
43 TCP application buffer size: High traffic in backbone and 5% packet drop in access network . . . 85
44 Emulation matrix - Blocking TCP . . . 86
45 Throughput: No traffic in backbone and blocking TCP in access network . . . 87
46 Statistics: No traffic in backbone and blocking TCP in access network . . . 87
47 Blocking TCP - deeper look emulation matrix . . . 88
48 Throughput: 512Kbps bandwidth and 0% packet drop in access network with no traffic in backbone . . . 89
49 TCP blocking implementation . . . 90
50 Delay variations for extra blocking tests . . . 91
51 Result matrix - the bottom line . . . 93
52 Statistics summary for non-blocking TCP . . . 94
53 Statistics summary for blocking TCP . . . 94
54 Jitter threshold with and without priority progress streaming . . 97
55 End to end delay computation . . . 98
56 One segment drop simulation with Reno TCP implementation . . 112
57 One segment drop simulation with SACK TCP implementation . 113
58 One segment drop simulation with FACK TCP implementation . 114
59 Two segment drops simulation with Reno TCP implementation . 115
60 Two segment drops simulation with SACK TCP implementation 117
61 Two segment drops simulation with FACK TCP implementation 118
62 Reno, SACK and FACK simulation over the same link with a traffic generator . . . 119
63 Reno, SACK and FACK congestion window and sequence number development . . . 121
64 Reno, SACK and FACK congestion window and sequence number development . . . 122

Abstract

The Internet has in the last decade experienced an explosive expansion with regards to both size and traffic volumes. Private users connected to the Internet are continuously presented with innovative ways of consuming their available bandwidth. In general these new ways of using the Internet initiate needs for more and more available bandwidth. In the final part of the last decade applications for watching video, listening to sound and interacting both visually and audio visually have become very popular. Such applications can be highly bandwidth consuming. In addition to the demand for high throughput they are also usually transported by UDP at the transport layer. Today it is possible to stream video and audio without specialized multimedia applications. HTTP streaming is such a technique where the stream is transported by TCP on port 80. Many streaming applications tries to use UDP as the primary transport protocol. If that fails it turns over to TCP. The personality of UDP makes it very suitable for video and audio. For video and audio applications loss is usually less critical than delay and UDP have the same priorities. The problem with UDP is that it does not react to congestion in the network from the sender to the receiver. In other words UDP does not incorporate a congestion control mechanism and sends data at the application specified rate. As the use of video and audio applications in the Internet increases so does the amount of connections without any form for congestion control. Transport Control Protocol (TCP) which still today is the dominating transport protocol in the Internet does incorporate congestion control mechanisms. TCP detects lost packets and reduces its sending rate when an assumed congestion is discovered. In brief this means that increasing use of UDP opens the possibility for TCP connections being strangled is higher. This problem has been widely discussed and alternative transport protocols and application extensions to UDP have been suggested. Application extensions to UDP that can cooperate with TCP in the network use a TCP-friendly algorithm at the application layer to adapt the application specific data rate. The carelessness of UDP is handled at application layer to make the long term throughput be similar to a TCP connection working under the same conditions.

In addition to be unaware of congestion, UDP is in many cases restricted from access through firewalls. More and more end users access the Internet from beyond a firewall. Since TFRC is typically implemented on top of UDP, rate controlled UDP solutions suffer the same problem as regular UDP with regards to access through firewalls.

In this work we investigate a streaming system with the means of a proxy which is situated just outside the access network firewall. The backbone transfer is carried by UDP and TFRC to control the rate at application layer. The proxy translates packets from UDP to TCP and forwards them towards the client. By using this architecture the transfer is TCP-friendly in the backbone and the firewall will let data through to the client assuming it is streamed using a well known port number like 80. The investigation also includes hierarchical layered multimedia to scale the throughput according to the TFRC rate control.

During our work we showed that the combination of TFRC in the backbone and TCP in the access network can improve regular HTTP streaming from server to client. The throughput and delay variations introduced by

TCP during packet loss in the access network are very limited due to the environment of access networks. The experiments also shows how the use of blocking TCP in the access network can regulate the TFRC backbone rate during high backpressure for scaling between the two environments. We also presents tuning options for the combined transport environment with the goal of maximizing the overall perceived quality and minimizing the delay variations.

1 Introduction

1.1 The big picture

Internet is, as most of us know it, a tool for information retrieval and exchange. Information is exchanged through the Internet in different forms and variations. Today, electronic mail is widely used and it is an essential tool for communicating fast, cheap and over vast distances. Web servers stand around the world waiting for anyone to request information from them, any time of day from everywhere in the world. We have FTP servers [31], NNTP servers [14], bulletin board systems and other services that allow creative ways of exchanging information electronically. We can, if these services are used as intended, identify some common behavior. They all receive requests and reply thereby. If such requests are served within a reasonable amount of time, most people will be pleased with that. If you request a file from an FTP server then that file will normally be downloaded to your computer. The total download time depends on the bandwidth available between your computer and the FTP-server, the congestion situation in the backbone, the size of the file and other variables. In the common case, the requested file will sooner or later be transferred to the client computer. It is usually not essential for you to receive that file in a certain amount of time.

Generally, clients of these services do not demand that their request should be fulfilled in a certain amount of time. If the response takes more time than expected, the value of the service will still be intact. The corner stone of most common services available in todays Internet is the transport protocol Transmission Control Protocol (TCP) [30]. TCP makes sure that the receiver does receive exactly the same information that was originally sent from the source. It is important that most services in the Internet can rely on TCP. If TCP discovers that information was lost or corrupted during transfer, it will react accordingly. If information is corrupted upon arrival, TCP discovers the incident and requests the corrupted information again. The general TCP implementation will act the same way if it experiences a timeout due to a loss of a packet. Therefore a file downloaded from an FTP server will not be corrupted, e-mail is received in the same state that is was sent. Basically, we can disclose one more common behavior of these traditional services. They demand a reliable transfer of information. So traditional Internet services are usually not time critical, but they demand a reliable transfer of data. As long as these services exist side by side without interference from other protocols and services, everything is very neat.

Nowadays, the traditional services are not alone in the Internet. In the recent years, a new kind of services has evolved. These new services can offer multimedia presentations as video and sound without transferring the whole multimedia file before it is viewed at the receiver side. In this context, multimedia is defined as digitally conversion of voice and moving pictures. Transferring multimedia based on the architecture of the Internet is basically an easy task. Sounds and motion pictures can be transferred by help of web servers, mail servers and FTP servers. When multimedia information is downloaded or partially downloaded to a potential client, the sounds can be heard and the motion pictures can be

viewed. This kind of technology is called download or progressive download.

Progressive download is a special kind of download where the multimedia is presented to the client before the whole presentation is downloaded. The viewer can view the portion of the file that has downloaded but can not move ahead to parts that have not been transferred yet. Progressive download files do not adjust to match the bandwidth of the users connection during transmission. Progressive streaming is sometimes referred to as HTTP streaming. This is because standard HTTP servers can deliver files via this method without the need for special protocols. This technology is widely used today, and many web sites offer progressive download.

In other situations the multimedia data should be presented as if you actually are nearby the source of multimedia. When you have a conversation with a friend, you hear everything your friend says. You respond in a split second and you can see the person you are talking to.

We can therefore divide real-time streaming in two broad categories: Streaming of stored content (for example video on demand) and live streaming (for example person to person conferencing). For the first case progressive download can be used to ensure the quality of presented material. Using progressive download gives the developer a chance to use large buffers or caching to ensure smooth presentation at the client side. Large buffers can and usually will have a sideffect. A large buffer will increase the latency between what is sent from the server and what is actually viewed at the client the very same moment. It will also introduce latency when starting the presentation. The clients have to fill up a rather large buffer before the video or sound is played. Live streaming presents actually the opposite problem. As it is presented live the latency will have to be very limited. When implementing live streaming services, large client buffers are not appropriate. If the buffers are to large, the latency can increase in such a way that the presentation is not actually live anymore. When limited to small buffers and no caching, the quality of service (QoS) can become a problem. If congestion occurs on the streaming connection, the client does not have much buffer for the client viewer. This can be experienced as poor quality for the viewer. Streaming of stored content and live streaming can be supplied with more or less human interaction. If you as a client of a video on demand (VoD) session want to fast forward, you push the fast forward button in the video viewer. Such an event is called real-time streaming with human interaction. The streaming technology gets more complicated when extensive use of interaction is involved.

Earlier in this section we stated that the traditional Internet services are based on reliable delivery of information with few time-scale requirements. Real-time streaming needs exactly the opposite. The time used for transferring information is absolutely critical. If some information is lost or corrupted during transfer it might not have too much impact on the perceived quality if those segments are discarded. A retransmission of such segments can lead to worse perceived quality than would originally be achieved if the lost or corrupted segments had been ignored. Loss of segments may result in picture distortion or voice noise. The packet loss will appear as irregularity and will be gone in a split

second. As a result, there has been a major challenge deploying robust solutions to real time streaming in the Internet. This thesis wishes to explore one solution and identify problems regarding the creation of acceptable real time streaming services.

1.2 The problem

The traditional services like web, FTP, NNTP and mail services fit nicely to the properties of TCP as it offers reliable transfer of data. The other main property of TCP is that it controls the sending rate of the data. When a network is heavily loaded (much traffic), TCP will decrease its sending rate to reduce the heavy load on the network. In general, TCP will give the receiver a reliable transfer while the transfer rate will vary based on network properties. As mentioned earlier, multimedia data is better off being transferred with stable sending rate with unreliable transfer. These wanted properties do not fit in with the TCP transport protocol. To solve the problem with transferring multimedia over TCP, several other protocols have been developed. These protocols aim to give a smoother sending rate than TCP with a more or less unreliable transfer of data. With the term "more smooth sending rate" we do not imply totally stable sending rate. These protocols must be able to react somehow like TCP to solve heavy load on a network. The protocols we mention here are also referred to as "TCP friendly protocols". In general they are TCP friendly because their overall amount of transferred data (throughput) is similar to the throughput of TCP. By smoothing out the sending rate and offering unreliable transfer of data they are more appropriate for multimedia transfer. Below we will highlight in some detail how TCP works and how "TCP friendly" protocols are meant to solve TCP's problems with regard to transferring multimedia data.

The rest of the paper will use terms like "segments" and "packets". The term "segment" is used in conjunction to TCP, while the term "packet" is used in conjunction with data packets in general. The separation of these terms is based on the transport protocol layer's standardized name for data units called "segments" or protocol data units (PDU's). When we use the term "packets", we refer to units of data that are not used in a direct TCP context.

In terms of transferring segments from source to destination, TCP detects and reacts to heavy traffic between the source and destination. When TCP discovers heavy traffic, it will decrease the rate of data transmission. In that way, TCP connections sharing a common physical line will adapt so that every connection experiences a fair share of bandwidth. TCP can be exploited and misused so that its innocent behavior fades. TCP also acts in such a manner that the closer you are to the server the better bandwidth you will experience (This is because of small round trip times, which will be introduced later in the paper). TCP does not send segments from the source at a constant pace. As written, when TCP recognizes that a packet is lost or corrupted during transfer (Usually corrupted packets represents less than one percent of retransmissions), it automatically decreases the rate of transmission. A lost or corrupted packet does most usually occur because one or more routers and switches connecting the sender and receiver experience congestion. Thereby, TCP connections will act in order to resolve the congestion. It is claimed that this behavior is essential

for the communication in the Internet to be stable. As explained in the previous section, TCP is probably not the ultimate choice if you want to develop streaming-services in the Internet. Many developers have avoided TCP as a transport carrier for their real-time services information. Retransmissions of packets combined with a very drastic rate control will in many cases be a major drawback for streaming purposes. The phrase "drastic rate control" should be explained more clearly. We will explain it here at a very abstract level as this behavior is an essential part of the discussions in this thesis. When referring to "TCP", it is based on a standard TCP implementation with the original congestion avoidance behavior. That means fast retransmit, fast recovery, rate-halving and other algorithms are not included. All those algorithms are explained in appendix A. But the original TCP congestion avoidance entered slow-start all the times a packet appeared lost or corrupted.

When TCP starts to transmit segments, it enters what is referred to as the "slow-start" phase. Initially it sends one segment to the destination. The sender will then receive an acknowledgement segment from the client. That action, from the time when the sender pushes out a packet to the time when an acknowledgement for that packet is received and treated at the server is called the end-to-end round trip time (RTT). If that segment is acknowledged by the receiver, TCP tries to send two segments. If both segments are acknowledged, TCP tries to send four segments and so on. This kind of behavior leads to an exponential rate increase. This behavior continues until one of four events occur. If the number of segments sent exceeds the number of segments the receiver can handle, TCP will stop the acceleration of segments sent. The exponential behavior ceases and TCP continues to send the maximum number of segments the destination can accept. When the receiver sends back an acknowledgement segment, it contains information about what size is left in the receivers buffer. As long as packet loss or corruption does not occur, the maximum transfer volume will be controlled by that information. The second event occurs when TCP discovers that a segment sent is not acknowledged in a fixed amount of time (retransmission timeout occurs). When such an event happens, TCP re-enters the slow start phase. This behavior will lead to an unsmooth transmission rate. The third event occurs when the sender receives duplicate acknowledgements. Duplicate acknowledgements normally tells the sender that one or more segments in one shipment are lost or corrupted. The fourth event occurs when the byte count of segments sent in one shipment exceeds the threshold value. The threshold value tells TCP when to cease the exponential behavior and increase linearly. If neither congestion occurs or the receivers buffer is exceeded before the threshold value is reached, TCP will stop the exponential behavior. TCP will then add one segment byte count (different behaviors for different implementations) for each successful round trip time. This behavior can in the best case continue until the TCP sending buffer reaches 64 kilobytes. TCP will not send more segments than can be represented in a 64 kilobytes shipment. These values are based on a standard TCP implementation. There exist implementations with larger TCP windows using a special technique called windows scaling which is explained later in the paper. When these events are combined with retransmission of lost or corrupt packets, it can be very troublesome to deliver good quality of streaming services. Especially will it be difficult to deliver good quality real time services over a long distance network with low bandwidth

and/or a lossy network.

The obvious solution to this problem is to use a transport protocol that does not inherit the TCP properties. The best way to deal with the oscillations of transmission rate of TCP delivering multimedia traffic is to use a protocol that at all times send out data at the application specific rate whether or not congestion occurs, supplied with a no retransmission policy. It is a totally selfish behavior and will most probably make several real-time clients happy. One such protocol is the User Datagram Protocol UDP [29]. Some Internet researchers see its behavior as a threat to the stability of the Internet. As explained, original TCP will re-enter a slow-start phase once it discovers a lost packet. A lost packet is most probably caused by congestion in the network. TCP streams will "cooperate" to resolve this conflict by reducing the transmission speed. A UDP connection can not adapt its sending rate and will not decrease the rate such as TCP does. The sending rate can be controlled at higher layers such as the application level. In this context we refer to how the transport level is implemented. UDP will still push packets onto to network as fast as the application layer wants to deliver data to the transport layer. When TCP backs off, UDP will just experience less loss of segments.

If too many such protocols are used as transport carrier for application data, TCP services in the Internet can be strangled. They may experience an increase of lost segments, mainly due to buffer overflows in intermediate routers between the sending and the receiving part. By the policy of common TCP implementations it can be strangled and go at minimum speed most of the time. Many of the packets sent and delivered can be dominated by handling retransmissions. The main concern of the future is that to many services rely on UDP and other "TCP-unfriendly" protocols. "TCP friendliness" is today a term with a well-defined meaning, and TCP unfriendly protocols are generally speaking protocols that "steal" bandwidth from other concurrent TCP based network connections. This fear has led to the development of a few transport protocols that inherit some, but not all of TCP's properties. TCP-friendliness is also implemented as independent rate adaption solutions that can be applied to existing protocols, and that themselves are not transport protocols. Most of these protocols are independent rate control solutions that fall under the category "TCP friendly". TCP-friendliness is implemented from everything between "some TCP-friendly" and "very TCP-friendly". One behavior has to be fulfilled to gain the title "TCP friendly". The protocols must have some sort of congestion control algorithm. The congestion algorithm must be built so that the long-term throughput is not more than TCP's long term throughput. The important part is that TCP-friendliness will try to smooth out the drastic rate control that is applied by TCP under multimedia data transfer. Appropriate actions must be taken when congestion occurs. TCP friendliness has been widely discussed and researched in the Internet community.

Many people in the Internet community are of the opinion that streaming services will deliver better quality using a TCP-friendly policy during transfer. As stated, the number of TCP-friendly protocols are many and they implement different functionalities and algorithms for achieving TCP friendliness. Some of them are more TCP friendly than others, so the protocol to use for one specific

solution will most likely not be suitable for another solution.

A great number of access networks in the Internet use one or more firewalls to protect and control access to and from the internal network. Nowadays one combination of protocol and port is widely open for access to and from most firewalls, namely the famous TCP HTTP port 80. If that port is closed in the firewall, network access users will have problems using web site services. TCP friendly protocols are not TCP. Therefore many streaming services based on non-TCP transport protocols can be stopped by firewalls. There is need for a solution where we can use TCP-friendly protocols in the backbone and at the same time process segments so that they are accepted by firewalls.

The main concern of this project is to explore how we can implement effective streaming services based on TCP-friendliness. We will not explore TCP-friendliness in detail, but concentrate on and discuss what TCP-friendly protocol is best suited for the assignment. We want to do research on how to make an effective TCP-friendliness to TCP conversion between the access network and the backbone. In order to get around the firewall problem, we will do this conversion before the packet enters the firewall. We also want to use an intermediate machine as a tool for controlling the virtual connection between streaming server and client. The benefit we can extract from this is a solution that uses TCP friendliness and at the same time can be accessed through most firewalls. Another positive effect is that the TCP connection at the client side will be served in a way that congestion is more or less avoided, which again can lead to more smooth transmission.

1.3 Description of this thesis

Some commercially available servers stream video to the end-user by means of the infamous HTTP streaming. While this technique is counter-intuitive to streaming, it has two advantages. It allows the user to receive streams through firewalls and it is compliant with the so-called TCP friendly behavior that is requested of multimedia protocols. In this work, the current implementation of the KOM(S) streaming system (Komssys) should be extended to support Real Time Protocol (RTP) filters that translate from RTP/UDP to RTP/TCP in proxy servers, to allow a mixed infrastructure of streaming transfer in an Internet backbone and HTTP streaming at its edge [35]. It should then be investigated how the streaming transmission in the backbone can be implemented in a TCP friendly manner.

The idea is to find out how a TCP friendly mechanism in the long distance will work together with a TCP mechanisms in the short distance. The conversion between TCP friendly transfer and TCP transfer will be done in an intermediate machine (also called a proxy) that is physically close to the client.

We have developed a piece of software along with this thesis. The product will be used as a tool to retrieve quantitative information about streaming in an environment as described. The product will be manipulated and changed during the work. The very first implementation will be able to receive a RTP/UDP packet from one network interface card and send an RTP/TCP packet out on a

second network interface card to the destination. The real application information in the packet is in our situation streaming information. The software will in practice be implemented in a proxy outside the firewall of the access network (or integrated with the firewall machine) where the streaming client is. The implementation will be able to receive a packet from a streaming server into the proxy server, translate the packet to TCP, and send it out on the client side interface card. The TCP connection at the client side must be opened before the first streaming data is received at the proxy. The TCP connection will be established at the moment the client requests the streaming file. If a streaming data packet sent from the proxy onto the access network hits a firewall on the rest of its journey it will most likely be accepted as most firewalls accept TCP connections on port 80.

The first version will be a plain RTP/UDP to RTP/TCP converter for a one way streaming scenario. TCP friendliness will later be added on the RTP/UDP connection. As stated earlier, TCP-friendliness is not a transport protocol itself, but a transport policy that can be added to an existing transport protocol such as UDP. All packets from the access network to the streaming server will be passed on as they are, that means no conversion will take place. The only time a conversion takes place is when a TCP friendly packet is received from the streaming server. With this solution at hand, we can execute experiments and gather results for further analyzing. First of all, we would like to see how such a solution works. If we assume that the proxy has no knowledge of buffering, caching or other ways to store data during the life of the connection, it will just be a simple TCP friendly to TCP router. The result of this combination will be analyzed and we propose enhancements to the solution. The solution will be integrated into the existing framework of Komssys.

With this solution set up and integrated into the existing framework, the real experiment begins. We do have a strong impression that the solution will not work very well. We will most likely have to implement some sort of behavior in the proxy, such that the client experiences the stream continuously and smoothly. The main problem is that the backbone usually has totally different criteria for transfer than the other side of the proxy. We have to do research regarding how we can use and analyze the client TCP connection behavior, and if necessary use that behavior to control the streaming flow from the backbone. If for example the client side of the connection experiences congestion and many timeouts, the streaming rate will not be smooth. Based on such congestion we might have to control and manipulate the backbone stream in such a way that the client side does not experience problems. A large part of this investigation is focused on the logic in the proxy to deal with these two different environments. The backbone will be implemented with a TCP-friendly transport protocol. The proxy solution will be integrated as a part of Komssys. I would like to emphasize that the software developed is a research tool for retrieval of quantitive data that can be used as arguments to strengthen or weaken the hypotheses raised.

The TCP friendly protocol to be implemented in the backbone is UDP with TCP Friendly Rate Control (TFRC) [6,13]. This TCP friendly solution will be described later in the paper. TFRC is basically an independent solution with a special transport policy that can be glued onto for example UDP. TFRC has

been specialized with a transport protocol named TFRC Protocol (TFRCP), but in this paper we will use UDP as the underlying transport protocol. The client will regularly report transmission behavior back to the server-side of the connection. The server then has to react accordingly to these reports. An interesting part of this assignment is to find out how the TFRC streaming in the backbone will act with TCP in the access network and what possible tuning options are possible. We assume it is necessary to control the backbone connection to manage TCP on the client side to avoid congestion problems. The result is meant to give a smoother transmission. We will discuss and evaluate several flavors of TCP (See Appendix A) and early buffer-full detection [23]. We have a hypothesis that the right choices can solve streaming through firewalls, since the client side will use a well known TCP port. And by having control over the transmission through a proxy we might make the TCP streaming smoother. Last but not least, we use TCP friendliness in the long distance to achieve fair sharing of bandwidth and be able to control the transmission.

The research in this paper is limited to streaming of stored content on a typical streaming server. The content will be delivered to the client in a one way fashion. The client requests are using the Real Time Streaming Protocol (RSTP), and the stored multimedia file will be streamed as a whole. It will not be possible to use interactions, such as play, pause and stop. We do these limitations to stay focused on the real question: How can we create a proxy solution that will help us deliver smooth multimedia streaming with TCP in the short distance ?

Now we will enter a discussion whether TCP can be used as a good alternative for video streaming transportation. It should be emphasized that this discussion is based on streaming of stored content.

1.4 Is TCP friendliness the only solution?

A few voices in the Internet community raise the question: Do we really need TCP-friendliness? By raising this question they have doubts whether TCP is that obsolete when it comes to streaming and suspect the future bandwidth to be vast enough that congestion control will not be necessary. Is it actually not fit to be used for streaming purposes? The subsection below introduces some questions related to future work on streaming. The questions are not intended to be completely analyzed and discussed in the paper, but we want to say that there are factors that possibly could give a solution to streaming that makes TCP-friendly solutions obsolete. The questions are raised to focus on the thought that TCP can be used as an alternative streaming transportation.

Do scientists in general agree upon the hypothesis that we must develop and use TCP friendly protocols to succeed with implementing good streaming solutions deployed in the Internet? The backbone and access networks might in the future have vast bandwidths. Is it possible that future bandwidth makes TCP appropriate enough for multimedia streaming? What about introducing new service modes in TCP, so that itself includes a kind of streaming-mode where appropriate transport properties are added or withdrawn (TCP friendly mode in TCP)? Is it possible to use TCP combined with todays bandwidths to achieve acceptable real-time streaming services? Are there any new behaviors in TCP

in the TCP/IP v6 stack [4] ? We do not intend to resolve all these questions here, but we merely state that scientists in general see TCP as obsolete for streaming due to retransmissions and drastic rate decrease. It is also said that as bandwidth increases the need to fill that bandwidth increases. And the IPv6 standard does not incorporate any new behavior that could make TCP more suitable for streaming.

The next section introduces an article on the need and use of TCP and TCP friendliness. The author of the article does not agree upon the hypothesis that TCP is obsolete as transport carrier for multimedia. We believe it is important to keep all possible solutions open. During our literature-studies early on in this thesis we discovered one common opinion. Most of the articles stated that a TCP friendly protocol should be used with multimedia streaming. The introduction of these articles informed us about the general behavior of TCP combined with the statement that TCP is not appropriate as carrier of multimedia. The rest of these articles mostly discussed TCP replacements and TCP friendliness. I asked myself: "Is there anyone who tries to figure out how and if TCP actually can be used for streaming purposes ?" I found one very interesting article on the subject. In the next section this article is described and challenged by the idea that TCP is obsolete for streaming purposes. The main arguments for such a contention is based on TCP's congestion control, reliability by retransmissions and it's lack of multicast support. It is assumed that the streaming type is based on stored content and not the conference or web phone type of real-time streaming. Video On Demand is used as a general example in the discussion.

A fair amount of research has been focused around TCP-friendliness. Focus has been set to how we can develop the best possible TCP-friendly protocol for our demand. It has been a matter of course that we need TCP-friendliness to successfully transport streaming media through wide area distributed networks. TCP is regarded as obsolete mainly because of retransmission and congestion control policy. In [21] a different point of view has been evaluated. The authors of this paper expose the idea that TCP is not obsolete for streaming transmission. Through careful content preparation and new compression technologies of stored content it is possible to use TCP for multimedia streaming. To support their view they introduce the use of Quality of service (QOS) adaptive video systems. Such systems can adapt transmission volumes based on available bandwidth. When bandwidth is low, the quality of the video decreases. The paper emphasizes that video quality is adapted at the time of transmission. The stored material can be encoded at good quality. Adaptation occurs at the time of transfer and not based on dynamic encoding of stored content. The adaption will not occur at system level (kernel), but at application level. The paper challenges the dogma that TCP's use of packet retransmission introduces unacceptable end to end latency. One condition that has to be met to support their view is that the streaming involves limited use of interactive events. Another objection states that abrupt rate variations due to congestion avoidance impede effective streaming. The dogma that retransmission gives unacceptable end-to-end latency can be solved. With Video On Demand (VOD) the criteria that has to be met is for the client to view the video in real time and with no interruptions. It is not that important that the client watches video that was

sent from the VOD server 1, 2 or 5 seconds ago. The client can use a large buffer to be sure that data continuously can be delivered to the application level. When retransmissions occur, the client side buffer will have time to wait for the retransmitted segment. Therefore with a large buffer, retransmissions can complete video data seconds before the client actually watches that chunk of data. Client buffering is also proposed to deal with strong variations of transmission rates. The solution presented in [21] is client-driven, meaning that the client requests streaming data from the server.

1.5 The vision

A vision is, as I define it, something you strive for, but that you most likely will not achieve. In this context a vision is the ultimate solutions to all the questions raised and all the problems defined in a research question. Such a vision is for example a streaming solution service that will deliver high quality video to every potential consumer. The video will be delivered smoothly and if necessary in real-time, so that every possible consumer will be satisfied with the service. Full duplex streaming with every possibility regarding interaction will also be served. All this should be implemented without disturbing the existing Internet architecture and network solutions. My goal is to strive for such a vision. Through the time I work with this thesis, I will hopefully learn a great deal about the subject of streaming and come a bit closer to my vision.

1.6 The motivation

Since my studies started I have personally been interested in and curious about how the transport of information in the Internet actually takes place. With this kind of thesis I am almost obligated to earn a deeper knowledge of the most widely spread transport protocol in the Internet, namely TCP. Such an obligation excites me. Multimedia streaming over the Internet is a very great present interest. Several companies and web sites today offer multimedia streaming. I want to explore how these work and what can be done to create more effective streaming solutions. Last but not least it is interesting to explore the distribution of non TCP-friendly protocols and the potential threat these are to TCP based services. For me, it is also exciting and encouraging to discuss and do research on the future situation and what impact badly implemented streaming services can have. We recently read an article on this subject [37]. The article was of the opinion that something had to be done to cope with streaming services "stealing" bandwidth from TCP. Multimedia streaming includes IP-telephony, audio streaming and video streaming. The greatest challenge is to make good quality of service regarding real-time streaming with substantial user interaction and full duplex action involved. One example would be IP-telephony with a web camera. The challenges in these area are still many. At least, I have a strong understanding that it is so. Combine these challenges with the possibility to get a deep knowledge of Internet's information transport details and the motivation is absolutely there. During the work with this paper I have indeed learned a lot about these issues. It was interesting to explore and investigate possibilities and limitations. Before the real research started I studied papers, articles and products written and made. The information gathered guided me on the journey I have been on.

1.7 Composition and methodology view

We have now introduced the topic of this thesis and the research goals we are pursuing. After this introduction to the paper, the remaining chapters will be outlined as follows.

Chapter 2 will concentrate on details about different protocols involved during the execution of the thesis. The protocols described range from the application level to the transport level in the OSI model, and they will be presented in OSI layer order, starting at the application level. The weight of chapter 2 lies on TCP and TFRC, as they are a crucial part of and will be a key factor when it comes to explaining the results of our experiments.

Chapter 3 will concentrate on the different protocols involved during the execution of the thesis. The difference from chapter 2 is that this chapter describes the protocols in the context of our research.

Chapter 4 will introduce the multimedia streaming system that will be the tool for our experiments. When the streaming system is introduced in some detail, we will concentrate on explaining the implementation changes necessary to support the experiments described.

Chapter 5 focuses on the preparations for the experiments. In addition, the chapter separates the complexity of the finished research setup and emulates the separated parts. By doing this we achieve a firm understanding of the combined experiment setup and thereby are more prepared to make the right choices and understand the final results.

Chapter 6 will present the combined experiments where the TFRC backbone transfers are combined with the TCP access network transfer. The observed results will be presented and discussed.

Chapter 7 presents suggestions for maximizing the value of our experiments by proposing several tuning options. The suggestions can be an outline for future work on the subject.

Chapter 8 concludes this thesis.

When a solid understanding of all protocols and how they relate to each other is acquired, we have a strong basis for actually implementing a prototype for research. The next goal is then to implement a proxy solution with our target streaming environment. Our hypothesis is that a mixed infrastructure of TCP friendliness and TCP can help us create a more effective streaming environment with more control over the intermediate stream. By controlling the TCP friendly backbone rate and use buffers in the proxy we believe the TCP connection can stream more smoothly. When TCP experiences congestion, the proxy solution has to react to compensate further congestion occurrence. TCP congestion leads to drastic rate regulations. We want to avoid such large variations in the transport rate. The proxy can act correctly by using buffers, faking feedback reports, manipulating the data and controlling the TCP friendly side of the connection.

This solution is different from many other solutions which only use one TCP or UDP connection between the streaming server to the streaming client. Other solutions discuss TCP friendliness and implement a solution where the whole connection is based on one TCP friendly protocol. We want to combine both types of protocols over one streaming connection by the help of a proxy.

1.8 Acknowledgements

I would like to thank my wife and my daughter who most patiently have waited for me to finish my MSc degree.

I would like to thank Carsten Griwodz who functioned as my tutor during this work. He has given me inestimable support and help. He has contributed with his expertise and raised questions that were a basis for deeper discussions. Michael Zink has given me input with regards to the TFRC implementation and I would like to thank him for that. Ashvin Goel also deserves credit for helping me out with some detailed information about TCP. I also thank Thomas Kvalvåg who has contributed by forcing me to do some other things than only work and studies for the last couple of years. Last but not least I would like to thank the Department of informatics for supplying all the hardware I needed to finish this work.

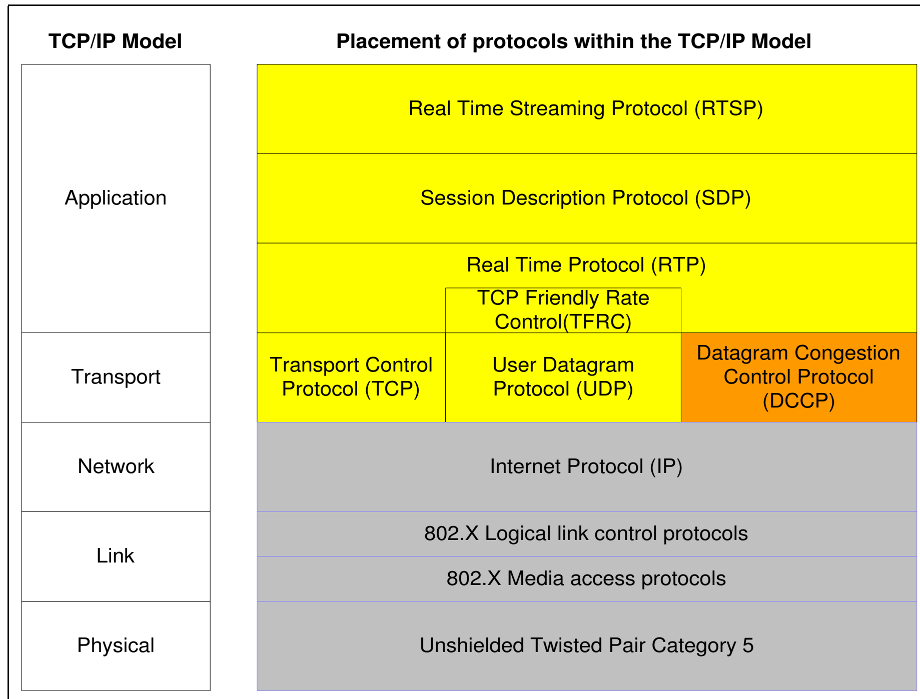

Figure 1: Protocol placement compared to the TCP/IP model

2 Protocol details - General overview

2.1 Chapter overview

This chapter is dedicated to a general insight into the protocols used and affected by our research. A thorough understanding of the different protocols used and how they relate to each other is necessary before entering the research setup and implementation detail chapters. In this chapter we give a short overview of the protocols succeeded by the next chapter where we highlight the protocols in the context of our research. As mentioned in the introduction, TFRC is not a protocol, but a rate control module for protocols in general. We will therefore describe it in this section. TCP and TFRC comprehension is important to be capable of manipulating and experimenting with the streaming implementation to achieve more effective streaming. The other protocols defined are necessary as part of the Framework for initiating, controlling and streaming multimedia sessions. For in depth understanding of the described protocols and TFRC we refer to the protocols' request for comments (RFC). Figure 1 gives the reader a visual overview of the affected protocols and their placement in the TCP/IP model. We emphasize that the figure presents one stack configuration. Other protocols exist at every one of the five layers.

This comparison of protocols in conjunction with the the TCP/IP model is not meant to be the one and only answer to where they belong. Different network functionality implementations and other papers might place them in

slightly different layers.

The grey squares in the figure represent the layers that will not be considered as interesting for our work. The network protocol IP is today the most widely deployed network protocol on earth. Its behavior is completely standardized and basing this thesis on manipulation of IP would give a highly theoretical piece of work not deployable in the Internet. Such a theoretical work could have been the focus of this paper, but instead we accept the IP behavior and move further up the protocol stack.

The yellow squares (including the red) represent protocols that directly or indirectly are interesting for our work. Why and how they fall under that category will be explained shortly as they are all described in detail. The red square represents a protocol that is not directly used in this paper, but it is presented because of its connection to TFRC as it incorporates the TFRC transmission policy. The protocol at the highest layer is introduced in the next section, and the rest of the protocols are presented in TCP/IP layer order from top to bottom. All sections first introduce the more general overview followed by chapter 3 which presents the protocols in the context of our research.

2.2 Real Time Streaming Protocol (RTSP)

Real Time Streaming Protocol (RTSP) is an application layer protocol [36]. The protocol is intended to control multiple data delivery sessions, provide a means for choosing delivery channels such as UDP, multicast UDP and TCP, and provide a means for choosing delivery mechanisms based upon RTP or other comparable protocols. It supports a signaling base for both streaming of stored content and live presentations. RTSP is a control protocol and not a data protocol. The most used control messages for establishing a streaming session are named DESCRIBE, SETUP and PLAY. RTSP control messages can be carried by unreliable transport protocols (for example UDP) or over reliable transport protocols (for example TCP). Usually it will be most convenient to use a reliable transport protocol, as the application itself does not require implementation logic for retransmission of potentially lost control messages. When a client wants to be added to or initiate a new multimedia streaming session its first communication with the server is via the transmission of a DESCRIBE request. The request consists of a URL describing the complete path to the known location of a presentation description. The server fetches the request, retrieves stored information on the requested presentation and builds up a DESCRIBE response. The response describes the presentation and makes the client capable of initializing itself and gather detailed information about the presentation. After processing this response the media initialization of the presentation is complete. The client will next build a SETUP request. The SETUP request describes a URI to the actually stored media file on the streaming server. In addition the request exposes what port numbers and transport protocols the client accepts for the actual data stream. The server receives the SETUP request, creates a session id for the streaming presentation, chooses transport protocol, server port(s), client port(s) (one or more of the announced ports) and returns a SETUP response. When the client receives the response, a streaming session with a unique session id is established. The server port(s), client port(s) and the selected transport

protocol are next used for transport of streaming data. Now the client can issue a PLAY request. The PLAY request contains the session id and the range of presentation time in seconds that is requested, expressed in normal play time (NPT). For each PLAY request the server streams data representing the NPT range. A PLAY request does not have to contain the NPT range. In that case the server streams the presentation until the end is reached or an interaction such as a PAUSE request is received. After the end of the presentation the client may send a TEARDOWN control message to the server signaling that it can release all resources held up by the session. If the client wants to view the movie again it has to issue a new SETUP request. The default server port for RTSP is TCP 554. The client port can be randomly chosen and should be any non-privileged TCP port.

