NTNU

Norwegian University of
Science and Technology

Network based QoE Optimization for "Over The Top" Services

Kristian Haugene
Alexander Jacobsen

Master of Telematics - Communication Networks and Networked Services (2 year)
Submission date: June 2011
Supervisor: Poul Einar Heegaard, ITEM
Co-supervisor: Bjørn Villa, ITEM

Norwegian University of Science and Technology
Department of Telematics

Problem Description

A major challenge in delivering Over The Top services with high expectation to the Quality of Experience is that the service provider cannot rely on that the network operators can or will control and prioritize the traffic according to the service providers needs. Therefore it is of great interest to investigate potential mechanisms that can be used to control traffic types outside the network operator domain. This assignment will take a closer look at the gateway located at customer premises, which has capabilities that may control and prioritize the traffic types.

Using an already established home gateway testbed based on Click Modular Router[1][2] the students will investigate and implement new components for the Knowledge and Action Planes. The Knowledge Plane components will specifically focus on the characteristics and behavior of the terminal equipment, the operating system and its applications[3], while the Action Plane components should address a selected set of functions that are required to support the optimization process. The modeling and description of a general QoE reasoning process is also of great interest.

Assignment Given: 13th of January 2011

Supervisor: Bjørn J. Villa

Abstract

This report focuses on the quality aspects of media delivery over the Internet. We investigate the constructs of Knowledge Plane, Monitor Plane and Action Plane as controlling functions for the Internet. Our goal is to implement functionality for monitoring services in a home network, allowing the router to reason and take actions to obtain an optimal traffic situation based on user preferences. The actions taken to alter ongoing traffic are implemented in a modular router framework called Click. We will use this router to affect the media stream TCP connections into behaving in accordance with the networks optimal state. New features are implemented to complement the functionality found in Click, giving us the tools needed to obtain the wanted results.

Our focus is on adaptive video streaming in general and Silverlight Smooth Streaming in particular. Using custom Silverlight client code, we implemented a solution which allows the applications to report usage statistics to the home gateway. This information will be used by the home gateway to obtain an overview of traffic in the network. Presenting this information to the user, we retrieve the user preferences for the given video streams. The router then dynamically reconfigures itself, and starts altering TCP packets to obtain an optimal flow of traffic in the home network.

Our system has been implemented on a Linux PC where it runs in its current form. All the different areas of the solution, ranging from the clients, router, Knowledge Plane and traffic manipulation elements are put together. They form a working system for QoE/QoS optimization which we have tested and demonstrated. In addition to testing the concept on our own streaming services, the reporting feature for Silverlight clients has also been implemented in a private build of TV2 Sumo. This is the Internet service of the largest commercial television station in Norway. Further testing with the TV2 Sumo client has given promising results. The system is working as it is, although we would like to see more complex action reasoning to improve convergence time for achieving the correct bit rate.

Foreword

This report describes the work we have done as part of our master thesis at NTNU. The work was done at Faculty of Information Technology, Mathematics and Electrical Engineering under the Department of Telematics. At the department we were guided by our supervisor Bjørn Villa and professor Poul Heegaard.

We would like to thank them for the help along the way and for showing initiative and interest in our work. This has motivated us greatly.

In addition to our mentors we have had support from a commercial actor. We would like to thank TV2 for allowing us access to their Smooth Streaming service TV2 Sumo. This service has been our main source of testing the Silverlight Smooth Streaming platform. In addition to this, they compiled our proposed Silverlight code into their own player for live testing. This custom player which is running our code has been a very important contribution as to demonstrating our solution.

Kristian Haugene & Alexander Jacobsen
June 2011

Contents

List of Figures

List of Tables

Acronyms

ACK Acknowledge

AP Action Plane

CPU Central Processing Unit

CSS Cascading Style Sheet

CTCP Compound Transmission Control Protocol

HTTP Hypertext Transfer Protocol

IIS Internet Information Services

ISP Internet Service Provider

KP Knowledge Plane

MIT Massachusetts Institute of Technology

MP Monitor Plane

NAT Network Address Translation

NIC Network Interface Card

OS Operating System

QL Quality Level

QoE Quality of Experience

QoS Quality of Service

RED Random Early Detection

RTCP Real-Time Transport Control Protocol

RTP Real-Time Transport Protocol

RTT Round-Trip time

RWIN Receive window

TCP Transmission Control Protocol

TTL Time To Live

VoIP Voice Over IP

webUI Web User Interface

XML Extensible Markup Language

Chapter 1

Introduction

Media streaming over the Internet has seen a tremendous growth and development over the past decade. YouTube was established in February 2005, and was at the end of 2006 sold to Google for $1.65 billion[4]. Early 2007, YouTube alone comprised approximately 20% of all HTTP traffic, or nearly 10% of all traffic on the Internet[5]. In December 2005, YouTube had 8 million views per day, and in May 2010 it exceeded 2 billion views per day[6]. Add movie rentals and music streaming by services like Voddler, Spotify and Wimp and it's clear that the Internet is receiving large amounts of traffic from media streaming. The Voddler movie rental service is illustrated in figure 1.1.

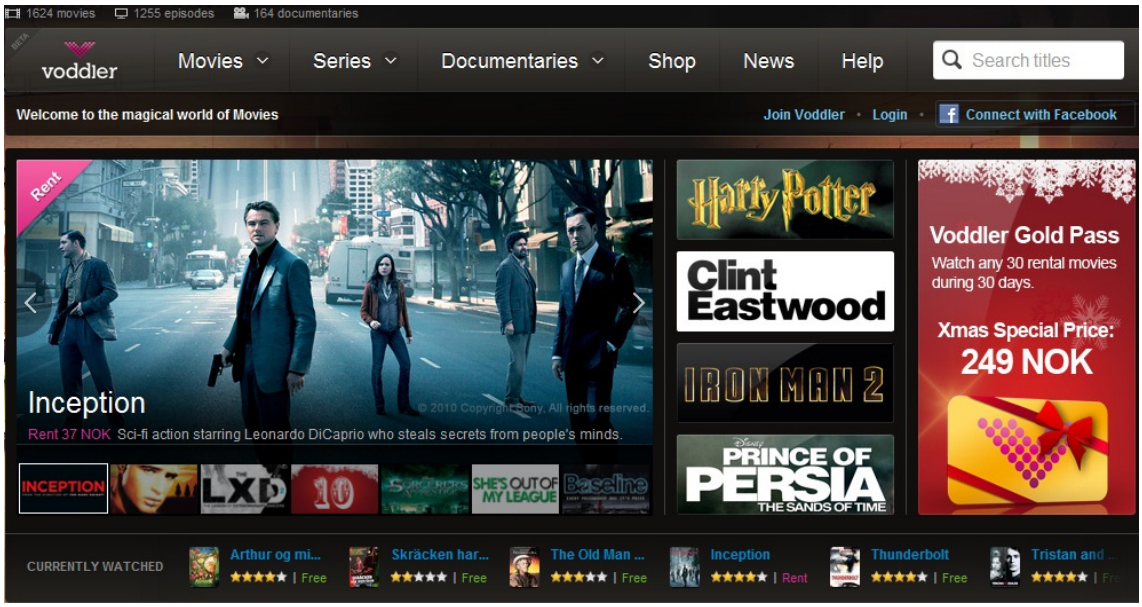

Figure 1.1: Voddler is an example of service providers that sell streaming media over the Internet. To attract paying customers they need to deliver high quality video over a best effort network.

With this growing trend in real time streaming media, the Internet is put to the test. The Internet is a dumb network with intelligence at the edges. By this we

mean that the Internet only forwards packets as directed by the end hosts. The Internet does not differentiate between a billion spam mails and a corporate video conference. If you are having a video conference with a business associate or a loved one at the opposite side of the globe, your communication can be disrupted by a swarm of spam mail.

This is in essence the topic for this thesis. How can we influence the Internet so that certain data gets increased priority, and therefore better Quality of Service (QoS)? The Internet today works, some might say. We can watch videos from YouTube and make calls over Skype. So why do we need this extra control?

We're rapidly moving towards the "everything over IP" world[7]. The new and brave world; where telephony, video telephony, video on demand, television, music and other media will be carried over the same network. We are getting closer each year and the amount of data transmitted is extreme. Figure 1.2 shows the usage statistics from Akamai on a Sunday evening. Akamai[8] is a major provider of content on the web. Akamai host amongst others TV2 Sumo, which is the Internet service of the largest commercial television station in Norway.