One interesting issue about RTSP is its ability to embed or interleave binary data with the signaling messages. Such a solution opens the ability to actually send the stored content or live presentation using RTSP, preferably over TCP. In fact the RTSP request for comments emphasizes that interleaved binary data should only be carried over TCP. This architecture should be used as a last resort, but may solve communication problems between server and clients due to firewall solutions. A normal way of using interleaved binary data in RTSP messages is transporting RTP packets as the binary data (See section 2.4 for a description of RTP). The streaming server interleaves RTP packets whenever a client issues a SETUP request with the "interleaved" keyword. Embedding RTP data into RTSP complicates client and server operation and imposes additional overhead. Figure 2 shows and example of the embedded binary syntax in the RTSP standard. The figure deserves some explanation:

The client issues a SETUP request to the server specifying TCP as transport protocol and by including the "interleaved" keyword in the "Transport:" header. The server responds with OK, and only moments thereafter the client submits the PLAY message with an OK echoed from the server. After the server has replied, it will send ranges of binary data (actual content) by submitting special RTSP message with a preceding "$" mark. Following the "$" mark, a one byte value identifying the channel to use. After channel information a two byte value representing the length of the binary data. Interleaving RTP data requires every interleave message to contain one and only one RTP packet. After the length data follows the actual binary content. As seen in the figure RTCP messages are also interleaved.

2.3 Session Description Protocol (SDP)

SDP is intended for describing multimedia sessions for the purpose of session announcement, session invitation, and other forms of multimedia session initiation [7]. Newer SDP ideas talk about session negotiation as well. On the Internet multicast backbone (Mbone), a session directory tool is used to advertise multimedia conferences and communicate the conference addresses and conference tool-specific information necessary for participation. SDP is primarily intended for use in an internetwork although it is sufficiently general that it can describe conferences in other network environments. A multimedia session for these purposes is defined as a set of media streams that exist for a duration of time. The

```
C->S: SETUP rtsp://foo.com/bar.file RTSP/1.0
      CSeq: 2
      Transport: RTP/AVP/TCP; interleaved=0-1

S->C: RTSP/1.0 200 OK
      CSeq: 2
      Date: 05 Jun 1997 18:57:18 GMT
      Transport: RTP/AVP/TCP; interleaved=0-1
      Session: 12345678

C->S: PLAY rtsp://foo.com/bar.file RTSP/1.0
      CSeq: 3
      Session: 12345678

S->C: RTSP/1.0 200 OK
      CSeq: 3
      Session: 12345678
      Date: 05 Jun 1997 18:59:15 GMT
      RTP-Info: url=rtsp://foo.com/bar.file;seq=232433;rtptime=972948234

S->C: $\000{2 byte length}{"length" bytes data, w/RTP header}
S->C: $\000{2 byte length}{"length" bytes data, w/RTP header}
S->C: $\001{2 byte length}{"length" bytes  RTCP packet}
```

Figure 2: RTP data interleaved with RTSP

time during which the session is active need not be continuous. SDP basically serves two primary purposes - as a means to communicate the existence and timing of a session, and as a means to convey sufficient information to enable joining and participating in the session. In a unicast session the need for SDP is limited as it is primarily a tool for organizing multimedia sessions. A common mode of usage is for a client to announce a conference session by periodically multicasting an announcement packet to a well known multicast address and port using the Session Announcement Protocol (SAP) [9]. The announcement itself consists of a SAP header and the SDP session description as a chunk of text following the header. The SAP packet is carried by the use of UDP at the transport layer using port 9875.

SDP is also aften used in conjunction with Session Initiation Protocol (SIP) [34]. SIP is a control protocol that can establish, modify, and terminate multimedia sessions (conferences), for example Internet telephony calls. In such architectures, SDP helps describing the multimedia sessions while SIP controls the sessions.

Figure 3 presents a non-formal view of the content of a SAP announcement packet. The header consists of several bits which are further explained in RFC 2974 [9]. The address type bit has the value 0 if this is an announcement for IPv4 and has the value 1 for IPv6. Message type with a value of 0 reveals this as an announcement packet with SDP payload. If the bit is 1 this is a session

SAP Header	SDP (application/sdp) payload
Version=1 Address type=0 Reserved=0 Message type=0 Encryption =0 Compressed=0 Auth.length =0 Mess.Id.Hash = Unique Orig.Source = 23.23.23.1	v=0 o=mhandley 2890844526 2890842807 IN IP4 126.16.64.4 s=SDP Seminar i=A Seminar on the session description protocol u=http://www.cs.ucl.ac.uk/staff/M.Handley/sdp.03.ps e=mjh@isi.edu (Mark Handley) c=IN IP4 224.2.17.12/127 t=2873397496 2873404696 a=recvonly m=audio 49170 RTP/AVP 0 m=video 51372 RTP/AVP 31 m=application 32416 udp wb a=orient:portrait

Figure 3: Example of a SAP announcement packet with session description payload

deletion packet. The payload of the SAP packet is an ASCII string representing the SDP packet as described in RFC 2327 [7]. Details about all the parameters can be found there. The "v" parameter gives the SDP version which currently only can be 0. The "o" parameter gives the originator of the multicast session by the originators user name and IP address along with session id information. The "u" parameter is a URI containing more information about the session. Connection data is followed by the "c" parameter. In the example the first two letters "IN" is an abbrevationfor Internet. Then follows "IP4" meaning IPv4 which we also could have read from the SAP header. Next follows the multicast IP address for the session again followed by a slash and then the value 127. This is a time to live value which is needed in addition to the multicast address to fully define the scope of the session. For sessions based on hierarchical or layered video encodings more than one multicast address is often needed. In such situations the "c" parameter has support for defining several multicast addresses (for example one for each layer) with the required TTLs. The "m" parameters consists of one or several media descriptions. And every "m" parameter introduces an entirely new block which again can include parameters like "a", "c" and so on. The session announcement example consists of audio, video and some sort of application media. Such a media could for example be a white board media description. And the last "a" parameter states that the application media should be portrait oriented.

SDP is also frequently used in conjunction with RTSP to describe presentations. Working with RTSP, the SDP information usually comes in shape of a file stored at the original multimedia server or other dedicated servers. The SDP file which exists for every stored presentation or stream contains necessary information that a potential client must retrieve and process before requesting

the actual content. The file is a text file that is structured as compatible with the SDP description, for example it can contain the information presented as payload in figure 3. SDP files (or presentation description files) can be retrieved by the means of an ANNOUNCE request or a DESCRIBE request depended on if the presentation(s) are under aggregate or non-aggregate control. Non-aggregate control allows the client to retrieve the SDP file with a DESCRIBE request.

2.4 Real Time Protocol (RTP)

RTP provides end-to-end network transport functions suitable for applications transmitting real-time data, such as audio, video or simulation data, over multicast or unicast network services [35]. There are split opinions whether RTP is a transport protocol or not. In this thesis we will look at RTP as a protocol that supports transport of data with real time properties. The actual protocol for data transport can be for example TCP or UDP. The most common way of using RTP is with a connection less transport protocol beneath it, for example UDP. The main property of RTP is that it carries additional information with the segment that contains streaming-specific attributes. In addition, RTP is augmented by a special protocol named Real Time Control Protocol (RTCP). RTCP is used for transferring special packets between participants of a multicast session for support for data monitoring, identification and control functionality. RTCP packets are classified as either sender reports or receiver reports. The sender report is used by any participant in a session who acts as sender. Receiver reports are sent by participants that are not acting as senders. Receiver and sender reports are initiated at periodic intervals.

The RTP header consists of payload identification, a sequence number and a times tamp. The payload identification is a 7 bit long information field telling the application extracting the header what format the payload is in. The sequence number is incremented by one for each RTP packet sent from the source. As RTP usually is carried over UDP, which looses, delays and reorder packets, the sequence number can help the receiver reconstruct the order of the packets and also detect lost packets. The times tamp and the marker bit are used by the application to know when to present the data contained in the RTP packet. The streaming connection can as mentioned delay packets, and the sequence number and times tamp can help reconstruct the originally sent stream except for lost packets. The mixer mentioned right below must also use the times tamp information for re synchronization purposes. The rest of the information in the RTP packet is described in 3.4.1

The RTP standard also supports translators and mixers. In short a mixer is able to take two separate streams and mix them into one stream. For audio streams the mixer is also able to convert the audio encoding to lower bandwidth. The translator is capable of translating between protocols meaning that it can swap the transport mechanism below the RTP's layer in the OSI model (Figure 1). The translator can also translate between different encoding formats for reduction of bandwidth demands. The RTP packet format is discussed in section 3.4.1, while the RTCP format is described in section 3.4.2. This thesis will refer to RTP numerous places in the paper. By the term "RTP" we refer to RTP and

RTCP together. RTP and RTCP packets will usually require two ports. An RTP streaming session will always use an even numbered port (2n) for transferring the payload, while the feedback reports will use the next odd port (2n+1). It should be noted that the IETF consider changing the requirement for the RTCP port to not have to be the following odd port.

2.5 TCP Friendly Rate Control (TFRC)

TCP Friendly rate control (TFRC) is one of many TCP friendly transmission policies [10]. In [38], several proposed TCP Friendly protocols and policies are described. In our work TFRC is chosen because of its reported smooth transmission rate and good TCP friendliness [10]. In addition TFRC was implemented in Komssys by Michael Zink during the time of writing this thesis.

TFRC is equation based, and the equation periodically computes the new transmission rate based on parameters and constants (figure 4). The parameters for the equation are derived by making the sender and the receiver exchange information about the underlying connection that normally would not be reported. In section 3.4.1 and 3.4.2 we show how the Komssys implementation copes with this extended information exchange. The TFRC equation (figure 4) is developed by Jitendra D. Padhye in his Ph.D thesis [26]. It is a rate control solution meant for high multiplexing scenarios involving large numbers of TCP sessions. The equation is included in a draft by Mark Handley, Jitendra Pahdye, Sally Floyd and Jörg Widmer which is a standardization effort for TFRC [10]. The result of the draft was the standardization of the Datagram Congestion Control Protocol (DCCP) which will be described in section 2.7. The TFRC equation is supposed to resemble average throughput of the Reno TCP which uses fast retransmit and fast recovery [1]. At the time the draft [10] was written, no equation based algorithm had been developed that could model SACK or FACK TCP behavior. We refer to appendix A for the details about Reno TCP, SACK TCP and FACK TCP congestion behavior.

The key to equation-driven TCP friendliness is a TCP throughput equation. Such an equation has to give a transmission rate that on average resembles TCP throughput. It is especially important that the equation reflects TCP's retransmission behavior, as this dominates TCP throughput at higher loss rates. TFRC was originally developed as a substitution to TCP for streaming media and telephony applications. Split opinions exist whether this formula is good enough to resemble overall TCP throughput. Other TCP friendly protocols use different equations and some of them do not use equations. In [13] the Square-Increase/Multiplicative-Decrease (SIMD) is presented. The SIMD-algorithm is an approach to TCP friendliness that is window-based rather than equation-based. In window-based congestion control schemes, increase rules determine how to probe available bandwidth, whereas decrease rules determine how to back off when losses due to congestion are detected. The control rules are parameterized to ensure that the resulting protocol is TCP-friendly in terms of the relationship between throughput and loss.

Below we will highlight the actions taken for TFRC to achieve TCP friendliness and how the two participants in the communication path receive and

send information that can influence the transmission rate. TFRC is unicast, but multicast extensions are part of the DCCP standard (see section 2.7) and a multicast extension to TFRC is presented in [39]. Different implementations can diverge from the general approach presented below. The reader may notice that the Komssys implementation does not follow this algorithm exactly.

When two endpoints communicate using TFRC regulated transmission rate, the following actions have to be taken to compute the new transmission rate:

- The receiver of the TFRC based transmission measures loss event rate and feeds this information back to the sender as receiver reports. A loss event is composed of one or more lost packets during one RTT, and does usually not reflect one single missing packet. TCP models use the packet drop rate which reflects the probability for the packet to be dropped (lost packets over packets sent).
- The sender uses the information in the receiver report to compute among other variables the round trip time of the connection.
- The loss event rate information and the round trip time is then fed into the TCP throughput equation (figure 4), giving the newly computed acceptable sending rate.
- The sender adjusts the transmission rate to match the calculated rate.

The parameter "s" (packet size) is usually known to the application using TFRC and thereby does not have to be measured, making it a constant. There are circumstances where "s" varies depending on the data, but TFRC is not adaptive to that. For example will an audio-streaming application need to deliver packets where a packet represents an audio tune interval. This means that the application instead will have to vary the packet size when congestion occurs. TFRC-PacketSize (TFRC-PS) is a variant of TFRC for applications that have a fixed sending rate but instead vary their packet size in response to congestion [10]. Usually the application can compute an average packet size for "s". The parameter "R" (round trip time) is computed based on receiver report packets delivered to the sender. The "R" parameter is updated every time a receiver report packet is delivered. When "R" is computed, TFRC has to update the "t-RTO" parameter (TCP retransmission timeout value) based on the new "R" value. For simplicity the TFRC draft suggests setting the "t-RTO" value to 4*"R". This factor is based on experiments and past experience. The "p" parameter (loss event rate) is the most important parameter and has to be computed with great care in TFRC. The loss rate measurements are actually computed at the client with the result shipped back in a receiver report. The "b" parameter (number of packets acknowledged for each acknowledgement) is not computed, but should be implemented as a constant. The TFRC specification suggests setting this value to 1. Today many TCP implementations use delayed acknowledgement, meaning that the parameter "b" should equal 2.

The equation given in figure 4 takes both triple duplicate acknowledgements and retransmission timeouts into consideration when computing the transmission rate. The left side of the '+' sign in the equation reflects the triple duplicate

$$X = \frac{s}{R*\mathrm{sqrt}(2*b*p/3) + (t_RTO * (3*\mathrm{sqrt}(3*b*p/8) * p * (1+32*p^2)))}$$

Where:

X is the transmit rate in bytes/second.

s is the packet size in bytes.

R is the round trip time in seconds.

p is the loss event rate, between 0 and 1.0, of the number of loss events as a fraction of the number of packets transmitted.

t_RTO is the TCP retransmission timeout value in seconds.

b is the number of packets acknowledged by a single TCP acknowledgement.

Figure 4: TFRC throughput equation

acknowledgements reductions and the right side reflects the timeout reductions. The right side will dominate more when the packet loss rate increases. Earlier and not so complex models assumed that all window reductions were based on triple duplicate acknowledgements. This is a fair assumption for lower loss rates. When heavy traffic and great number of packet losses occur even the acknowledgements can fail reaching the sender before the retransmission timeout puts the congestion window to 1 and executes a slow start. The TFRC equation assumes that when one packet is lost in a window all the subsequent packets in the window are lost. This assumption fits well for drop tail routers, but will be more inaccurate for newer packet dropping schemes like Derivative Random Drop (DRD) and Random Early Detection (RED). As the packet losses occur more randomly it is not that likely that all the subsequent packets are lost.

The equation is modeled after Reno TCP, but the specification states that experiments and simulations point in the direction that a SACK modeled equation would be quite similar. Anyway the throughput for SACK modeled TFRC would be higher as SACK is more aggressive than Reno.

Between two feedback reports, the application sender will use a constant transmission rate. If no feedback reports are received within two round trip times, the sender cuts the transmission rate in half. This behavior is necessary for the sender to react to assumed heavy congestion in the network. So in certain situations the TFRC implementation can experience 50 percent decrease in

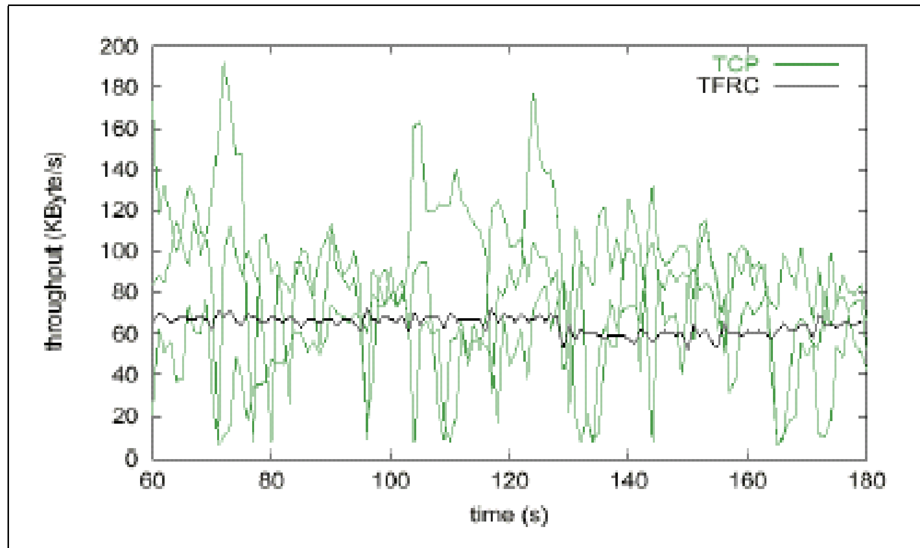

Figure 5: TCP vs TFRC transmission rate

transmission rate. This halving is not smooth and is executed like any ordinary TCP implementation reacts when congestion is discovered (if timeout does not occur) (see appendix A).

When all parameters are computed and updated, the equation will give the application the number of bytes pr. second that should be sent to the underlying transport layer. This equation will respond to congestion, but it will give a smoother change of transmission rate. In figure 5 we have tried to visualize the difference. The graph is retrieved from a presentation of equation based congestion control by Sally Floyd, Mark Handley, Jitendra Padhye and Jörg Widmer.

2.6 Transport Control Protocol (TCP)

2.6.1 General problems with streaming multimedia data over TCP

In the introduction we gave a general description of the Transmission Control Protocol (TCP) and highlighted the main drawbacks for using TCP as a carrier for real time streaming data. Most of the drawbacks using TCP are based on its behavior during congestion occurrence. In general, four main drawbacks can be detected:

- The sending rate is not stable due to congestion control algorithms in the sender.
- Lost or corrupted segments are retransmitted. This is not a drawback in itself, but it delays further streaming. For some streaming applications it is very likely that the retransmitted segments arrive to late to be of any value for the application layer.

- TCP does not support multicast.
- Concurrent UDP sessions can degrade TCP throughput.

The next section introduces the most frequently used TCP implementations that exist today. Their features mainly differ in how they react to congestion occurrence. Neither of them support multicast sessions and all of them serve reliable transfer.

2.6.2 Different dominant TCP implementations and their features

We used FACK TCP in the access network during the implementation, which uses fast retransmit and fast recovery and is further described in appendix A. Several different dominant TCP implementations are deployed in the Internet at present, each and one of them with their special behavior. FACK is the default TCP implementation in the operating system used during the research (Linux Mandrake 9.0), and the choice of using it therefore came natural. Below we list some of the current dominant implementations.

- Tahoe TCP
- Reno TCP
- New Reno TCP
- Vegas TCP
- Reno TCP with selective acknowledgments (SACK)
- SACK implementations with forward acknowledgments (FACK)

Appendix A contains the results of a comparison of Reno, Reno with SACK and FACK implementations. The appendix was created in parallel with the work of this paper. The first part of the appendix discusses detailed differences between the three variations. The second section includes simulation results with the help of the network simulator (NS2) [24]. We encourage the reader to study the appendix for detailed information on these simulations.

The results of the simulations showed us that Reno, SACK and FACK basically give similar throughput when 0 or 1 packets are lost from one window of data. When multiple packets from a window are lost, Reno tends to lag behind SACK and FACK. The problem with multiple packet drops in one window using Reno is a known issue and is described in detail in the appendix. The distinction between SACK and FACK is getting clearer as the link delay increases and the packet loss rate is high. FACK uses an algorithm that at all times maximizes the amount of packets on the line, while the algorithm in SACK does not maximize the use of the ether.

The next section introduces an example of how the Tahoe TCP implementation reacts to network congestion.

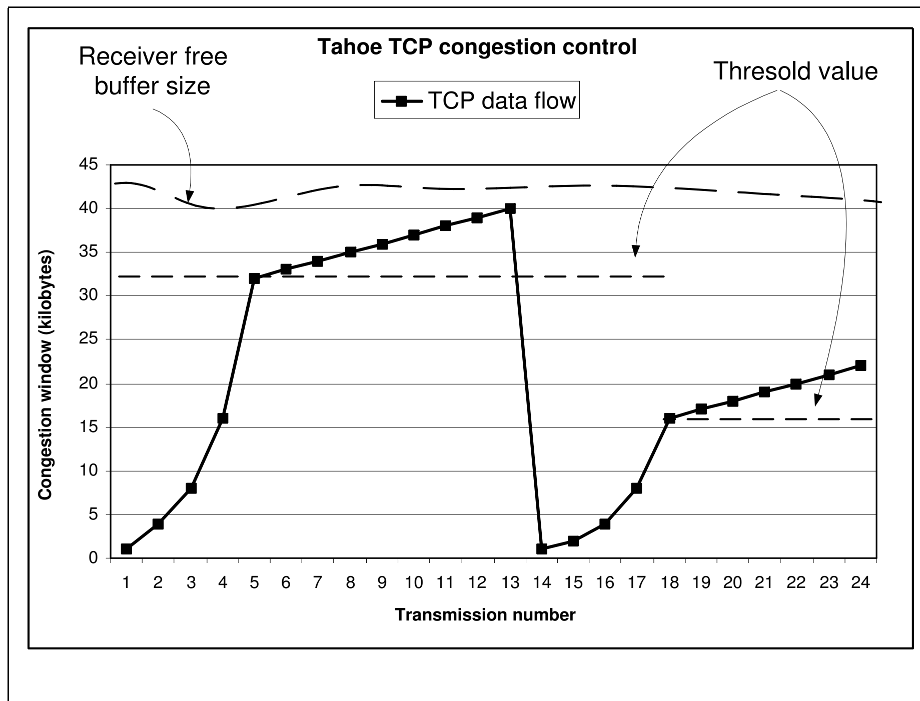

Figure 6: Tahoe TCP slow start congestion policy

2.6.3 Example of congestion control in a Tahoe TCP implementation

In figure 6 we have visualized how a standard Tahoe TCP implementation responds to congestion. The y-axis represents the size of the congestion window in kilobytes. TCP is a sliding window protocol, and the y-axis represents the size of this window. The congestion window will at all times represent how many bytes worth of segments can be sent out on the cables during each burst. The x-axis represents the transmission number, which in this case is not equal to the sequence number. A transmission number n represents the n-th time TCP has pushed all segments in the window. This means that the transmission number equals the number of round trip times since start. TCP will as shown increase the congestion window by one segment worth of bytes when it enters congestion avoidance mode when the congestion window equals 32 kilobytes.

From transmission number 1 we can see TCP's slow start exponential behavior. TCP (if a send timeout does not occur) will first send 1 segment in a burst, then 2,4,8 and so on as long as no congestion symptoms are identified. The congestion window will increase exponentially until it meets the threshold borderline. When the threshold borderline is met (the number of bytes in the window equals the threshold value) TCP ceases its exponential behavior and starts increasing the congestion window linearly. Now the congestion window will increase by 1 segment worth of bytes for each successful burst (one round trip time). In figure 6 the threshold value is met for the first time when the congestion window is 32 kilobytes. TCP will increase the congestion window

linearly until a timeout occurs. For each segment TCP ships into the congestion window it starts a timer. If TCP does not receive an acknowledgement for the byte range of this segment until the timeout clock hits zero, it assumes the segment as lost due to buffer overflows in routers, corrupted segment e.t.c. The important part is that TCP will react to the timeout. As we can see a timeout occurs at transmission number 13. This means that one or more segments from the burst in the transmission number are not acknowledged before a timeout for that segment occurs. TCP now re-enters the slow start phase and initializes the threshold value to half of the congestion window at the time of congestion assumption. As we can see from the figure the threshold value is now met at 16 kilobytes. In this scenario we assume that the receiver always reports a free buffer size greater than 40 kilobytes. The ideal situation would be that slow start never occurred after the initial one. In such a scenario the only action that can reduce the congestion window is the flow control feedback from the receiver.

A number of TCP standards and experimental implementations of TCP with diverging solutions to congestion control exist in todays Internet. The slow-start behavior described above is more or less the same among all TCP implementations. The different solutions tend to focus on how to avoid slow start or how to avoid waiting for a retransmission timer to expire, thereby keeping higher throughput. The FACK TCP implementation in our research environment uses fast retransmit, fast recovery and other solutions to avoid slow start. Fast retransmit alone does not limit slow start behavior, but makes TCP not have to wait for a timer to expire. Fast recovery is used in conjunction with fast retransmit with the main goal of quickly retransmitting apparently lost packets combined with an effective way to recover from the occurrence of the lost packets.

As mentioned, several different experimental and published TCP implementations exist in todays Internet. When we describe the research specific issues in the next chapter we focus on how TCP over RTP was implemented in Komssys. Our intention is not to explain the different TCP implementations, but to focus on the solution used in this paper. We do understand that we most probably will achieve higher throughput with FACK TCP than using less aggressive versions as SACK and Reno. Testing our scenario with different TCP implementations is unfortunately outside the scope of this paper (see appendix A).

2.7 Datagram Congestion Control Protocol (DCCP)

The DCCP protocol is not included in our research model. We wanted to add a brief description of this protocol because it (like TFRC) is a TCP friendly protocol well suited for media streaming transmission [10]. DCCP is derived from TFRC and it is currently being standardized.

DCCP is a TCP friendly transmission protocol incorporating a transmission policy and the underlying transport protocol [18]. DCCP can be used to achieve TCP similar flow based mechanisms without in-order delivery and reliability. In addition, the congestion control behaves different than TCP's congestion control. DCCP implements a congestion-controlled, unreliable flow of datagrams suitable for use by applications such as streaming media.

The kind of applications which could make use of DCCP are those which have timing constraints on the delivery of data, such as reliable in-order delivery. When combined with congestion control it is likely that some information will arrive at the receiver after it is no longer of use. Such applications might include streaming media and Internet telephony. To date most such applications have used either TCP, with the problems described above, or used UDP and implemented their own congestion control mechanisms (or no congestion control at all). The purpose of DCCP is to provide a standard way to implement congestion control and congestion control negotiation for such applications. One of the motivations for DCCP is to enable the use of Early Congestion Notification (ECN) [32], along with conformant end-to-end congestion control, for applications that otherwise would be using UDP. In addition, DCCP implements reliable connection setup, teardown, and feature negotiation. Even though it is unreliable, the sender receives ACK's telling if the packets arrived and if they were ECN marked. The main goals of the DCCP initiative are:

- To be an alternative protocol for applications currently using UDP without end-to-end congestion control. The goal is that when finally implemented DCCP can replace UDP with little reason not to replace it.
- To provide an alternative to current TCP-style implementations halving the congestion window in response to a congestion notification. The motivation is for the application to choose among different TCP friendly policies, for example TFRC.

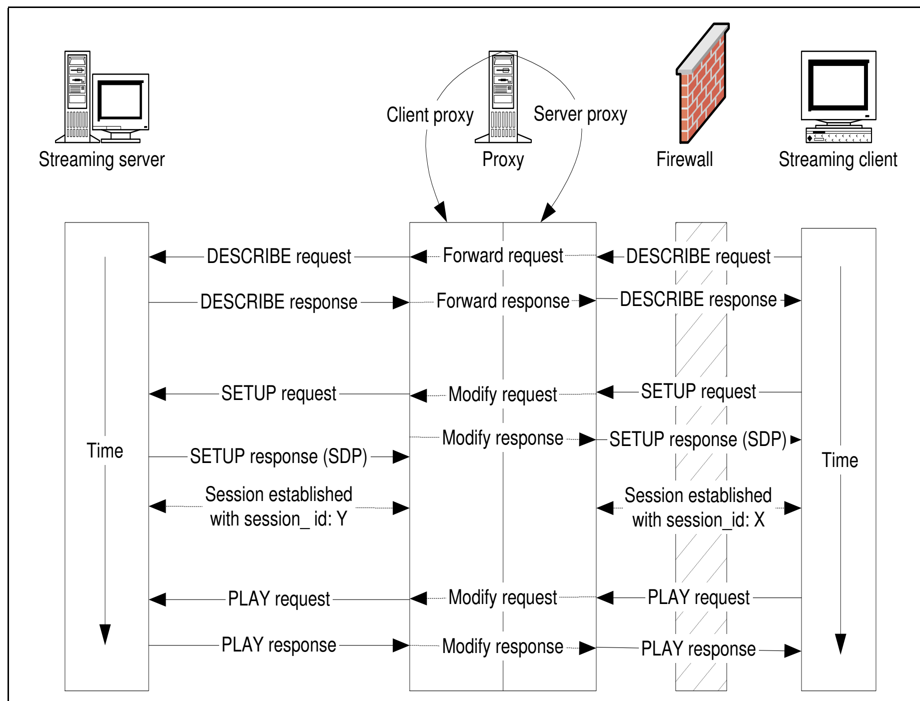

Figure 7: RTSP session establishement

3 Protocol details - Research specific overview

3.1 Chapter overview

As mentioned early in the previous chapter, this chapter explains the protocols in the context of our research. The protocols appear in exactly the same sequence as in chapter 2.

3.2 Real Time Streaming Protocol (RTSP)

In our work, the proxy had to act as a streaming server to the client and as a client to the streaming server. We also had to be certain that the transport method used between the proxy and the client was based on TCP. The transport method between the proxy and the server had to be based on UDP. To arrange such a setup, some considerations had to be taken for the client to establish a virtual streaming session with the streaming server. We will now describe and highlight actions performed by the proxy during session and presentation establishment between client and server in the context of RTSP. Figure 7 visualizes the necessary logic that the proxy had to implement.

When the client issued a SETUP request, the proxy had to send a SETUP response to the client that chose RTP/TCP as transport protocol and the most suited ports announced by the client. When the response was received by the client, one or more TCP connections between the proxy and the client would

exist (one for each stream). The proxy now had to be sure that the forwarded SETUP request to the streaming server specified RTP/UDP as the only transport choice. The port(s) announced did not have to be changed even though we may be using the same set of port numbers. One set is TCP and one set is UDP. The operating system can distinguish between a connection on port 80 with UDP and a connection on port 80 with TCP. The streaming server sent back a SETUP response to the proxy. After the session setup was finished, the client had a streaming session with the proxy based on RTP/TCP, and the proxy had a streaming session with the actual streaming server based on RTP/UDP. We had now created a session in the backbone that was ready for streaming using the TFRC congestion policy over UDP. On the client side, an established TCP session was created where data would be pushed out at a controlled rate from the server side of the proxy. Chapter 4 discusses the implementation details necessary to accomplish this arrangement, and the reader should navigate there for more details on the RTSP logic of the proxy.

3.3 Session Description Protocol (SDP)

The research model described in this work does not incorporate a multicast architecture. As mentioned in the last chapter where SDP was described, the need for SDP in unicast architectures are limited. The properties of TFRC (described later in this chapter) and TCP force us to implement our model in a unicast environment. SDP is used to arrange conferences/multicast presentations where equal users join and leave conference sessions. In our case SDP is used as a tool for arranging the virtual session between the streaming server and the client in the initial RTSP setup negotiation. It existed two unique RTSP sessions in the research environment. One session with a session id between the streaming server and the proxy client and one session with another session id between the proxy server and the client (Figure 7). SDP was used by the means of SDP files (presentation description files) which are initially retrieved when the client issued a DESCRIBE request. In the research environment all streaming is therefore under aggregate control.

3.4 Real Time Protocol (RTP)

When an RTSP session was established between two endpoints and the PLAY request was processed, the streaming server started to send data representing the requested presentation. The content was split up in small chunks of data and wrapped in RTP packets and shipped down to a predetermined socket on the server. Beneath the socket interface the RTP packet was encapsulated in a transport protocol packet and later sent out on the network to a predetermined end point (computer and port). This Transport Service Access Point (TSAP) could be a specific address at the destination computer (unicast) or a multicast address (several potential computers). As described in section 2.4 the ports used for RTP is predetermined during the RTSP SETUP process. At regular intervals the server released a new chunk of multimedia data and sent it out on the network. As the session propagated, the destination computer did with periodic intervals return RTCP receiver reports giving connection, identification and status information. The server would likewise send out RTCP sender reports. For multicast sessions all RTCP packets are sent to the multicast address

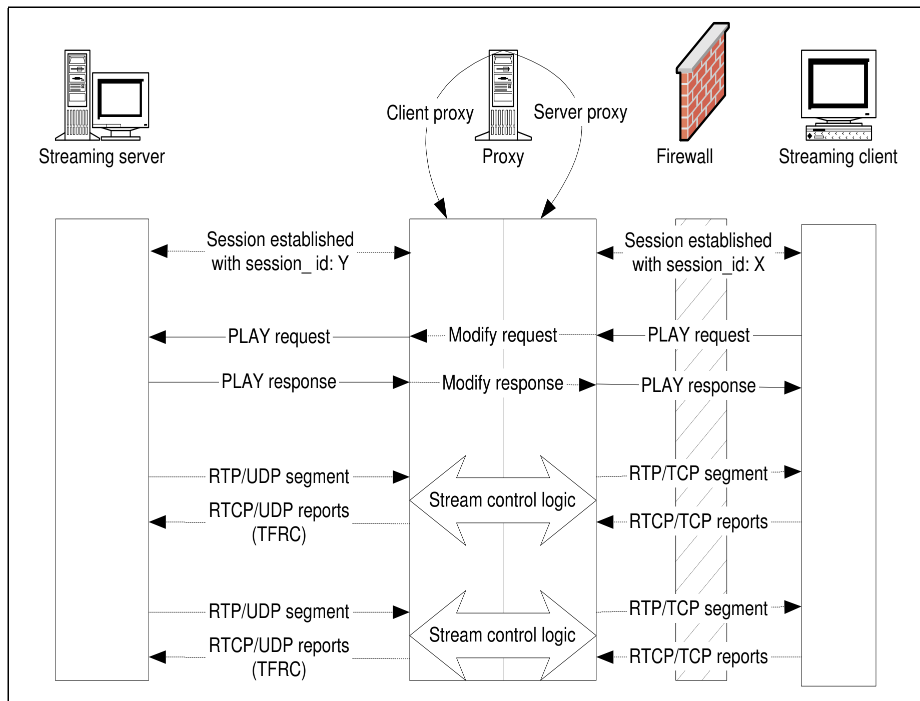

Figure 8: RTP and RTCP (TFRC reports) flow

and every participant in the session can read all RTCP packets (both sender and receiver reports).

Figure 8 gives an overview of how RTP and RTCP was used in this work. It should be noted that our environment is strictly unicast and all issues regarding multicast architectures are left out. The server streamed data to the proxy computer and the proxy computer streamed data to the client. The figure shows that the RTP/UDP packets were translated to outgoing RTP/TCP packets. On the client side we used the TCP connection for streaming, and TCP does not loose packets neither reorders them. The use for RTCP receiver reports are rather limited as we used a reliable transport protocol. The RTCP receiver reports sent from the proxy to the origin server did in our scenario not only function as RTCP receiver reports are intended to be used. Komssys uses RTCP functionality to transport information that is important for the TCP friendly mechanism, TFRC, to adjust to the right transmission speed. These reports were sent in addition to standard RTCP reports. The information units were periodically transmitted to the server. The RTP standard allows for totally application specific RTCP packets by the means of the APP packet (See section 3.4.2). In the next section we also reveal that the RTP sender had to transmit information to the proxy via the RTP extension header. This functionality is part of creating the basis for calculating the correct transmission speed. The extension header functionality is part of the RTP standard and allows the sender to transmit additional application specific information with the

RTP packet. Applications that do not understand potential extension headers should be implemented in such a manner that the extensions will be ignored. The two next sections introduce the RTP and the RTCP packets in the context of the Komssys streaming system. The RTP extension header information and the RTCP APP packet information are explained in detail.

3.4.1 The RTP packet format and Komssys specific features

Figure 9 presents the RTP packet format. Below we describe the different attributes of the packet.

The **"V"** parameter describes the RTP version which currently should be set to 2. The **"P"** bit is used to inform if the packet is padded with octets at the end of the payload that are not part of the actual payload. Padding is necessary mainly for encryption algorithms with fixed block sizes or to carry more than one RTP packet in one transport level protocol. The **"X"** bit equals 1 if the extension header is applied. **"CC"** is an abbreviation for Contributing Source Count. The number in "CC" gives the number of contributing sources identified in the CRSC list. This parameter can be used by mixers to mix two streams onto one stream. An RTP mixer is an intermediate system that receives and combines packets of one or more RTP sessions into a new packet. The two separate streams have two separate SSRC (Synchronization Source) identifiers and the value 0 in the CC parameter. After the mixer has done its work the outgoing RTP packet has the mixer as SSRC and the value 2 in the CC parameter. The previous SSRC's identifiers are put into the CSRC parameter. The **"M"** and the **"Payload type"** parameters are used for describing profile specific information. Whether the "M" bit is set the packet depends on payload profile described in the "Payload type" field. It is for example possible to send Vorbis encoded audio packets with the RTP protocol. If one Vorbis packet can not fit in one RTP packet, the Vorbis packet must be fragmented. To preserve payload specific frame boundaries the "M" bit is set to the value of 1 every time the existing packet completes one whole Vorbis fragment. For an RTP packet containing a fragment of a Vorbis packet, the payload specific header (8 first bits of payload) contains information that this payload contains 0 Vorbis packets by the 5 last bits of the 8 bit profile specific header. The **"sequence number"** parameter is chosen randomly and is increased by 1 for every RTP packet sent. Following the sequence number, a **"timestamp"** parameter giving the exact time of the sampling of the first octet in the RTP payload. **"SSRC"** is as described earlier the synchronization source for this packet, which usually is a streaming server or a mixer. **"CSRC"** contains the identifications to any sources contributing to this packet. The original RTP header is now complete. If the "X" bit equals 1, an extension header follows with a profile depended field, a length field containing the size of the extension payload and then at last the extension payload itself.

The Komssys streaming system takes advantage of the RTP extension header in two scenarios:

- **TFRC:** The origin sender of the stream must send additional information to the client when streaming by the means of the TCP friendly mecha-

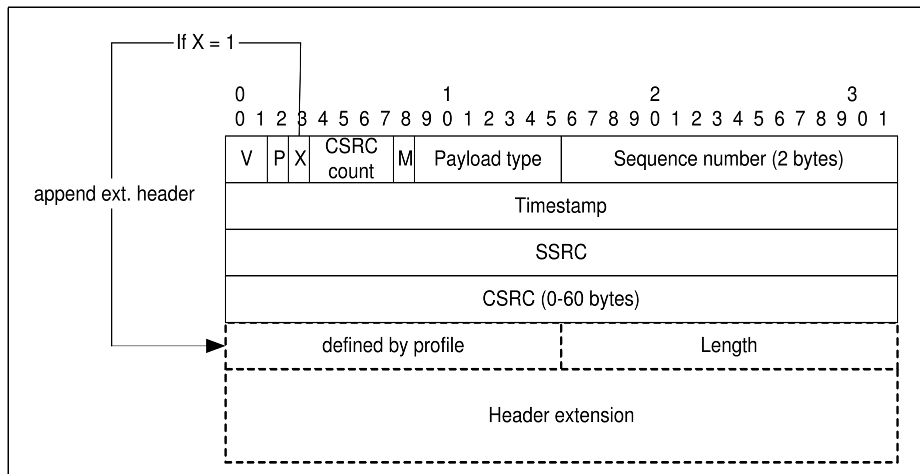

Figure 9: RTP fixed header and extension header

nism TFRC. The receiver uses this information to calculate and build the RTCP application specific packets (RTCP APP packets) which later will be returned to the server. The RTP extension header is part of the framework for regulating the transmission speed to be TCP friendly.

- **LC-RTP:** LC-RTP is defined here [40]. It is an abbreviation for Loss Collection RTP. Since RTP is usually transmitted over a non-reliable transport protocol packets my be lost on their way to the destination. For regular streaming this is not a problem because packets are expected to be lost. In other situations clients initiate streaming sessions from its nearest streaming proxy server. A streaming proxy server holds all or some content of the requested media. If the cache does not have the requested content it will usually retrieve the content from the origin server. During this process it is desirable that the cache contains the whole content with the same quality as the origin server after retrieving the content. LC-RTP can be used for this process as all lost packets are recorded by the cache and retransmissions of these packets occur at the end of the delivery session. LC-RTP offers reliable transport at the application level.

In figure 10 the Komssys specific extension header is presented. The extension header is equal for both LC-RTP and TFRC streaming, but different units in the extension header are extracted in the different cases. The fields in the extension header are:

- **Defined by profile:** The field is a word describing the extension profile. Different values for TFRC and LC-RTP are used. LC-RTP does itself use two different values in this field for identifying if it is currently in normal streaming or in the retransmission phase.
- **Length:** Length of the extension payload in bytes.
- **Round trip time:** This parameter reflects the senders current estimation of the round trip time.

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
```

Defined by profile	Length
Round trip time (TFRC)	
Bitrate (TFRC)	
Loss measurement round (TFRC)	
Mode (TFRC)	
Byte count (LC-RTP)	
Byte count (LC-RTP)	

Figure 10: RTP extension header containing TFRC and LC-RTP specific information

- **Bitrate:** This parameter reflects the current transmission speed.
- **Loss measurement round:** Loss measurement round determines the interval that is regarded for loss measurement at the receiver. It is initialized to zero and increased whenever a new loss measurement round is initiated.
- **Mode:** This parameter is used to inform the receiver about the current transmission mode of the underlying transport protocol. As TFRC has to be TCP friendly and therefore also act according to TCP during the slow start and congestion avoidance modes the part of the framework for TFRC in Komssys must include this information.
- **Byte count (64 bits):** This is a 64 bit long number compromising the byte count of the current packet. The LC-RTP implementation uses this field to detect loss, reserve a gap on storage medium so that lost packets can be retransmitted at the end of the session and be stored at the reserved gaps on the storage medium.

The first four parameters are TFRC specific while the two last parameters are used for LC-RTP. More details on LC-RTP can be found in [40]. The use of TFRC parameters are more clear in section 3.5 which describes TFRC and the equation based framework for computing TCP friendly transmission speed. In addition to these 4 parameters the TFRC framework also takes advantage of the sequence number and timestamp in the standard RTP header.

3.4.2 The RTCP packet format and Komssys specific features

The specification defines several RTCP packet types to carry a variety of control information. These are:

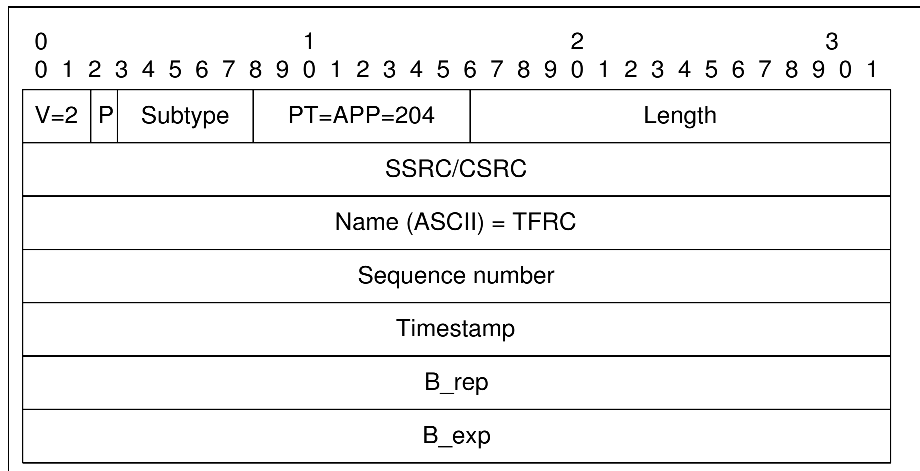

Figure 11: RTCP application defined packet in Komssys

- **SR:** Sender report, for transmission and reception statistics from participants that are active senders. This packet is equal to receiver reports with the exception of a sender specific header. SR is transmitted at regular intervals and should be sent as often as bandwidth constraints allow.
- **RR:** Receiver report, for reception statistics from participants that are not active senders. RR is transmitted at regular intervals and should be sent as often as bandwidth constraints allow.
- **SDES:** Source description items, including CNAME. The source of a packet is identified through the SSRC field, but in many cases this field can happen to be no persistent. The SSRC field might change because of an SSRC conflict or a program is restarted. The CNAME is also used for other purposes. To obtain a persistent transport-level identifier (CNAME) for the source a SDES packet is interpreted by the receiver.
- **BYE:** Indicates end of participation if used with multicast and several participants. In a unicast environment, sending this packet tears down the whole session.
- **APP:** Application specific functions.

In normal multicast several of the defined packets are sent in one compound packet and sent at transport level as one protocol data unit (PDU). For example, the sender will build a stack of RTCP reports before sending. The stack will start with an SR packet followed by one or more RR packets depending of how many sources the sender can hear. At the end of the stack an SDES packet is applied. These RTCP packets are sent as one PDU. If RTCP packets are compounded the RFC states that it must comprehend an SDES packet for new receivers to learn the CNAME of the source as quickly as possible.

For deeper details about any of these packet formats we refer to [35]. The focus of this section is to describe how Komssys uses the APP packet to support

TFRC and LC-RTP. The APP packet defined in the Komssys implementation is shown in figure 11. The Komssys specific APP packet is built like this:

- The **"V"** parameter gives the current version of RTP which should be set to 2.
- The **"P"** field is used for identifying padding information. If the value is 1 this RTCP packet contains some additional padding octets at the end which are not part of the control information.
- The **"Subtype"** parameter can be used to make several APP packets to be defined under one unique name.
- The **"PT"** field is set to 204 for APP packets. So the standard says.
- The **"Length"** field gives the length of this RTCP packet in 32-bit words minus one, including the header and any padding.
- The **"SSRC/CSRC"** field contains the synchronization source and contributing synchronization source identifiers.
- The **"Name"** field is set to "TFRC" and is completely defined by the developer of the APP packet.

The standard APP header is now given. Any information in the packet beyond this point is totally application specific. Like the reader can see out of figure 11 the rest of the Komssys defined APP packet is constructed as followed:

- The **"sequence number"** reflects the sequence number of the last RTP packet received.
- The **"timestamp"** reflects the timestamp of the last RTP packet received.
- The **"B_rep"** parameter is the actual measured reception rate at the receiver. It is an abbreviation for "Bandwidth reported".
- The **"B_exp"** parameter is the TFRC calculated transmission rate. It is an abbreviation for "bandwidth expected". As this parameter implies, the actual computation of the new expected transmission rate of the stream is executed at client site.

Now the reader should have good knowledge of how the RTP framework is used in Komssys and how support for information exchange needed for TFRC and LC-RTP is implemented. The next section gives a general discussion of TFRC with some level of details.

3.5 Transport Friendly Rate Control (TFRC)

The main concern in the communication between the origin server and the proxy client was that we achieved good TCP friendliness and at the same time experienced smooth transmission rates. Our choice of TFRC was based on earlier documented experiences like [38] and the fact that TFRC was just recently implemented in Komssys during our work by Michael Zink. During this thesis

the origin server and the proxy client were extended to make the RTP transmission with TFRC function the way we needed it. The TFRC implementation resides at application level and was used to control the transmission rate over the UDP datagram service between the origin server and the proxy client. The TFRC implementation at the origin server used RTCP receiver reports sent from the proxy client to extract enough information to be able to adapt to the new transmission rate (Figure 4). The proxy implementation therefore used RTCP packets, not only as regular RTCP packets, but as building TFRC receiver reports [42]. The TFRC receiver reports contained information based on the properties of the UDP connection between the proxy client and the streaming server. It is now explained how the different parameters in the TFRC equation is calculated in Komssys.

In all the emulations the "s" parameter was set to 1460 bytes. 1460 resembles Maximum Segment Size (MSS) on an ethernet with a Maximum Transfer Unit (MTU) of 1500 bytes. The MTU is the largest packet the MAC layer can have as payload. MSS is the maximum payload for layer 4. With a TCP header of 20 bytes and an IP header of 20 bytes this adds up to 1500 bytes for layer 2. To be very specific since Komssys' TFRC rate control works over RTP/UDP, the MSS should be set to 1472 in the normal case assuming that the RTP headers are part of the transport protocols payload. 1472 + UDP header of 8 bytes and IP header of 20 bytes adds up to 1500 bytes. Note that our TFRC emulations might have given slightly lower throughput since the UDP payload was set statically to 1024 bytes + RTP header. The ideal setting would have been to set the "s" parameter to 1024 + RTP header, but the code was hard coded with 1460 and this situation was discovered after all emulations. The impact of this situation is that the TFRC transfers in the emulation part are somehow less aggressive than using the correct packet size of 1024 bytes + RTP header.

The "R" parameter is calculated as follows: When the receiver receives an RTP packet, the timestamp and the sequence number are extracted from the RTP header. The receiver returns this information in the TFRC feedback packet. When the sender receives a feedback, it can calculate the end-to-end round trip time of the packet by subtracting the current time with the timestamp from the feedback report. The RTT is then fed back to the receiver again in the RTP extension header of the RTP packet. The receiver now has the calculated "R" parameter for the equation.

The "p" parameter is calculated at the receiver site based on loss events. Loss events are calculated in loss event rounds. The loss event round starts at 0 and increases for every loss measurement round. The current loss measurement round is sent to the receiver by the sender in the RTP extension header.

The "t_RTO" parameter should resemble the TCP retransmission timeout value. This value is calculated at the receiver based on the reported RTT from the sender. The t_RTO parameter is calculated as: maximum of (4*RTT, 500ms). The standard implies that the smallest t_RTO value should be set to 1000 milliseconds to confirm with RFC2988. The RTT does not equal the "R" parameter. To adapt smoothly to variances in round trip times a decay parameter of 0.01 is used. The standard suggests setting the decay parameter to 0.1.

The RTT is therefore calculated as the maximum of (_rtt_ema, _rtt_cur) where _rtt_cur equals the "R" parameter and the _rtt_ema resembles the smoothed RTT. The _rtt_ema is calculated as follows: ((1 - RTT_DECAY) * _rtt_ema + RTT_DECAY * rtt_cur).

The "b" parameter was set statically to the value 2, which means delayed acknowledgements from concurrent TCP sessions are assumed.

As seen when we presented the RTP extension header and the feedback packets in the RTP section of this chapter the sender and the receiver exchanges more information. The receiver side calculated expected transmission rate is sent back to the receiver. In addition the current transmission rate at the receiver is reported back to the sender. The sender uses this transmission rate during slow start mode. During slow start mode the new transmission rates should calculate to the minimum of (2*rate at receiver, 2*rate at sender). The sender also reports the actual bitrate to the receiver in the RTP extension header. The receiver uses this bitrate to calculate right transmission rate during initial loss.

3.6 Transport Control Protocol (TCP)

Application level code in the proxy supervised the TCP connection between the proxy server and the client and detected heavy client site transmission congestion. When congestion occurred during transmission, the TCP connection between the proxy server and the client was probably heavily loaded due to concurrent traffic. As mentioned earlier, congestion will lead to a "drop to one" transmission rate using the Tahoe implementation (Figure 6). A heavily loaded TCP connection can deliver very low throughput. The transmission rate can frequently drop to one or be halved, and the overall throughput follows the transmission rate. In chapter 7 we present tuning solutions with the goal of dealing with heavy congestion. We suggest different solutions to adapt our stream throughput to the transmission rate. Based on the documented emulation results this paper can hopefully give ideas on how to achieve good QoS with our research setup. Basically we wanted to achieve a smooth and stable transmission rate on the server side. On the client site the transmission rate did vary drastically (TCP), but because of properties in the access network the general transmission rate was higher than the TFRC backbone transmission rate. Chapter 6 presents the emulation results, describes if the suspected behavior exists and chapter 7 suggests several potential tuning options for the scenario.

We used FACK TCP as transport protocol between the server proxy and the client (defended in section 6.3.4) in the experimental setup as we wanted to implement a solution that would give us higher throughput and limited drop to one behavior on the TCP connection. The TCP implementation used in our target scenario follows RFC793, RFC1122 and RFC2001 with the New Reno and SACK extensions. RFC2883 duplicate SACK support (DSACK) is enabled by default. It also supports RFC1323 which is extensions in TCP for higher performance on high bandwidth delay products (BDP). RFC1323 proposes a technique called windows scaling which makes it possible to advertise a window size larger than 65536 bytes.

During the emulations described in chapter 6 we started out with a pure reflection proxy. A reflection proxy takes all incoming data and forwards it immediately without any regards. Such a scenario would function great if at all times the client side TCP connection gives equal or higher throughput than the backbone TFRC connection. When this was not true, the proxy had to implement some sort of behavior to cope with the increased TCP forwarding buffer fill up. We discovered 2 different angles to attack the problem from:

- The more theoretical angle would be to manipulate FACK TCP's original behavior to achieve what we want.
- The more practical angle would be to supervise TCP from application level and try to act in such a manner that the application would relax the data pressure in case of high back pressure.

We did not want to modify TCP's original behavior. In most POSIX systems TCP is part of the kernel and manipulating the standard congestion policy would be the last step in order for us to create effective streaming. Such a scenario could be extremely interesting in a more theoretical context, but if any interesting results were found they would most likely be quite useless in a practical manner. Deploying a quasi-TCP version with streaming friendly congestion policy on the Internet would most likely not be very appreciated and also be somehow useless as the most suited congestion policy would end up like UDP with no congestion control at all. We therefore chose the second angle of the two.

Below we have summarized ideas on how to tune TCP throughput between the proxy server and the client. As mentioned it was appreciated that the Komssys proxy extension operated only at application level. The intention was to monitor TCP throughput and react to congestion occurrence in the application and at application layer. When the proxy server monitored the TCP connection with the client, application level code in the proxy might execute actions to deal with potential low throughput and congestion. We had several ideas that could be implemented to see if they would solve the potential problems. These ideas could lead nowhere but they could also give better results on the client side TCP connection. It was up to the TCP research and quantitive data to figure out if any of these ideas actually could improve throughput and smoothness. In chapter 6 the proxy solution was put to the test in an emulation test bed. The bulleted list below adds up our ideas.

- The proxy will be able to sense when the client side TCP connection experiences congestion. When the proxy application tries to send more data out on the TCP connection it will discover if TCP's send buffer is full. A full buffer at the TCP sender side indicates congestion. To avoid further congestion the proxy has to inform the original sender that it should slow down its sending rate. The proxy generates fake TFRC receiver reports or fake losses and send them back to the sender to make it reduce throughput. This behavior can help the client side connection avoid further congestion.
- The proxy can be placed physically close to the streaming client. Physical closeness between computers in a TCP session usually result in shorter

round trip times. Shorter round trip times result in shorter timeout intervals for segments and faster recovery periods. A packet loss can be identified faster when the physical distance between 2 computers involved in a session is shorter. In addition the congestion window will build up more rapidly.

- The streaming media can be based on hierarchical layered frames. Layered frames present adaptivity possibilities.
- The client side TCP connection can be based on the newer more aggressive FACK TCP flavor for improved throughput.
- The TCP queue in the proxy can be managed by priority progress streaming. This means dropping lower prioritized layers of a frame for an incoming higher prioritized layer of a frame.
- The TCP socket buffer can be minimized for maximum control over the queue using for example priority progress streaming.
- "Bad" layers received from the backbone stream can be thrown away so that they are not transferred to the client. Badput and goodput are explained in section 5.1
- For improved backbone transfers it might be valuable to implement TFRC rate control modeled after FACK TCP instead of TCP Reno. This is especially important if the forward TCP queue to the client is empty most of the time. If it is mostly empty it is an indication that the client access connection has potentially higher throughput capabilities. It can also be invesigated how to send faster than the application specific bitrate ig allowed by TFRC.
- The client can buffer an amount of the stream before presenting it to the user. The proxy can manage the application queue in such a manner that the total playback time for the part of the stream in the queue does not exceed the playback time buffered at the client. The queue can therefore be limited by a playback threshold.

We could attack from numerous angles to experiment with TCP. In this subsection we have touched some of the ideas and it is up the experimental part of the thesis (Chapter 6) to present the controlled experiments. Based on those experiments we further present tuning options for the testbed (preferably one or more of the ideas mentioned above). If any ideas are disregarded we explain why.

3.6.1 The stream oriented TCP versus message based transport protocols.

In chapter 4 we describe in detail the necessary implementation changes to the Komssys streaming system to support our research model. The details presented require the reader to have knowledge of differences between stream oriented and message oriented transport protocols. This distinction is important to understand as the modified implementation extends Komssys to handle the stream oriented transport protocol TCP.

Message based transport protocols send and receive messages. Messages are composed of one or more bytes. The transport protocol treats these bytes as one logical unit. The message is transmitted and hopefully received at the target computer. The receiver can read the whole message or no message at all. Only when the message is conveyed to application level the inner details of the message can be extracted. Komssys supports UDP as transport protocol which is a message based transport protocol. Application level bundles RTP packets and transmits the packets as messages to the receiver. This behavior makes it easy for the receiver to know the boundaries for an RTP packet. The message itself consists of one or several whole packets.

During this thesis Komssys was extended to support TCP as transport protocol for RTP data. TCP is stream oriented, which does not preserve logical boundaries between data. All data sent to TCP are treated as a sequence of bytes. When the receiver wants to read RTP data at application level on the other side it only sees a sequence of bytes. It is up to application level protocols to agree on how these bytes should be interpreted. TCP did in our case complicate the existing message based solution. We had to add logic that made the receiver discover when an RTP packet data started and ended in the sequence of bytes.

The next chapter is dedicated to the implementation changes necessary to conduct the experiments.

4 Implementation changes in Komssys

4.1 General description of Komssys

Komssys is a platform for experimental VoD research [43] developed mainly in C++. The platform was developed to support scientists and students doing research on streaming audio and video content over wide area networks (WAN). Since the platform was created, its functionality has been continuously extended. When developers designed this system they therefore had to focus on making the platform:

- Reusable from the technical as well as the legal point of view.
- Modular with well-defined interfaces
- Interoperable with other standard compliant tools
- Integratable with existing code

Reusable means that the system modules are easily moved in and out from different parts of the implementation. For example can numerous modules in the client implementation also be found in the serverside implementation. The intention was that such modules, when first created, could be plugged into both sides of the implementation. Interoperability is supported by choosing the most widely used protocols at all levels. The protocols used are RTP/RTCP for streaming and feedback, RTSP for signaling and SDP for session control. These protocols are standardized by the Internet Engineering Task Force (IETF). Integratable with existing code means that modules and solutions from other projects could easily be converted to fit into the Komssys framework.

As mentioned, Komssys is based on RTP/RTCP for streaming and RTSP for signaling. It consists of a server, different proxy solutions and a client. At present the different proxy solutions are:

- **Reflection proxy:** A reflection proxy forwards all incoming data immediately to the outgoing port. This is at the moment implemented as a zero-size cache.
- **Caching proxy:** The primary task for caching proxies are keeping exact copies of original multimedia presentation delivered from origin streaming servers and be able to deliver these presentations to clients when requested. The main intentions of a caching proxy are reducing server load by distributing replicas of presentations and being able to stream these presentations from caches near the client. In Komssys the cache retrieves exact copies of multimedia files by the means of the earlier described LC-RTP.
- **Patching proxy:** A patching proxy is a special kind of a caching proxy where the whole multimedia file is not stored on the patching proxy. Instead the proxy keeps a predetermined piece of the start of the file. When a client requests a presentation, the patching proxy immediately starts to stream a presentation to the client. At the same time the patching proxy

initiates a transfer of the rest of the requested file from the origin streaming server. A patching proxy shortens the average response time per client and can give a higher feeling of true on demand video capabilities.

- **Gleaning proxy:** Gleaning combines patching and caching. When a client at time t requests a movie, the request is first received at the caching proxy. The caching proxy does not have the requested content and initiates a multicast session with the origin streaming server which delivers the requested file to the cache. The cache starts a unicast session with the client. If another client at time t+1 requests the same movie, the request might end up at another caching proxy. The caching proxy does not have the content and requests the content from the origin server (the same server as the first client uses) by the means of a multicast session. But the session the proxy cache joins is the same as the the multicast session established for the request from the first client. Since time (t+1)-t has passed since the first multicast was initiated, the second proxy will miss the start of the media. The start of the media will be served by the means of a unicast patch stream between the origin server and the caching proxy from the second client request.

UDP is used for transport of RTP/RTCP data, while support for TCP was developed through the time of working on this thesis. RTSP is transported with TCP, RTSP over UDP is not supported. RTSP is used for controlling delivery of real time data. For example when the client user presses the 'PLAY' button in the application GUI, RTSP will signal the server that the client is expecting streaming of the defined multimedia content. When such actions occur, there is a need to define and control communication between the modules of the system based on the current state. To control state transitions, a finite state machine (automaton) was developed. This solution solved several issues regarding intermediate states in the system. For example can one RTP connection to a video server be established, while the RTP decoder is not yet initialized. The automaton takes care of solving such issues. The state machine controls establishing RTP sessions, initializing decoders and in general function as a link between RTSP and RTP. Based on signals from the RTSP layer, the automata controls the RTP level of the system. When a 'PLAY' is requested, the client has initialized a graph manager that will control the reception of incoming RTP packets. The graph manager supervises several streamhandlers (SH). The streamhandlers have one endpoint for reception of data and one endpoint for sending data. In between, the streamhandler will execute specific actions on the data. In figure 12 the abstract overall solution of Komssys is presented. It does not include the proxy. For further details on the design of Komssys see [43].

To implement the solution described in this thesis, the choice of integrating new code into Komssys came natural. To make the platform support our basic experiments with a reflection proxy, we had to:

- Extend the TFRC implementation
- Extend Komssys to stream RTP over TCP.
- Extend Komssys to translate incoming RTP/UDP packets to outgoing RTP/TCP packets in the proxy.

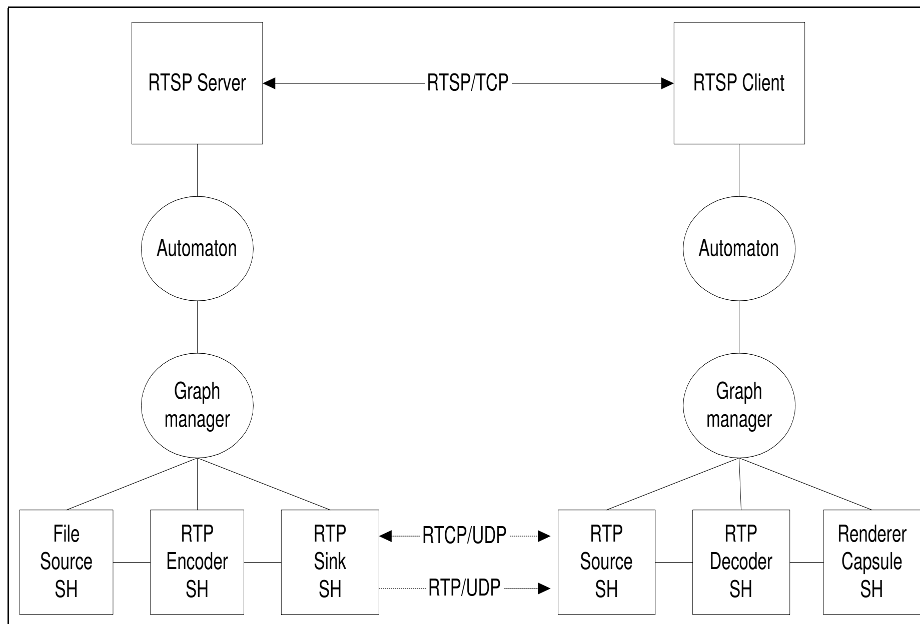

Figure 12: Komssys overview

In the following three subsections we will present the implementation changes needed for extending Komssys with the described functionality.

4.2 Extended the TFRC implementation

Details about the implementation of TFRC in the RTP framework can be found here [10]. Support for the additional information required by TFRC is in Komssys implemented by adding information to the RTP and RTCP packets. As described in [10] the sender has to forward a sequence number, a timestamp, an estimation of the round trip time and the senders current transmit rate to the receiver in every RTP packet. The first two parameters are already contained in the regular header of RTP. The two other parameters are contained in the RTP extension header (see section 3.4.1). The regular header can be marked as including an extension header. The extended header carries additional information that can be interpreted and used if the RTP receiver understands the extension. If the extension is not understood, it is simply ignored.

If the RTP receiver is TFRC capable, it will extract the extension header information and use this for TFRC specific purposes. Because the sender needs information about how the client experiences the transmission environment, the client has to regularly send feedback packets to the sender. The feedback packets contain information necessary for the sender to compute the new TFRC transmission rate. The TFRC feedback reports are carried by the means of application defined RTCP packets. It is possible to use RTCP packets to transmit totally application defined information. The different RTCP packet types were described in section 3.4.2. The application defined RTCP packets are discovered

by the sender by looking at bits 8 to 16 in the header (See figure 11). The value 204 means that it is an application defined RTCP packet. Values 200 to 203 indicates other types of RTCP packets. We refer to section 3.4.2 for further details on the RTCP packet format.

In order to compute the new transmission rate, the TFRC receiver has to send this information back to the TFRC sender (In Komssys the implementation is slightly different):

- The timestamp of the last data packet received.
- The amount of time elapsed between the receipt of the last data packet at the receiver, and the generation of this feedback report.
- The rate at which the receiver estimates that data was received since the last feedback report was sent.
- The receiver's current estimate of the loss event rate.

As mentioned in [42], Komssys also supports LC-RTP. This protocol is an extension to RTP including application layer loss control. Loss controlled RTP can be used for reliable transfer over a non-reliable transport protocol. In Komssys it is used for transferring multimedia data to cache servers. Cache servers should contain exact copies of the data stored on the origin server. LC-RTP specific information is carried in the extension header. To support both LC-RTP and TFRC, the extension header is as described in section 3.4.1 in this paper and in 4.1.1 in [42]. In section 3.4.1 the RTP header and the RTP extension header were presented. If the X bit is set, the extension header will immediately follow the CSRC list in the fixed header.

In order to use TFRC as intended in this research, the Komssys implementation had to be manipulated some to make it support the framework for the experiments:

- TFRC congestion control support between origin server and proxy had to function as intended.

The next section describes implementation changes necessary in the context of TFRC.

4.2.1 TFRC congestion control support between proxy and origin server

As the client and server were already TFRC capable just prior to our work, only minor changes had to be made in order for the proxy client to be TFRC capable. Changes included RTSP implementation changes and minor changes to the retransmission classes in Komssys.

4.3 Stream RTP over TCP

At present, Komssys supports streaming multimedia data over UDP. It is the most common transport protocol to use for multimedia streaming applications.

UDP is not reliable, has no congestion control and does not use retransmissions. As long as RTP packets are available from the application level, UDP wraps them with UDP headers and sends the new packets down to IP at a predetermined stable rate. This behavior suits particularly well when it comes to streaming. Some packets will never reach the destination, and the client user will experience video disturbance. The most important property of UDP is that it keeps sending out packets at constant pace with no regard to concurrent traffic, network congestion. It does not retransmit packets or back off during packet losses.

To add support for RTP/TCP we have to understand the main differences between UDP and TCP. These are:

- UDP is connection less while TCP is connection oriented and requires a 3-way handshake connection negotiation.
- UDP is datagram oriented while TCP is stream oriented. The sender must preserve RTP packet boundaries by implementing an application defined datagram service.
- The receiver must understand and use the application defined datagram service to read RTP packets from the network.

To achieve the goals stated in 1.3 we had to add support for streaming data with TCP between the proxy server and the client. Based on the differences between TCP and UDP we implemented necessary changes as described in the following subsections.

4.3.1 TCP connection negotiation

Unlike UDP, TCP is a connection oriented transport protocol. We had to change Komssys to be capable of establishing 2 TCP connections between sender and receiver. During connection establishment, one side of the connection was acting as the TCP server while the other side acted as the TCP client. The TCP server created one or more sockets, and listened for connection requests on those sockets. To avoid potential problems with firewalls it would make sense making the RTP sender the TCP server in the TCP setup negotiation. Because of firewall policy and Network Address Translation (NAT) it would be possible to connect to a TCP server from inside the firewall to a machine outside the firewall. The machine behind the firewall would serve as the multimedia client. This solution would limit difficulties with connecting through firewalls. Unfortunately we implemented our solution the other way around, making the multimedia client the TCP server. This setup could cause problems when an outside machine tries to connect to a socket on the machine behind the firewall. The problem is not the firewall, but the potential use of Network Address Translation. Switching the TCP server to act as the TCP client and visa versa is work that might be executed post ceding this thesis.

The RTP RFC did not specify any rules for TCP connection negotiation. Implementation changes made it necessary to create a new class named RTPServerSocket. It is a child class of TCPServerSocket which again is a child class of

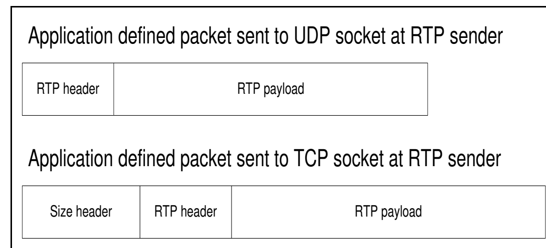

Figure 13: UDP and TCP application defined packets

TCPSocket. This class sets up 2 listening TCP sockets, one for RTP and one for RTCP. After these sockets were created and in listen state, the client issued the RTSP SETUP request with the port numbers of the listening ports. The RTP sender received this information and called connect to the TCP endpoints. The RTP receiver then accepted the connection requests. The TCP connections were established. After connection negotiation the server and the client functioned as equal peers. The next step was for the RTP sender to transmit a SETUP response back to the RTP receiver.

4.3.2 Main implementation changes in the RTP sender application

TCP is a transport protocol that does not support any kind of packet boundary logic if an application wants to read data from a socket. When the sending application sends an RTP packet to a TCP socket, the packet will not longer be treated as a separate packet, but just added to the stream of data in the TCP send buffer. The receiving application has no knowledge about the size of these packets. The implementation had to make sure that the server informed the client about how much to read from the stream. The server had to report the RTP packet size before the actual packet was read. Figure 13 shows how we defined packets when streaming using TCP.