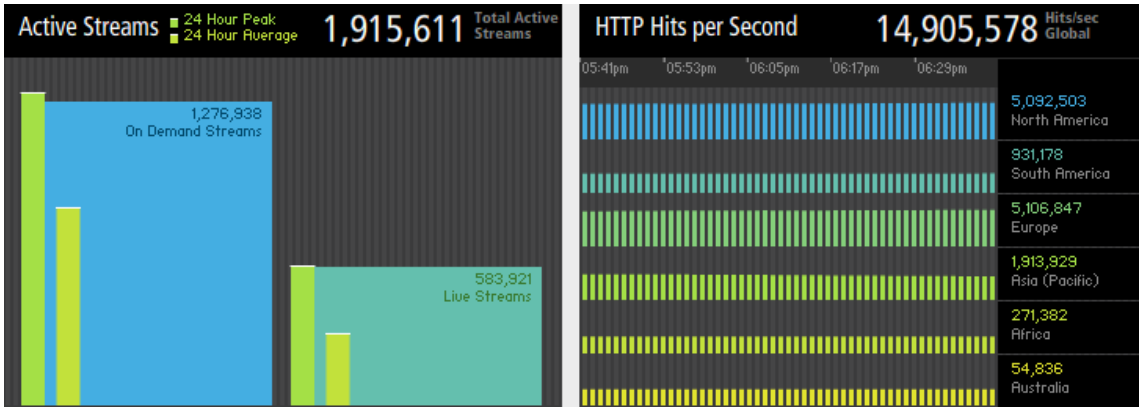

Figure 1.2: This figure is a screen dump of Akamai streaming statistics. At the time of this screen dump, a Sunday evening, they had in excess of 1.9 Million active streams and over 14.9 Million HTTP hits per second. According to their web pages they handle 20% of the worlds total web traffic.

With everything over the same network we need to exercise control over what gets priority. We can accept some quality degradations when we use Skype, and we are used to allowing hiccups in web telephony services because they are cheap. However, when our living room television cannot deliver the news uninterrupted, that will be a problem. This is why we work towards a future Internet where the network itself can make decisions based on the content it's transmitting. This way the network could provide high quality uninterrupted service to voice communications and television even if the network is experiencing congestion. During congestion, voice and video communications are types of services that experience vast quality degradation. These types of services are more sensitive to delay and jitter than other types of traffic like web browsing and file transfers.

If the network was more intelligent and could do service differentiation based on content, we could allow higher utilization of links in the infrastructure. The current situation of our infrastructure is that it is scaled up to keep the link utilization and therefore the loss, delay and jitter down. Merging television and telephony to be carried over the Internet represent tremendous value to the service providers given that the network is able to successfully transmit with high quality and reliability.

We talk about this new Internet with service differentiation based on content. We envision everything over the same network without quality degradation for the user when we move away from dedicated television and telephony networks. Making the entire Internet compatible for such traffic handling will be a "work in progress" for many years. The complexity in control systems usually increases drastically with size, and we're talking about the Internet. This will not be easy.

As this transformation cannot be done in one simple act of revolution, the solution must be to start somewhere and continuously add layers of functionality. For our work we have chosen to examine traffic at the end user premises, that is the home gateway. The home gateway is often the bottle neck of bandwidth for a typical user. On the home network, several clients can be connected with gigabit Ethernet connections. The home gateway link towards the Internet is often below 10Mbps for a typical subscriber line. The competition for resources between clients in a home network often result in severe quality degradation. This is where our focus will be.

Most Internet subscribers have experienced a slow network caused by other traffic from a process sending or receiving large amounts of data. Peer-to-peer networking is extremely efficient in claiming the entire bandwidth of a subscriber link. While such traffic is active, keeping a Skype conversation or streaming video is relatively unbearable. This is an obvious case where traffic collides, and it's not hard to identify the problem and fix it. You'll simply have to do the downloading after finishing your conversation. If you cannot control the download, then you have a problem. Even though you only need 80Kbps to obtain a voice conversation, a peer-to-peer download running at 6000Kbps is making it impossible. This is when you would want the home gateway to recognize your traffic as more important, passing your traffic in front of the queues giving you quality as if you were the only one utilizing the link. The download would run at 5920Kbps instead of the 6000Kbps, which would give a time difference of 9 seconds for downloading a 500MB file. Instead of 11 min 6 seconds, the download would take 11 min 15 seconds. The positive effect is that you would have a voice conversation with excellent quality. Clearly it is worth it.

Another mechanism that has been introduced in streaming services with great success is adaptive streaming. A comparison between this technology and traditional streaming can be seen in figure 1.3. The concept and theoretical aspects will be introduced in this report. For now it suffices to say that it allows the client to dynamically change the video quality of an ongoing stream. This is done to adapt to changes in network conditions. This form of video streaming is increasingly popular and is the focus of our report. The authors of [9] argues that this streaming technology has its problems when it comes to competing traffic. Two clients streaming simultaneously will, according to their findings, cause frequent quality variations.

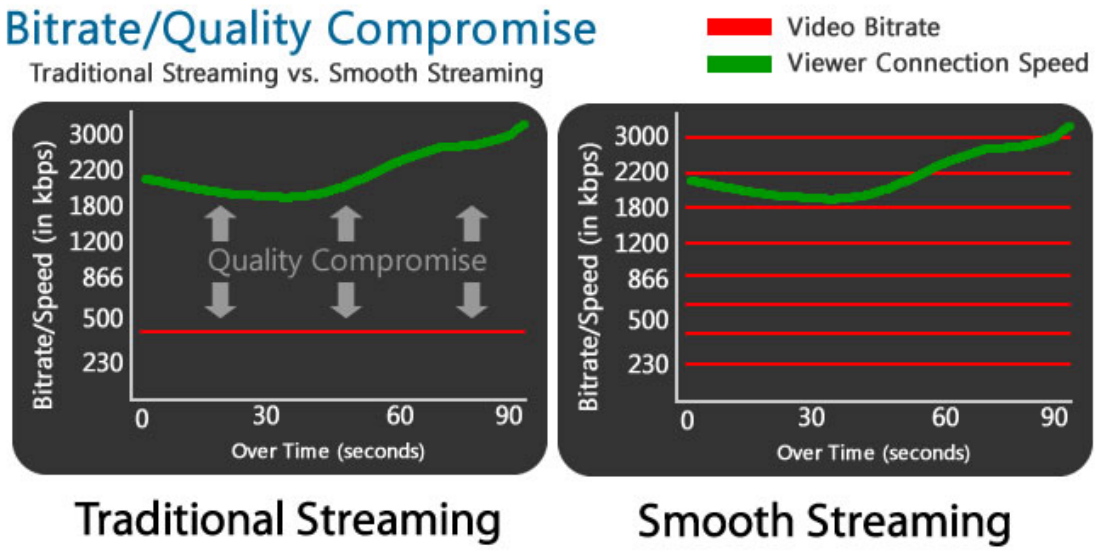

Figure 1.3: Traditional streaming offers a low quality video stream that all users have enough bandwidth to view. Adaptive streaming will automatically adjust on the fly to the viewers changing network conditions, calculating and delivering the best possible quality level.

We will in this report focus on delivering video streams in a home network using Silverlight Smooth Streaming technology. We will focus on the home gateway and its possible role in the provisioning of resources. In our previous work[10] we explored a similar solution where we dynamically configured a home gateway to set a strict limit on bandwidth for certain services. Setting a strict limit on bandwidth will cause excess data to be dropped, which is a wasteful way of controlling traffic. In this report we will propose less intrusive ways to control traffic rates on a per stream basis in the best interest of the network. This will be achieved by manipulating the transport protocol TCP, which is used in Smooth Streaming.

In order to be able to do such manipulation, the home gateway must obtain information about active video streams. In our previous work obtained such information by monitoring traffic by packet count and a recognizing a pre-defined list of video sources. Here we will explore a form of application involvement where the Silverlight clients communicate directly to the home gateway control function. The home gateway control will in turn use this information in addition to user preferences to obtain a stable streaming rate for the given client. Our results show that this form of control can be quite effective and with the information from the clients we can present an overview to the user. This way the user can monitor the home network and possible reasons for a low quality video stream might be visualized.

The goal of our work was to implement a complete testbed for this architecture. We have therefore implemented a Silverlight client that retrieves end host information and transmits this to the home gateway control function. The home gateway is implemented with custom tools to manipulate data traffic. The control

function includes a web user interface where the user can make decisions as to what quality the video should be delivered in. Using this information the home gateway control function will try to alter traffic flows to obtain the quality wanted by the user. Using our solution, when several clients are streaming video, we should be able to control what bit rate the streams should use by simply inputting it to the user interface. This should be regarded as a proof-of-concept solution for content aware home gateways with mechanisms for influencing quality of service.

Chapter 2

Background

2.1 Autonomic Access Networks

In this section we will explain the concept which our work is founded upon. The constructs of Knowledge Plane, Monitor Plane and Action Plane will be introduced and their functions explained. First, in section 2.1.1, we will present an overview of the network and in turn look at the elements it contains and interaction between them in sections 2.1.2 through 2.1.5. After this, a short presentation of our previous work and system is found in section 2.1.6. This is helpful background material for understanding the system we want to present here.

2.1.1 Overview

There have been several proposals for autonomic access networks. Amongst them are [11]. These solutions mainly consist of three layers of functionality. These layers are called Knowledge Plane(KP), Monitor Plane(MP) and Action Plane(AP). The idea is that the network through information gathering in MP should be able to interpret and classify current traffic patterns and possible complications. This reasoning process is done by the KP which allows the network to identify problems in Internet traffic. The KP is then able to use a diverse set of tools to resolve the problem it has identified. The action chosen by the KP is realized by the AP which applies a change in the network configuration. In figure 2.1 the placement of KP, AP and MP is shown in a distributed example where each node has its reasoning process. These nodes are able to communicate with a global KP that allows the network to define an optimized solution.

A scenario could be that the MP reports that a video stream is experiencing low throughput or high delay due to a congested link. The KP could then, through the AP, initiate counter measures in the network to guarantee a level of resources available to the service. The actions taken by the KP will vary according to where in the network the problem occurs. An example of such mechanisms can be adding of Forward Error Correction(FEC) packets in the core network[11]. At the edges, bursty traffic can be shaped by the home gateway to prevent it from causing lag spikes in a video stream.

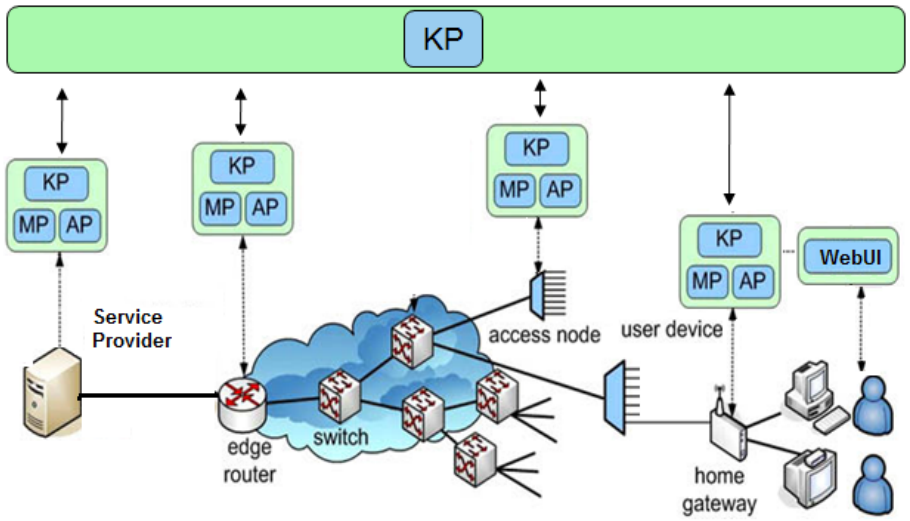

Figure 2.1: Complete architecture for a global autonomic network where information can be collected, distributed and acted upon. Our model includes a web user interface where we envision that users can influence the policies enforced at the edges of the network.

The webUI allows for interaction between the user and the network at the edges. Obtaining this information may allow the KP reasoning process to make optimized decisions.

There have been several contributions proposing design and architecture for the KP and its role in an autonomic network[11][12][13][14]. This work has been focusing on the intermediate nodes in the network and how a network as a whole can be tied together by a KP layer. These ideas and proposals together with future work in this area will define the structure of such a network. The design and implementation is, needless to say, a formidable challenge that will require optimized solutions in every level to assure the wanted functionality without extreme amounts of overhead.

In our previous work[10] we have been focusing on the home gateway. This is the node that connects users to the network and this node is of special interest to the network as a whole. Users are connected to the home gateway with Gbps capacity and a typical home gateway often serves several users simultaneously. The home gateway link to the network is normally in the order of some Mbps that is shared amongst the users. Web browsing and downloading emails are by nature bursty traffic and the user bandwidth towards the home gateway allows very sporadic flooding of the home gateway. This is a problem if the gateway simultaneously serves a video streaming service. Our previous work was therefore focused on shaping this other traffic to protect video streaming services from traffic bursts that will manifest as interrupted video for the user.

The following sections will present the KP, AP, MP approach to an autonomic network as well as describing the previous state of our system.

2.1.2 The Knowledge Plane

The Internet is a best effort stupid network with intelligence in the end-systems. It does not discriminate between packets and has, for the most part, no interest in what it's transporting. This simplicity has been an important factor in the success of the Internet. The IP protocol has taken its grasp on world communications.

Despite its success, we should always try to find new ways to improve the Internet. This is why the idea of a KP has been proposed[12]. The Internet transports data without knowing what it's transporting or why it's transporting it. If we could put another layer on top of the Internet, this layer would allow the Internet to know what it's asked to do. The proposal is therefore a distributed cognitive system that permeates the network, called the Knowledge Plane.

2.1.3 The Action Plane

When we design a system for optimizing traffic patterns through the Internet we need to, at some point, alter the traffic traversing it. Our goal is to make small adjustments to the ongoing flows that tilts the traffic conditions in favour of our services of interest. These services are mostly media streaming services, that is a category of services with higher requirements for throughput, delay and jitter than traditional web traffic as HTTP and email.

The AP is our tool to make these adjustments. Through it we aim to make dynamic changes to packets and flows in order to increase the overall QoE perceived by the user. The AP is the tool set of the KP. When the KP has obtained a picture of the network environment it will try to determine what could be done to improve the current state of the system. This could mean that a typical download should be delayed in order to allow a competing video streaming service better traffic conditions. If the KP decides that this would be valuable to the user, then it will signal the AP to take actions to achieve this. The AP is provided with the necessary parameters to identify the services in question and will then apply the changes. The AP therefore needs access to the packets flowing through the node, and it needs to be able to make changes to the packets or even dropping them.

2.1.4 The Monitor Plane

To be able to improve the network conditions for specific services we need to identify and obtain information about them. As nodes in the Internet do not have any information of what the packets they process contain, we need to achieve this by adding a separate service. This is a task for the MP.

The MP will provide the system with information of ongoing traffic. The KP will maintain an overview of the current traffic through the home gateway. By informing the MP of what traffic to look for, the MP can inspect packets and report back on the current state of given streams or even application specific information from within the data packets. This helps the KP to do correct interpretations of the traffic at any time and will make our home gateway content aware.

A MP component could also be active. Such a component could probe the operating system(OS), applications or network nodes in order to provide the KP with information. Any form of information gathering from the OS, its applications or the network, is the responsibility of the MP.

2.1.5 Interaction

The interactions between the KP, MP and AP are centered round the KP. The KP collects information through different channels and tries to identify actions that may be taken to optimize the current traffic situation. The KP can use the MP to intercept control information in certain transport protocols or provide overall traffic statistics. Using information from the MP and information provided from the user via the webUI, it can determine a proper action. This action is then communicated to the AP which initiates the actual traffic manipulation.

Results from the manipulation are monitored by MP and reported to KP which then needs to reason again to check if the wanted effect has been obtained. If it has, KP can go back to reading system information and reasoning. If or when a new imperfection is found, the process starts over again.

2.1.6 Previous Work

We implemented a functioning system for dynamic reconfiguration of the home gateway allowing the gateway to adapt to the current traffic pattern. This was done by the use of the Click[1] modular router, explained in section 3.3, for monitoring and routing of traffic. Java applications was reading and interpreting the behavior and configurations of the gateway and was responsible for the reconfiguration. Most of our effort was used to combine the tools presented to us by Click in a Java system and creating mechanisms for exchanging information.

Because of the amount of time put into creating a working system, our methods for shaping and controlling of traffic flows were quite simple. Traffic was classified according to user preferences and put in several queues that were shaped by a mechanism that most accurately could be described as a leaky bucket. This resulted in an even flow of traffic and heavy bursts were punished by packet dropping. The packet dropping was implemented using Random Early Detection(RED). This algorithm randomly drops packets from a queue when the queue is approaching its capacity. Since most of the traffic in the Internet is sent over TCP, a lost packet will result in lowered transmission rate. This lowered the rate to our target rate set by the leaky bucket.

This solution has its pros and cons. The tradeoff was simplicity versus waste. The traffic was successfully shaped and we observed an increase in video streaming quality when the system was active. The negative effect of this approach was the high amount of waste that appears when the leaky bucket fills up and packets are dropped. As previously described, the bandwidth from the home gateway towards the users is definitely not the bottleneck. Discarding packets at the home gateway that have already been sent through the network will trigger a retransmission, which

results in increased bandwidth use to send the same amount of information. This is quite the opposite of our overall goal which is to optimize the transfer of data. A more appropriate approach would be to forward the packet and signal the sender to decrease its transmission rate. That way we could decrease waste of bandwidth and still be able to control the flows.

Our system included a KP implemented in Java, a MP that was a Java-Click hybrid and an AP that were realized by the Click router. In addition to these layers, two entities were added. These were entities we found could improve the KP decision making and in that effect further optimize the system. The first addition to the classic three layer architecture was the webUI that allowed the user to input its preferences. This allowed the KP to differentiate between video streams according to user preferences and served as a visual tool presenting traffic information to the user. The second entity was a service registry web service. Our vision was that service providers could register their services and relevant information in a common registry. This allows the KP to easily identify services and to present better information to the user.

Our KP would gather information from the MP in order to get a picture of the current traffic. In addition, information from the service registry would be collected to create a lookup table for recognizing and classifying services. Communications between KP and the external service registry was done by the use of web services, which allows for a standardized way of machine to machine interactions. The list of registered services was presented to the user for his or her input. This information was then aggregated so that the KP knew what services the user preferred and which services had an active flow. Using this information, the KP tried to find an optimal allocation of resources and used the AP to enforce them. Statistics were collected and reported to the webUI which presented the user with a graphic representation of traffic history and identified flows.