When reading from a UDP socket, the receiver will receive a whole message (if it is not lost). The message received is exactly equal to the message sent. The receiving socket will receive 1 whole RTP packet (or no packet) in its buffer for every read. The nice property of UDP socket read system calls is that they either receive 1 message or no message at all. UDP sockets will never read parts of a message. TCP works the other way around. When reading from a TCP socket it is certain that no more than the application defined buffer size is read. But the read call can catch less data than can be contained in the free buffer. It can catch everything from 0 bytes up to the application defined buffer size. The solution we chose during implementation was to add a 5 bytes information unit preceding the RTP header (figure 13) in the RTP sender application. Before the RTP packet was shipped down to TCP, the application retrieved the size of the whole RTP packet (including its header) in bytes and added the size as a 5 byte header preceding the RTP packet itself. The packet was then dispatched to the TCP layer using the `sendmsg()` socket system call. When `sendmsg()` was called on the TCP socket it was not certain that the whole packet was successfully sent

to TCP. If the TCP buffer was almost full or full when `sendmsg()` was called, only a part of the packet might be transmitted .The application code had to consider this and make sure that everything in the application buffer was sent when the application wanted to transmit a new RTP packet.

If a partial packet was sent during a `sendmsg()` call, the remaining bytes were added to a queue of bytes. This queue contains all bytes not yet sent. The queue remained in an own class with the name MNSocketQueue. This class does the rest of the processing of the outstanding bytes. The MNSocketQueue did during RTP initialization register itself to the MNSelector (Figure 15). When the MNSelector discovered free buffer space in the socket send buffer, it notified the function `MNSocketQueue::send_bytes()`. Figure 15 shows how classes register themselves to the MNSelector. The RTP and RTCP classes in figure 15 would ask to be notified when incoming bytes are added to the socket buffer (Stream receiver). MNSocketQueue asked to be notified when the socket buffer had additional space (Stream sender). When the socket buffer discovered available space, the MNSelector class notified MNSocketQueue by calling the registered method `MNSocketQueue::send_bytes()`. Figure 14 shows a scenario where a chunk of multimedia data is received by the `MNRTP::rtp_send()` method from the encoder. As the figure implies, packets could be partially sent because of missing buffer space in the socket. The socket returned the number of bytes sent. If this number is not equal to the size of the whole RTP packet, the remaining bytes must be sent later. The `MNRTP::rtp_send()` cut off already sent bytes and added the remaining bytes to a byte queue managed and contained in the MNSocketQueue class. After a while, the MNSelector discovered that the socket had available buffer space. It notified the MNSocketQueue class by calling its registered method `MNSocketQueue::send_bytes()`. This method tried to send everything that was in the queue. If everything was successfully sent, the queue was emptied. If not everything was sent, `MNSocketQueue::send_bytes()` cut off the sent part and the rest of the data was still in the queue. At a later moment the `MNSocketQueue::send_bytes()` was called again.

With this solution at hand it is surtain that TCP sent exactly the data we wanted and in the order we wanted. That means 5 bytes header, RTP packet, 5 bytes header, RTP packet, 5 byte header and so on. This scenario was implemented for sending RTP packets without extension headers. Indirectly this implies that LC-RTP or TFRC support is not supported using TCP as transport layer protocol. Support for LC-RTP or TFRC can at any time be implemented, but neither LC-RTP or TFRC support over a TCP connection would make much sense.

4.3.3 Main implementation changes in the RTP receiver application

Before a client issued the RTSP PLAY request, it had already created two UDP sockets. One for receiving RTP data and the other one for sending and receiving RTCP packets. The socket descriptors for these sockets were registered in the MNSelector object. The MNSelector administered all sockets registered in it using a select set, and took care of executing appropriate actions when data was received at a socket where the socket descriptor matched the socket descriptor of a registered application socket class. Figure 15 shows what happened when

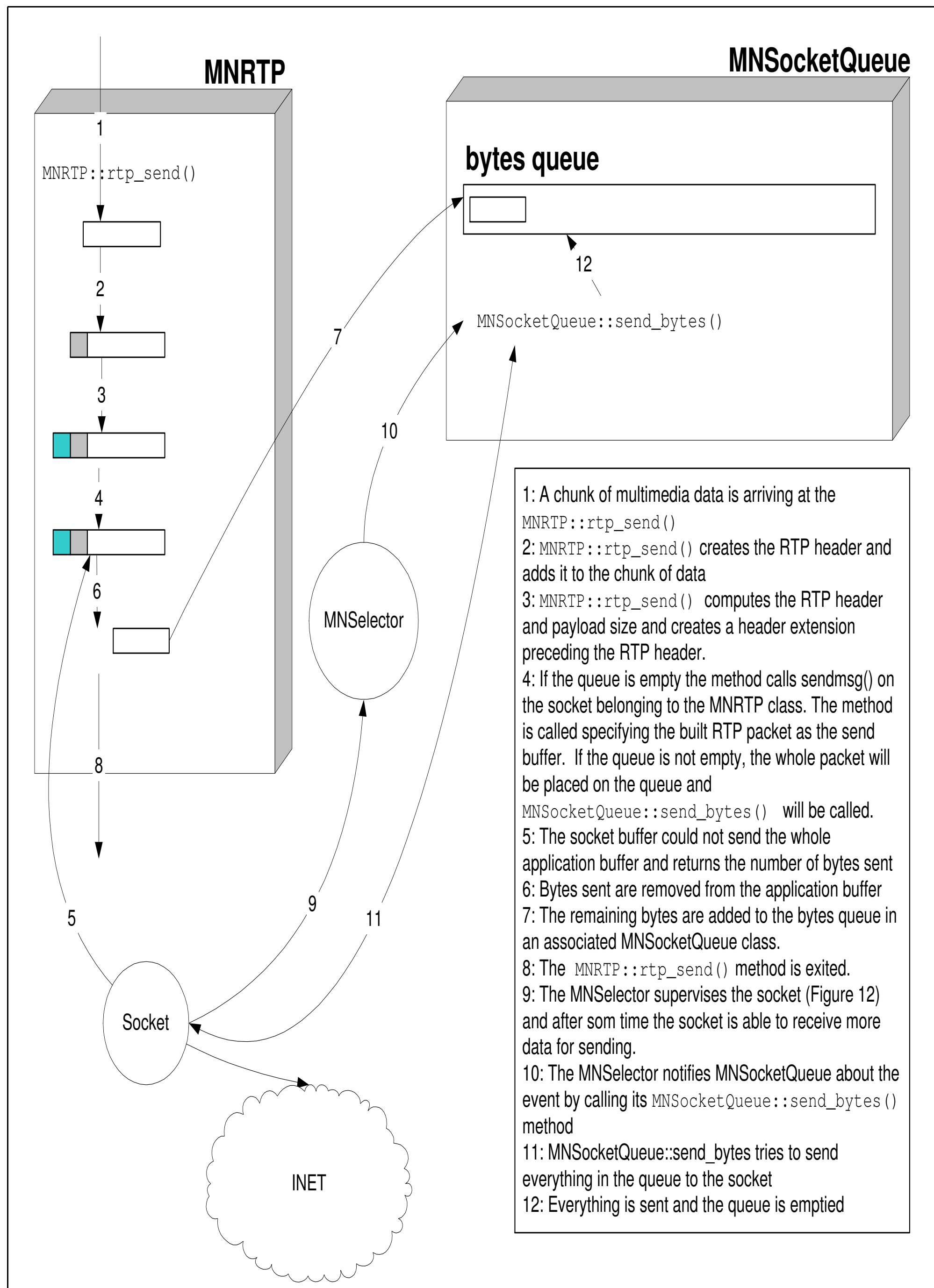

Figure 14: Administration of not sent bytes after a `sendmsg()` call

RTP data was received at the Network interface card (NIC).

The RTP and RTCP application socket classes are indirectly instances of the MNUDPServer class. In a class collaboration diagram this class is holding a UDP socket descriptor, and is a child class of MNUDPServer, which again is a child class of MNUDPSocket. The implementation of the callback function lies in the MNUDPSocket class. For our implementation we wanted to preserve the existing classes and make as small changes as possible. The callback function in the RTP and RTCP application socket classes had to be able to handle both TCP and UDP traffic. After a while manipulating the callback function, we understood that it was better to separate the callback functions. If UDP streaming was enabled, then the regular `callback()` in MNUDPSocket was called. If TCP streaming was enabled, then a new method `tcp_callback()` would be called. The MNSelector was not changed and would still use `callback()` on the socket class, but the call was redirected to `tcp_callback()` if it was based on a TCP socket. The `tcp_callback()` was implemented in a new class called DatagramSocket. The Datagram socket class was part of the RTP and RTCP application socket classes and was placed above MNUDPServerSocket in the class collaboration diagram. `Tcp_callback()` would first read the size header (figure 13) from the buffer. It then made sure that this number of bytes were read from the socket. When the whole packet was received the packet was propagated up the class hierarchy.

The sender had to be sure that the whole packet was sent (figure 14) and the receiver had the opposite problem. In the `tcp_callback()` in DatagramSocket, the callback would not call `reader()` before the whole packet was received. If the `recvmsg()` socket system call did not grab the whole packet, we stored what was read and waited for the MNSelector to call `tcp_callback()` again. When the whole packet was completely read it was propagated up the system by calling the `reader()` method. In figure 12 we show that the packet finally enters RTPSourceSH which again passes it on to RTPDecoderSH.

4.4 Create RTP/UDP to RTP/TCP filter in proxy

For details on the Komssys proxy solution design we refer to [6]. The proxy basically consists of one proxy client (which acts as a client for the streaming server) and one proxy server (which acts as a server for the streaming client). The proxy client and the proxy server are finite state machines (automata), both running in the process RTSPProxySession. Because of Komssys' good re usability and modularity, changes in the proxy were minor. The proxy uses mostly all of the modules we changed earlier, and the implementation changes consisted of forwarding and converting RTSP messages. If the client wanted to stream over TCP the RTSP SETUP message specified that. The proxy server would receive this message and (because of re usability) would automatically make appropriate actions to establish a TCP session with the client. The proxy client can not send this RTSP message to the origin server before the support of transport was changed from TCP to UDP. This change required minor modifications to the RTSP classes of Komssys. The proxy client would request UPD transport with the origin server. If the proxy client was configured to ask for TFRC transmission control, we had achieved a streaming session with a mixed transport

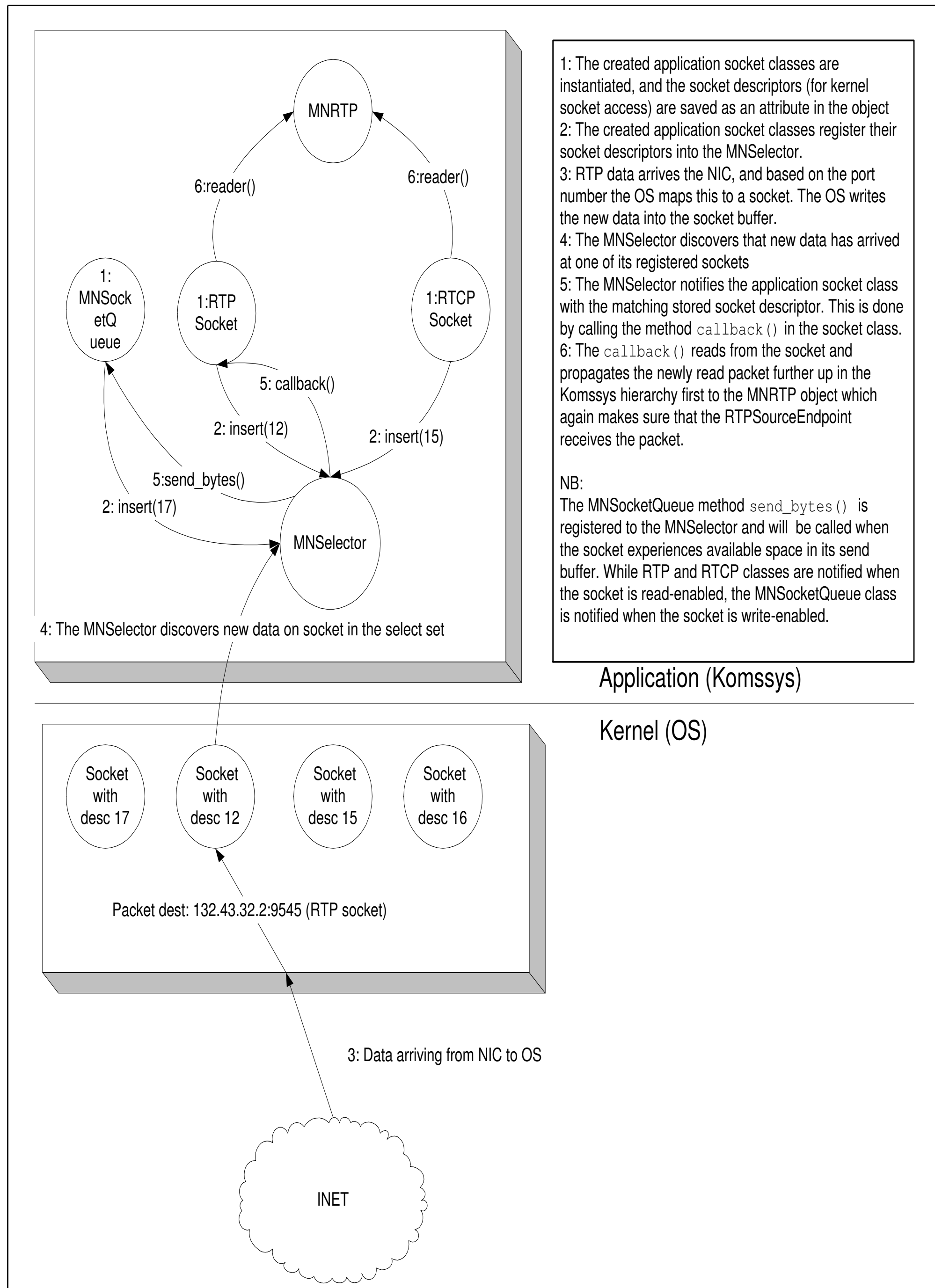

Figure 15: Incoming RTP data packet

environment: TFRC/RTP/UDP streaming between the server and proxy client and redirect the stream to RTP/TCP between the proxy server and the client (figure 8). Figure 16 shows an example of an RTSP session establishment with the RTP/UDP and RTP/TCP solution where the proxy is reflecting from origin server.

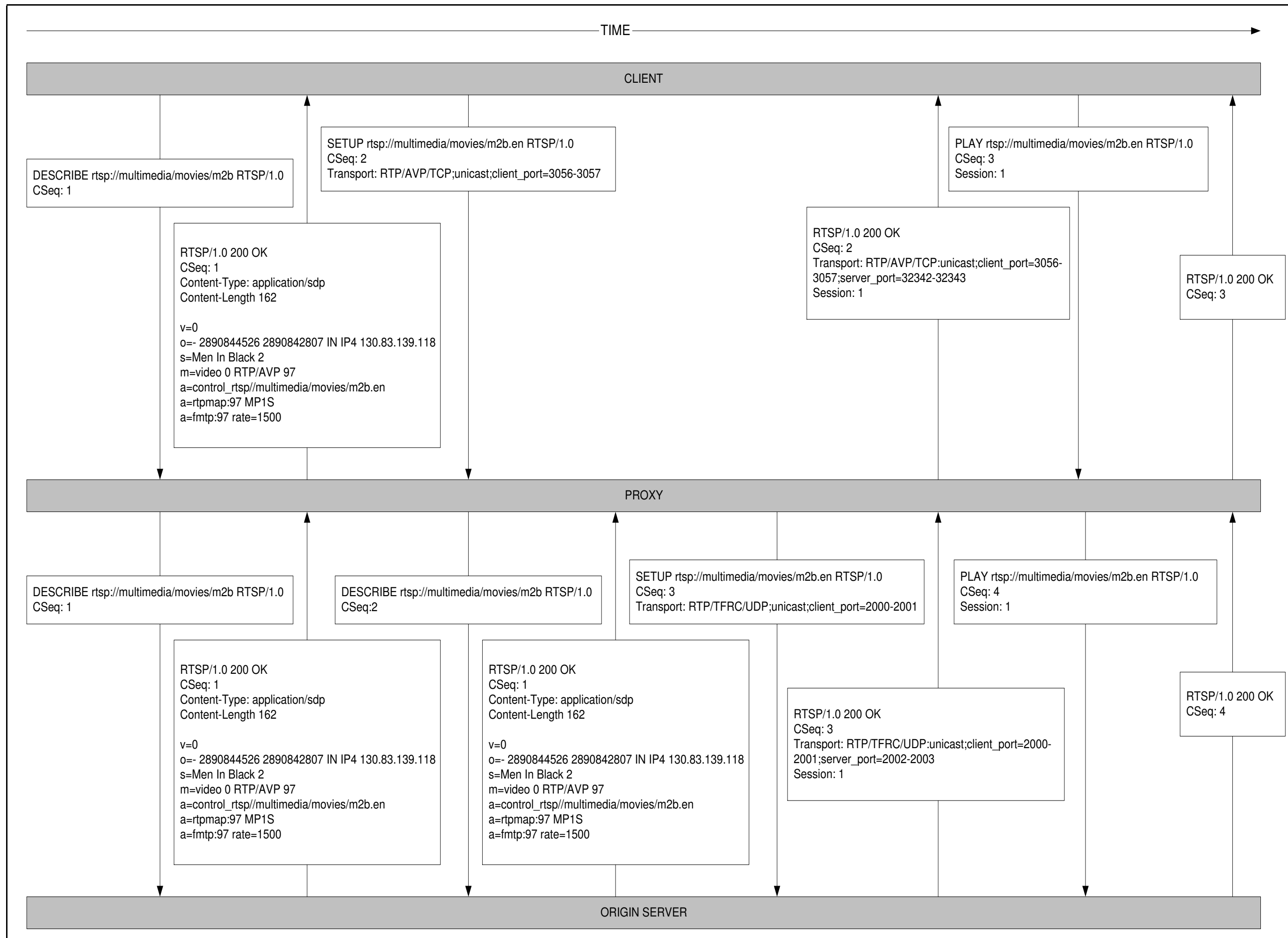

Figure 16: RTSP signaling with mixed transport protocol streaming through proxy

5 Experiment preparations

In the previous chapters we tried to give the reader a fair understanding of the framework for our experiments. The focus of this chapter is to describe how we proceeded to execute the experiments in a controlled manner. To understand how a TFRC based streaming session can be combined with a TCP based streaming session in the access network this chapter is divided into three parts. First we emulate a TFRC transfer between a client and a server, describe the observed results and end up with a discussion based on the results. This transfer emulates round trip time parameters that can typically be observed in backbone networks as the TFRC session will be used in the backbone network. After the emulation of TFRC sessions, we emulate a TCP streaming session that would resemble round trip time parameters seen in typical access networks in todays Internet. We will also in this scenario describe the observed results and finally discuss the results. After we finish these emulations the TFRC streaming session emulation is combined with the TCP streaming session emulation to discuss how these could work together in a combined setup. This discussion resembles a pure reflecting proxy. All packets arriving at the proxy are forwarded to the client. Such a discussion combined with the two emulations gave us a fair insight to the behavior of a mixed environment transfer and made us well prepared for the next chapter.

The next chapter will be dedicated to several experiments using the full proxy included solution. The emulations documented in the next chapter will resemble all parameters found in typical backbone and access networks respectively. The choice of parameter values like round trip time, bandwidth, forwarding queue size and traffic load are of crucial importance to gain realistic results. After the emulations we discuss several tuning options for the scenario.

The three next sections in this chapter will document the two emulations just recently described and the discussion of a combined setup. These emulations should make us more prepared for understanding the results in the following chapter. Note that the results gained in this chapter is not meant to resemble accurate real-world transfer parameters, but merely a tool to observe the behavior of TFRC and TCP respectively.

5.1 Emulation of TFRC backbone transfers

The TFRC based streaming session did in our scenario use a hierarchical layered video format that somehow resembles Scalable MPEG [19]. The scalable media format implemented in Komssys generates what is called dummy layers. The layers are created dynamically at runtime and contain no actual video information. The layers are just stuffed with empty data based on a dummy layer SDP description file. For more information about the Komssys dummy layer solution we refer to [41].

In a hierarchical layered streaming scenario a frame contains a specified number of layers. For one layer to be valid and contribute to the overall quality of the frame, all layers below it will have to be received on time and used as a basis for the layer. Streaming hierarchical layers over a lossy transport protocol intro-

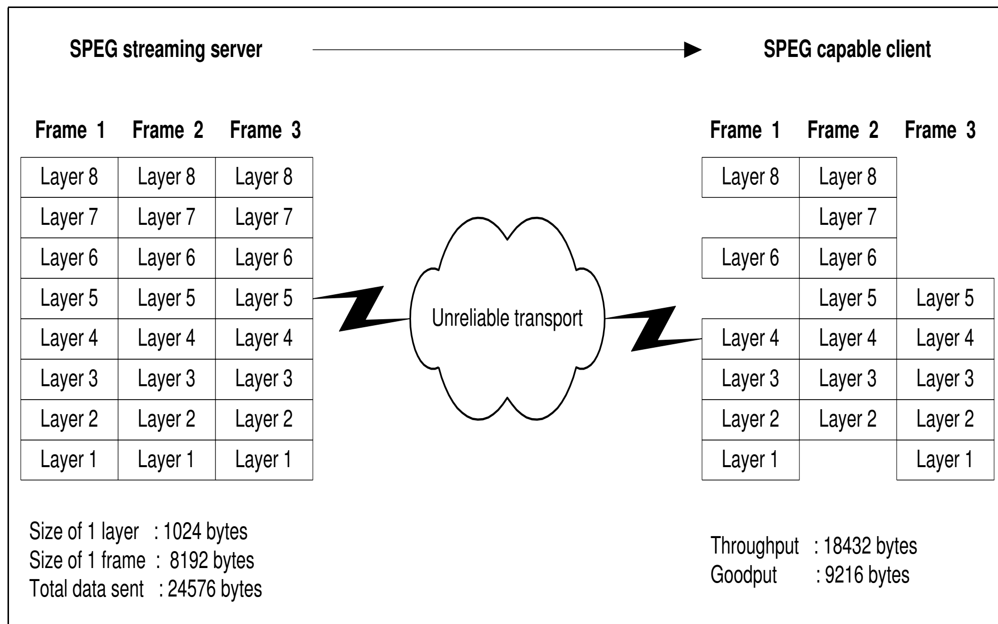

Figure 17: Throughput versus goodput using hierarchical layered streaming

duces a distinction between goodput and throughput. If the sender transmits for example all 8 layers of a frame to a client or a multicast group some of the layers might not be received. Based on which layers are actually lost during the session, throughput and goodput can be quite distant from each other. In figure 17 we describe the overall idea.

The results presented later in this chapter is based on throughput/goodput graphs. We now describe the experiment setup and why we chose this particular solution. The testbed is presented in figure 18.

We chose NIST Net as a tool for emulating the network environment [2]. As seen in figure 18 a computer running NIST Net was placed between the streaming server and the client. In addition one computer was used to create concurrent traffic on the network path towards the client. To create parallel TCP and UDP traffic in this chapter we used a tool called "ttcp" which is a simple and well known tool for generating TCP and UDP traffic [25]. Ttcp can with one process instance create one TCP session between two computers and send data at full throttle. Sending data at full throttle means constantly keeping the TCP socket send buffer filled with data. The bandwidth between the streaming server and the client was set to 100Mbps. The delay on the network was 30 milliseconds, giving a roughly round trip time of 60 milliseconds. As NIST Net is basically capable of emulating many aspects of network properties it also includes the emulation of a router where forward queues can be configured as wanted. During this experiment we chose two different router queue properties to show how they affect our results. For each of the policies we show the observed results. NIST Net supports the Derivative Random Drop (DRD)

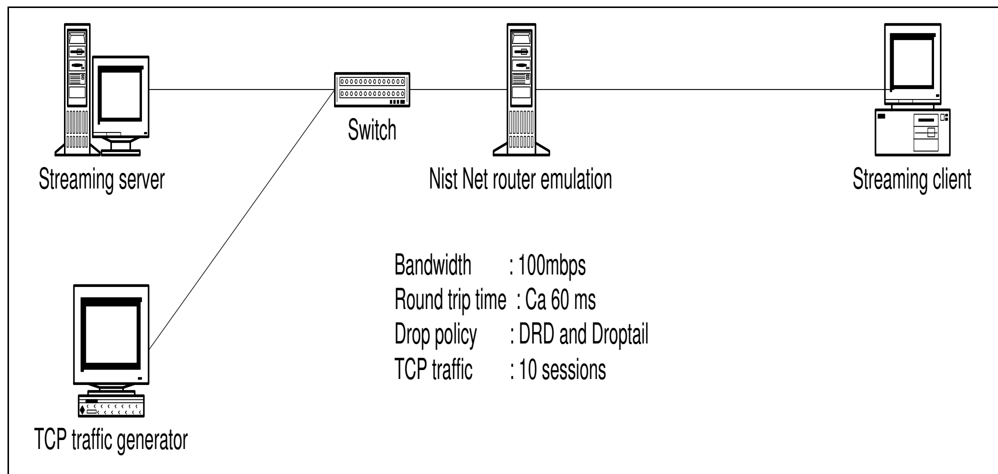

Figure 18: TFRC emulation testbed

algorithm, which is a follow up to Random Early Detection [5]. DRD uses a progressive drop scale where minimum and maximum threshold values are set. When the router queue passes the minimum limit the router starts dropping randomly 10% of the incoming segments. If the router queue exceeds the maximum threshold, 95% of the incoming packets are dropped. Between these thresholds a progressive scale is used. We show the results of two emulations, one with the DRD policy and one with the regular old fashion tail dropping policy where any incoming packet is dropped immediately if the buffer is 100% full.

For DRD the minimum threshold was set to 5 segments and the maximum threshold was set to 25 segments. For taildropping the queue size was set to 25 segments. The thresholds were set at this size to achieve a stressed network situation. The most optimized buffer size for the concurrent TCP sessions should probably follow the rule of the bandwidth delay product (BDP) [12]. The BDP states that the buffer size should be set to the product of the maximum bandwidth on the link and the end to end delay on the link. In our scenario a more realistic buffer size based on the BDP and the number of traffic sessions would be much higher. For the streaming sessions the buffer size should not give much affect as UDP does not perform flow control. In Chapter 6 where we combine TFRC and TCP streaming using mixed transport protocols separated by proxy we increase the buffer size parameters for getting closer to the BDP to give as realistic results as possible.

For streaming we used an 8 layer hierarchical dummy layer solution where each layer of the frame was of equal size. The size used was set to 1024 bytes pr. layer giving a total of 8192 bytes pr. frame. In figure 19 and 20 we show the observed results using DRD and tail dropping respectively.

If we concentrate on figure 19 we can see that hierarchical layered streaming using UDP can waste a lot of bandwidth. Right over 50% of all throughput is

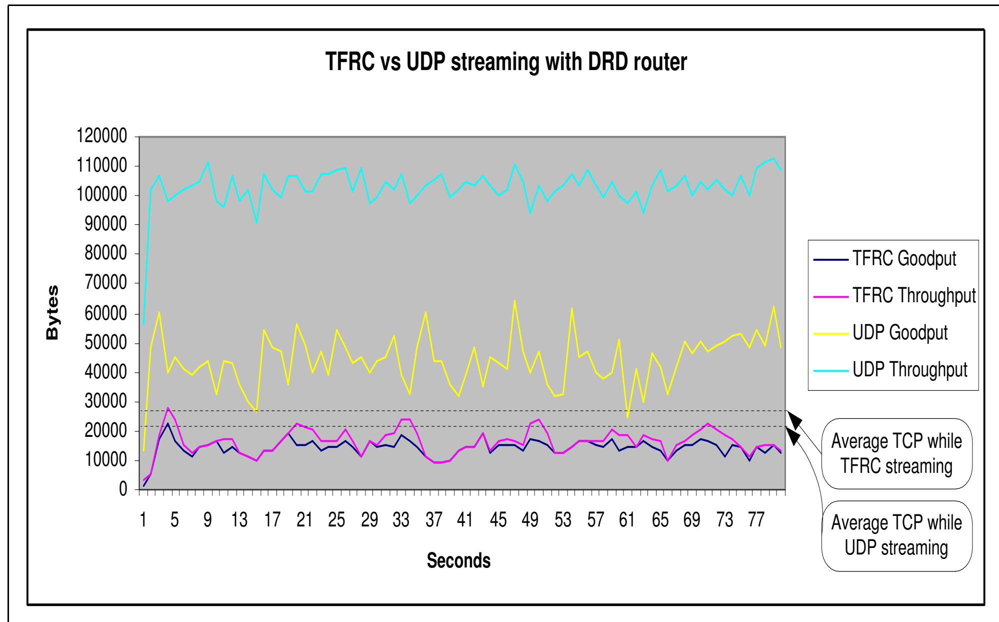

Figure 19: Hierarchical layered streaming using TFRC with DRD in router

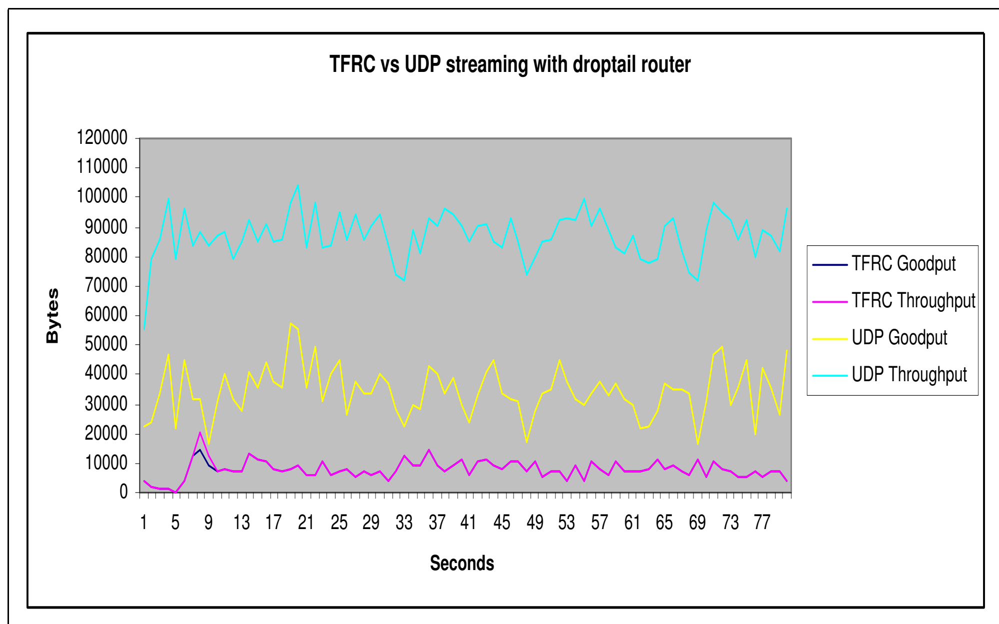

Figure 20: Hierarchical layered streaming using TFRC with tail dropping in router

useless. It's not only useless, but the UDP behavior decreases the throughput of the parallel ongoing TCP sessions. The lowest black dotted line shows the average throughput of all 10 TCP sessions lasting for 90 seconds. The streaming session lasted for 80 seconds. Figure 21 presents a table to show how TCP is degraded by streaming UDP instead of TFRC. Even though average goodput of UDP is considerably higher than TFRC goodput the UDP goodput varies a lot during the 80 seconds session. If we showed throughput pr. frame instead of average throughput pr. second we would discover the behavior presented in [41]. The number of valid layers pr. frame varies drastically with UDP giving unstable quality when the stream is presented to the user. For non-hierarchically layered solutions UDP would give good results. In the figure we can observe how TFRC goodput and throughput are very close giving minimal bandwidth waste. Most of the throughput is valid and considered as goodput. The highest dotted black line shows the average TCP parallel sessions throughput during the TFRC streaming session. A strange observation is that the average TCP throughput is quite much higher than average TFRC throughput. At first we did not understand why, but after a while we discovered that FACK was enabled on the TCP traffic generator and that SACK was enabled at the streaming client. So the parallel TCP sessions were based on FACK which is a relatively aggressive TCP implementation. The TFRC implementation in Komssys is as mentioned earlier modeled after Reno TCP which is more passive, and this test generated very high packet loss ratios. The figure therefore gave us the side effect that shows how the TFRC implementation in fact gets lower throughput than TCP because it is modeled after Reno. If we used Reno as a basis for figure 19 and 20 the TFRC throughput and goodput would much likely correspond better to TCP throughput. In addition, the TFRC throughput graph is calculated using the RTP payload size. The RTP header (12 bytes) and RTP extension header for TFRC (28 bytes) are not added as throughput in the graph. That means 40 bytes pr. received RTP packet are disregarded when computing the throughput. Even though FACK TCP is more aggressive, the actual TFRC throughput would have climbed a bit higher if we calculated the RTP header as throughput as well. Anyway the results are as expected and we can observe that (in the context of hierarchical layered streaming and highly congested networks):

- UDP wastes bandwidth.
- UDP degrades parallel TCP sessions.
- UDP gives large variances in goodput pr.second (quality jitter).
- UDP goodput is higher than TFRC goodput.
- TFRC does not waste bandwidth.
- TFRC does not degrade parallel TCP sessions
- TFRC gives small variances in goodput pr. second.

TCRC gives less throughput and therefore lower quality video sessions, but the sending rate is fairly stable and almost all throughput is valid. These properties are quite good for our final scenario. The receiver in this emulation was in the final setup the streaming proxy (Chapter 6). Since TCP was used as

	TFRC	UDP	Diff
tcp1	2657080	2192272	464808
tcp2	2094248	2512280	-418032
tcp3	2694728	1659408	1035320
tcp4	2258880	2422504	-163624
tcp5	2887312	1697056	1190256
tcp6	3157520	2235712	921808
tcp7	3205872	1944664	1261208
tcp8	1487096	1542120	-55024
tcp9	2699952	1849096	850856
tcp10	2350104	1699952	650152
Total	**25492792**	**19755064**	**5737728**

Figure 21: TCP traffic throughput with TFRC and UDP streaming

transport in the access network, the demanded throughput is much lower than the original throughput sent from the streaming server and almost all throughput delivered at the proxy is valid for further transport into the access network. By using this solutions the TCP session in the access network can be capable of delivering all received throughput to the access network client in time. The term "in time" basically means that the client application can access the data at the time of playback. The user in the access network might receive generally low and stable quality video streams.

In figure 21 we can see how TCP is forced to lower throughput when streaming non congestion controlled. In average, the TCP throughput using UDP streaming is 5.7 megabytes less than using TFRC streaming. It should also be noted that this is FACK TCP results. If Reno was used in this emulation the difference would most likely be much higher.

5.2 Emulation of TCP access network transfers

For emulation of TCP access network transfers we used the setup shown in figure 22. The setup is equal to the testbed in section 5.1. The difference lies in bandwidth, transport protocol and round trip time.

The bandwidth between the streaming server and the client was set to 10 Mbps. The end to end delay was set to 1 millisecond, giving a roughly round trip time of 2 milliseconds. Such bandwidth and round trip times can be found in typical access networks today, even though it is reaveled in chapter six that this is a very small access network round trip time. The choice of parameters is defended in the next chapter where it is crucial to achieve a realistic environment. For DRD the minimum threshold was set to 3 segments and the maximum threshold was set to 5 segments. The thresholds were set at this size to achieve a stressed network situation. An optimized buffer size should as mentioned follow the BDP. In addition we used 5 TCP and 5 UDP parallel sessions for traffic.

For streaming we used an 8 layer hierarchical dummy layer solution where all layers of the frame were of equal size. The size used was 1024 bytes pr. layer giving a total of 8192 bytes pr. frame. In figure 23 we show the observed results

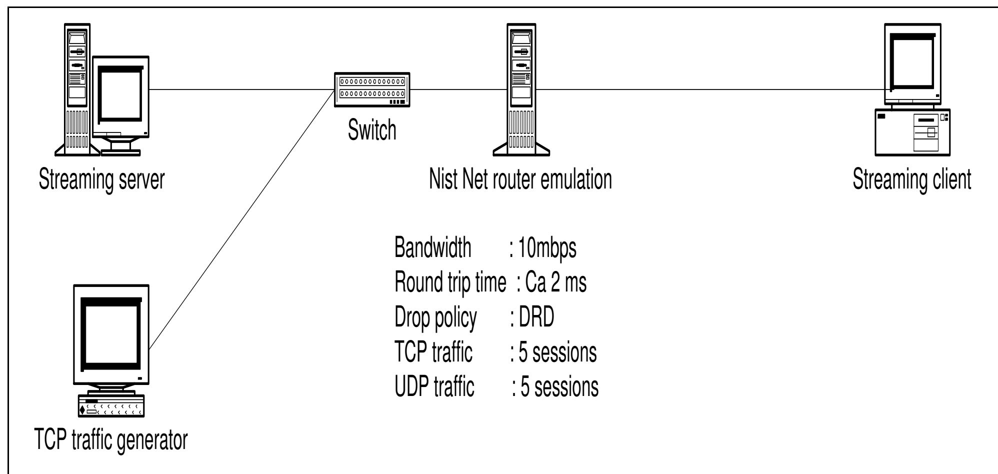

Figure 22: TCP emulation testbed

using DRD dropping.

Because changing the TCP flavor in newer Linux installations is quite easy the emulation was done for TCP Reno, SACK TCP and FACK TCP respectively. The 5 background TCP sessions were all changed to the same flavor as the streaming session TCP. For example the SACK TCP graph in figure 23 is a result of 5 UDP sessions and 5 SACK TCP sessions for background traffic. Appendix A contains a deep study of Reno, SACK and FACK TCP flavors.

As shown in figure 23, Reno lags behind FACK and SACK TCP when focusing on the average throughput lines. One interesting issue to note is that SACK and FACK give approximately the same average throughput. This is most likely due to small round trip times in the emulation. This behavior confirms the results in appendix A where it is stated that FACK is increasingly better than SACK when the round trip time increases. The throughput pr. second varies tremendously during the emulation. For multimedia streaming this is not a positive behavior, but the results are as expected. Why this variance might not a large problem in our work is discussed in the following subsection. The average throughput of all three flavors lies well above the throughput for the TFRC backbone session emulations executed earlier.

5.3 Discussion of a combined setup with a pure reflection proxy

This subsection discusses how our already executed emulations would function in a combined setup. Actual emulations of the total combined setup are described in the next chapter.

In figure 24 the TFRC streaming emulation graph is shown together with the FACK TCP emulation graph. The figure gives a good understanding of why

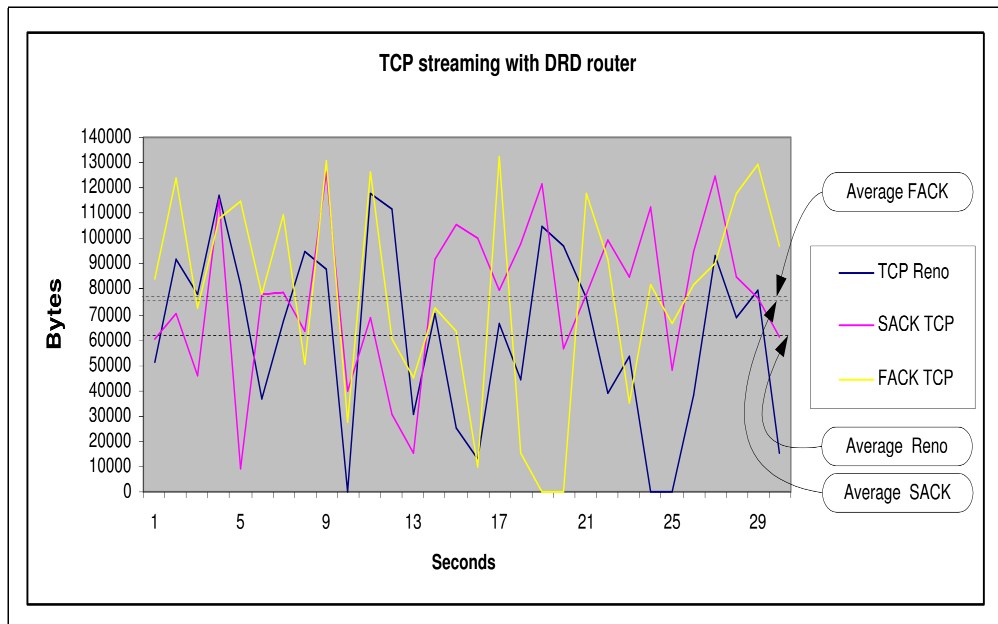

Figure 23: Hierarchical layered streaming using TCP with DRD dropping in router

steady TCP throughput might not be necessary when TCP is used in the access network. This assumption is based on the fact that TCP does not change its parameters and variables during idle time. In chapter 6 we investigate how TCP acts when its socket queue is close to empty and the congestion window is not filled up completely. The question to be answered is what TCP does with ccwnd, sshtresh and other parameters during idle periods. The TCP graph in figure 24 shows a TCP session that has data to send during the whole emulation period. In a combined setup this will not be true, as TFRC can deliver far less data to the proxy than the client site TCP connection can handle. That means TCP in periods might lie idle. Below we discuss this setup based on a situation where idle times do not change TCP parameters.

The TFRC throughput requirement is indeed low (very low because it is modeled after Reno) compared to TCP. TCP can deliver immediately (no queuing) except for during two regions in the graph. For the region between seconds 15-16 and the region between seconds 17-20 the TCP socket buffer will increase. Queuing in a pure reflecting proxy presents potential delay and jitter in the video stream. Packet loss is not a concern because we stream with TCP. To solve the delay problem the client could in this scenario buffer 3 seconds of the stream and start playing. The client would then receive all layers from the TFRC backbone streaming at the right time. This assumption is based on a worst case scenario where the client has not buffered at all and all intermediate queuing present a delay/jitter situation. Another solution would be to drop low priority layers from the queue in such circumstances. During the next chapter these issues are dealt with and several tuning possibilities for the heterogeneous

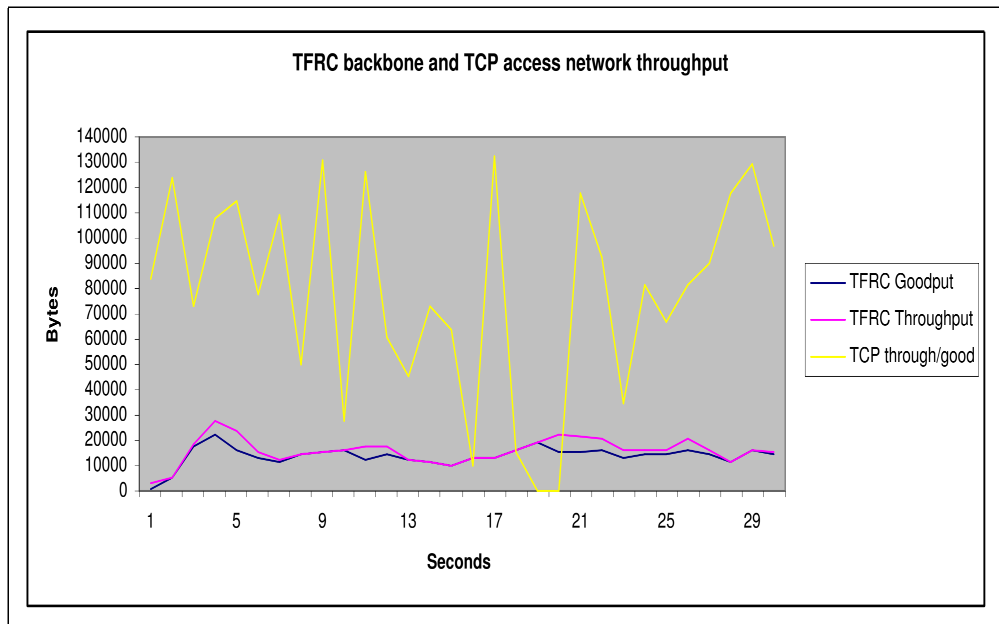

Figure 24: TFRC backbone and TCP access network throughput

transport environment are presented.

We have now tried to visualize the main goal of this research. By using TCP in the access network we believe it is possible to deliver smooth video streams to HTTP streaming clients. Even though TCP throughput is highly unstable it is generally giving much higher average throughput in the access network than the TFRC controlled backbone throughput. During the two next chapters we emulate and discuss several tuning possibilities for the heterogeneous transport environment based on real world emulation parameters.

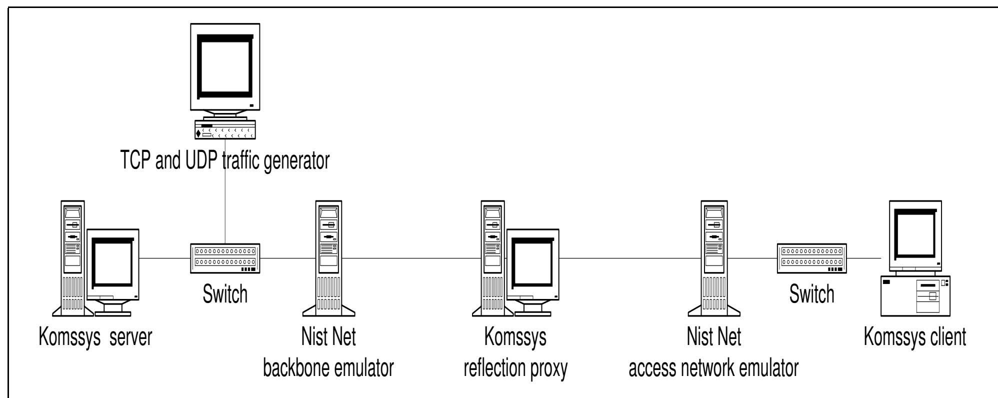

Figure 25: Emulation testbed

6 Main emulations

6.1 Introduction

In the previous chapter we emulated separate TFRC and TCP streaming sessions using hierarchical layered frames. The benefits and drawbacks of introducing these protocols in the context of streaming were explained. In this chapter the complete research setup is introduced. The focus of this chapter will concentrate on throughput, delay variations and queuing of data. The benefits and drawbacks presented in the last section do also apply here, but they are not discussed beyond what was revealed in chapter 5.

6.2 Emulation testbed

Figure 25 presents the emulation testbed. It consists of a Komssys origin streaming server, a NIST Net backbone emulator, a Komssys reflection proxy, a NIST Net access network emulator and a Komssys client. In addition a traffic generator is placed on the backbone. The emulation testbed therefore consists of a total of 6 computers, all running Linux (Mandrake for Komssys and SuSe for NIST Net).

6.3 Emulation parameters

During all emulations described in this chapter we urged to find emulation parameters that resembled real world Internet scenarios. These parameters include in particular round trip time, bandwidth and size of forwarding queues in routers. These parameters should range between typical values found in backbone and access networks respectively. The choice of which flavor of TCP to use in the access network is also of importance. In addition we defend our choice of background traffic load. The 5 next subsections describe and defend our emulation values for round trip times, bandwidth, size of forwarding queues in routers, flavor of TCP used and background traffic load respectively.

Destination	Country	# pings	Average RTT (ms)
karakalpakstan.freenet.uz	Russia	4	813
www.ort.ru	Russia	4	465
www.webwombat.com.au	Australia	4	340
www.terra.com.ve	Venezuela	4	142
www.mit.edu	USA	4	140
www.cuny.edu	USA	4	115
www.alitalia.it	Italy	4	102
www.oui.net	France	4	72
www.si.se	Sweden	4	20
www.ntnu.no	Norway	4	20

Figure 26: Round trip times in the Internet backbone

6.3.1 Round trip time

In order to find a suitable range of round trip times for the Internet backbone we used the well known ping utility. In figure 26 round trip times for several web-sites spread all our the world are presented. The pings were executed from a computer with an ADSL line. The emulations in this chapter therefore lie between 20 milliseconds and 813 milliseconds for backbone round trip time values. For access network round trip times we discovered a range from 2 to 20 milliseconds by pinging several servers at our ISP's site. This range was used to emulate access network round trip times. In NIST Net the delay from sender to receiver was set to equal the round trip time which basically means that packets go slow to the receiver and extremely quick back to the sender. The total RTT is therefore basically equal to the sender-receiver delay. When we mention round trip times from this point, the reader should understand that this is how we emulated round trip times in our setup. The most correct way of emulating the RTT would be to emulate about 0.5*RTT from sender to receiver and 0.5*RTT the other way around.

6.3.2 Bandwidth

For access network bandwidth we based our choice on what a typical private consumer can expect to receive from its Internet Service Provider (ISP). We considered ISDN and old fashion POTS to be obsolete and decided to choose the bandwidth on the current emerging Digital Subscriber Line (DSL). Many DSL types exist and examples of DSL flavors are Asynchronous DSL (ADSL), Synchronous DSL (SDLS), High-bit-rate DSL (HDSL) and Integrated DSL (IDSL). All the different types of DSLs give various bandwidth and some of them are asynchronous and some of them are synchronous. At the time of writing this paper ADSL is a very popular and widespread type of DSL and the access network will use bandwidths in the range of what ADSL can deliver. Currently a typical ADSL service from an ISP to a private consumer gives 700kbps - 2000kbps download and 300kbps to 600kbps upload speed. Note that both higher and lower rates are possible with ADSL. Based on these facts we chose our access

network bandwidth to be 1000Kbps download speed. We do not evaluate cable modem solutions which provides shared access service. ADSL provides a dedicated circuit connection and is more reliable in terms of bandwidth and delay. For the "blocking TCP" emulations (described shortly) we lowered the access network bandwidth to 512, 256 and 128 kbps to see the influence on the backbone bitrate. The bandwidth for the backbone emulation was chosen to be 100Mbps. We would of course like to use a more realistic value for the backbone bandwidth, but the best we can do in the lab is 100Mbps.

6.3.3 Router queue capacity

For router queue capacity we used the well known BDP algorithm [12]. It states that the size of a routers forwarding queue should be: (Maximum bandwidth on link)*(Delay on link). This equation gives us the size of the routers forwarding queue. It is a well known and widely distributed computation of forwarding queues. When emulating a router with NIST Net high amounts of internal memory is needed when emulating high bandwidth lines with high delays. In the testbed the backbone router emulator had 3 gigabytes of internal memory. During tests NIST Net will use all the memory it demands, and if the memory is full NIST Net will not function. For our backbone emulations which use 100Mbps, the maximum link delay can be computed: Delay * 100Mb/s = 3000*8 Mb -> Delay = (3000*8Mb) / (100Mb/s) -> Maximum delay for 3 gigabytes of internal memory is = 240 seconds, so we should not get into trouble.

6.3.4 TCP flavor

For the access network we chose FACK TCP for the streaming TCP flavor. In up to date Linux installations FACK TCP is default enabled after installation. This gives us reason to believe that access networks today generally use FACK TCP. This assumption is based on the fact that the operating systems in the access networks are up to date. In our work we assume that it is so.

In addition to the streaming session TCP flavor it had to be defined what flavor to use for TCP background traffic in the backbone network. For the access network we still assume that FACK is default enabled. Even though we used a packet drop ratio instead of parallel traffic for the access network, all potential competing TCP sessions in the access network should have used FACK TCP. For the backbone the situation was more delicate. The TFRC equation used for rate control in the backbone is modeled after TCP Reno. In figure 19 we visualized TFRC throughput compared to ongoing parallel FACK TCP sessions during an emulation. The figure shows that the background TCP sessions achieve higher throughput than the TFRC session. We assume this happened because all the background TCP sessions were FACK enabled. It seems that FACK TCP tends to supress Reno modeled TFRC. In the emulations we therefore used Reno TCP as the choice of background TCP traffic. This might not be the case in the real world, but as FACK modeled TFRC is not yet created we have to adapt to that and use TCP Reno. In addition to Reno TCP sessions we generate Pareto distribution like traffic for the backbone using "tg". See the next section for deeper explanation of traffic load.

6.3.5 Background and access network traffic load

The access network should emulate a standard ADSL line between an ISP and a typical private consumer. On such a line parallel traffic is generally indeed limited. Theoretically, a consumer with ADSL connection to its ISP should have 1Mbps available for itself. Anyway ISPs tend to over scale the number of ADSL lines connected to an ISP backbone router. For example if a link to an ISP backbone router is 100Mbps, that router should in theory be connected to a maximum of 100 ADSL lines with the capacity of 1Mbps. Instead ISP's can over scale and use a number well above 100 for such a scenario. Over scaling is done because all consumers do not use all their bandwidth at all times. In our emulations we will both use no cross traffic and increasing cross traffic for the access network ADSL emulated link to gain some packet losses in the access network emulation router. To get realistic results we did not generate actual parallel traffic on the DSL emulation, but rather used static packet drop ratio in NIST Net. We assume that a consumer only watches the movie and consume no other bandwidth than the multimedia stream uses. The packet drop rate was used to emulate potential full router buffers in the ISP backbone router.

Generating realistic traffic and the right type of traffic on a backbone link is indeed challenging due to the nature of the Internet backbone [28]. The type of traffic flowing through the Internet includes among many other HTTP, FTP, TELNET, SMTP, Video and Audio. Earlier backbone measurement work documents describe the type of traffic observed and the volume of each traffic type, for example in [17]. Even though the measurements in the paper belong to the year 1998, we assume the correlation between the traffic types are somehow the same today. We note that this might not give an accurate overview of the situation in 2003, and video and audio traffic have surely increased since 1998. Later studies has shown the Internet backbone to follow a self-similar Pareto distribution pattern [3]. In the backbone we used Pareto to create heavy loads of self-similar traffic. In addition a few Reno TCP sessions are created to emulate other potential sessions between the streaming server and the access point to the Internet backbone.

We will keep the type of backbone traffic to resemble these measurements. The size of the Pareto traffic will of course be limited because we have a 100Mbps backbone, but a down-scaled traffic pattern to suite 100Mbps. It is important to emulate the type of traffic and the traffic load realistic. In the emulations, the router is not asked to drop packets randomly (like for the access network). NIST Net can be configured to drop for example 1% of all incoming packets. Instead the packet drops occur as a natural consequence of the type of traffic and the traffic load. For the load to be realistic (in a down scaled manner) the packet drop rate will vary between typical drop rates found in Internet traffic measurement work [16]. PingER (Ping End-to-end Reporting) is the name given to the Internet End-to-end Performance Measurement (IEPM) project to monitor end-to-end performance of Internet links. The project now involves hundreds of sites in many countries all over the world. Based on packet drop rates observed at the PingER web-site, we find realistic values for packet drop rates to roughly lie between 0 and 5%. We will in our emulations generate enough background traffic to make the average packet loss rate stay within these two boundaries in