2.2 A Brief Introduction to Video Streaming

This section provides a brief introduction to video streaming and underlying concepts. We also present the most widespread transport protocols used in media streaming, beginning with presenting media streaming in general.

Video streaming gives clients the opportunity to access video and audio content over the Internet. The streaming content is made available by service providers with the use of streaming servers. The distributed content is either real-time or on-demand. Real-time is typically live, while on-demand is recorded content. Examples of real-time services are television broadcasting and video conferences, while on-demand services are movie rentals and YouTube videos. We will mainly focus on live streaming in this thesis, because it is the most demanding service related to network resources. In addition, live content is more sensitive to fluctuations in the network such as bandwidth and delay.

An example of a video streaming service in use today is the television provider TV2 in Norway which offer streaming services through their TV2 Sumo service. Other similar services are YouTube, NRK and Netflix. We believe that media

streaming has achieved its popularity because it gives users a more interactive experience compared to downloading a full length video and then viewing it. This interactivity can be an interface for changing channels, the opportunity to chat with other viewers or view previously shown television shows, etc. Another advantage of media streaming is the "view and waste" property where the client views the content without storing it locally. This enhances protection of copyrights and reduces the problem of redistribution of media content through file sharing. However, providing extra interactivity through streaming have costs relating to resource consumption and high QoE/QoS demands.

2.2.1 Traditional Streaming Protocols

The Real-Time Transport Protocol provides end-to-end network transport functions for real-time multimedia content (audio, video, etc.). RTP is only a transport protocol and has no means of providing interactivity to the client. The interactivity gives users the opportunity to play, pause and fast forward, which is provided by RTSP. Hence, RTSP can be considered an extension of the RTP protocol. RTSP is used to set up the session between the server and a client. This session between server and client is maintained until the media file transfer is complete. This differs from HTTP streaming which is stateless. HTTP streaming is presented in section 2.2.3. An example of streaming using the RTP combined with RTSP is shown in figure 2.2.

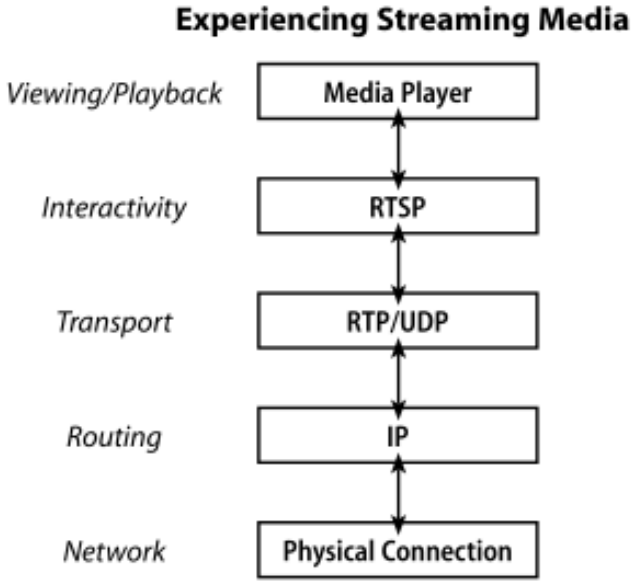

Figure 2.2: Protocol stack example when streaming using RTP/RTSP.

RTP was published as a standard by IETF in 1996 (RFC 1889) and an enhanced version followed in 2003 (RFC 3550). It provides services such as payload type

identification, sequence numbering, time stamps and delivery monitoring. However, RTP does not provide a reliable delivery or provide QoS guarantees, but relies on lower layer protocols. RTP can be used in combination with for example UDP or TCP. Usually UDP is the preferred protocol. This gives the server the opportunity to push traffic at a bit rate decided at the application level. The server can then control the streaming session using reports generated at the client and transported to the server using RTCP[15].

As mentioned, RTP uses the RTCP for probing for feedback on the quality of transmission sessions. RTCP is based on periodic transmission of control packets. The feedbacks which can be gathered from using RTCP are number of lost packets, jitter and RTT. These parameters can then be used by RTP to perform adaptations to the quality or transmission rate to be used for the data distribution [11].

The reason for RTP not being as popular as expected is that it runs on special ports which could be blocked by firewalls and NATs. Another disadvantage is that RTP streaming needs special servers configured for this technology. To enhance the streaming technology the next evolution step was to use HTTP instead of RTCP. HTTP is stateless and runs on port 80 which is supported by almost every device connected to the Internet. HTTP streaming will be introduced in the section 2.2.3.

2.2.2 TCP in Media Streaming

It was believed that TCP did not have the properties needed in order to provide media streaming because of the contribution to extra delay when retransmission is activated. Another reason for the scepticism in the streaming capabilities of TCP is the congestion avoidance algorithms used. It was believed that congestion avoidance would contribute to large fluctuations in the video bit rate and viewers QoE. Hence, there were proposed several TCP friendly protocols with smoother handling of congestion avoidance. It has been shown that TCP can be used for video streaming. This is because the real-time requirements for video are less strict than for VoIP communication. Two main methods of dealing with the retransmission issue are buffering and bit rate adjustment. A buffer at the client side will handle the extra delay from retransmissions. Offering the video in a set of different bit rates will allow the client to choose a bit rate best suited to its traffic condition. This way the client can handle large bit rate fluctuations, and in combination with a buffer also deal with retransmission delay. Using custom players we can obtain smooth transitions between bit rates. Having a smooth transition to a lower bit rate gives a higher QoE than if the video was to stop playing in order to fill the buffer again. These properties in combination with TCP functions such as fairness is why TCP is being used as the transport protocol of choice for video streaming [16].

2.2.3 Progressive Download

There are several solutions to providing media streaming. One is to transfer the video content from the streaming server to the client using a TCP session at the highest possible bit rate. A buffer is then used at the client-side which starts play-

ing the video content after some byte threshold in the buffer is reached[15]. This will reduce the clients waiting time, compared to downloading the full length video file, but it will not perform well under fluctuating network conditions. This can be prevented using large buffers, but the client waiting time until it starts playing the video content will approach the video download time, which again reduce interactivity. This technique is referred to as progressive download and is used by services such as YouTube, Vimeo and MySpace[17].

Progressive download uses HTTP for session initialization. Traversing firewalls is one problem in using RTP/RTCP[18]. HTTP is among the most widely used protocols in the Internet. Nodes will therefore recognize HTTP traffic as normal web traffic, which eases the traversal of firewalls. It is hard to adapt the network to media streaming, instead we can use HTTP and adapt media streaming to the network.

One disadvantage of this technique is that if the viewer decides to stop watching the video half way and the full length video is contained in the buffer. Then the unused part of the video has been transmitted over the network and contributes to a significant amount bandwidth waste [17]. Another disadvantage is a lack of the ability to adapt to current network state. Consider a scenario where the network traffic approaches the saturation point, then this type of streaming technology has no means of reducing the bit rate at which it is distributed. The result is that the user will experience hiccups in the viewing session which is considered a severe drop in QoE.

2.2.4 Single Rate Adaption

A development in traditional streaming is to determine the user bandwidth in the startup of the streaming session. This allows for choosing a video quality for this specific user. This technique is used by the Norwegian television provider NRK and others. Using the bandwidth measurements, a static bit rate is chosen for the client. Some fluctuations in the network conditions can be handled by using this technology in combination with a client side buffer. This approach performs well under stable network conditions, but it does not cope well with large fluctuations in network conditions. This technique can be referred to as single rate adaption[19]. An enhancement to this is the concept of self-adaptive streaming which will be presented in the following section.

2.2.5 Self Adaptive Streaming

Self-adaptive streaming is a concept developed for handling fluctuations in a network environment. In general the streaming server advertises a set of available streams with distinct bit rates and other relevant parameters to the client, using a manifest file written in a markup language like XML. The streaming server achieves different bit rates by dividing a video file into small segments (some seconds long) with different resolution, hence file size. The client will then monitor network parameters such as display resolution, CPU usage, available bandwidth, etc. Based on these

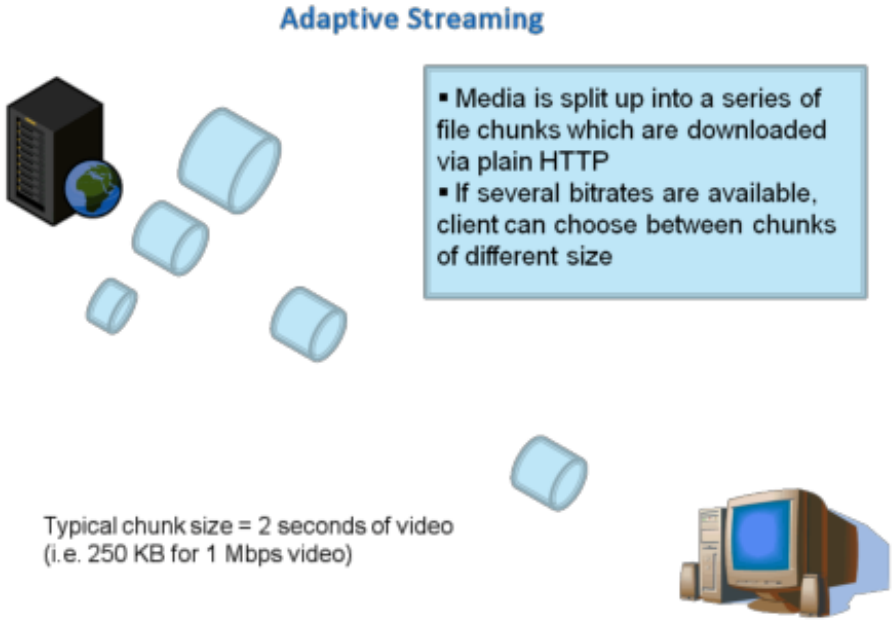

Figure 2.3: Example showing adaptive streaming using HTTP [17].

parameters the client will use HTTP GET requests to retrieve segments with the most appropriate bit rate in accordance to the current network conditions[15]. This server-client signaling is presented in figure 2.4.

The network conditions are monitored periodically to detect fluctuations and the switching between different bit rates is done seamlessly. It is believed that this is the technique of choice without doing changes to the current Internet architecture. Another advantage is that by using adaptive streaming there is a possibility to constrain the buffer size and therefore reduce the wasted bandwidth if a client decides to stop watching the video half way in. Wasted buffering was presented as a disadvantage with progressive download in section 2.2. This technology can adapt to different users with different connections to the Internet. All the users can view the media without the need for the media itself being scaled for the user with the lowest bandwidth. Hence, a client with high bandwidth can view a video with high quality, while clients with low bandwidth can view videos with lower quality[17].

A negative property of adaptive streaming is that there are no standards for this technology, but there is ongoing standardization work. This has resulted in an IETF draft[20]. Some examples of proprietary solutions using this HTTP streaming are Adobe Dynamic HTTP Streaming, Apple HTTP live streaming and Microsoft Smooth Streaming. Another property lacking in HTTP streaming is the support for multicast. By using multicast video content can be transported to central points in the network and clients can connect to these points on demand. This will enhance the network usage because only one video stream is sent from the streaming server to the multicast point. Then there is no need for having one end-to-end unicast stream for every client.

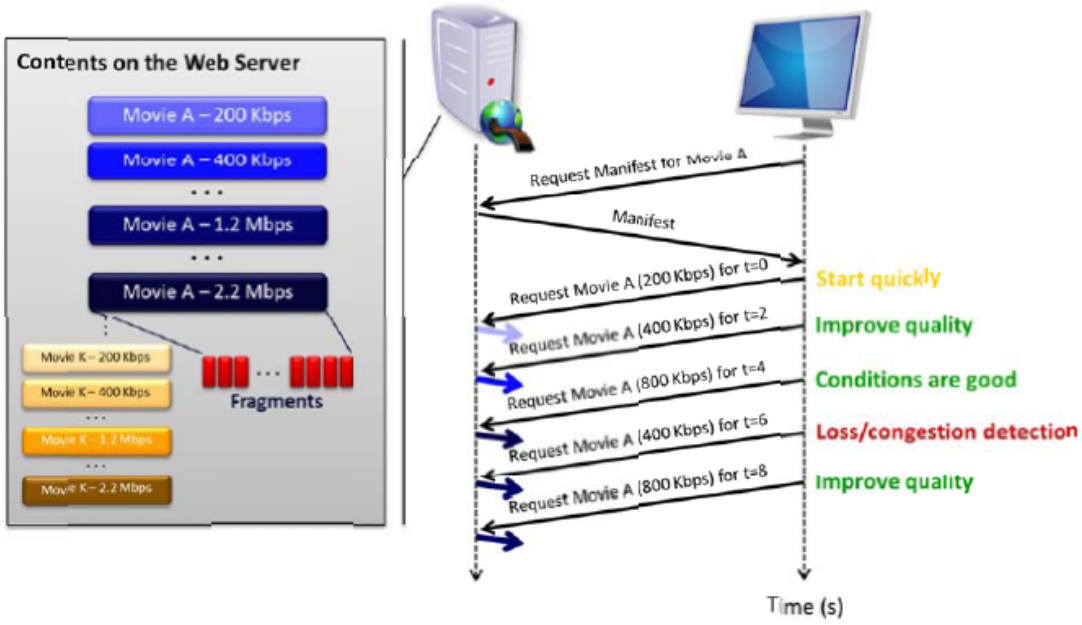

Figure 2.4: Example showing adaptive streaming HTTP server-client signaling. At first the client requests the manifest file, and then the movie transfer starts. At time unit 6 some loss/congestion is detected and the client requests a segment with lower bit rate. At time unit 8 the loss/congestion is no longer present, and a segment with higher quality is requested. [15].

The lack of multicast support is often countered by the use of content delivery networks. The media is sent to servers closer to the end users, and all end users closest to this server will retrieve their data from that server. Properties from multicast are then imitated by network nodes. Multicast is built on a different approach and a direct comparison is not possible. From the perspective of the network both these approaches allows for data to be transmitted down through a hierarchy. Their differences apart, these methods are putting the server load on the closest node. In addition to this, proxy caches are already in place all across the Internet. These caches tend to cache smaller files that are requested frequently. As more and more clients are connected and watching a particular video, the caches will recognize this and cache the media segments[21]. In essence these proxies become an extension of the content delivery network. The service providers will benefit from these caches and as this caching happens without any request for it by the provider, this is a free extension to the delivery network.

2.2.6 HTML5 Video Streaming

HTML5 is not a web standard yet. Still it is of great interest to the future of video on the web. Most browsers support the <video> tag introduced in HTML5 which allows for easy addition of video in a web site, without the need for browser-specific

third-party plug-ins. These plug-ins are activated by the <object> tag in HTML. The problem was that the video players and content were a black box, as seen from the browser. This means that CSS is not able to style or apply transitions to the video being showed on the web page[22].

Using the <video> tag a web developer can easily add a video while developing a website by simply writing:

```
<video src="myVideo.webm"></video>
```

This is basically all that is needed, but height and width attributes should be included, as with the <img> tag[23]. This is simply done by:

```
<video src="myVideo.webm" width="320" height="240"></video>
```

Since the <video> tag is a regular tag like <p> and , it can be styled directly using CSS. If a fully functioning video interface is needed, then a control panel can be added using the controls attribute like this:

```
<video src="myVideo.webm" width="320" height="240" controls></video>
```

HTML5 offers a fallback mechanism allowing the developer to specify alternate sources using different formats or codecs[22]. Research is also being done to allow support for peer-to-peer connections in a specification called HTML Device.

The web is designed to be open and platform independent. Many vendors have, despite this, implemented their own technology to offer rich web applications. Silverlight is one of these technologies, accompanied by Adobe Flash and Apple QuickTime amongst others. These technologies provide additional functionality to the web, but impair the webs ability to be uniformly usable across platforms. The World Wide Web Consortium is therefore developing HTML5 that should provide enhanced functionality without proprietary technologies[24].

The question then becomes if HTML5 will be able to provide the wanted functionality. The usability across platforms is a plus, but to replace the proprietary technologies it must also provide technology that makes it a viable substitute. In February 2011 the World Wide Web Consortium held the second workshop on "Web and TV". A summary of the workshop is available at

`http://www.w3.org/2010/11/web-and-tv/summary.html`

They concluded that "Most players of the Web and TV world who need to stream video on the Web require some adaptive mechanism to quickly react to network fluctuations and ensure a smooth user experience while watching videos. (...) Integration within HTML may require exposing new functionalities in the browser, such as QoS (Quality of Service) counters, control of the adaptive streaming process, and access to the streaming manifest. The Web and TV Interest Group should prioritize and discuss these needs on a second step."

The development of HTML5 is ongoing and the focus on adaptive streaming is present as network fluctuations should be handled. The downside is that the work group does not seem to be very close to finding a solution. The goal is to

develop a comprehensive test suite to achieve broad interoperability for the full specification by 2014[25]. This was announced by Jeff Jaffe, the CEO of World Wide Web Consortium. As of now it seems that the proprietary adaptive streaming solutions will not get any competition in the near future.

2.3 Overview of TCP

TCP is a widely used protocol in the Internet today and provides a reliable transmission of data segments and congestion control mechanisms. As elaborated in section 2.2.2, despite TCPs reliable characteristics it is still found to be an effective transport protocol for video content. We will in this section provide a brief overview of some TCP protocol versions to highlight some important characteristics. We will not provide a complete overview of all mechanisms in TCP, but rather highlight those that are important for our work.