```
#host1 is the alias for the traffic source
alias host1 192.168.2.2
#host2 is the alias for the traffic sink
alias host2 129.240.66.37

#HTTP EMULATION
#One UDP flow from host1 port 2000 to host 2 port 2001.
#Follow a pareto distribution.
#Use packet size of 1500 bytes and send in average 30000 packets per second
#The grade of self-similarity is reflected in the Hurst parameter.
#set to 0.5 in this test which gives an average self-similarity.
flow udp host1 2000 host2 2001 send pareto 1500 30000 600.0 32 0.5

#FTP EMULATION
#One TCP flow from host1 to host2.
#The listening port is 2008 on host2.
#Send greedy flow
#Use packet size of 1000 bytes
#send a total of 100 megabytes
#Stop anyway after 600 seconds
flow tcp host1 host2 2008 send greedy 1000 10 100000000 600.0
#Host2 is the TCP server and sets up the listening socket
flow tcp server host2 2008 recv
```

Figure 27: Tg traffic generation script example

the general case. Much higher loss peak rates may exist in different locations in the the Internet backbone. We did also do a couple of tests with higher loss rates than 5%.

We used the "tg" traffic generator to create background traffic load [15]. "Tg" was developed as a part of a package that provides an implementation of the Resource Reservation Protocol (RSVP) [33]. It can produce UDP and TCP traffic and it has to be started at all sources and sinks that are a part of the traffic generation. The traffic is controlled via a script that is executed equally on all computers. Figure 27 shows an example script for traffic generation that is executed at both the source and the sink.

It can produce constant bit rate (CBR) traffic, variable bit rate (VBR) traffic, greedy traffic and Pareto distribution modeled traffic. "Tg" is especially neat for our emulations as we needed to emulate FTP traffic (greedy) and HTTP like ON/OFF traffic (Pareto). A large effort has been invested in creating accurate models for web-traffic like HTTP ON/OFF traffic. Earlier the web-traffic was assumed to follow a Poisson process like distribution, but later studies tend to show that web-traffic have self-similar and long-tailed distribution like behavior which fit well with the Pareto distribution implementation in "tg" [3].

6.4 Structure of the emulation presentations

The next three subsections present the emulation results. The emulations are categorized based on the amount of background traffic in the backbone. We start off with no or minimum background traffic in the first subsection while in the second and third subsection the total traffic is continuously increasing. For the different emulations we vary round trip times on both links and the amount of cross traffic in the access network. We present throughput, jitter and jitter statistic graphs for all emulations. In addition we show how the TCP application queue behaves during the emulations.

Non-blocking TCP gives us the ability to make the TCP socket send buffer to grow in the application queue. When the TCP send buffer is full when we try to send, the rest of the unsent data is placed on the application queue (MNSocketQueue).

Figure 28 presents a matrix that shows the combinations used for the emulations. All emulations have their separate subsections from this point. For the first emulation we show graphs visualizing different aspects of the stream. As we proceed with the emulations we find it unnecessary to present all the graphs for all the emulations. For each emulation we select graphs that show interesting behavior. The values presented in graphs are throughput, jitter, jitter sample, statistics and the size of the TCP application queue. In figure 51 we present the result matrix that resembles the matrix found in figure 28.

In 6.8 we conduct some blocking TCP experiments. In these tests the application queue is not applicable. It is interesting to see how the system behaves during blocking TCP. When TCP blocks the send-call, the proxy will lock (it is working in one thread at the moment) and not be able to receive UDP packets from the server. When UDP packets are dropped in the proxy, TFRC in the backbone will decrease as the loss event rate increases. We wanted to investigate if blocking TCP in the proxy could give us a smart way to decrease the backbone TFRC rate when TCP back pressure occurs.

6.5 No cross traffic in the backbone

6.5.1 No cross traffic in backbone, 0% packet drop in access network

First we show how the streaming environment behaves when both the backbone and the access network lies idle. During this emulation the packet drop is constantly 0%. This might not be too realistic most of the time (especially for the backbone), but periods occur when one can experience no packet loss on a UDP based Internet connection both with a QOS agreement and also with best effort traffic. This emulation shows absolutely best case scenario. The video was streamed at a rate of 1 Mbps and the round trip time between server and proxy was set to 200 milliseconds. The round trip time between the proxy and the server was set to 10 milliseconds. The router queues followed as described the BDP. Note that the BDP for the access network emulation would give a to small router buffer (1 packet assuming IP-packet size of 1500 bytes). We set a limit to the router buffer to 10 packets of size 1500 bytes, which is the well

AN / BB	0% packet drop	2% packet drop	5% packet drop
No traffic	Bandw. BB : 100 Mbps RTT BB : 200 ms Bandw. AN : 1 Mbps RTT AN : 10 ms	Bandw. BB : 100 Mbps RTT BB : 400 ms Bandw. AN : 1 Mbps RTT AN : 13 ms	Bandw. BB : 100 Mbps RTT BB : 200 ms Bandw. AN : 1 Mbps RTT AN : 10 ms
Medium traffic	Bandw. BB : 100 Mbps RTT BB : 50 ms Bandw. AN : 1 Mbps RTT AN : 7 ms	Bandw. BB : 100 Mbps RTT BB : 200 ms Bandw. AN : 1 Mbps RTT AN : 15 ms	Bandw. BB : 100 Mbps RTT BB : 100 ms Bandw. AN : 1 Mbps RTT AN : 15 ms
High traffic	Bandw. BB : 100 Mbps RTT BB : 300 ms Bandw. AN : 1 Mbps RTT AN : 5 ms	Bandw. BB : 100 Mbps RTT BB : 300 ms Bandw. AN : 1 Mbps RTT AN : 20 ms	Bandw. BB : 100 Mbps RTT BB : 50 ms Bandw. AN : 1 Mbps RTT AN : 12 ms

Figure 28: Emulation matrix - Non blocking TCP

known default maximum transfer unit (MTU).

Figure 29 shows the development of throughput on both links over a 60 seconds streaming session. The graph shows that the proxy-client throughput closely follows the server-proxy throughput. It takes about 9 seconds for the TFRC rate to reach a stable sending rate, which in this case is limited not by the network but by the application restricted throughput of 1024 kilobits/second. The stable sending rate lies stable at around 91000 bytes. One would in the case where the network can deliver 1 Mbps assume that TFRC would choose a higher stable sending rate, but the implementation in Komssys incorporates a maximum bitrate that can be set to tell when TFRC should cease throughput increase. In our test this variable was set to limit the bitrate to what can be seen in figure 29. A more interesting phenomena is how the access network succeeds in delivering the data. In this case (since no packets are lost and the access network can deliver up to 1Mbps), TCP throughput is not limited by the network, but rather by the application. We wondered how TCP would react to such a situation as TCP is usually designed for bulk data transfers. We suspected it to reduce its congestion window when the congestion window was not fully utilized. What happens is that since TCP does not experience packet loss, all acknowledgements from the receiver contributes on expanding the congestion window. Assuming that the both the sender and receiver has a 64 kilobytes TCP socket buffer, the congestion window will expand up to the maximum 64KB in a short period of time. Because of the application limited bitrate, the congestion window will not nearby be fully utilized. Therefore all data sent to the TCP socket buffer will be sent nearby immediately. This suits well for our work, as TCP does not reduce its congestion window even when the send rate is application limited. We also note that this situation can make (also during packet loss) the congestion window grow beyond the capabilities of the network. Even though the connection can deliver the application defined bitrate, the congestion window grows out of proportions. A sudden burst of 64KB of data to the TCP socket buffer could create unfairness in terms of other TCP sessions as the congestion window is not adapted to the limitations of the network. This situation has been discovered and a suggested solution is

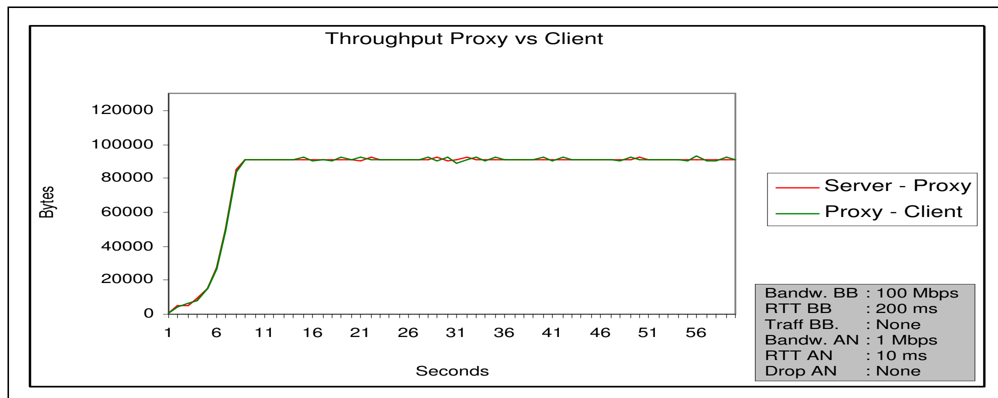

Figure 29: Throughput: No traffic in backbone and no packet drop in access network

proposed in this RFC [8]. When TCP is application limited (the congestion window is not fully utilized), the congestion window is suggested halved after a fixed amount of time. Obviously this RFC is not implemented in our FACK TCP implementation.

Figure 30 presents experienced jitter between server/client, server/proxy and proxy/client. The graphs shows the amount of time passed from the server pushed a layer packet to the receiver unpacked it. The server stamps packets with a fairly constant time interval, and this graph visualizes the delay variations for all layers. Figure 31 shows a closeup jitter sample of 100 continuous received layers. The blue line plots the delay between the proxy and the client for all layers received. The green line represents the delay between the server and the proxy while the red line represents the total delay for all layers from they are sent from the server to they are conveyed at the receiver. As we can observe delay variations are minor, but some small peaks appears regularly. These peaks are a result of router buffering and application jitter. The average delay between the server and the proxy lies just below 200 milliseconds, while for the proxy-client situation the average lies at about 35 milliseconds. As mentioned the RTT's were set to 200ms and 10ms respectively, so the averages are not very surprising. The reason that server-proxy delay average lies below 200 milliseconds is because of clock synchronization issues. We note that the jitter on the TCP connection is quite low when we compare it to the UDP based connection. This is off course based on the fact that no packet drops were detected and that TCP behaves like it does during application limited bit rates. For access networks with no cross traffic on the private ADSL consumer line, our architecture seems like a reasonable approach for improved HTTP streaming.

Figure 32 shows the data more statistically correct. We show the average delay, the 5/95 percentile and the standard deviation around the average delay. The right graph is just a zoom to see the dots better. We note that the 95 percentile is located about 10 milliseconds above the average delay. The spread is quit limited. This is as expected based on that no packet drops occurred,

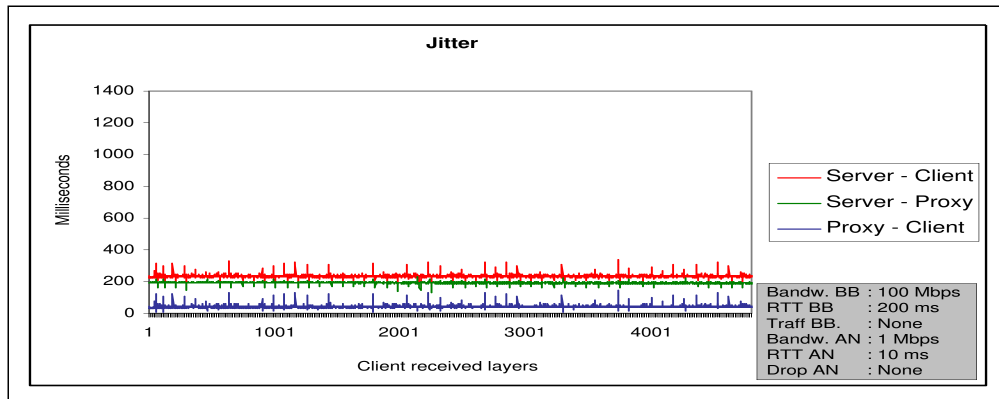

Figure 30: Jitter: No traffic in backbone and no packet drop in access network

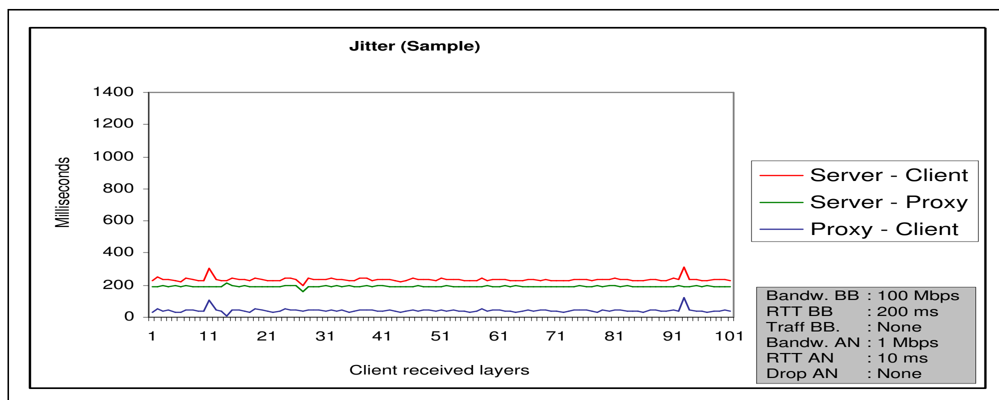

Figure 31: Jitter sample: No traffic in backbone and no packet drop in access network

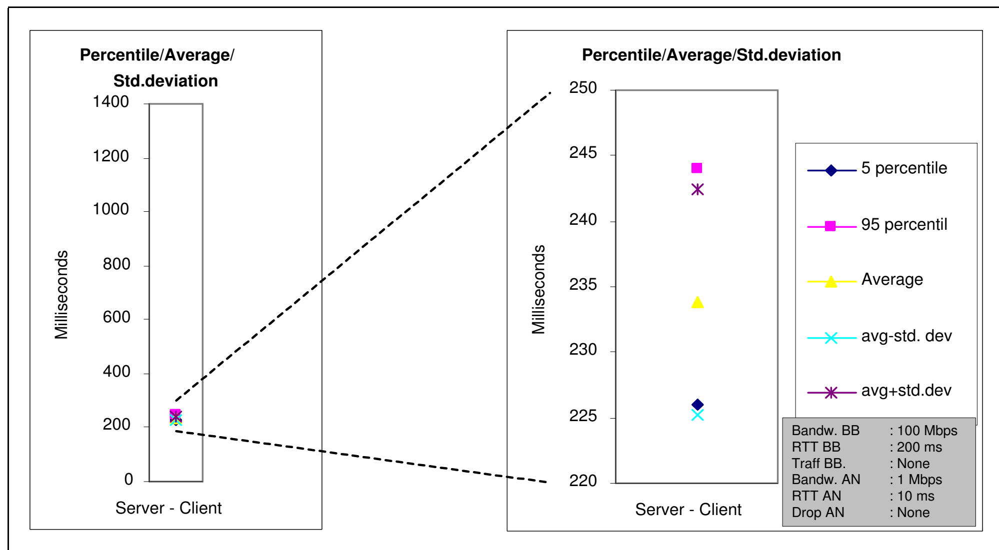

Figure 32: Statistics: No traffic in backbone and no packet drop in access network

small processing time in routers and the fact that TCP behaves like it does under application limited bit rates.

6.5.2 No cross traffic in backbone, 2% packet drop in access network

In this section we test the streaming with medium cross traffic in the access network by dropping 2% of the packets. Figure 33 presents the results, which shows that the 95 percentile lies just above 550 milliseconds.

Figure 34 shows that the TCP application queue experiences a few peaks. This back pressure does not contribute to much instability as shown by the jitter statistics.

6.5.3 No cross traffic in backbone, 5% packet drop in access network

This setup really puts the access network to the test. The bitrate in the backbone is maximum of what TFRC will deliver and the access network experiences high packet loss rates. When setting a constant 5% packet drop rate in NIST Net we do not achieve the Poisson process like traffic that might occur in access networks, but we show how TCP behaves during high average packet drop rates. A 5% packet drop rate can be used to simulate pretty constant high cross traffic in the access network. Figure 35 presents the throughput graphs of backbone and the access network respectively. TFRC lies constantly on just above 90 kilobytes/second as in the last test after a 11 second long climb to the peak rate. TCP shows the well known zig-zag pattern centered around the TFRC backbone rate. But after a down period, TCP climbs fast and exceeds the TFRC bitrate and again drops below it. When TCP lags behind, it quickly

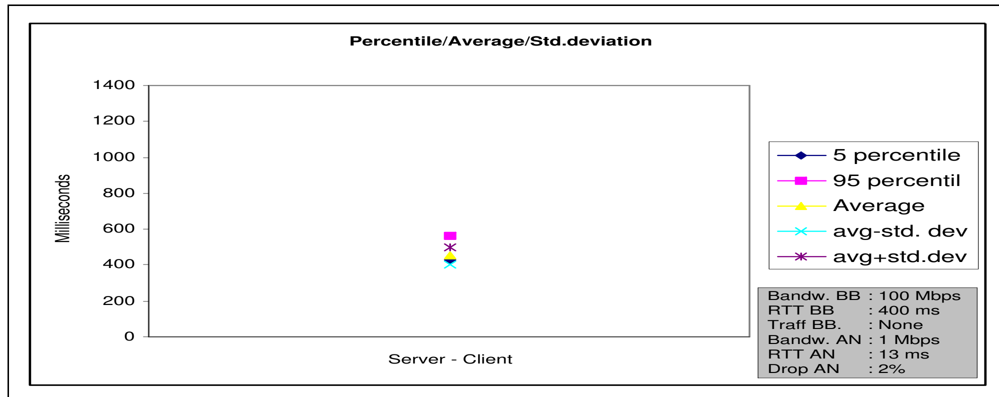

Figure 33: Statistics: No traffic in backbone and 2% packet drop in access network

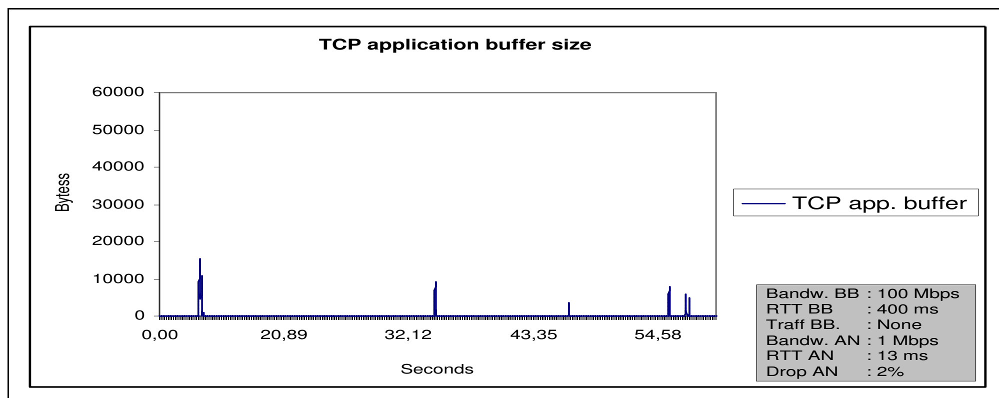

Figure 34: TCP application buffer size: No traffic in backbone and 2% packet drop in access network

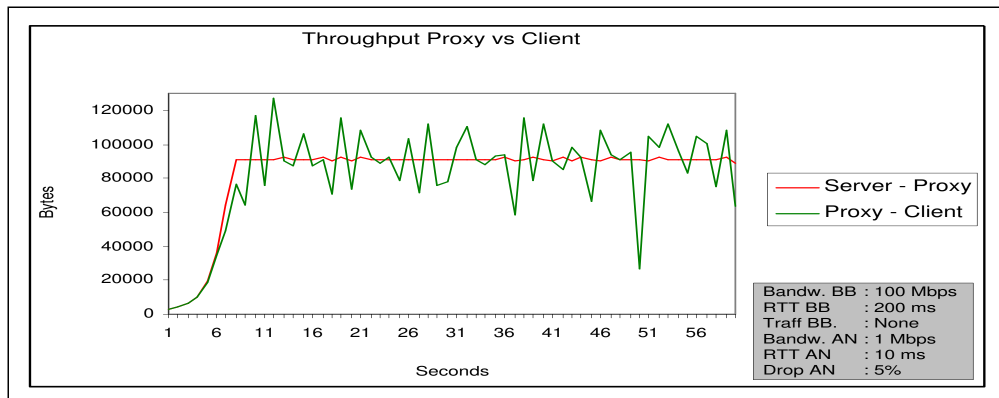

Figure 35: Throughput: No traffic in backbone and 5% packet drop in access network

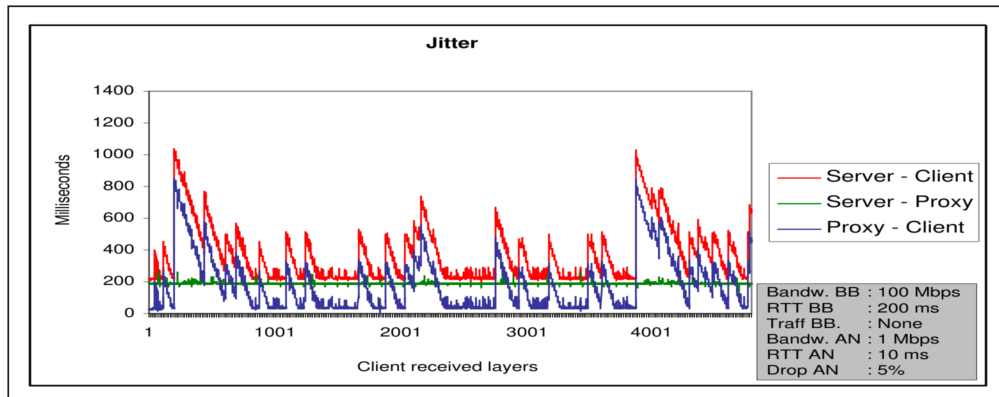

Figure 36: Jitter: No traffic in backbone and 5% packet drop in access network

thereafter transmits the delayed data.

The jitter between the server and proxy is naturally just about non-existent. The blue graph shows why TCP is stated to not suit very well for streaming applications (interactive applications in particular). TCP contributes much to the server-client delay variations. The peak delay variation is just above 1000 milliseconds. Figure 36 show the jitter values for all layers received at the client.

Figure 37 shows the statistical values for the emulation. The average delay lies just below 400 milliseconds. The 95 percentile lies at about 750 milliseconds which is 350 milliseconds from the average point. The 95 percentile actually shows that 95% of all layers are received within a 750 millisecond delay. As we wrote, this emulation is particularly challenging for the access network and the delay variations observed are not intimidating. If this emulation was based on a real world scenario, the client could buffer 1 second of the movie for quit successful presentation. We note that high packet rate losses or longer periods of

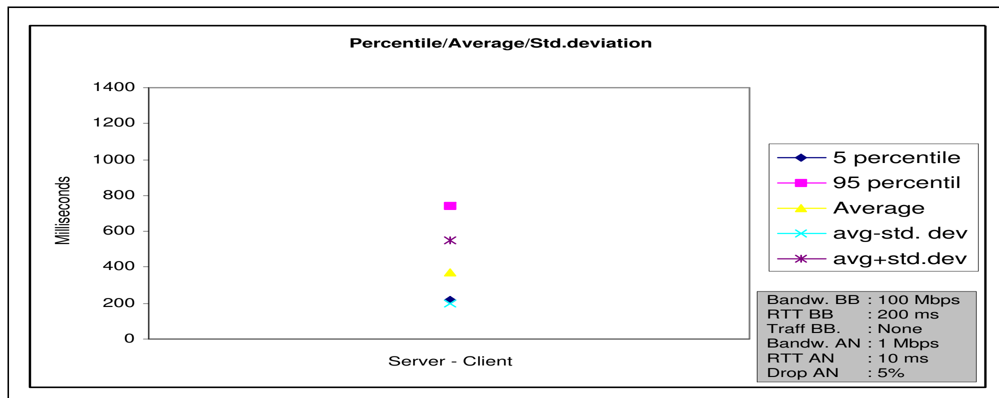

Figure 37: Statistics: No traffic in backbone and 5% packet drop in access network

decreased bandwidth on the access network also will demand quality adaption actions in the proxy. Chapter 7 deals with potential tuning options, including suggestions for quality adaption in the proxy.

Figure 38 shows how our implemented application queue behaved during this test. This application queue was documented in section 4.3.2. The y axis shows how many bytes sent are stored in the queue. The x-axis shows the time elapsed. The graph corresponds closely to the graph in figure 36. When TCP throughput is limited, the application queue grows. At the very most the queue rises to 60 kilobytes. These 60 kilobytes are in addition to the full TCP socket send buffer of 64 kilobytes , giving a total of about 124 kilobytes. As shown, TFRC delivers about 90 kilobytes pr second giving just above 1000 milliseconds of playback time residing in the application buffer. This value corresponds roughly to the peak jitter value in figure 36.

We believe this emulation indicates that TCP is suited for HTTP-streaming in the access network. TCP presents delay variations and varying throughput, but because of access network properties a very small amount of buffering can guarantee jitter free presentations. One of the main objections to use TCP for streaming applications is its reliable behavior with packet retransmissions (And congestion avoidance behavior). Retransmissions introduce jitter and prevents the progress in the transfer. Jitter increases with higher round trip times. In access networks the RTT is very limited and we have tried to show that TCP connections with small RTTs do not introduce very high delay variations. If the access network is heavily congested and the TCP throughput is substantially reduced over a long period the application queue can be exploited to reduce delay variations. The tuning options are further described in chapter 7.

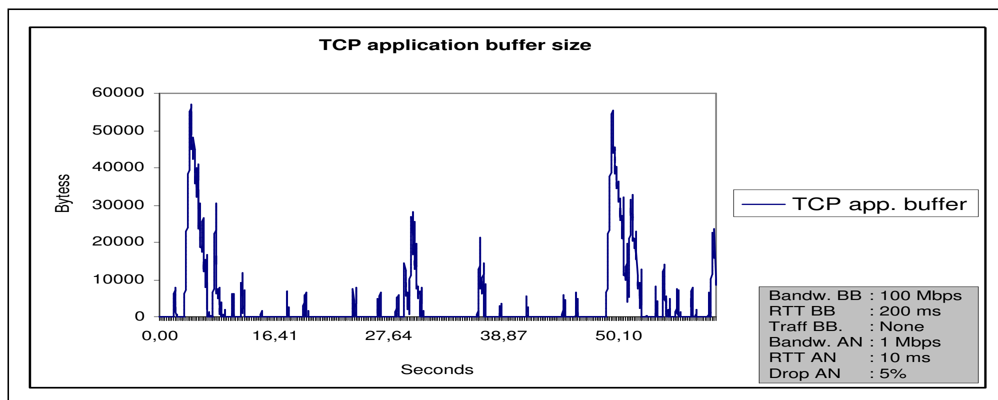

Figure 38: TCP application buffer size: No traffic in backbone and 5% packet drop in access network

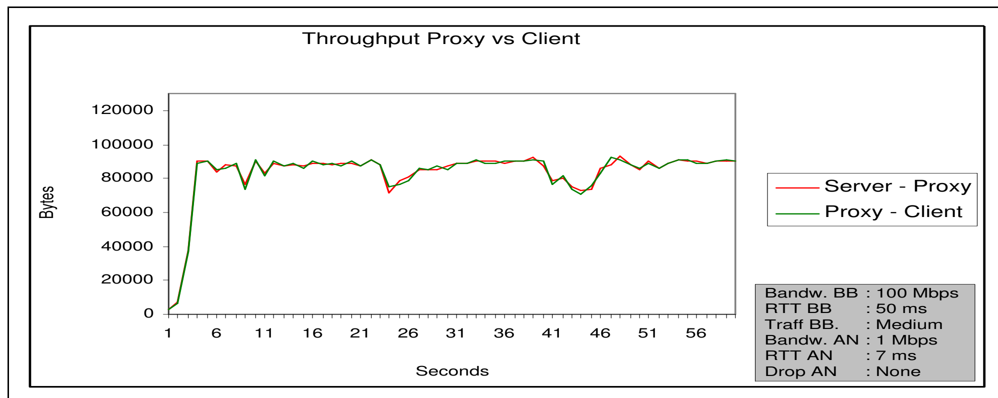

Figure 39: Throughput: Medium traffic in backbone and no packet drop in access network

6.6 Medium cross traffic on the backbone link

6.6.1 Medium cross traffic in backbone, 0% packet drop in access network

In this emulation we generated traffic in the backbone. The traffic was based on 1 large Pareto distribution with a Hurst parameter of 0.5 giving average self-similarity. Two TCP sessions did also compete for bandwidth. We observed that 2.8% of all packets in our dummylayerstream sent from the server never reached the proxy. In figure 39 we can see how TFRC degrades throughput on the backbone link. As expected will TCP adapt immediately to any changes. The access network does not experience any problems. For statistics about the jitter we refer to figure 51.

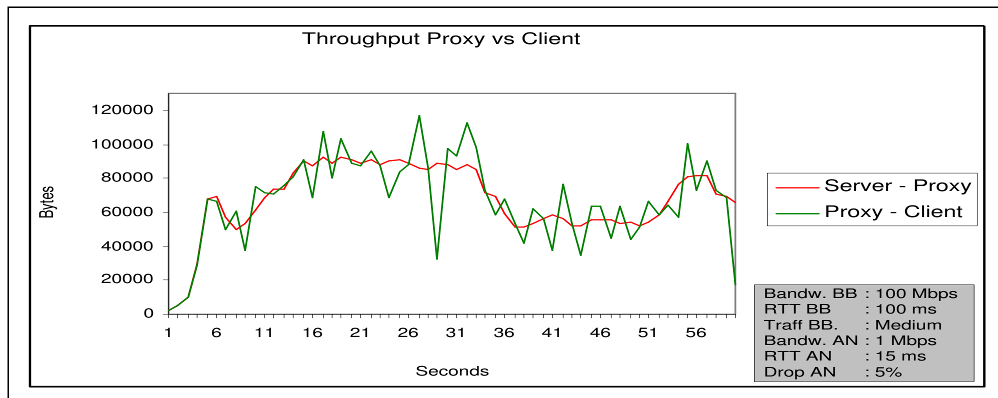

Figure 40: Throughput: Medium traffic in backbone and 5% packet drop in access network

6.6.2 Medium cross traffic in backbone, 2% packet drop in access network

In this emulation we generated traffic in the backbone. The traffic was based on 1 large Pareto distribution with a Hurst parameter of 0.5 which gives average self-similarity. Two TCP sessions did also compete for bandwidth. We observed that 1.6% of all packets in our dummylayerstream sent from the server never reached the proxy. For statistics about the jitter we refer to figure 51.

6.6.3 Medium cross traffic in backbone, 5% packet drop in access network

This test shows that the access network throughput varies quite a bit from second to second. Throughput correlation between the two environments is shown in figure 40. The 95 percentile delay stays at 577 milliseconds which is quite good for so much emulated traffic. Jitter statistics are presented in figure 51. The average drop rate in the backbone is 3.3% for the dummylayerstream.

6.7 High cross traffic on the backbone link

6.7.1 High cross traffic in backbone, 0% packet drop in access network

The Pareto traffic in the backbone was very aggressive and a total of 11.2% of all dummylayerstream packets were dropped at the router. In figure 41 it is shown what happens with the throughput. In the start of the streaming it is still space in the router buffer, as the Pareto traffic seems not to fill it up completely. At about 11-12 seconds after the start the TFRC backbone rate has increased to a rate where the router buffer goes full. Large amount of drops occurs and the rate is drastically decreased and sustained fairly stable at just a few kilobytes pr. second in the beginning, but increases some in the end of the stream. The TCP connection has as expected no problems forwarding all

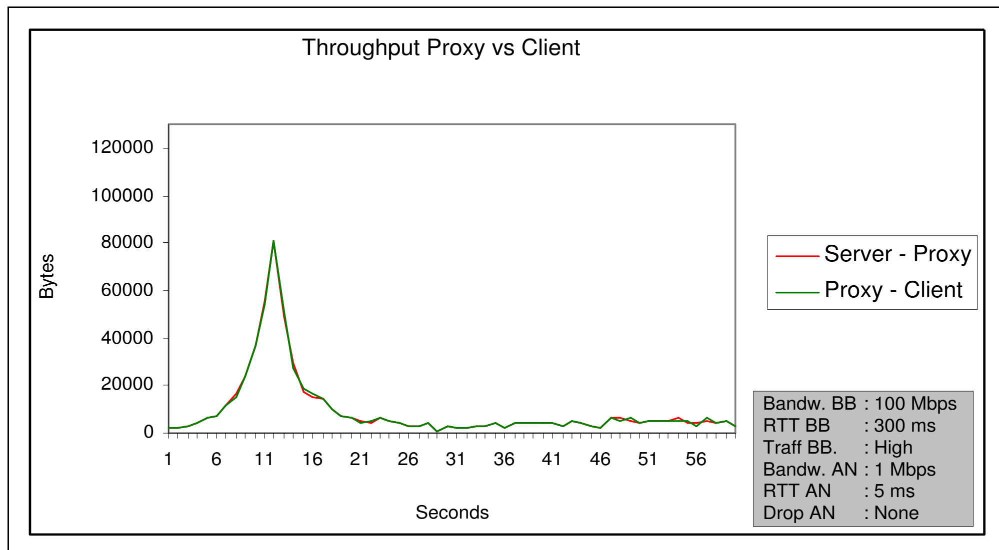

Figure 41: Throughput: High traffic in backbone and 0% packet drop in access network

packets immediately.

6.7.2 High cross traffic in backbone, 2% packet drop in access network

In this test the backbone dropped 4.6% of all packets in the stream. The throughput of this test shows a very nice and smooth throughput curve. The backbone dropped packets from the TFRC flow early in the presentation so that TFRC established the right, smooth sending rate quite early. The small delay variations occur at the client site during packet drops at the TCP connection.

6.7.3 High cross traffic in backbone, 5% packet drop in access network

In figure 42 we can see that the TFRC rate is varying quite a bit during the test. 3.4% of all packets were dropped in average. The Pareto traffic and the two TCP sessions create varying traffic which is reflected in this graph. The TCP access network throughput has the normal zig-zag behavior around the TFRC throughput. In figure 43 we observe that the back pressure is smaller than the back pressure in our third test where 5% packet drop in access network and no cross traffic in the backbone was used. Figure 51 shows the jitter statistics.

6.8 Blocking TCP

In this section we present results for non blocking TCP tests. Figure 44 presents the three initial tests conducted. The three emulations are described in the

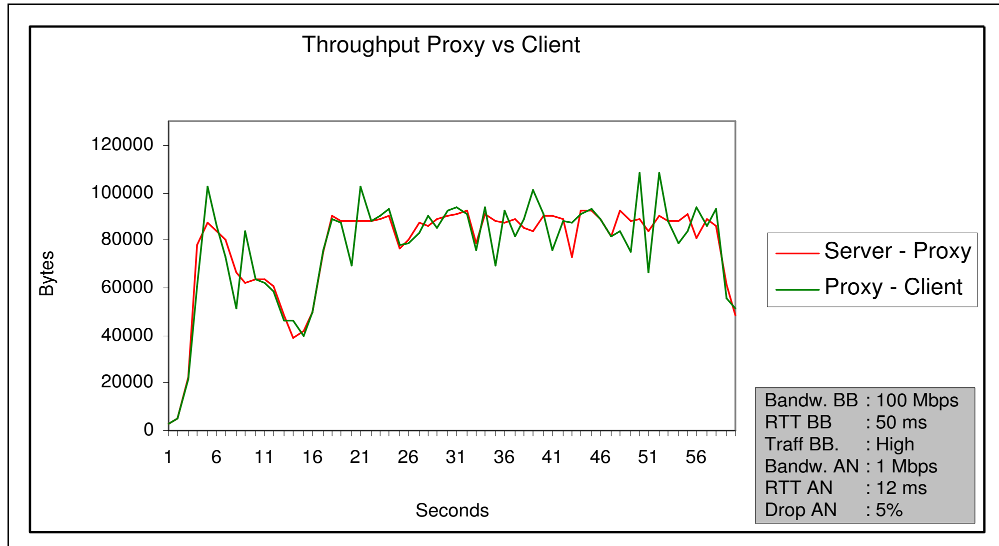

Figure 42: Throughput: High traffic in backbone and 5% packet drop in access network

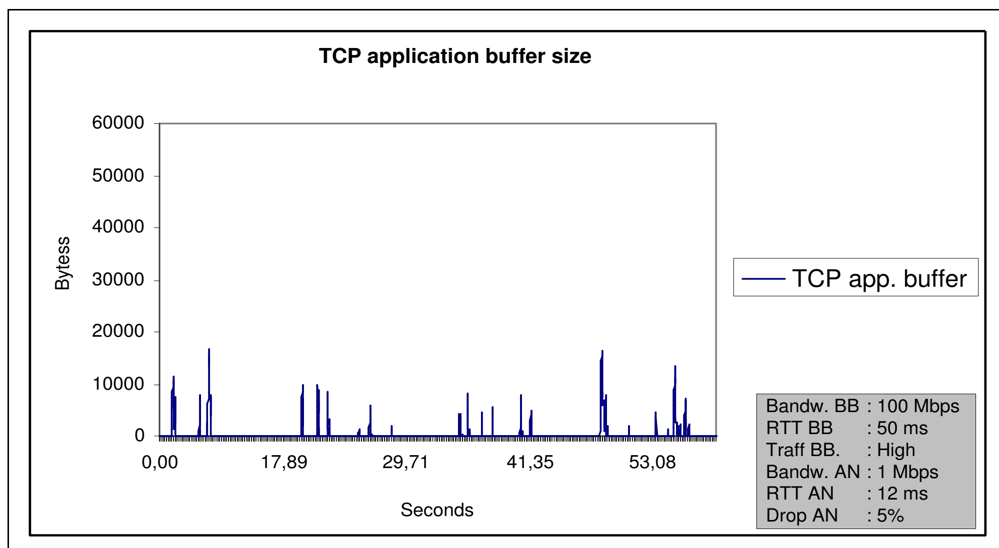

Figure 43: TCP application buffer size: High traffic in backbone and 5% packet drop in access network

AN / BB	Blocking TCP
No traffic	Bandw. BB : 100 Mbps RTT BB : 100 ms Bandw. AN : 512 Kbps RTT AN : 8 ms
Medium traffic	Bandw. BB : 100 Mbps RTT BB : 100 ms Bandw. AN : 256 Kbps RTT AN : 15 ms
High traffic	Bandw. BB : 100 Mbps RTT BB : 50 ms Bandw. AN : 128 Kbps RTT AN : 12 ms

Figure 44: Emulation matrix - Blocking TCP

following three subsections. In section 6.8.4 we take a deeper look into blocking TCP in order to explain the behavior of the experiments.

6.8.1 No cross traffic in backbone, blocking TCP in access network

With blocking TCP the application buffer is not used. Whenever the TCP socket send buffer is full, a write to the socket will block the thread it was called from. The proxy solution in Komssys does at present run in one thread. This implementation results in that the RTP packets received at the proxy from the server during a block will be lost. When these RTP packets are lost, TFRC will compute a higher loss event rate and reduce its sending rate. We wanted to investigate if blocking TCP would give an easy way to control the backbone rate to adapt to the access network rate whenever the access network is the bottleneck. In this test we set the bandwidth in the access network to 512 Kbps. The multimedia stream is still encoded and sent at 1 Mbps and the backbone is capable of delivering at just above 90Kbps as documented earlier. Figure 45 shows that regular blocking TCP can be used to control the backbone bandwidth. TFRC reports high loss event rate during blocking periods and the backbone bandwidth regulates the bandwidth down to the access network bandwidth. We can see that the server-proxy bandwidth has a peak at just above 80Kbps. During this period, the TCP socket send buffer fills up during the 6 first seconds. It then blocks, and the proxy is unable to receive packets from the backbone. The TFRC equation regulates bandwidth to match the access network bandwidth. The average delay lies at 799 milliseconds as shown in figure 51. For a packet that is placed in the back of a full TCP send queue it is theoretically estimated that it will reside in the queue for about 1000 ms. The queue is 64 kilobytes large and the bandwidth is 64 kilobytes pr. second. The peak delay lies at 1349 ms. During the emulation 1.6% of the UDP packets were lost only because of blocking TCP. The behavior in the graph is further analyzed in section 6.8 where it is described what happens when TCP blocks in the proxy.

The 95 percentile lies at just above 1000 milliseconds as shown in figure 46, which is not intimidating. We do not show the jitter graph in this section, but

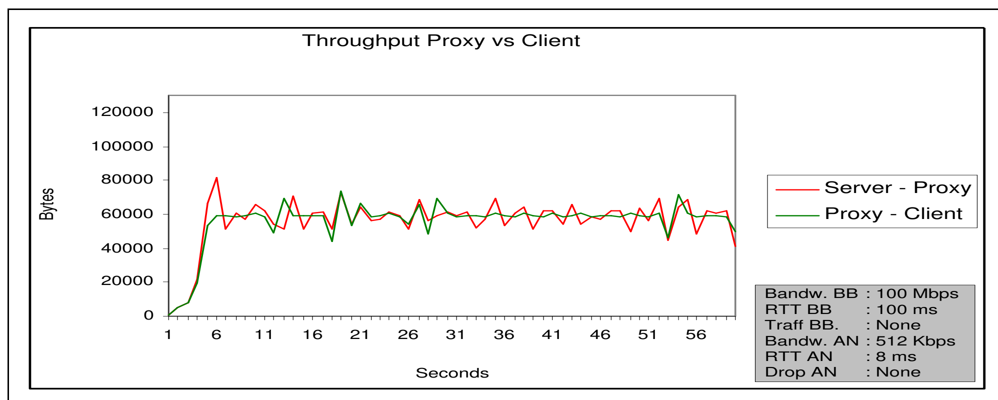

Figure 45: Throughput: No traffic in backbone and blocking TCP in access network

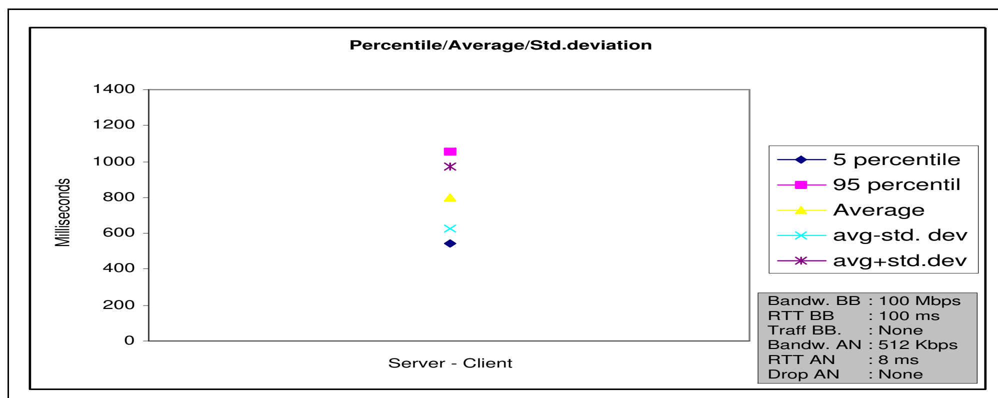

Figure 46: Statistics: No traffic in backbone and blocking TCP in access network

the results are that the server-proxy delay variations are not nearly as stable as in the earlier emulations. Both server-proxy and proxy-client jitters are quite similar.

6.8.2 Medium cross traffic in backbone, blocking TCP in access network

The behavior looks like the previous blocking TCP test. The difference is that in this test 5.4% of the packets were dropped in the backbone. This number results from Pareto parallel traffic in addition to packets dropped in proxy because the TCP buffer was full. Therefore the back pressure on the 256 kilobit TCP link is not as high as the previous test, and the delays lie just above the results found in the previous blocking test. We do not distinguish between packets lost due to traffic and packets lost due to blocking. In this test it could be interesting to see how many of the 5.4% dropped packets that happened because of blocking.

AN \ BB	0% packet drop	2% packet drop	5% packet drop
No traffic	Bandw. BB : 100 Mbps RTT BB : 200 ms Bandw. AN : 512 Kbps RTT AN : 10 ms	Bandw. BB : 100 Mbps RTT BB : 200 ms Bandw. AN : 512 Kbps RTT AN : 10 ms	Bandw. BB : 100 Mbps RTT BB : 200 ms Bandw. AN : 512 Kbps RTT AN : 10 ms
No traffic	Bandw. BB : 100 Mbps RTT BB : 200 ms Bandw. AN : 128 Kbps RTT AN : 10 ms	Bandw. BB : 100 Mbps RTT BB : 200 ms Bandw. AN : 128 Kbps RTT AN : 10 ms	Bandw. BB : 100 Mbps RTT BB : 200 ms Bandw. AN : 128 Kbps RTT AN : 10 ms

Figure 47: Blocking TCP - deeper look emulation matrix

That number can again tell us something about the amount of back pressure.

6.8.3 High cross traffic in backbone, blocking TCP in access network

In combination with high backbone traffic many packets were also dropped in the proxy because of the bottleneck bandwidth of 128 kilobytes per second. A total of 21.5% of the dummylayer packets are dropped, and we suspect more than half of this to be drops due to blocking in proxy. As the statistics show, a high delay is achieved which witnesses of many periods with full socket buffer. A full socket buffer on a TCP connection with a 128 kilobits per second line means that a packet entering the queue after a block can theoretically stay in the queue for (64/(128/8)) 4 seconds.

6.8.4 Blocking TCP results - a deeper look

The blocking tests summarized in the table below in figure 51 with statistics in figure 53 shows that when bandwidth capability is degraded on the access link the delay is getting very substantial. This is as expected because packets will stay in the send buffer for a long time in addition to delay at the router emulator. Figure 45 presented a throughput pr. second graph of a blocking test. It is evident that the TFRC backbone rate has to decrease to access network bottleneck link. It does not adapt smoothly but in fact introduces a very variable sending rate. Figure 48 shows the problem even better. It is a throughput graph of the test that resembles the upper square in figure 44. The line with the name "Dropped" visualizes what is sent from the server minus what is received at the proxy pr. second. We now try to explain this behavior.

The 9 first seconds the backbone does not feel the bottleneck bandwidth on the client link. For about the 9 first seconds the TCP send buffer is filling up in the proxy. At about 9 seconds the proxy blocks after calling write on a socket with a full send buffer. By looking at the throughput difference between server and proxy it is evident that a large amount of packets are dropped for each time the proxy blocks. After this blocking TFRC reduces throughput very fast

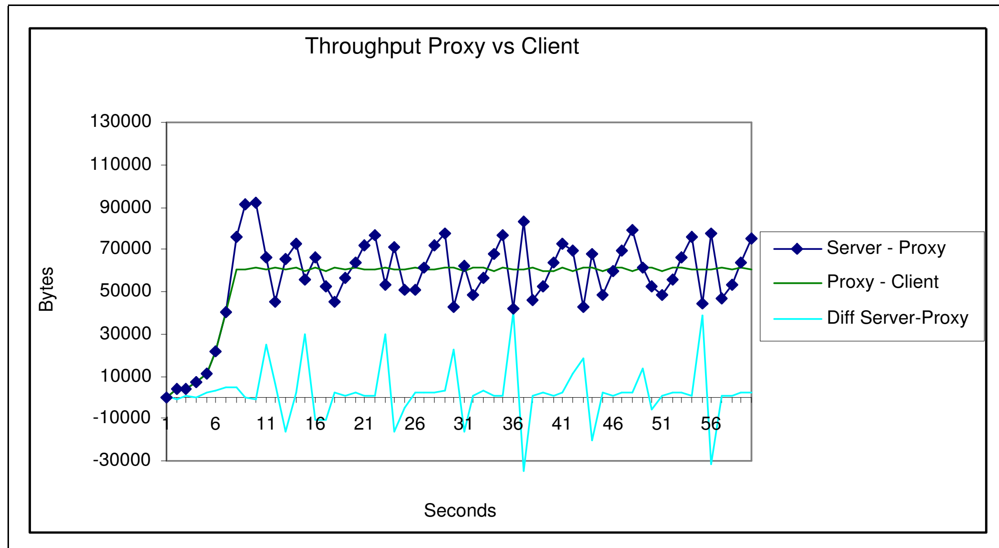

Figure 48: Throughput: 512Kbps bandwidth and 0% packet drop in access network with no traffic in backbone

due to these losses. When the proxy blocks the TFRC sender does not receive feedback reports from the receiver. The retransmission timeout is triggered and the sender halves the sending rate according to the standard. As no more losses occur it increases rapidly again. This creates unstable sending rate between the server and the proxy.

Anyway it seems like the proxy blocks for a long period. By looking at the TCP implementation in the Linux distribution in the testbed the reason for this behavior shows. In figure 49 the functions used for deciding when to stop blocking is presented.

in `tcp_poll` the test for stopping blocking is:
`tcp_wspace(sk) >= tcp_min_write_space(sk)`. If we resolve these functions the test concludes to: `sk->sendbuf - sk->wmem_queued >= sk->wmem_queued/2`. If we solve this equation with respect to `sk->wmem_queued` we get: `sk->wmem_queued <= 2/3 sk->sendbuf`. The bottom line is that whenever TCP blocks because of a full send buffer it does not write and return data until at least 1/3 of the total send buffer is freed.

When a block is started the send buffer is full with 64 kilobytes of data. It will block until TCP has been able to send just above 21 kilobytes. During this period all packets received at the proxy by the backbone is lost. This is the reason why we achieve the zig-zag behavior in backbone throughput as shown in figure 48. TFRC experiences sequential losses in bulks followed by no losses at all. The TFRC rate in figure 48 could be smoothed out by tweaking the TCP implementation to block for a shorter period. By more rapid blocks for shorter periods the TFRC would experience more average drop rates and give smaller

tcp.h

```
static inline int tcp_min_write_space(struct sock *sk)
{
        return sk->wmem_queued/2;
}

static inline int tcp_wspace(struct sock *sk)
{
        return sk->sndbuf - sk->wmem_queued;
}
```

Function tcp_poll(3) in tcp.c

```
if (!(sk->shutdown & SEND_SHUTDOWN)) {
        if (tcp_wspace(sk) >= tcp_min_write_space(sk)) {
                mask |= POLLOUT | POLLWRNORM;
        } else {  /* send SIGIO later */
                set_bit(SOCK_ASYNC_NOSPACE, &sk->socket->flags);
                set_bit(SOCK_NOSPACE, &sk->socket->flags);

                /* Race breaker. If space is freed after
                 * wspace test but before the flags are set,
                 * IO signal will be lost.
                 */
                if (tcp_wspace(sk) >= tcp_min_write_space(sk))
                        mask |= POLLOUT | POLLWRNORM;
        }
}
```

Figure 49: TCP blocking implementation

quality jitter. It had to be figured out what would be the optimal relation between context switch overhead and blocking periods.

In figure 53 the statistics for these extra blocking experiments are presented. It is evident that the delay variations and the average delay does decrease as we introduce packet drop in the access network. In figure 50 jitter graphs for all six blocking experiments are presented. The y-axis represents delay in milliseconds with a maximum of 8500ms. The x-axis represents sequence numbers for layers. In a normal scenario it would be natural that delay and delay variations increased with higher loss rates. By studying the 0% packet drop rate graphs in figure 50 it can be seen that (because of the way NIST Net works) the delay increases in the beginning. This is because Nist Net queues all packets and pulls them out of the queue at a rate of respectively 128kbps and 512kbps. We did not specify any queue size for these emulations but merely restricted the bandwidth and entered the static packet drop rate. So when TCP tries to deliver just above 90 kilobytes pr. second the queue in NIST Net increases rapidly. This large queue introduces increasing round trip times. TCP reduces throughput because of these large round trip times (ACKs don't come back very fast). After a while the TCP send buffer stacks up and the proxy blocks. The zig-zag behavior coming next is a visualization of what happens when the proxy blocks. When it blocks the send buffer has time to drain some. When it unblocks again the next few frames introduce smaller round trip times. The queuing ramps up again and the proxy blocks. Now when we introduce packet drops in the access network the NIST Net queue does not increase that much as TCP reduces it's send rate to half for every packet drop. This is why the delay decreases as packet drop rates are introduced.

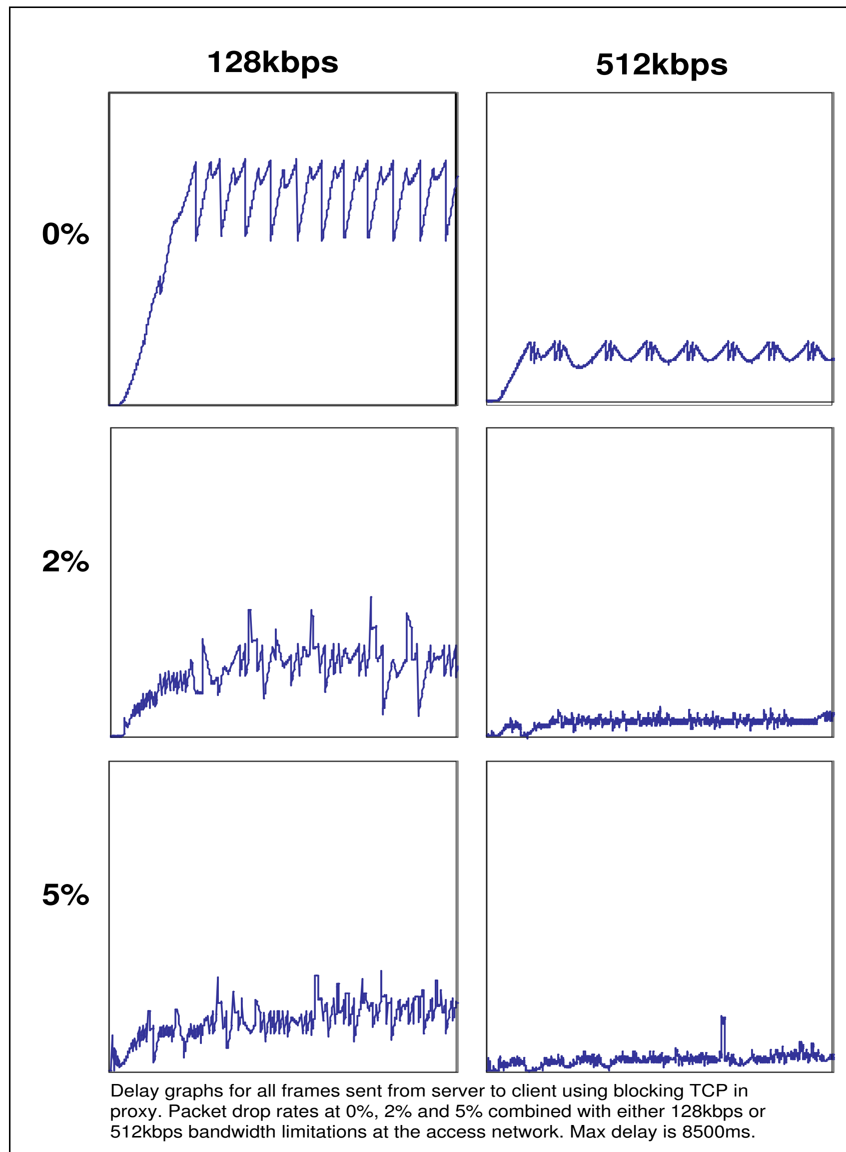

Figure 50: Delay variations for extra blocking tests

6.9 Result matrix - No blocking and blocking TCP tests

The result matrix for all presented emulation matrixes is presented in figure 51. **Figure 52 presents statistics summary for for non-blocking TCP. These statistics are not based on the emulations just described as we varied round trip times and other parameters during these tests. The statistics are based on a repeated emulation like done for blocking tests but all parameters were set to the same values to be able to analyze and compare the different emulations**. For the statistics in figure 52 the backbone RTT was set to 200ms and the access network RTT was set to 10ms. The backbone used a bandwidth of 100Mbps and the access network used 1Mbps. Figure 53 presents statistics for the extra blocking tests (Corresponds to figure 47). As the extra blocking test did use the same parameters for all emulations the statistics are based on the extra blocking tests recently described. For non-blocking tests it is evident that the delay variations increase as the packet drop rates in the access network increases. As traffic is introduced on the backbone link, the delay variations are in general smaller due to lower backpressure in the TCP socket queue. For blocking TCP the delay variations decrease as we introduce higher packet drop rates in the access network. When the bandwidth on the blocking TCP connection decreases the delay variations naturally increases.

AN / BB	0% packet drop	2% packet drop	5% packet drop	Blocking TCP
No traffic	Average : 233.8 5 perc. : 226.0 95 perc. : 244.0 Std.dev. : 8.6 Peak high : 336.0	Average : 452.2 5 perc. : 426.0 95 perc. : 565.0 Std.dev. : 46.4 Peak high : 719.0	Average : 372.6 5 perc. : 217.7 95 perc. : 744.0 Std.dev. : 174.4 Peak high : 1036.0	Average : 799.0 5 perc. : 545.0 95 perc. : 1052.0 Std.dev. : 172.5 Peak high : 1349.0
Medium traffic	Average : 80.4 5 perc. : 72.0 95 perc. : 92.0 Std.dev. : 8.5 Peak high : 180.0	Average : 240.6 5 perc. : 219.0 95 perc. : 314.0 Std.dev. : 34.9 Peak high : 496.0	Average : 246.0 5 perc. : 127.8 95 perc. : 577.25 Std.dev. : 153.3 Peak high : 950.0	Average : 683.6 5 perc. : 141.1 95 perc. : 1156.0 Std.dev. : 294.8 Peak high : 1444.0
High traffic	Average : 324.6 5 perc. : 315.0 95 perc. : 340.4 Std.dev. : 9.6 Peak high : 408.0	Average : 346.1 5 perc. : 331.0 95 perc. : 422.0 Std.dev. : 42.3 Peak high : 593.0	Average : 144.1 5 perc. : 66.0 95 perc. : 322.0 Std.dev. : 90.6 Peak high : 564.0	Average : 3439.7 5 perc. : 2130.7 95 perc. : 4561.2 Std.dev. : 975.0 Peak high : 5455

Blocking tests:

AN / BB	Blocking TCP
No traffic	Average : 799.0 5 perc. : 545.0 95 perc. : 1052.0 Std.dev. : 172.5 Peak high : 1349.0
Medium traffic	Average : 683.6 5 perc. : 141.1 95 perc. : 1156.0 Std.dev. : 294.8 Peak high : 1444.0
High traffic	Average : 3439.7 5 perc. : 2130.7 95 perc. : 4561.2 Std.dev. : 975.0 Peak high : 5455

Extra blocking tests:

AN / BB	0% packet drop	2% packet drop	5% packet drop
No traffic	Average : 1453.8 5 perc. : 479.05 95 perc. : 1885.7 Std.dev. : 375.5 Peak high : 1997.0	Average : 726.7 5 perc. : 246.1 95 perc. : 952.0 Std.dev. : 181.5 Peak high : 1166.0	Average : 650.8 5 perc. : 231.0 95 perc. : 1023.0 Std.dev. : 254.3 Peak high : 1907.0
No traffic	Average : 5959.7 5 perc. : 473.1 95 perc. : 7802.35 Std.dev. : 2079.2 Peak high : 8161.0	Average : 2606.4 5 perc. : 588.0 95 perc. : 3826.6 Std.dev. : 886.4 Peak high : 4823.0	Average : 2153.4 5 perc. : 754.4 95 perc. : 3159.3 Std.dev. : 689.8 Peak high : 3609

Figure 51: Result matrix - the bottom line

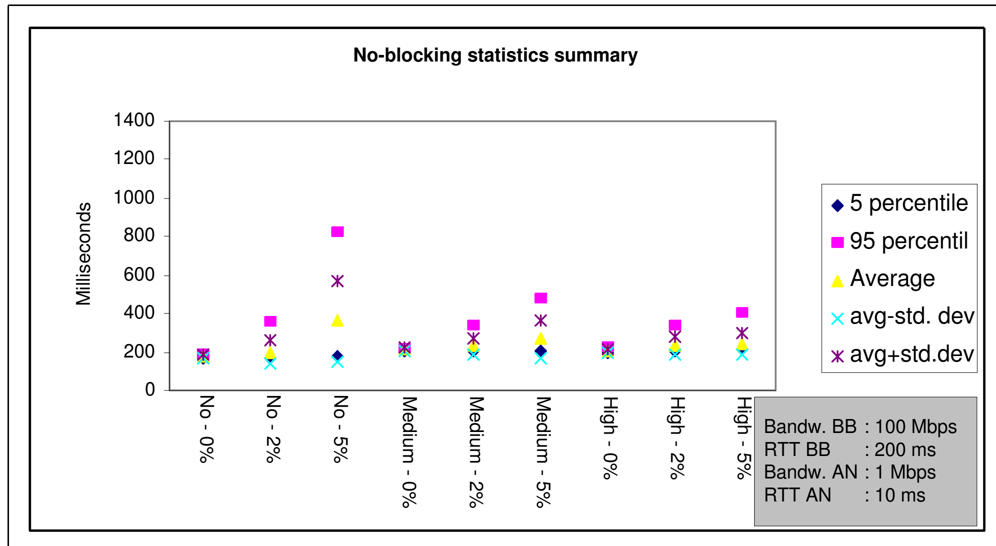

Figure 52: Statistics summary for non-blocking TCP

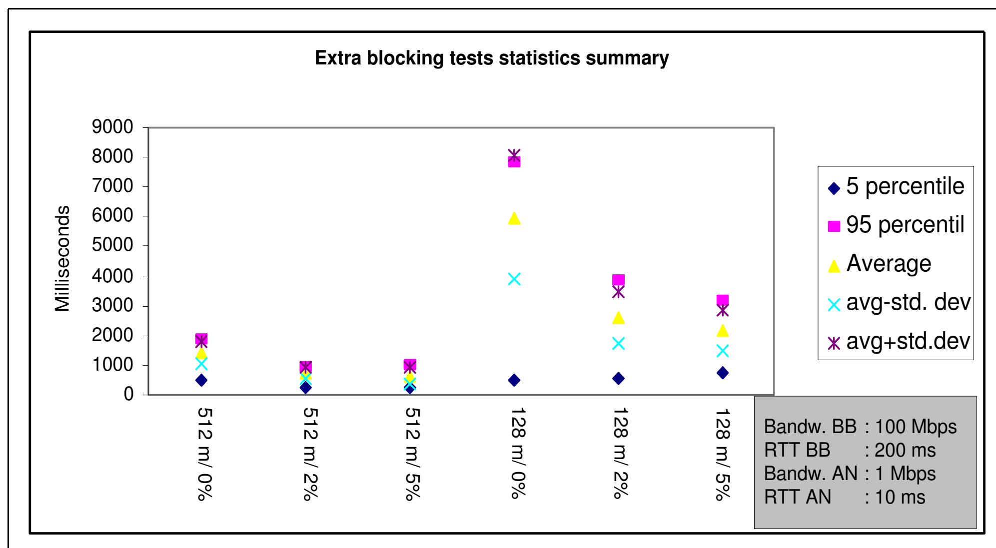

Figure 53: Statistics summary for blocking TCP

7 Tuning options

7.1 Use of priority progress streaming in proxy

When TCP in the access network experiences high back pressure with much streaming data in the application queue, it might be necessary to drop layers from the queue to prevent time-jitter and perceived quality-jitter. There are many algorithms to use for dropping layers from the queue. To evaluate the importance of a packet it makes sense to observe the timestamp of the packet and the priority of the layer in the packet like it is investigated in this work [20]. The application queue could either implement regular priority dropping thresholds, giving potential unstable quality in presentation, or priority progress streaming where both time-jitter and quality-jitter are handled in a graceful manner.

7.2 Minimize the TCP socket buffer in proxy for greater application level control

In our testbed, the TCP socket send buffer used was not manipulated in any sense. We used the default value of 64 kilobytes and did not touch any parameter file that could change the default behavior. In our scenario the best solution would be to have as much data as possible in the application queue and as less data as possible in the TCP send buffer. When data is sent to the TCP send buffer we no longer have the ability to manipulate the content. In this work [22], the authors present an architecture where the TCP socket send buffer is dynamically increased and decreased to the size of the congestion window. By doing this the authors show how TCP end-to-end latency can be reduced. This reduction of TCP end-to-end latency is suitable for (but not only) interactive multimedia applications. The implementation would fit well in our work to have greater control of all data waiting to be sent. We especially have the priority progress streaming scheme in mind in this context. This work could also be combined with our idea in section 7.3.

7.3 Introduce a jitter threshold in front of the application level TCP queue in proxy

When a client issues a HTTP-streaming session from a server, it is natural to buffer some amount of streaming data in the client buffer before presentation. If the client buffers X seconds of playback time, the client will start off playback with a head start of X seconds. This head start is meant to reduce jitter experience and function as buffer for longer periods of degraded throughput.

If the client negotiates an appropriate buffer time with the proxy during setup (new extension to RTSP ?), the proxy can use that information to make choices for the application queue. All packets in the application queue are provided with a timestamp when the streaming data was encoded. When back pressure occurs, the proxy can use the timestamp information and the amount of buffering at the client to predict if the user would detect jitter in the presentation upon decoding of the particular layer in the queue. When the proxy controls a packet in the queue it extracts the timestamp of the packet. The current clock

time of the server is subtracted with the timestamp to calculate the jitter at the moment. If this jitter value + estimated end-to-end delay is larger than the client buffering time, the layer will most likely contribute to observed jitter at the client. The proxy can thereby pop the layer from the queue. We note that the packets should be investigated from the front of the queue, as these packets have stayed in the queue the longest time.

The packets can also be controlled in the context of minimal socket send buffers (Section 7.2) for better estimation of end-to-end delay. The idea is presented in figure 54. The first solution is as simple as dropping any packet that breaks the jitter threshold. It also drops all other layers for the frame. This is not very pleasant when a packet with a high priority breaks the threshold. All other layers in the frame will be dropped as well. The second solution takes regard to priority when dropping packets based on the jitter threshold. The idea is to drop the lowest layers of the queue when the front of the queue gets close to the jitter threshold value. By doing this the higher layers get a better chance of passing the jitter threshold control. This prevents foremost the time-jitter but also smoothes out the layers so that the quality-jitter also can be prevented. It could be interesting to investigate such an architecture. In the figure we assume all layers of a frame has the same timestamp.

The end-to-end delay to compute is the number of milliseconds that has passed from the application calling the socket to write the packet until the receiver reads the packet from the socket. An estimation of this end-to-end delay can be computed as described in figure 55. The computation takes advantage of the RTP extension header and RTCP reports.

This idea with jitter threshold and how to compute it is not implemented and tested in our work. It is merely a suggestion for future work.

7.4 Degrade TFRC throughput during high TCP backpressure

All packets dropped in the proxy might not have to be transferred to it in the first place. The proxy can implement logic that makes the server decrease its TFRC based sending rate in times of high back pressure. In Komssys this is easily implemented by manipulating values in the TFRC APP packets sent to the server whenever an RTP packet is received to the proxy. As long as we decrease and not increase the sending rate outside the TFRC computed rate such a solution is acceptable. Anyway this makes TFRC too TCP-friendly. Other concurrent TCP sessions will steal this additional released bandwidth. By decreasing TFRC sending rate during high back pressure periods in the proxy we reduce sending packets from the origin server that would be received too late at the client in any case. The context of this behavior is that TFRC has to be viewed as TCP-friendly if its rate is not higher (but can be substantially lower) than other TCP sessions.

We demonstrated a maybe not so elegant, yet easy way of achieving this behavior by using blocking TCP in the access network.

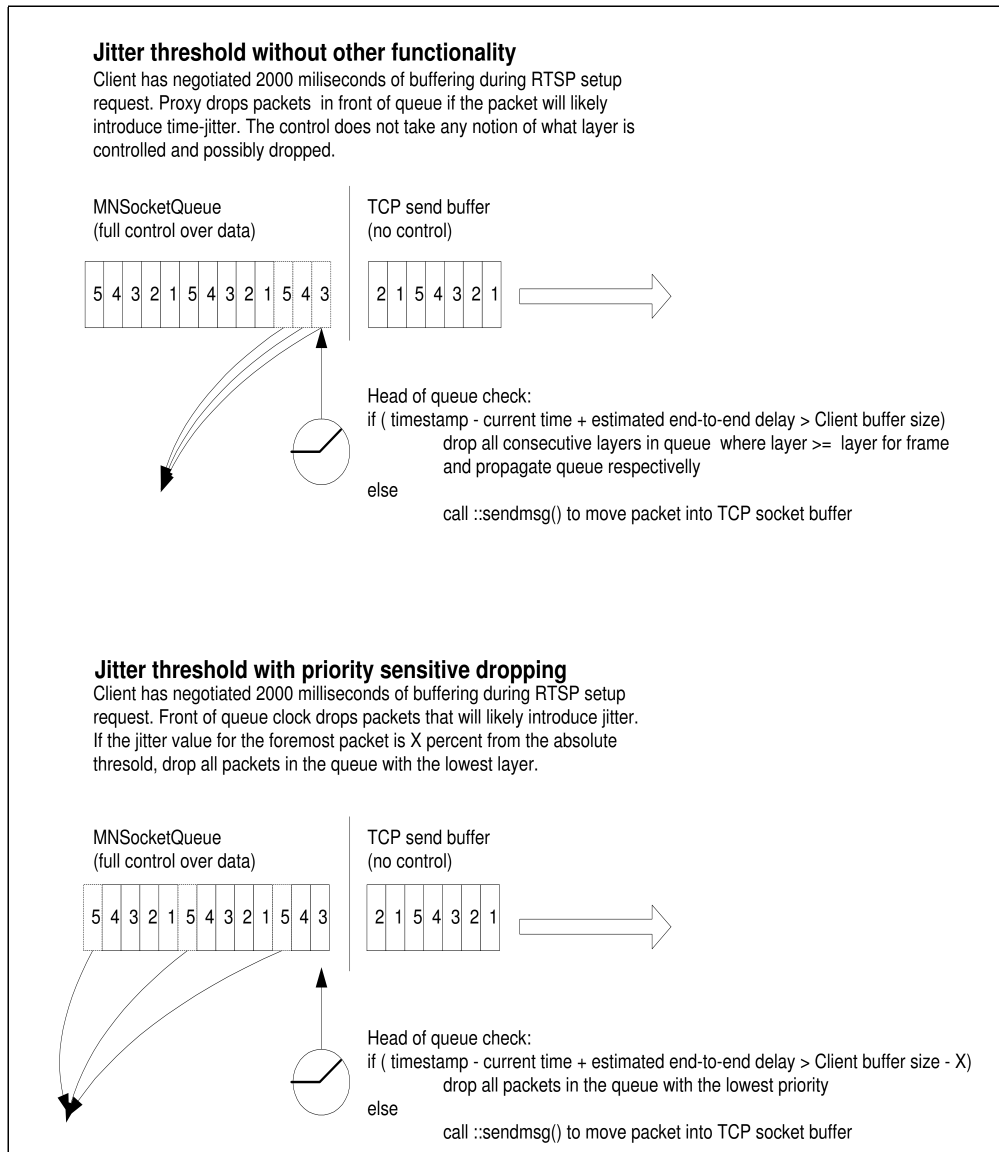

Figure 54: Jitter threshold with and without priority progress streaming

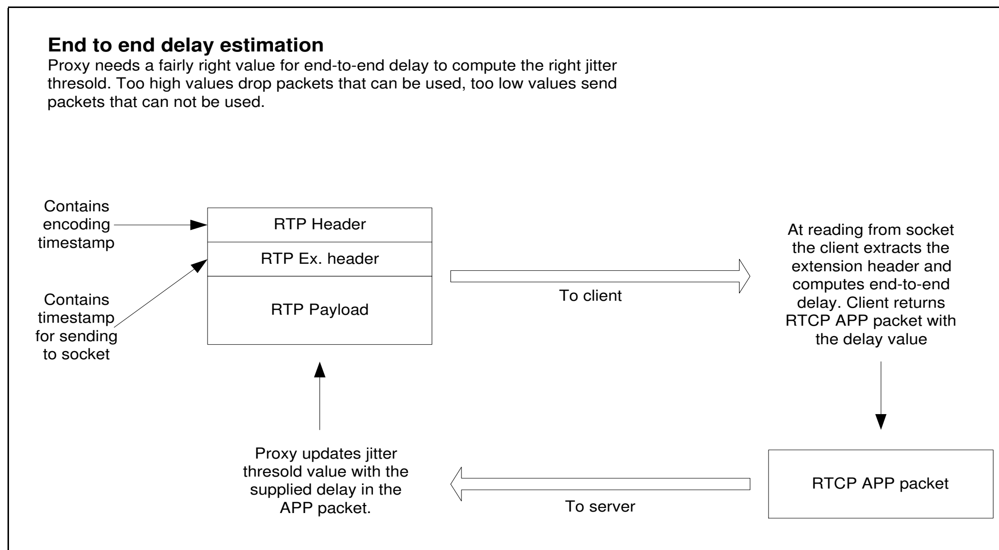

Figure 55: End to end delay computation

7.5 FACK modeled TFRC in backbone

TFRC is as described modeled after Reno TCP. For improved throughput between server and proxy a FACK modeled TFRC rate equation could be developed. This is particularly suitable when the access network has no problems delivering all data to the client (The opposite of high back pressure).

7.6 Drop TFRC badput in proxy

As presented in figure 19, the proxy can receive "bad" layers from the origin server. These bad layers (because of cumulative layered frames) can be dropped in the proxy. Our testbed did not include this behavior, with the result of badput being forwarded to the client.

7.7 Keep TCP session busy for improved throughput in access network

The only way of maximizing TCP throughput (with no regards to TCP flavor, window sizes or other TCP parameterization) is to keep it constantly busy, meaning that the send buffer should be kept as full as possible during a transfer. As shown in our work the access network TCP connection was sometimes limited by application bit rates instead of network bit rates. This scenario does not exploit TCP to its fullest. It is not possible to send data from the proxy faster than it is received by it from the server. It could be investigated how the server could deliver streaming content to the proxy faster than playback rate, so that the TCP session was kept busy. This again expands the need for buffering space at the client.

8 Conclusion

In this paper we have evaluated multimedia streaming of stored content in the context of a mixed transport environment using TFRC and TCP. The motivation for this work was to improve the infamous HTTP-streaming, which is end-to-end TCP based and firewall friendly. HTTP streaming is based on an end-to-end TCP connection from the server to the client. This introduces long round trip times. For TCP the throughput is inversely proportional with the RTT. Our proposal was to use TFRC over UDP in the long run with TCP in the access network. Because of access network properties (supplied by for example ADSL) we showed that TCP varies throughput over time, but the delay variations are limited for the normal backbone - access network scenarios. For the client to view a HTTP-streaming movie, only a small amount of buffering is needed. For all cases in our work (except TCP-blocking tests) the client would need a maximum of 744 milliseconds of buffering for very successful presentation based on the 95 percentile. The emulation scenario was based on an ADSL access network connection with static packet drop rates. The backbone was filled with parallel traffic using a Pareto distribution. Some TCP blocking tests were executed to see how this could limit the backbone rate. It is evident that the backbone rate drops in average to the access network rate during blocking tests. The problem is among other challenges that TCP blocks for a long period creating large variances in throughput on the backbone link. We investigated the strange effects of blocking TCP and tried to explain the behavior seen in the emulations. The conclusion of blocking TCP tests is that blocking TCP introduces high throughput variances in the backbone due to long sequences of packet losses. It was discovered that when TCP blocks it does not unblock until it has drained one third of the send buffer. In our proxy solution all packets sent to the proxy from the backbone were considered lost during blocking periodes.To reduce the perceived quality jitter we suggested a trade off between TCP blocking time and the overhead of context switching into the kernel for smoother sending rate in the backbone.

We suggested an extension to RTSP where the client can negotiate the total buffering time. This extension is particularly useful when streaming is requested over TCP. The proxy can exploit this information to make decisions on which packets to send to the client. ADSL lines are supposed to be fairly stable in terms of bandwidth (a dedicated line for every customer) and the fact that the access network delay is very limited, TCP shows a nice average throughput which can vary quite a bit (when high packet loss rates are introduced) from one second to the other. This is not an issue (for streaming of stored content) with just a small amount of buffering at the client. If the client requests a stream that exceeds the ADSL bandwidth, the proxy can implement priority progress streaming or a jitter threshold in the proxy queue to adapt to the network condition. In this paper we assumed no client requests a stream that is defined to exceed the general supplied bandwidth (1 Mpbs) on the ADSL line with the exception of the blocking TCP tests.

For future work we presented a solution where the client and the proxy during setup negotiation can agree on a total amount of client buffering time. If the proxy knows the maximum buffering time on client it can make intelligent

choices for improved perceived quality and minimize delay variations. Several other future tuning scenarios were also presented, for example dropping unusable layers in the proxy and implementing minimal socket send buffers as presented in [22].

The bottom line is that the target scenario can be used as a basis for improved HTTP streaming in a more complex 2 phase transport environment. For live streaming or interactive traffic like video conference sessions our results show that greater care must be provided to deliver the responsiveness required by these applications. Delays in the order of a couple of hundred milliseconds can easily create problems in such applications. Our target scenario was streaming of stored content in the context of HTTP streaming.

A Reno, SACK and FACK TCP details

A.1 Introduction

The main goal of this appendix is to understand different implementations of congestion control in the transmission control protocol (TCP). The paper consists of two parts. The first part contains a description of three different TCP implementations. The second part of the paper consists of simulation results using the Network Simulator 2 (NS2) [24].

The chosen TCP variations to investigate are Reno TCP (not New Reno), Selective acknowledgement TCP (Reno with SACK) and Forward acknowledgement TCP (FACK TCP). NS2 supports these three TCP variations. This paper begins with an overview of congestion control behavior of the earlier TCP implementations [11]. Following we give an overview of congestion control in the three different implementations. The intention is not to focus on the general implementations, but details about reaction to lost segments, corrupt segments and timeouts. The different implementations will react differently to such events. Congestion control and data recovery algorithms vary in their effort to maximize throughput and at the same time preserve the properties of TCP congestion control as described in [11]. In the next section the original TCP implementation is described. It reacts drastically when congestion is discovered. The three newer implementations deal with this drastically decrease by implementing different algorithms and solutions to congestion control. There are other improvements in these three implementations, but in this paper we focus on congestion control and data recovery.

A.2 Description of Reno, SACK and FACK TCP

A.2.1 The earlier TCP implementations

With the terms "earlier TCP" and "TCP" in this specific section, we refer to implementations based on [30] with the enhancements described in [11].