2.3.1 TCP Reno

The most important aspect of TCP in relation to our work, configuring a home gateway, is the rate adjustment mechanisms built into the different TCP versions. Versions like TCP Reno use packet drops as an indication of congestion. First, TCP Reno starts with sending one segment and waits for a corresponding ACK. The sender continues to send segments as ACKs are received. The number of segments which are allowed in transit is called the transmission window. The period when TCP Reno increment its sending rate exponentially is called slow start. When a packet drop is detected, due to receiving duplicated ACKs, the window is halved and it enters a new period where the window is increased with one segment every time an ACK is received. This period with linear increase in the transmission window is called congestion avoidance[26]. This is the basic mechanisms of TCP Reno. Other mechanisms like fast retransmit have been complemented in newer versions. Fast retransmit includes a timer at the sender which sets the amount of time before a segment is considered lost. However, the main mechanism used to control TCP Reno implementations is packet drop. This type of TCP traffic can be controlled using queues with RED, which we did in our previous work. The randomly dropping of segments will provoke a reduction of the senders throughput and be fair among simultaneously active TCP flows[10]. Next, we introduce other TCP versions which are delay based.

2.3.2 TCP Vegas

Other versions of TCP use RTT calculations based on the time between a segment is sent and the corresponding ACK is received. One example of this is TCP Vegas. When sending the first segment, Vegas calculate a BaseRTT which is the RTT of the first segment. The window size advertised from the receiver in the ACKs, is also used. These parameters are used to calculate expected throughput as follows:

$$Throughput = \frac{WindowSize}{BaseRTT}$$

When the next ACK arrives at the sender, a new throughput called actual throughput is calculated and compared to the expected throughput. The formula is the same as previously shown; however the window size and BaseRTT might have changed.

Next, the difference between expected and actual throughput is calculated and compared to initial thresholds. The upper and lower threshold is denoted as α and β respectively. These thresholds set the limits on how the protocol reacts to the difference in actual and expected throughput.

$$\alpha < (Expected - Actual)Throughput < \beta$$

If the difference between expected and actual throughput is in between the upper and lower threshold, then the sending rate is kept at the same level. However, if it is below α, then the window size is reduced linearly. Similar, when the difference in throughput is above β, then the window is increased linearly[26]. This is typical behavior for basic TCP Vegas, but other version can do this differently.

2.3.3 Compound TCP

With the two distinct TCP versions presented in previous sections we move to the Compound TCP (CTCP). It is based on both the Reno and Vegas approach. CTCP has emerged due to the steady increase in transmission rate. One major disadvantage of Reno like approaches is that when the transmission rate is high, then a reduction of window size to half of its original size will represent a large amount of bytes. Therefore, reducing the window size to half of its original size and entering congestion avoidance to rebuild transmission rate will be time consuming. Versions like HSTCP and STCP have been developed to overcome this issue[27]. Another disadvantage is the fairness between delay and loss sensitive TCP versions. Loss sensitive versions do not back off until they experience duplication of ACKs. Then these versions will overflow queues compared to delay sensitive version, because queues with an increasing amount of packets will contribute to a delay increase. Hence, delay sensitive TCP versions will back off at an earlier stage than loss sensitive versions. FAST TCP is developed to handle these challenges[27].

Even though both delay and loss sensitive TCP versions have solutions for high bandwidth transmission, there are still some problems when these two types of flows are present in the network at the same time. This is why CTCP was developed [27].

CTCP uses three windows to calculate its TCP sending window. These are calculated using the following formula:

$$win = min(cwnd + dwnd, awnd) \text{ , where:}$$

cwnd = congestion window,
dwnd = delay window,
awnd = advertised window from the receiver.

From this formula it can be seen that the congestion window, the delay window and the advertised receive window all contribute to the transmission rate adaptation in CTCP. The congestion window is controlled by packet drops explained in section 2.3.1 and the delay window is controlled by increase/decrease in transmission delay.

It is worth noting that the transmission window is the sum of both the congestion and delay window. The increase of the congestion window is described with the following formula[27]:

$$win = \frac{cwnd+1}{cwnd+dwnd}$$

Another important characteristic of this approach is that the delay component is not used until the transmission session leave the slow start mode and enters congestion avoidance. The slow start period follows the same principles as in TCP Reno and dwin is set to zero. However, in congestion avoidance the delay mode of CTCP follows the same principle as in TCP Vegas presented in section 2.3.2. The only difference from CTCP and TCP Vegas is that CTCP also reacts to packet losses with reducing its window size aggressively if packet loss is detected. If a transmission timeout occur, the TCP sender enters slow start again. Hence, the delay component of CTCP is disabled.

2.4 RTT Measurements

As we investigated how to manipulate TCP flows according to user preferences, we discovered a significant need to gather accurate information from ongoing flows. TCP characteristics where presented in section 2.3 and common for several TCP versions is the throughput formula. RTT is a fluctuating parameter which needs to be estimated accurately in order to correctly set a target bit rate for individual TCP flows. There are different methods for estimating RTT and these are mainly divided into active and passive categories. They will be addressed in the following subsections.

2.4.1 Active Measurements

Active measurements are where a network measurement probe actively injects packets into the network and awaits a response from the designated destination. An example of such tool is ICMP ping messages[28]. A user sends ICMP packets and the destination node will respond if it is configured to support ICMP ping. The user which sent the ping can then measure the RTT (delay of the echo reply) and packet drop ratio. An issue with using ICMP ping is that some network administrators deactivate it for security or other reasons. The ping message will then be discarded at the receiving node.

2.4.2 Passive Measurements

Another method used is passive measurements where an intermediate node analyzes the traversing traffic flows and through this gathers statistics about them. An

example is where the three-way handshake period of a TCP session is analyzed. The time interval between when a segment with the SYN flag set, and when a corresponding ACK passes through the measuring node, can be used to calculate the RTT[29]. This is illustrated in figure 2.5. This type of passive measurement makes use of well-known TCP behavior. The behavior pattern used could be complex and unpredictable. As a result, TCP timestamps where proposed.

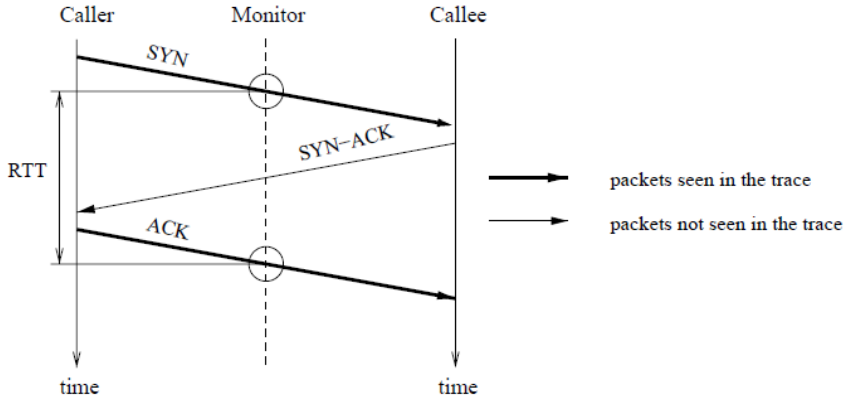

Figure 2.5: An example of an intermediate node which analyses TCP RTT in a three-way handshake passively[29].

TCP timestamps are described in RFC 1323 and is an extension to the option field found in the TCP header. An illustration of the TCP option field can be seen in figure 2.6. An issue with using TCP timestamps is that both client and server have to support it. Tests have shown that on average 76,4% of all nodes support TCP timestamps[30]. Hence, this solution needs some other mechanisms if TCP timestamps is not supported in both client and server. One solution could be to first use TCP timestamps, and if this fails ICMP ping could be used as backup.

2.4.3 Comparing Active and Passive Measurements

An objection to using ICMP ping to retrieve RTT for TCP flows is that some nodes in the network will treat ICMP and TCP traffic differently. In some cases nodes give TCP flows more resources when saturation is experienced. Measurements of RTT are then dependent on the protocol in use to do the measuring. This issue has been addressed in previous work and it has been shown that both ICMP ping and TCP analysis methods give approximately equal RTT[28]. Most servers do not block ICMP ping messages, sending ping messages is therefore a valid solution.

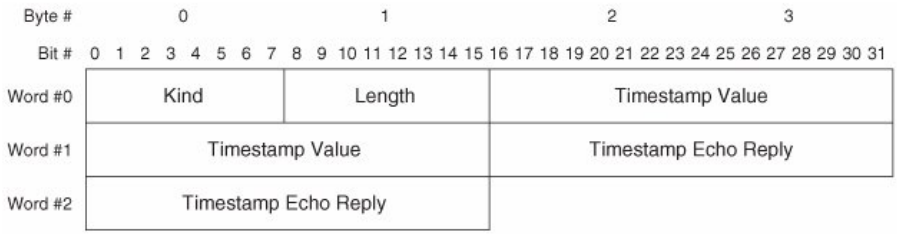

Figure 2.6: An illustration of the option field in the TCP header. The first and second byte identifies the option selected and data length, while the following 8 bytes represent the timestamp values. The timestamp value field indicates when the segment was received and the timestamp echo reply is used to store the timestamp of when the last segment was received.