When TCP starts to transmit segments, it enters a so called "slow-start" phase. Initially it sends one segment to the destination. The sender will later receive an acknowledgement segment from the client. The acknowledgements are cumulative and inform the sender that everything up to byte "X" is received. From the time when the sender ships data down the TCP stack to the time when an acknowledgement for that segment is received is called the round trip time (RTT). When the first segment is acknowledged by the receiver, TCP tries to send two segments. If both segments are acknowledged, TCP tries to send four segments and so on. This kind of behavior leads to an exponential rate increase. This behavior continues until one of four events occur:

- If the number of segments sent exceed the number of segments the receiver can handle, TCP will stop the acceleration of segments sent. The exponential behavior ceases and TCP continues to send the maximum number of segments the destination can accept. When the receiver sends back an acknowledgement segment, it contains information about what size is left in the receivers buffer. As long as segment loss,corruption or packet

reorder do not occur, the maximum transfer volume will be controlled by that information.

- The second event occurs when TCP discovers that a segment sent is not acknowledged in a fixed amount of time (retransmission timeout occurs). When such an event is triggered, TCP re-enters the slow start phase. This behavior will lead to bad throughput during heavy load.

- The third event occurs when the sender receives a duplicate acknowledgement. A duplicate acknowledgement tells the sender that one or more segments in one shipment are lost or corrupted. The sender waits for a third duplicate acknowledgement. When received, the sender assumes one segment is lost and retransmits that segment before its timer expires. After the fast retransmit, the sender enters "slow start".

- The fourth event occurs when the byte count of segments sent in one shipment exceeds the threshold value. The threshold value tells TCP when to cease the exponential behavior and increase linearly (congestion avoidance mode). If neither congestion occurs or the receivers buffer is exceeded before the threshold value is reached, TCP will stop the exponential behavior. TCP will then add one segment byte count (different behaviors for different implementations) for each successful round trip time. To be more specific: If W is the size of the congestion window and W segments are outstanding in the network, the congestion window will increase by 1/W for every incoming acknowledgement. When W segments are received the congestion window has increased by one segment worth of bytes.

To summarize, the earlier TCP implementations have these major concerns:

- The acknowledgements are cumulative and give the sender little information about what segments are actually lost. The sender can only learn about and retransmit at most one lost segment pr. round trip time.

- The sender enters slow start after a fast retransmit when a constant number of duplicate acknowledgements are received.

- Under heavy network load TCP can perform poorly because of frequent initiation of the slow-start behavior.

A.2.2 Reno TCP

The major change when it comes to congestion control is how Reno implementations react to three duplicate acknowledgements from the receiver. As explained earlier, TCP will complete a fast retransmit due to three duplicate ACK's. This makes TCP retransmit the segment so that TCP does not have to wait for a timer to expire. Reno also does this but will not enter slow start after the retransmit. Reno enters fast recovery mode [1]. The fast recovery mode governs data transmission until a non duplicate ACK arrives that acknowledges new data. The argumentation for not entering slow start is that the sender does receive acknowledgements from the receiver. These acknowledgements indicate that segments are arriving at the receiver. Segments are therefore assumed to be flowing through the connection because acknowledgments are only sent

back whenever a segment arrives (Unless the receiver uses delayed acknowledgments). Reno combines fast retransmit and fast recovery in the algorithm presented below. The algorithm text assumes the reader has good knowledge of TCP parameters such as sshtresh, ccwnd and other specific TCP terminology. For an explanation of terms and variables in the algorithm text we refer to [1]:

- When the third duplicate ACK is received, set sshtresh to no more than the value max (Flight Size / 2, 2*SMSS) .
- Retransmit the lost segment and set ccwnd to sshtresh plus 3*SMSS. This artificially "inflates" the congestion window by the number of segments (three) that have left the network and which the receiver has buffered.
- For each additional duplicate ACK received, increment ccwnd by SMSS. This artificially inflates the congestion window in order to reflect the additional segment that has left the network.
- Transmit a segment, if allowed by the new value of ccwnd and the receivers advertised window.
- When the next ACK arrives that acknowledges new data, set ccwnd to sshtresh (the value set in step 1). This is termed "deflating" the window.

  This ACK should be the acknowledgement elicited by the retransmission from step 1, one RTT after the retransmission (though it may arrive sooner in the presence of significant out- of-order delivery of data segments at the receiver). Additionally, this ACK should acknowledge all the intermediate segments sent between the lost segment and the receipt of the third duplicate ACK, if none of these were lost.

Today this algorithm is heavily implemented in Internet based computers, but the SACK implementation will eventually take over. The Reno implementation is widely distributed and gives better throughput than TCP implementations without fast recovery. This implementation is known to not recover gracefully when multiple segments are lost from one single flight of segments. Beneath we try to explain why by giving two detailed example scenarios.

One segment loss in one flight of data Assume a senders congestion window equals the size of 50 segments (ccwnd) and that the segment size is 1024 bytes. The threshold size (sshtresh) is 30. The max window send size equals 60. At present the sender has 50 outstanding segments in the network. Since ccwnd does not allow for more segments on the network, the sender awaits acknowledgement segments from the receiver. Assume the last acknowledged segment is segment 250 (Segment 253 will be lost)

- The sender receives a regular acknowledgements for segment 251. (Ccwnd is increased with 1/50)
- The sender transmits a new segment (301) based on the acknowledgement.
- The sender receives a regular acknowledgement for segment 252. (Ccwnd is increased with 1/50)
- The sender transmits a new segment (302) based on the acknowledgement.

- The sender receives the first duplicate acknowledgment for segment 252.
- No new segments are sent because no new segments are acknowledged.
- The sender receives the second duplicate acknowledgement for segment 252.
- No new segments are sent because no new segments are acknowledged.
- The sender receives the third duplicate acknowledgement for segment 252.
- The sender assumes segment 253 is lost based on 3 duplicate acknowledgements.
- The sender retransmits segment 253 and starts a retransmission timer.
- The sender halves the threshold size to $50/2 = 25$
- The sender sets the ccwnd = sshtresh
- The sender virtually adds 3 segments to ccwnd (called inflating the window), since 3 duplicate acknowledgements signalizes that the receiver has received and cached 3 segments that are not following the segment sequence.
- The data in flight is now assumed to be 47 segments worth of bytes and the virtually congestion window equals 28 segments worth of bytes.
- Before the sender can send more data it has to virtually increase the ccwnd by one for each duplicate acknowledgement received (251). When half the original ccwnd size of duplicate acknowledgements are received, ("virtual" ccwnd = old ccwnd) the number of outstanding segments is equal to half the original ccwnd size.
- The sender will now with every duplicate acknowledgment received (252) send out new segments (303 and beyond) because the "virtual" ccwnd allows it.
- The sender now receives a new acknowledgment which acknowledges all segments cached at the receiver (The ones that triggered totally 48 acknowledgements for segment 252) including segment 253. When the 253 segment was received, the segment filled a gap and suddenly created a long row of sequentially right ordered segments.
- The sender will receive an acknowledgement for segment number 302. (The filled gap at 253 acknowledges up to segment 302)
- The sender will now really set the ccwnd = sshtresh (25), that in practice will cut the ccwnd in half its original size before the fast recovery algorithm was triggered. This is called deflating the window.
- The sender exits fast recovery.
- The sender is now in congestion control mode with linear increase of the congestion window, and a full window of segments (25) are assumed outstanding.

This scenario shows that Reno will successfully execute fast recovery when 1 segment is lost during one flight of data. No slow start is performed, no retransmission timeout has to be awaited and the pipe is not flushed.

Multiple segment loss in one flight of data Assume a senders congestion window equals the size of 16 segments (ccwnd) and that the segment size is 1024 bytes. The threshold size (sshtresh) is 19. The maximum send window size equals 19. At present the sender has 16 outstanding segments in the network (during slow start). The sequence numbers of these outstanding segments range from 15 to 30. Assume segments 28 and 30 are lost due to a overflowed buffer in an intermediate router.

- The sender receives regular acknowledgements for segments 15 to 17.
- The sender continuously transmits new segments with sequence numbers 31 to 36. (exponential behavior)
- The flight size is now 19 as is the congestion window.
- The sender receives regular acknowledgments for segments 18-27.
- The sender transmits new segments (38-47) based on the acknowledgements. Now only 1 segment can be sent out in reaction to one new acknowledgement as the maximum sender window size is 19.
- The sender receives the first duplicate acknowledgement for segment 27.
- No new segments are sent because no new segments are acknowledged.
- The sender receives the second duplicate acknowledgement for segment 27.
- No new segments are sent because no new segments are acknowledged.
- The sender receives the third duplicate acknowledgement for segment 27.
- The sender assumes segment 28 is lost based on 3 duplicate acknowledgements for number 27.
- The sender retransmits segment 28 and starts a retransmission timer.
- The sender halves the threshold size to flight size/2 = 9
- The number of outstanding segments is now (19-3) 16.
- The sender sets the virtual ccwnd = sshtresh
- The sender virtually adds 3 segment to ccwnd (called inflating the window), since 3 duplicate acknowledgments signalizes that the receiver has received and cached 3 segments that are not following the segment sequence. The congestion window now equals 9 segments.
- The number of segments in flight is 16 segments

- Duplicate acknowledgements based on the 16 outstanding segments arrive before the retransmitted lost segment is acknowledged. No new segments are sent because these are duplicate ACK's and the sender can not send as long as a full window of data is assumed outstanding.
- Now a new acknowledgement comes in (The fast retransmitted one). This acknowledgments carries sequence number 29 as segment 30 is lost.
- The sender now sets the real ccwnd to (19/2) 9 segments and exits fast recovery.
- Now the fast recovery for segment 30 can not be executed as the connection is idle.
- The retransmit timer goes off and segment 30 is retransmitted.
- Reno enters slow start mode.
- An acknowledgment for sequence number 46 arrives, as the 16 duplicate acknowledgements received while in fast recovery really was a response to the receiver receiving segments 31 to 46.

We can see that multiple loss of segments from one flight of data can have drastic impact on throughput. The problem is that fast recovery was optimized and supposed to solve a 1 segment drop pr. window situation. A solution to this problem is called New Reno. The fast recovery algorithm in New Reno can recover from multiple lost segments in the same window. Reno will fail to recover when a small number of segments are lost (if the congestion window is relatively small and/or the congestion window is close to the receivers advertised window when fast recovery is invoked) . With other parameters, Reno might survive a 2 segment loss, but react according to this scenario when 3 or more segments are lost. If Reno survives a 2 segment loss, the congestion window will anyway be halved two times from it's original size. This behavior slows down throughput considerably.

A.2.3 SACK TCP

SACK TCP is defined here [23]. SACK TCP implementations solve the cumulative acknowledgement problem by making the receiver send SACK segments back to the data sender. The SACK packets inform the sender which segments are received successfully and reveals which segments did not arrive. The sender can thereby choose to retransmit only the missing segments. This improves performance and prohibits redundant segment transmissions. The sender does not have to wait one round trip time to learn about each and every single segment drop. Already sent and received segments will not be retransmitted. SACK is not intended to give better throughput when only one segment is lost in a window. Reno TCP will give good comparative throughput in such circumstances. SACK TCP is meant to solve, among other problems, the poor performance when multiple segments are lost from one window (Like New Reno does), but SACK also tells the sender what is actually received.

SACK is an extension to TCP implementations, and SACK permitted Reno

implementations are very common. The receiver informs the sender about its SACK capability by sending a SACK permitted option in a SYN segment. The sender (if SACK capable) will then understand and interpret SACK blocks in the acknowledgments.

As long as segments are received in order, the client will acknowledge with regular acknowledgement segments. SACK option packets are sent once the receiver discovers reception of segments with sequence numbers that do not fit in the left edge part of the receiving window. This means one or more segments are lost, corrupted or reordered. The SACK acknowledgement informs the sender which segments are received and retransmits what appears to be the missing segments. When retransmitted segments arrive, the receiver acknowledges the data normally (cumulative). A SACK segment is structured as one or more blocks. These blocks contain definition of continuous received segments. A SACK option contains 3 important parameters: "ACK", "Left edge" and "Right edge". "ACK" contains the sequence number that a regular acknowledgment segment would contain (Reno). The "Left edge" parameter contains the lowest sequence number beyond "ACK" that is received. Right edge contains the highest sequence number received after "ACK". All segments between left edge and right edge are received successfully and in order.

The table below shows an example of an acknowledgement with 3 SACK blocks. Segments are 500 bytes in size. The last acknowledged byte is 5000 and in a burst of 8 segments, segments 2,4,6 and 8 are lost. For each of the 8 acknowledgements the SACK blocks will contain the information as shown in the table below.

Triggering Segment	ACK	First Left Edge	Block Right Edge	2nd Left Edge	Block Right Edge	3rd Left Edge	Block Right Edge
5000	5500						
5500	(lost)						
6000	5500	6000	6500				
6500	(lost)						
7000	5500	7000	7500	6000	6500		
7500	(lost)						
8000	5500	8000	8500	7000	7500	6000	6500
8500	(lost)						

The first segment is acknowledged as a regular ack. The second segment is lost. In the second acknowledgment the receiver adds one SACK block. It informs the sender that 5500 is acknowledged and the earliest segment thereafter starts with byte 6000. The sender learns that the segment containing bytes 5500-6000 is lost. Further on the fourth packet is lost, and the third acknowledgment contains 2 SACK blocks. Note that the highest received segments are always defined in the first SACK block. These blocks inform the sender that segments containing bytes 5500-6000 and 6500-7000 are lost. Packet number six is also lost, and the fourth acknowledgment contains 3 SACK blocks with the same pattern as described earlier.

SACK TCP is meant to solve the cumulative acknowledgement problem, giving higher throughput and reduce transmission of already successfully received segments. If for example a Reno implementation is extended to support SACK, the congestion control implementation should remain unchanged.

A.2.4 FACK TCP

FACK TCP is designed to be used in conjunction with the SACK option described in the previous section [23]. The main concern for SACK is to address data recovery, not congestion control. SACK will help TCP survive multiple segment losses within a single window without incurring a retransmission timeout. SACK information can be used to let the sender know about the current congestion status on the connection as well. This information can be used to optimize TCP during data recovery. Pure SACK implementations do not contain such behavior. The FACK is intended to decouple data recovery and congestion control and improve both.

The data recovery solution is covered by original SACK, while FACK focuses on improving congestion control algorithms during data recovery. FACK got its name because the implementation makes the receiver inform the sender about its forward most received sequence number (SACK does too, but does not exploit the information). A regular SACK implementation preserves the existing fast recovery algorithm as described in section A.2.3. Because of SACK acknowledgments, fast recovery will not be ended prematurely because of partial acknowledgments. FACK contains detailed changes to the algorithm that calculates outstanding segments during recovery. This is the difference between SACK and FACK. FACK aims to change algorithms for when to send data, while SACK only improves the algorithms for what to send. Below we try to explain the exact differences between SACK and FACK implementations.

- In SACK, the sender uses SACK blocks to figure out what to send, what not to send and when to exit fast recovery. FACK also does that, but in addition it uses information in SACK blocks to record the highest received acknowledgment number in a new state variable called "snd.fack". The sender will always know about the highest received sequence number.
- FACK introduces another state variable with the name "retrans_data". This variable will reflect the quantity of outstanding retransmitted data in the network.
- The existing "snd.una" variable has the same value as "snd.fack" during non fast recovery. The highest received sequence number is also the highest acknowledged sequence number. During fast recovery "snd.fack" will cointain the highest received sequence number, while "snd.una" will contain the highest acknowledged sequence number. During fast recovery these values will differ.
- **During fast recovery FACK will compute the estimated amount of outstanding data to be "awnd = snd.nxt - snd.fack + retrans-data". The "snd.nxt" state variable holds the highest sequence**

number sent. SACK will only base its knowledge of outstanding data by counting duplicate acknowledgments from the receiver. FACK will continuously update its state variables during fast recovery. The result is a better knowledge of what amount of data is outstanding in the network. As long as awnd < ccwnd the sender can push new segments onto the network. These segments can either be new data or retransmissions.