Chapter 3

Theory

In this chapter we will outline how the theory on the KP-MP-AP architecture is adapted into our system. We will first focus on the Monitor Plane to see how the information gathering can be done. We need some functionality to retrieve information of the services that are running at the terminal equipment connected to our home gateway. Then we turn to the specifics of the Silverlight Smooth Streaming technology. After this, the Action Plane will be addressed; the entity that manipulates traffic in order to obtain the network goals. The Action Plane is realized by the Click modular router framework, which will be introduced here. Finally we present an overview of the architecture and how information will traverse the entities of our system.

3.1 Information from Terminal Equipment

Some video streaming clients collect information from the terminal equipment. Adaptive streaming services will monitor the bandwidth obtained while transferring chunks of data to approximate the available bandwidth. In addition, CPU usage can be used to determine if the terminal equipment has the processing power to receive video at a higher rate. Much of our testing has been on the TV2 Sumo service which uses Silverlight Smooth Streaming. This service monitors bandwidth and CPU load, and uses this information to determine which quality to request from the streaming server.

Each video service utilizes the collected information to improve its QoE. We want to explore how this kind of information could be used by the KP to optimize video services in the home network.

The methods for collecting information vary in many areas. We can passively collect data from traffic running through nodes in the network. By this approach we can collect the information that is communicated between the client and the streaming server, and in addition aggregate traffic statistics. Alternatively we can have an active approach, where data is collected at the end system and transmitted to the closest KP instance. For a home network this will normally be the home gateway. There are several places within the end system where an active approach could be placed. We will now have a closer look at passive, and several active,

techniques for information gathering from the end systems.

3.1.1 Monitoring Traffic

This is, as described earlier, a passive approach. The goal here is to make use of the information that is transmitted between the client and server. In adaptive streaming services there is usually an initial message sequence where the server advertises its services and bandwidth options. In addition, the client often provides information for authentication. As mentioned in section 2.2.5, the former can be done using a manifest file. The latter is often done in a similar matter with XML encoded strings presenting the user information to the server.

Silverlight Smooth Streaming uses this approach. After logging in to a service provider web page, the server communicates user details to the client as shown in figure 3.1.

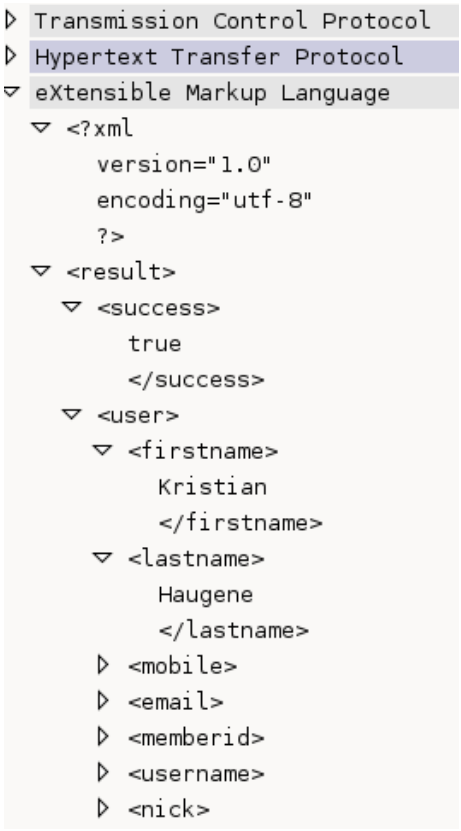

Figure 3.1: This is a cropped view of a Wireshark capture where the streaming server sends user information to the client application. The service used here is TV2 Sumo and the user details are retrieved from their database of users. This information can be used to identify users which can be useful in the KPs decision making.

Using this information we can recognize, separate users and give dynamic per-user configurations. We can see which clients are accessing video services and this

can be utilized when determining how to treat them. If a client name is livingroomTV it will tell us that this is a service that should receive QoE above the best effort, at the expense of the others. It can also be argued that when the father or mother in a household is streaming the news, this should have priority over the 14 year old daughter streaming Justin Biebers latest sightings in LA from a US gossip website.

Following our example on IIS Smooth Streaming, the successive messages between the server and client is the clients retrieval of a manifest. This manifest contains information of all available video and audio streaming rates as shown in figure 3.2. Being aware of this information allows the KP to set a target streaming rate for each service, giving one service a slightly elevated QoE. This is especially useful when two adaptive streaming services are competing for bandwidth.

Figure 3.2: Yet a Wireshark capture where the streaming server sends video stream details to the client application. Once again, this is a cut from TV2 Sumo. This information is very useful for the KP to determine an optimal resource sharing policy. Knowing what bit rates each video can be offered can greatly improve KP decision making.

According to [9], competing players will cause frequent quality variations. They argue that they observe congestion at the same time and responds by lowering the quality. When they both lower their rates, bandwidth is available and they both increase their rates simultaneously. These fluctuations can be prevented by tilting the resource provisions slightly in favor of one service. From a global perspective, this is a more stable use of bandwidth and the users will avoid annoying quality changes. Even though one of the players will have a constantly lower rate which

causes a lower QoE, the video will be streamed at a constant rate. The constant streaming rate will have a positive effect on the QoE. In sum the user getting lower rate will have a slightly lower QoE, but the user getting the high rate will have a significantly higher QoE. From a global perspective the case of service differentiation is definitely closer to the optimal solution.

After the initiation process, the client will frequently send GET messages to the server where it specifies what quality the video should be delivered in. By collecting all the mentioned packets the KP should know who the user is, what service the user is streaming, what rates are available and what rate is currently being streamed.

These files are, as described, sent in Silverlight Smooth Streaming. Similar files are also sent in Adobe OSMF player and Apples adaptive streaming services. The gains of passive monitoring can therefore be significant. An even bigger advantage is that this is monitored from the home gateway. No end user software or alterations in software are needed; it's plug and play from the moment it is installed in the home gateway. And in addition to getting direct information from the signaling between client and server, statistics can also be gathered. Bit rates can be logged and traffic analysis can be aggregated. This allows for the implementation of a learning ability. The home gateway can recognize different scenarios and how its previous actions have taken effect. Based on this information it can make the right configuration in the network using both pre-programmed parameters and the learnt behavior of the network.

3.1.2 Altering Streaming Client Code

This approach is motivated by the observations we have done while using the TV2 Sumo Smooth Streaming service. The client provides information to the user of what bit rate is currently being used and what the users maximum bit rate is. In addition to this, the current CPU load is displayed. The latter is of special interest because allowing the network to know the CPU load will contribute in the decision of what stream should receive more bandwidth. There is no point in allocating extra network resources to a computer that will have trouble decoding it.

In the previous section we saw that Silverlight Smooth Streaming, and the Adobe OSMF adaptive streaming, retrieve user and server capability information. This was the information we were going to capture in our passive approach. As shown in figure 3.3, it also collects information from the processing done in the end system. A suggestion is therefore to collect all this information directly from the client instead of going to great lengths to capture and interpret this information from other sources. Other useful information could also be available from the video client, depending on what the framework supports. We can make use of the information gathering already taking place in the streaming video client.

A solution could be to write code that could be added to the existing client and would function as a communicator towards the KP. There are several ways to signal the KP with information from the client. In our case, having main focus on Silverlight applications, a HTTP approach seems to be a reasonable solution. Building a HTTP client is simple in Silverlight and parameters could be passed to a

Figure 3.3: Screenshot of the Tv2 Sumo Silverlight client where some system parameters are read and posted to the user. There is lots of information available in the Silverlight Player

web server enabled home gateway with POST messages. At the home gateway, the parameter passed in the HTTP request can then be read and utilized.

Another benefit of this approach is the simplicity of upgrading it. The video streaming clients are downloaded from the service provider every time the user watches a video. This means that users would get the software and successive upgrades without noticing it. Of the approaches for end system reporting of data, this is the most user-friendly approach as upgrading is done server side.

3.1.3 Collecting Information From the Browser

The web streaming services are presented from the providers web site using a client. One thing the clients have in common is that they are all viewed in a web browser. Most web browsers allow for custom third-party plug-ins or add-ons. By creating an add-on we could retrieve information about the resource consumption and service provider by collecting it in the browser.

This method has several imperfections that, in our view, disqualify it. First of all, the number of browsers is quite large. A custom add-on would have to be implemented for each browser and constantly be updated as the browsers often have version upgrades. In addition, video player clients presented by Silverlight or Flash are black boxes to the browser. As discussed in section 2.2.6, most clients are not HTML encoded. If HTML5 someday presents a good solution for adaptive streaming, the browser might be able to read information from this player.

The clients are often represented by a <object> tag, and its content is not visible to the browser. Adding together the concerns of this approach we see that we would have to do several implementations trying to get information hidden in a black box.

3.1.4 A Standalone Application

Another solution for the active approach is to have a standalone application running on the end system. This application could be written using any programming framework and language. But a platform independent language would save the trouble of multiple implementations.

This application would run in the OS directly. This would give easier access to a lot of OS parameters. We could retrieve the CPU load and available memory directly. In addition we could get information on what type of interface the end system uses to connect to the home gateway. Wired and wireless Internet connections have different levels of QoS. This is especially a concern if the signal strength is low on a wireless link. The network link load could also be collected, but this would be a total of all traffic originating and terminating in this end system.