- With SACK, fast recovery is triggered based on three duplicate acknowledgements. With FACK, fast recovery is triggered when the sender discovers that (snd.fack - snd.una > 3*MSS) the highest acknowledged segment is beyond 3 segments from the highest received segment. If three duplicate acknowledgements are received before this condition is met, fast recovery will also be executed.
- When a large number of segments are lost, SACK will overestimate the amount of data currently in the network (underestimate the congestion window) and inhibit sending of new data or retransmissions. It will often have to wait several round trip times to fully recover. FACK will recover faster because it will know about the amount of lost segments (better estimation of outstanding segments by using snd.fack), thereby retransmitting new segments earlier than SACK will.
- FACK will use less time in the recovery phase than SACK which results in higher throughput over lossy network conditions.

To really see performance differences between SACK and FACK, we will later in this paper simulate heavy traffic so that several segments will be dropped in several windows of data. During simulation of one and two segment drops, SACK and FACK will generally perform equal.

A.3 Simulation results using Reno, SACK and FACK

A.3.1 Simulation setup

We present the simulation results in two parts. In the first part (Section A.3.2 and A.3.3) the environment was configured to loose one or two packets during slow-start. In the second part (Section A.3.4) we configure NS2 to create heavy network load with different loss and delay situations with concurrent transmission between the three implementations.

A.3.2 Simulating one segment drop during slow start

In this section we present graphs showing the development of the congestion window and the maximum sequence number acknowledged during a 5 second simulation. The simulated protocols are Reno, SACK (Reno with SACK) and FACK (Reno with SACK with FACK). The properties of the simulation setup are as follows:

- 1 client, one router, 1 server.
- 5 seconds FTP application transfer between client and server

- Bandwidth client-router = 1Mb
- Delay client-router = 50ms
- Bandwidth router-server = 1Mb
- Delay router-server = 50ms
- Segment size = 1024 bytes
- Initial threshold value = 19 segments
- Initial receiver window buffer size = 19 segments
- Initial congestion window size = 1 segment
- Maximum congestion window size = 50 segments
- Maximum buffer client-router = 9 segments
- Maximum buffer client-router = 100 segments

Reno The results from the simulation are presented in figure 56. The top graph shows the development of the highest numbered segment sent during the 5 second test. As the test starts the development of the sequence number acknowledged is exponential because of Reno slow-starting. At circle one the development of the sequence number stops. Because the receiver has discovered one out of order sequence number, it sends several acknowledgments for the last successfully sent segment. The development of the last acknowledged sequence number halts. The development can only continue when the sender has received 3 duplicate acknowledgements (or timeout), retransmitted the lost segment and received a new acknowledgment from the receiver. The round trip time in the test is set to 0.3 seconds, and we can clearly see that no new sequence numbers are acknowledged in that time. The gap in the test will exceed 0.3 seconds, as more than one round trip time passes from the receiver initiates fast recovery and a new acknowledgment is received. At circle two we can see that as time goes by the acknowledgments are coming in more and more seamlessly.

In the second graph we can see the development of the senders congestion window. The gap at circle one is there because the graph will only give output when the window changes its value. Initially it is set to one and the first dot (after about one RTT) appears at the value 2, which is when the first acknowledgment is received. The congestion window increases exponentially. At circle three the sender has discovered one (or more) lost segment by the reception of three duplicate acknowledgments. The sender enters fast recovery, retransmits the indicated lost segment and halves the congestion window. The simulation does not tell us exactly what is going on. The "virtual" inflation of the congestion window while the sender is in fast recovery is not discovered (See section A.2.1-A.2.2). Anyway, the congestion window is halved. Between circle three and four duplicate acknowledgments arrive (if there is anything to acknowledge). Just before circle four the retransmitted lost segment is acknowledged. The sender exits fast recovery. At circle four these acknowledgments come in and the sender is linearly evolving its congestion window. At circle five we can see

the acknowledgments coming in more and more seamlessly because of the linear growth of the congestion window. The fast recovery is executed successfully.

SACK There are really no other special observations to comment when we simulate this using Reno's SACK option implementation. In general the exact same is happening compared to the last simulation. The difference lies in that the receiver will not send regular acknowledgment segments when the lost segment is discovered. It will send acknowledgments with one SACK block. The block is, as earlier mentioned, an indication that one or more segments are lost from the window and that they are in sequence. In this case one segment is lost. As with regular Reno, three duplicate acknowledgments with SACK options makes the SACK implementation enter fast recovery. After that the behavior of Reno is followed. The result can be viewed in figure 57. The comments to the circles are the same as in the last simulation.

FACK As expected, FACK behaves like SACK and Reno (figure 58). We could have skipped the simulation of one segment drop as long as we do not simulate the Tahoe implementation (which would go into slow-start after retransmitting the lost segment). For the sake of understanding the protocol implementations we wanted to simulate one segment drop. But one interesting detail is how fast FACK is out of fast recovery compared to Reno and SACK. The simulation indicates that the congestion window will go down to one and climb fast up to half the original congestion window.

A.3.3 Simulating multiple segment drops during slow start

In this section we present graphs showing the development of the congestion window and the maximum acknowledged sequence number pr. second. The simulated protocols are Reno, SACK (Reno with SACK) and FACK (Reno with FACK). The parameters are as followed (if not listed, the value is equal to the previous documented simulation):

- Maximum buffer client-router = 8 segments

By manipulating the queue we achieve the wanted effect of two dropped segments in one flight of data.

Reno The first graph in figure 59 shows that the acknowledged sequence numbers are increasing exponentially prior to circle one. The time between circle one and two really shows the problem with multiple dropped segments in a Reno implementation.

In the second graph in figure 59 we can see what happens with Reno when multiple segments are dropped in one flight of data. At circle one the first acknowledgement arrives. At circle two the sender has received three duplicate acknowledgements and it enters fast recovery. At circle three the congestion window is halved and the presumed lost segment is retransmitted. As time flies by, if anything to acknowledge, duplicate acknowledgments come in. At circle four the acknowledgment for the retransmitted lost segment is received. This acknowledgment is a partially acknowledgment and the sender does exit fast

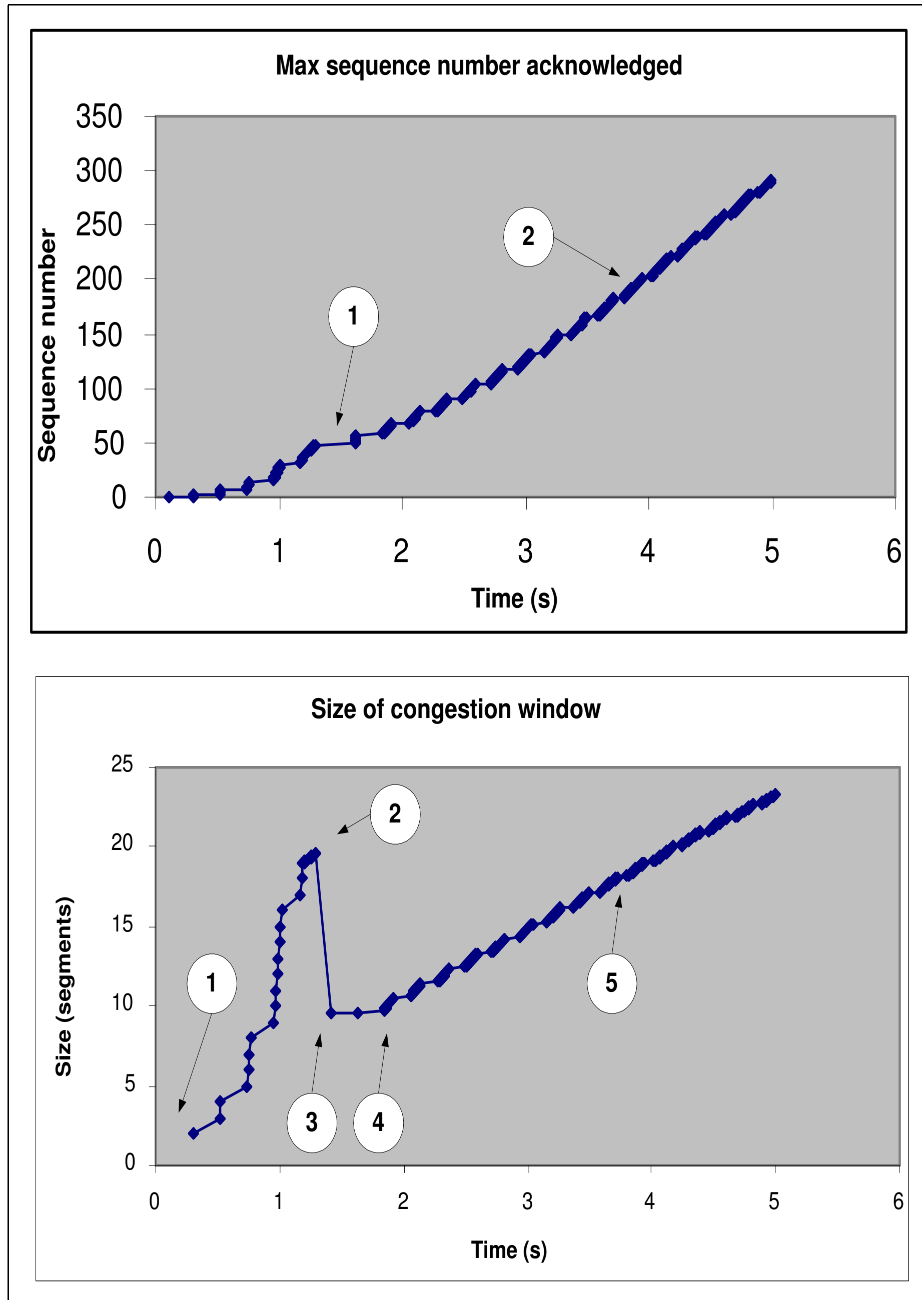

Figure 56: One segment drop simulation with Reno TCP implementation

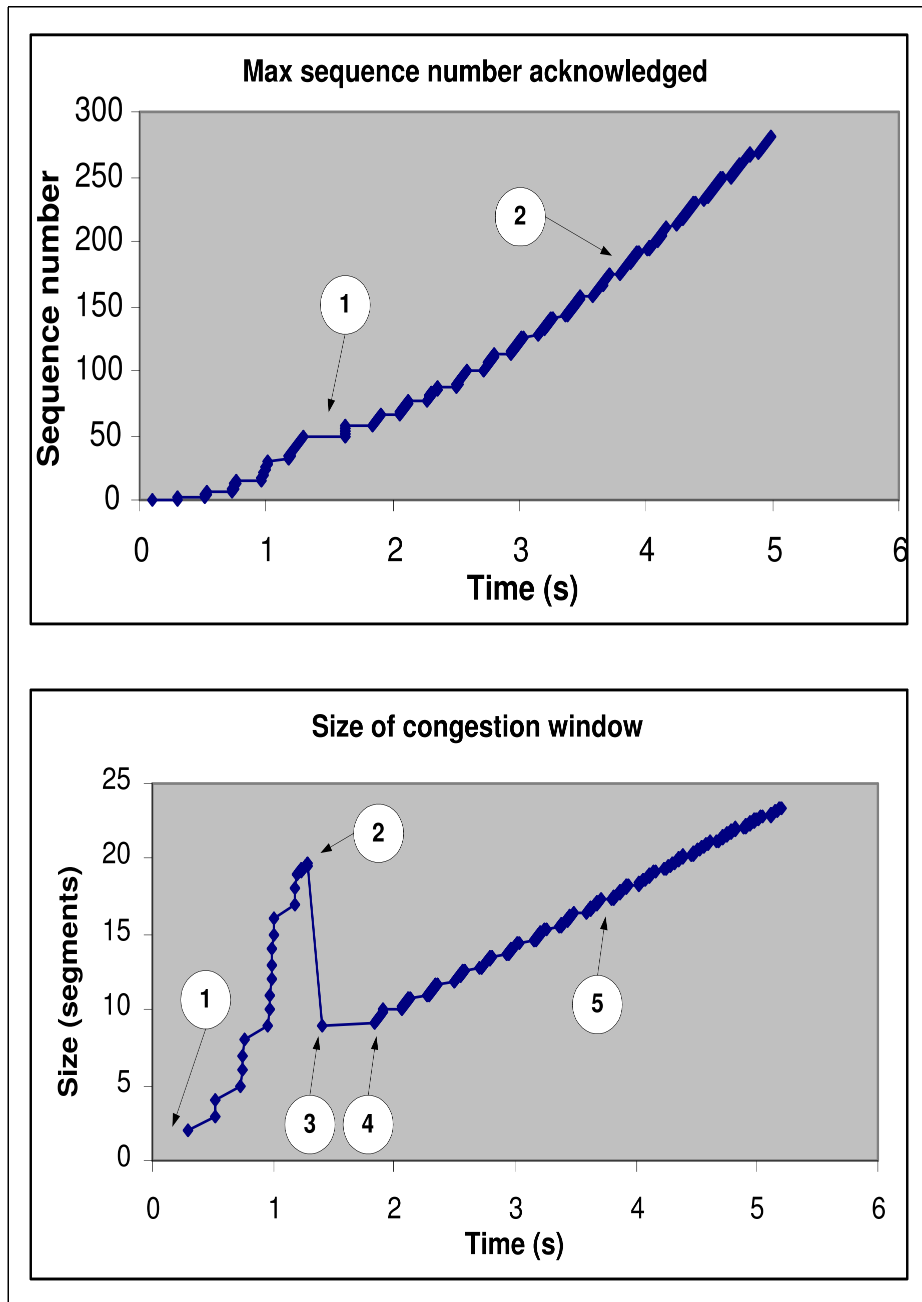

Figure 57: One segment drop simulation with SACK TCP implementation

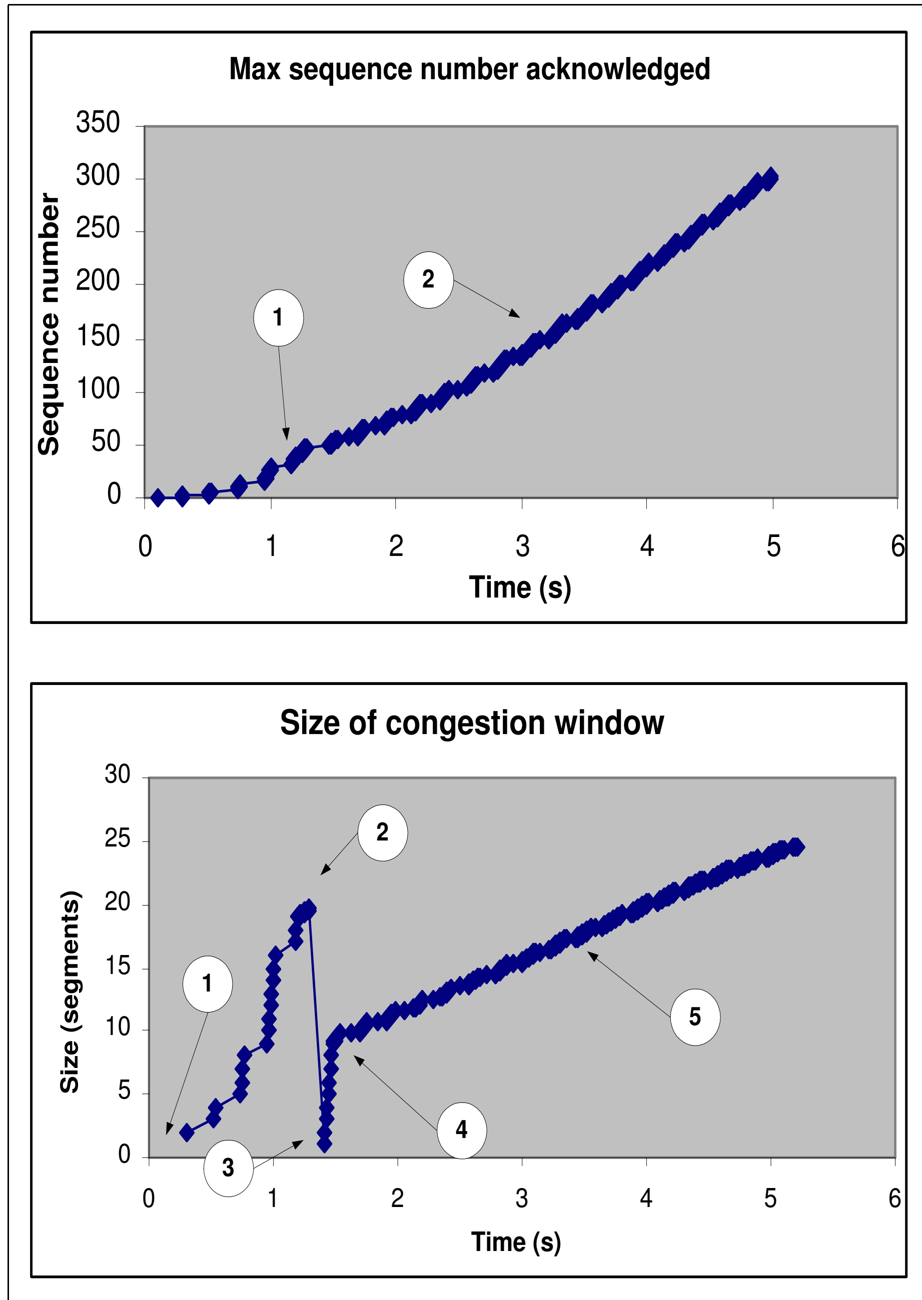

Figure 58: One segment drop simulation with FACK TCP implementation

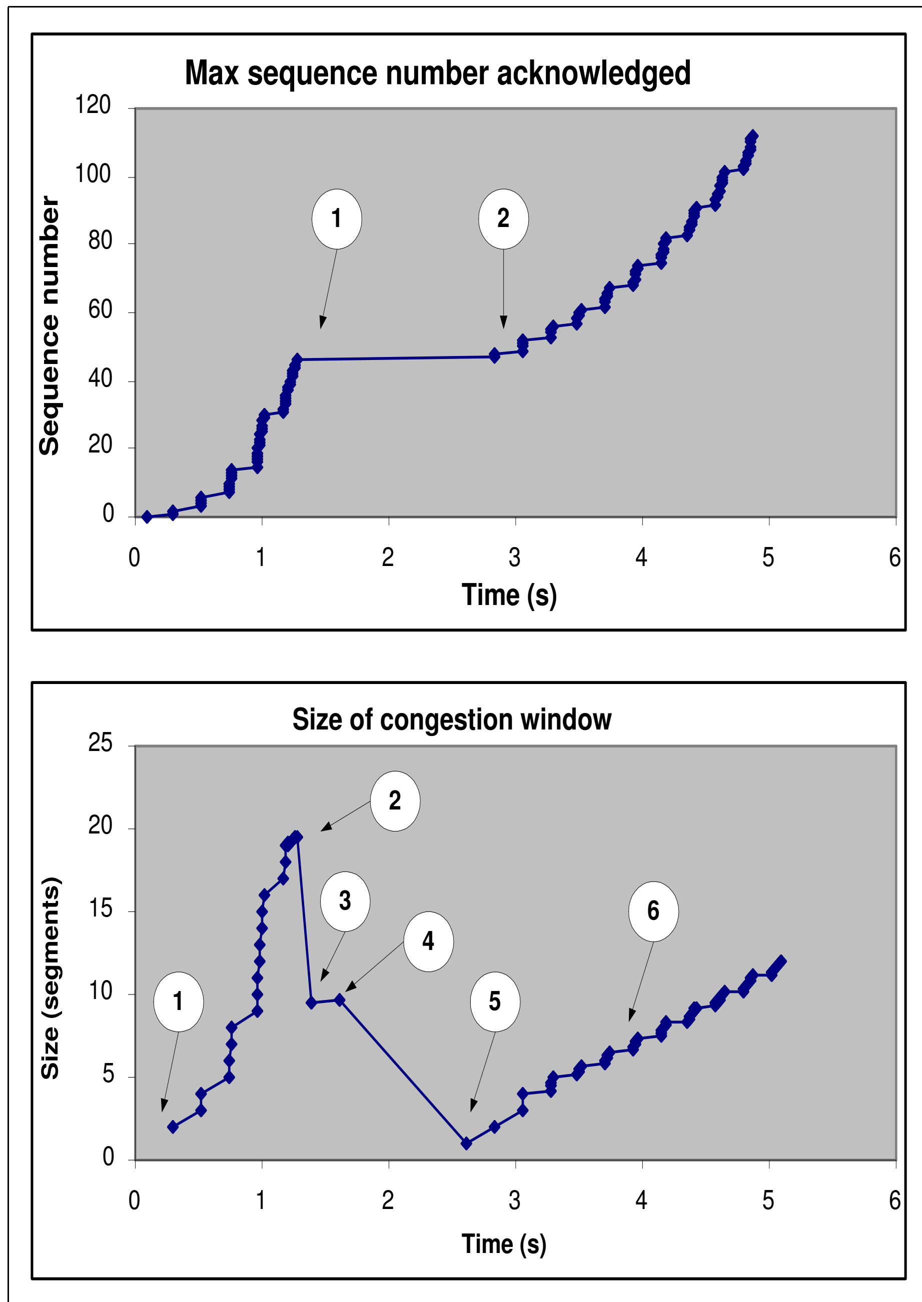

Figure 59: Two segment drops simulation with Reno TCP implementation

recovery (not good). Now the sender will not be able to send more segments as there is a full window of data outstanding and at the same time it is not likely that any duplicate acknowledgments to trigger fast recovery will be received (This behavior is as mentioned in A.2.2 based on the variables when fast recovery was initiated). The sender must wait for the retransmit timer of the partially acknowledged segment + 1 segment. The timer expiration makes TCP initiate slow-start mode. At circle five the congestion window is down to one segment, and at circle six the acknowledgments start to come in more and more seamlessly.

SACK The first graph in figure 60 shows a better development of the throughput compared with Reno. At circle one, duplicate acknowledgments are starting to come in. After a successfully fast recovery (because of SACK) the development increases exponentially. At circle two we can see the smooth out of incoming acknowledgements.

In the second graph in figure 60 we can see that SACK successfully completes a fast recovery with two lost segments in one flight of data. At circle two the three duplicate acknowledgments are received. The sender enters fast recovery. But where regular Reno would exit fast recovery based on a partial acknowledgement, the SACK implementation does not exit. The partial acknowledgment will carry SACK blocks that will reveal we are dealing with a partial acknowledgment. For details on what SACK does when a partial acknowledgement is received we refer to [23]. The fast recovery is exited when all segments sent upon fast recovery initiation are acknowledged. That event is indicated by the sender receiving an acknowledgement containing no SACK blocks. This behavior makes the sender successfully recover.

FACK As expected, FACK behaves somehow like SACK (figure 61). To visually discover performance differences between the two implementations, a larger number of segments have to be dropped from sent windows more frequently. Under such circumstances SACK's long recovery period will make it give lower throughput than FACK. In 3.4 we will simulate a network load to provoke the throughput difference between SACK and FACK. But like the first FACK simulations shows, the fast recovery period is shorter.

A.3.4 Reno, SACK and FACK in a concurrent transfer with a traffic generator

Reno, SACK and FACK will in the two following simulations have to "compete" for bandwidth. Three FTP connections will have to use the same link between two routers. In figure 62 we present the topology for the two simulations. For each simulation we present parameters regarding delay, bandwidth, congestion windows and other parameters.

Simulation one: Lossy conditions with small delays The parameters used between source nodes and the router are the same, referred to as "source-router" parameters, and the parameters between the other router and the sinks are the same, referred to as "router-sink" parameters.

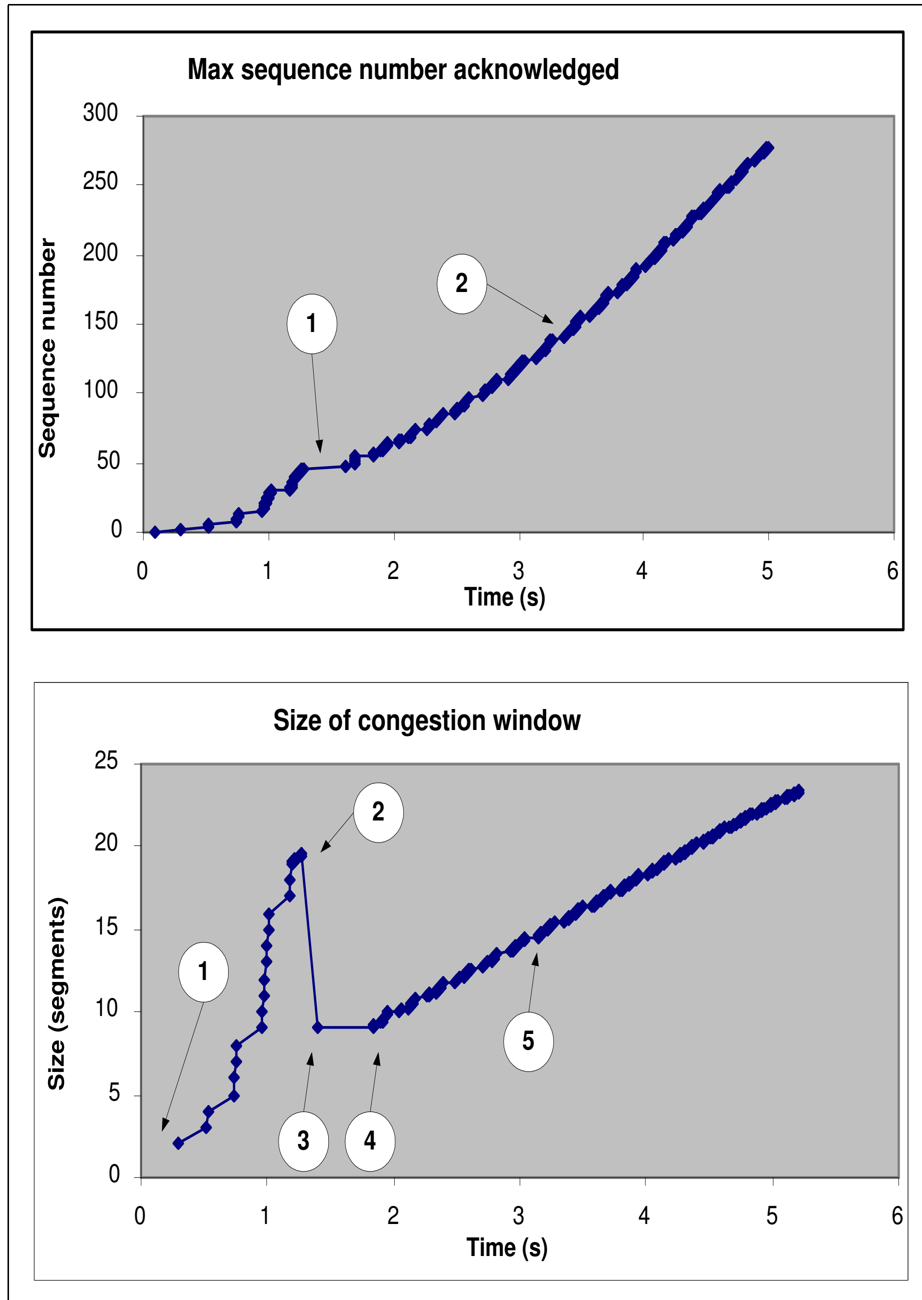

Figure 60: Two segment drops simulation with SACK TCP implementation

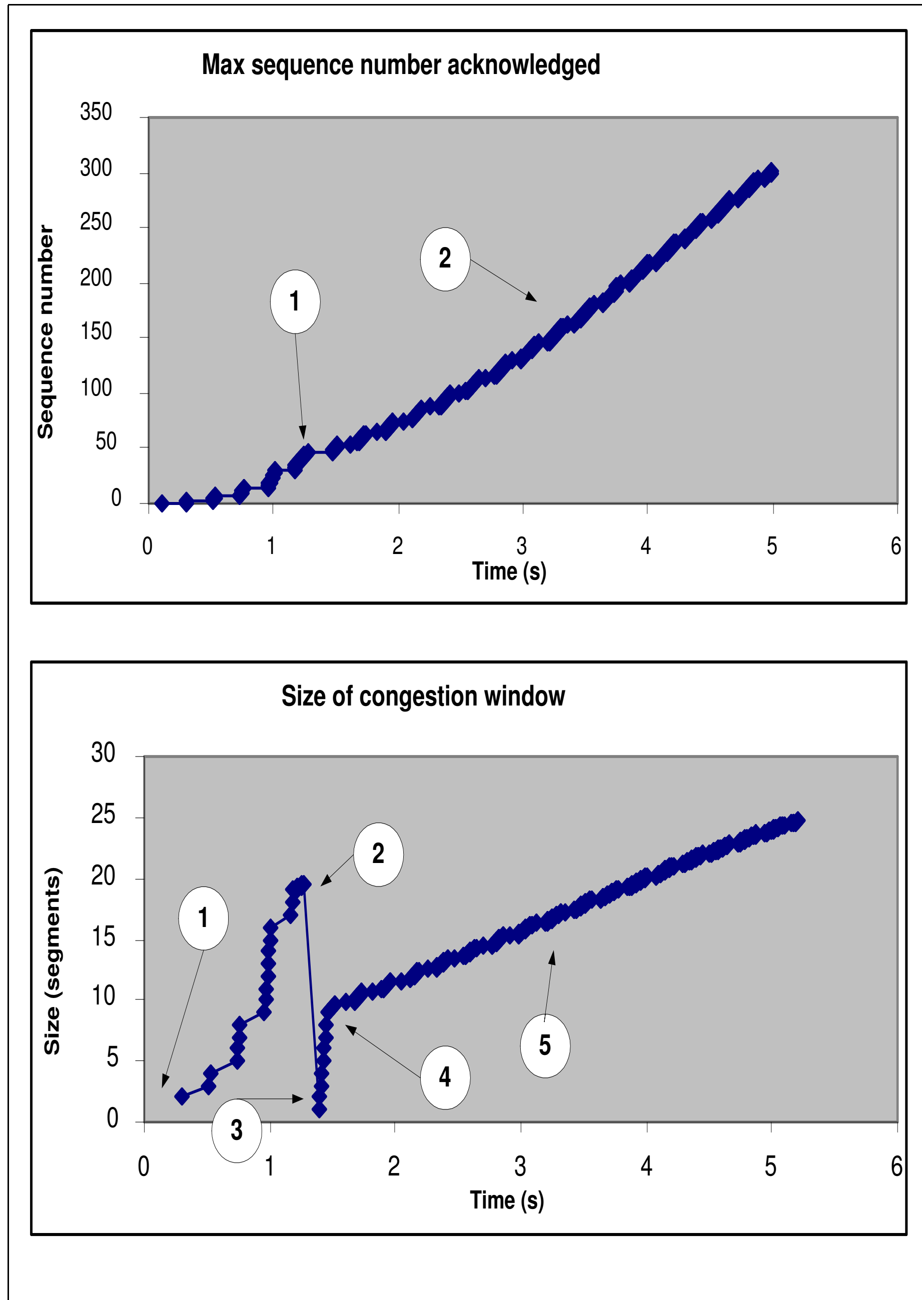

Figure 61: Two segment drops simulation with FACK TCP implementation

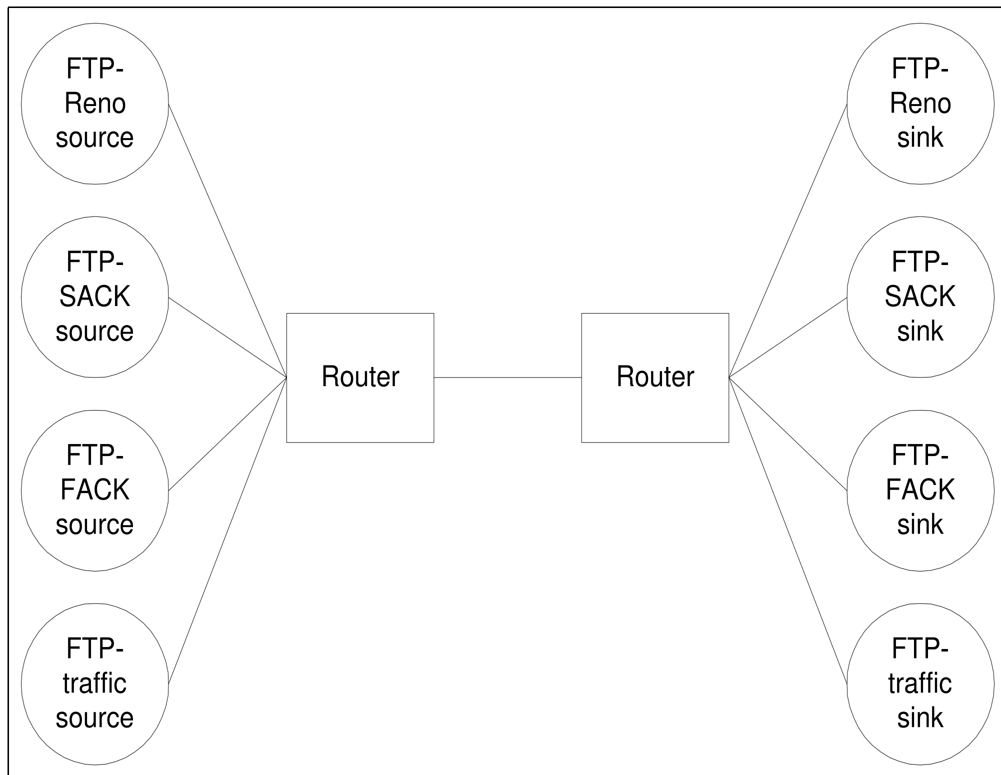

Figure 62: Reno, SACK and FACK simulation over the same link with a traffic generator

- Bandwidth source-router = 0.5Mb (Except for the traffic generator which has the value set to 1Mb)
- Bandwidth router-router = 100Mb
- Bandwidth router-sink = 0.5Mb (Except for the traffic generator which has the value set to 1Mb)
- Delay source-router = 50ms
- Delay router-router = 5ms
- Delay router-sink = 50ms
- Max buffer size source-router = 4 segments
- Max buffer size router-router = 4 segments
- Max buffer size router-sink = 4 segments
- Segment size source = 1024 bytes
- Initial threshold value source = 10 segments
- Initial source window buffer size = 19 segments
- Initial source congestion window size = 1 segment
- Maximum source congestion window size = 30 segments
- Simulation time = 20 seconds

The simulation lasts for 20 seconds. The results for the development of the congestion windows and the sequence numbers (figure 63) are presented. The simulation shows that FACK achieves much better throughput than both Reno and SACK. In the figure we see that SACK uses more time in fast recovery and therefore lags behind compared to FACK.

Simulation two: Lossy conditions with large delays Below we present parameters that have changed compared to the last simulation

- Bandwidth source-router = 10Mb
- Bandwidth router-router = 0.1Mb
- Delay router-router = 500ms
- Max buffer size router-router = 4 segments
- Simulation time = 60 seconds
- Bandwidth router-sink = 0.5Mb

The delay between the two routers are increased by the factor of 100 since the last setup. In addition we have shrunk the available buffer size in the routers. This setup should lead to very bad throughput because of long round trip time and many lost packets. Figure 64 shows the results. An interesting observation is that SACK performs worse than Reno. We suspect this is because Reno was "luckier" and was able to use the small router buffers more that SACK. This is just speculations and would have to be further investigated.

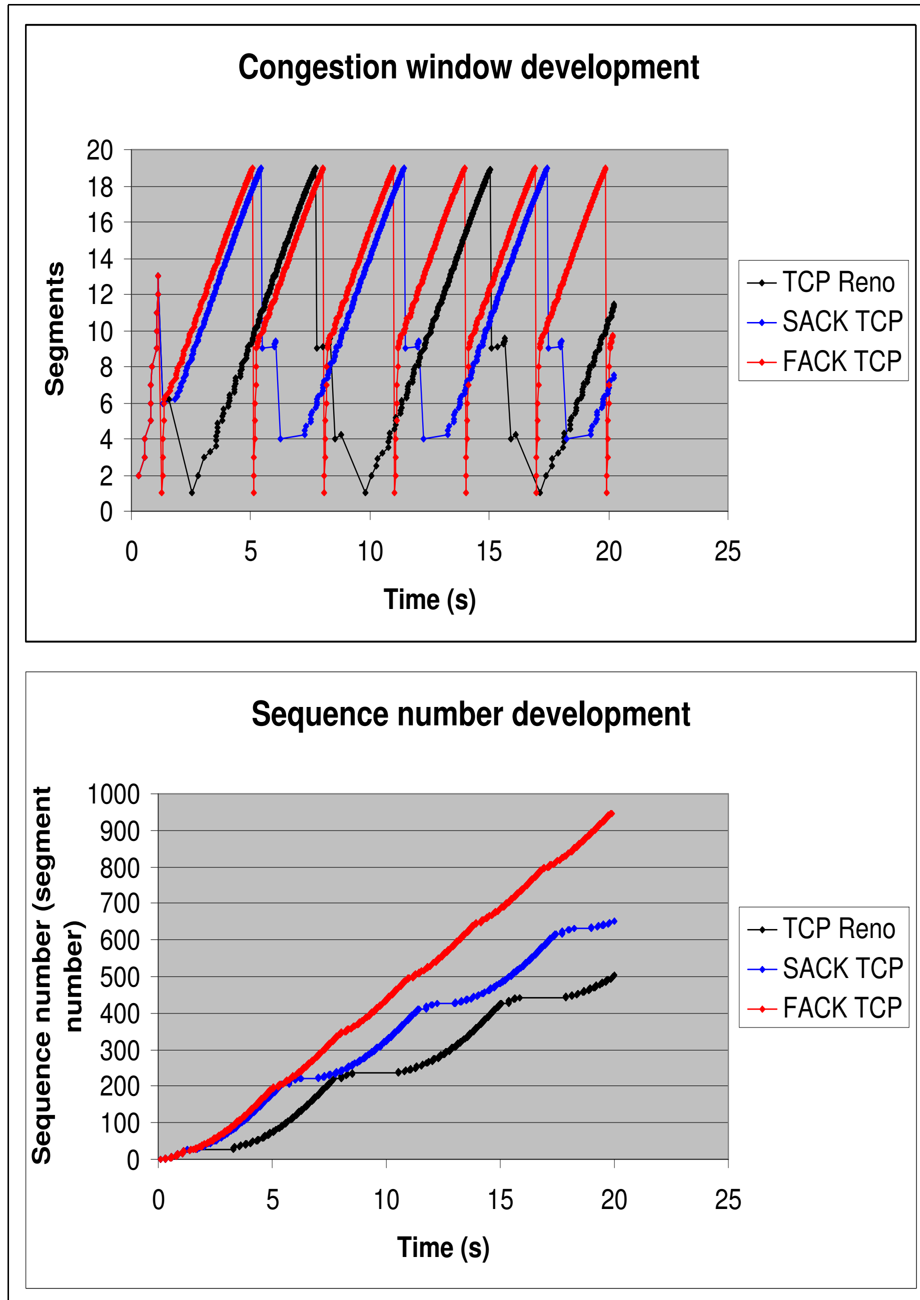

Figure 63: Reno, SACK and FACK congestion window and sequence number development

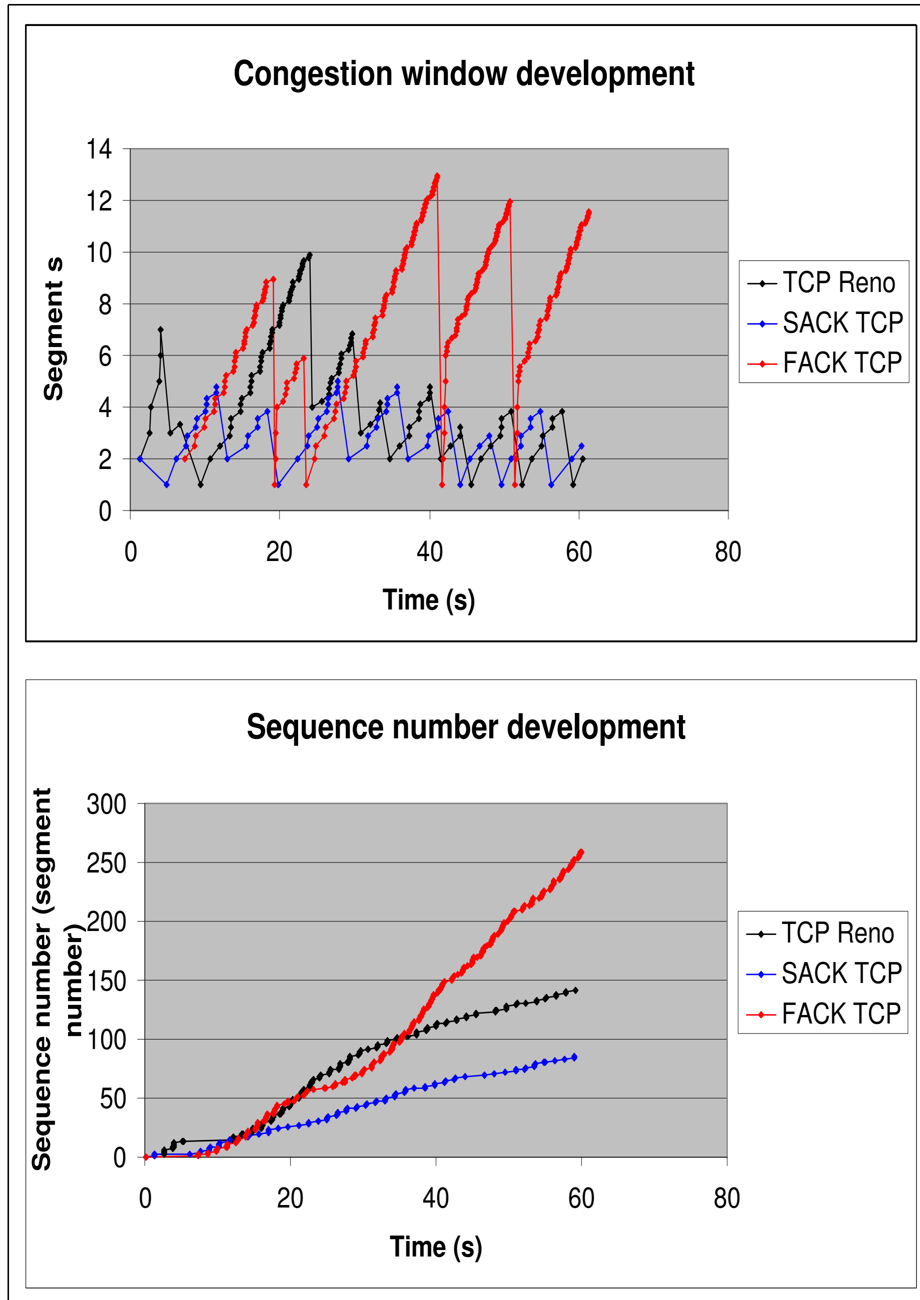

Figure 64: Reno, SACK and FACK congestion window and sequence number development

A.4 Conclusion

The simulations described in this paper shows that FACK is superior to SACK, while Reno mostly lags long behind. (Except in the very last simulation).

But we would like to repeat the two questions Jitendra D. Padhye asks in the conclusion part of his Ph.D thesis [27]:

- In an environment, where the total available bandwidth remains the same, will widespread replacement of TCP-Reno by TCP-SACK (or TCP-FACK) result in increased packet loss rate ?
- If the answer to the above question is affirmative, then what is the benefit of widespread deployment of TCP-SACK (or TCP-FACK) ? Would the send rate of TCP-SACK (or TCP-FACK) have less variance compared to TCP-Reno ?

References

[1] M. Allmann, V. Paxon, and W. Stevens. Tcp congestion control. RFC 2581, April 1999.

[2] Mark Carson. Nist net. obtain via: http://snad.ncsl.nist.gov/itg/nistnet/.

[3] Mark E. Crovella and Azer Bestavros. Self-similarity in world wide web traffic. evidence and possible causes. Technical report, Boston University, 1996.

[4] S. Deering and R. Hinden. Internet protocol, version 6 (ipv6) specification. RFC 2460, December 1998.

[5] Mark Gaynor. Proactive packet dropping methods for tcp gateways. Technical report, Harvard University, November 1996.

[6] Gunnar Gudmundsson. Design of a multiformat capable cache for video streaming. Technical report, Fachbereich Electrotechnik und Informationstechnik, Darmstadt University of Technology, September 2000.

[7] M. Handley and V. Jacobson. Sdp: Session description protocol. RFC 2327, April 1998.

[8] M. Handley, J. Padhye, and S. Floyd. Tcp congestion window validation. RFC 2841, June 2000.

[9] M. Handley, C. Perkins, and E. Whelan. Session announcement protocol. RFC 2974, October 2000.

[10] Mark Handley, Jitendra Padhye, Sally Floyd, and Jörg Widmer. Tfrc specification. Technical report, Universitet Mannheim, Juli 2001.

[11] V. Jacobsen. Congestion avoidance and control. In proceedings of the ACM SIGCOMM '88, pages 314-329, August 1988.

[12] V. Jacobsen, R. Braden, and D. Borman. Tcp extensions for high performance. RFC 1323, May 1992.

[13] Shudong Jin, Liang Guo, Ibrahim Matta, and Azer Bestavros. Tcp-friendly simd congestion control and its convergence behavior. Technical report, Computer Science Department, Boston University, May 2001.

[14] Brian Kantor and Phol Lapsley. Network news transfer protocol. RFC 977, February 1986.

[15] Martin Karsten. http://dmz02.kom.e-technik.tu-darmstadt.de/rsvp/.

[16] Martin Karsten. http://www-iepm.slac.stanford.edu/pinger/.

[17] K. Klaffy, Greg Miller, and Kevin Thompson. The nature of the beast: recent traffic measurements from an internet backbone. Technical report, Caida, MCIv/BNS, 1998.

[18] Eddie Kohler, Mark Handley, Sally Floyd, and Jitendra Padhye. Datagram congestion control protocol (dccp). Technical report, The ICSI Center for Internet Research, June 2002.

[19] C. Krasic and J. Walpole. Priority-progress streaming for quality-adaptive multimedia. Technical report, In ACM Multimedia Doctoral Symposium, October 2001.

[20] Charles Krasic and Jonathan Walpole. Priority-progress streaming for quality-adaptive multimedia. Technical report, Oregon Graduate Institute, 2001.

[21] Charles Krasic, Jonathan Walpole, Kang Li, and Ashvin Goel. The case for streaming multimedia with tcp. Technical report, Oregan Graduate Institute, March 2001.

[22] Charles Krasic, Jonathan Walpole, Kang Li, and Ashvin Goel. Supporting low latency tcp-based media streams. Technical report, Oregan Graduate Institute, May 2002.

[23] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanov. Tcp selective acknowledgment options. RFC 2018, October 1996.

[24] S. McCanne and S. Floyd. ns-lbnl network simulator. obtain via: http://www-nrg.ee.lbl.gov/ns/.

[25] Mike Muuss and Terry Slattery. ttcp. obtain via: http://renoir.csc.ncsu.edu/ttcp/.

[26] Jitendra D. Padhye. Towards a comprehensive congestion control framework for continuous media flows in best effort networks. Technical report, Department of Computer Science, University of Massachusetts, March 2000.

[27] Jitendra D. Padhye. Towards a comprehensive congestion control framework for continuous media flows in best effort networks. Technical report, Department of computer science, University of Massachusetts, March 2000.

[28] Vern Paxon and Sally Floyd. Why we don't know how to simulate the internet. Technical report, Lawrence Berkeley National Laboratory,University of California, 1997.

[29] J. Postel. User datagram protocol. RFC 768, August 1980.

[30] J. Postel. Transmission control protocol. RFC 793, September 1981.

[31] J. Postel and J. Reynolds. File transfer protocol. RFC 959, October 1985.

[32] K. Ramakrishnan, S. Floyd, and D. Black. The addition of explicit congestion notification (ecn) to ip. Technical report, EMC, September 2001.

[33] R.Braden, L.Zhang, S.Berson, S.Herzog, and S.Jamin. Resource reservation protocol (rsvp). RFC 2205, September 1997.

[34] J. Rosenberg, H. Schulzrinne, and G. Camarillo. Sip: Session initiation protocol. RFC 3261, June 2002.

[35] H. Schulzrinne, S. Castner, R. Frederick, and V. Jacobsen. Rtp: A transport protocol for real-time applications. RFC 1889, January 1996.

[36] H. Schulzrinne, A. Rao, and R. Lanphier. Real time streaming protocol (rtsp). RFC 2326, April 1998.

[37] Jorg Widmer, Robert Denda, and Martin Mauve. A survey on tcp-friendly congestion control (extended version). Technical report, University of Mannheim, 2001.

[38] Jorg Widmer, Robert Denda, and Martin Mauvre. A survey on tcp-friendly congestion control (extended version). Technical report, Universitet Mannheim.

[39] Jorg Widmer and Mark Handley. Extending equation-based congestion control to multicast applications. Technical report, Universitet Mannheim, AT&T Center for Internet Research at ICSI (ACIRI).

[40] Michael Zink, Carsten Griwodz, A. Jonas, and Ralf Steinmetz. Lc-rtp (loss collection rtp): Reliability for video caching in internet. Technical report, Proceedings of the Seventh Internation Conference on Parallel and Distributed Systems: Workshops, pages 281-286, July 2002.

[41] Michael Zink, Carsten Griwodz, Jens Schmitt, and Ralf Steinmetz. Scalable tcp-friendly video distribution for heterogeneous clients. Technical report, Darmstadt University of Technology.

[42] Michael Zink, Carsten Griwodz, Jens Schmitt, and Ralf Steinmetz. Scalable tcp-friendly video distribution for heterogeneous clients. Technical report, KOM, Darmstadt University of Technology, IFI, University of Oslo, 2002.

[43] Michael Zink, Carsten Griwodz, and Ralf Steinmetz. Kom player - a platform for experimental vod research. Technical report, Darmstadt University of Technology, German Nathional Research Center for Information Technology, 2001.