The standalone approach has some advantages in retrieving network link information. The retrieval of application information would on the other hand be a significant challenge. Application specific information, like the current quality level used by a Smooth Streaming application, will be difficult to retrieve. The application is run in a black box, on top of a browser. It is not made for easy access.

The conclusion is that some parameters could be easily gathered from the OS through a standalone application. These could aid us in determining what actions to take. Especially CPU, memory and network link signal strength are good candidates.

3.2 Silverlight Adaptive Streaming

In our previous proposal[10], we have already implemented some traffic monitoring. Using known media services, video streams are monitored and bandwidth usage is calculated. Even though this monitoring could have been improved by the measures explained in section 3.1.1, we want to explore new ways. As a proof of concept solution, perfecting the traffic monitoring is not the goal. We will therefore focus on a different approach.

Implementing a browser add-on, as described in section 3.1.3, is not desirable. The standalone application described in section 3.1.4 could provide useful information. The information would however be high level usage statistics and not detailed information of a video stream and its properties.

Therefore we have chosen to explore the option of altering the client code, as described in section 3.1.2. In accordance with previous work, we have yet again focused on Silverlight Smooth Streaming. Our custom client code is therefore written in Silverlight for a Silverlight Media Framework video player. This implementation process will be explained in section 5.1.

Silverlight provides an adaptive streaming technology called Internet Information Services Smooth Streaming or IIS Smooth Streaming for short. The Silverlight framework provides a media player that has support for multiple streaming technologies. The player can be instantiated to provide progressive download or adaptive streaming of media along with other forms of media delivery.

The properties of an adaptive stream have been covered in section 2.2.5. The adaptive streaming provided by the Silverlight framework is the technology we have investigated in this report. Microsoft announced in October 2008 that they would feature a new streaming extension called Smooth Streaming. For promotion purposes Microsoft announced an initiative with Akamai and launched a showcase web site; SmoothHD.com[17].

The Smooth Streaming technology uses MP4 media format. This is the first time in over a decade that Microsoft uses a file format other than ASF as a media format. The MP4 container contains less overhead, is based on a widely used standard, has H.264 video codec support and native support for payload fragmentation within the file.

Adaptive streaming requires that the video is coded into several different bit rates. A manifest file that has an overview of these bit rates is created; as shown in figure 3.4.

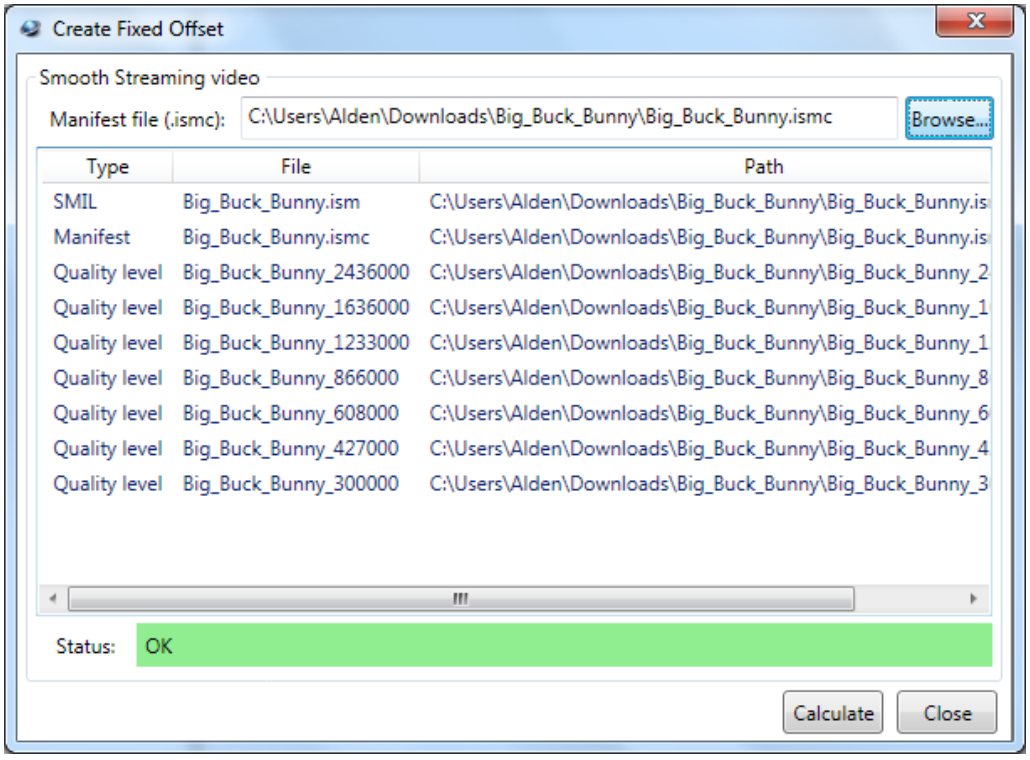

Figure 3.4: Expanded view of the information present in a manifest file. The available bit rates are listed and a path to the file containing each bit rate is given. This file is then retrieved by the client which will choose a bit rate to retrieve.

As described, these video files are partitioned into chunks of data. The chunks

contain a given time interval of video for a given bit rate. This way the client has an array of video files to request. It chooses the wanted bit rate, and then selects the time interval it wants to retrieve from within the video.

The Smooth Streaming specification defines each chunk as a MP4 fragment and the chunks are stored as a contiguous MP4 file. Each fragment can be chosen from this file at random access. When a client asks for a specific time segment, the server finds the appropriate chunk and sends it over the Internet as a standalone file. This means that the chunk files are created virtually upon request while the video is stored on disk as a single full-length file encoded per bit rate. This is possible due to the mentioned support for payload fragmentation in MP4.

Using this scheme, the Smooth Streaming servers offer the content to the Silverlight clients. A client is free to request a video file with the wanted bit rate and specify a time interval to retrieve. In the Silverlight framework the Smooth Streaming media players have a pre-defined way of determining what bit rate to request and when to adapt.

For greater flexibility the developer is free to implement his own version of this selection process. This allows service providers with a choice to either use the predefined player settings or rewriting the logic for rate adaption. The rate adaption logic is included in the source code, allowing for small customizations or completely rewriting it. The custom implementation of such mechanisms is complex, and most service providers would be better off using the provided code.

3.3 The Click Modular Router Framework

There are many different router architectures in use today. Most of them are proprietary solutions (e.g. Cisco) that give network administrators the opportunity to use different built in functions. There are no mechanisms for extending the already built in functionality. This is supported by the Click framework, without severe cost of routing performance. It has been shown that Click only contributes to a 10% cost of computing resources, compared to normal Linux routing operations[1].

Click is an open source project developed at MIT. It offers a flexible architecture for developing routers with configurations aimed to support different scenarios. This is what we need when we wish to explore how a home gateway can be used to enhance QoE offered by adaptive streaming services. The Click framework is built upon several different fundamentals which will be presented in the following sections.

3.3.1 Click Elements

A Click configuration is built upon several smaller processing entities called elements. When an element finishes its designated task it will hand the packet to the following element in the routers configuration. An example of an element task is to decrease the TTL field in the IP header. A series of elements which execute different tasks is called a forwarding graph. An illustration of an element can be seen in figure 3.5.

The reason for having many elements with a restricted area of responsibility is to give network administrators the possibility to add and remove elements in order

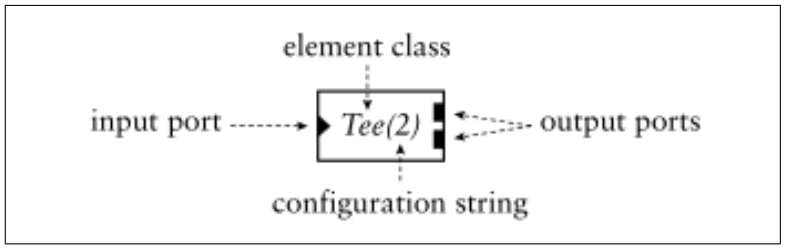

Figure 3.5: Example showing a Click element with its corresponding class name, ports and configuration string. This is just a dummy element which does not do anything else than having one input and two output ports. This element is provided for illustration purposes[1].

to tune the routing behavior to their needs. The Click framework now offers a wide collection of elements which can be used to build routers with different behavior. These elements and their corresponding documentation can be found on the Click website[31].

However, if an element for a specific need is not already built, then this element can be built as a class in C++, a widely used programming language. The syntax and how an element is built will be explained further in section 3.3.3. We present the most basic element building blocks in the following subsections[1]:

Element Class

Element class is like a class in classical programming (e.g. Java, C#). This is where the element behavior is defined.

Ports

All elements have a set of both input and output ports. A packet enters an element through an input port, is processed, and leaves the element through an output port. These ports can be agnostic, pull or push.

Pull is when an element asks for a packet whenever it is idle. The element which receives a pull request will then return a packet or a null pointer if there are no waiting packets. A push port is used when an element wants to send a packet without being concerned if the receiving element is idle or not. This can for example be when an element sends a packet to a queue element. Agnostic connection is used when the connection is not defined. Hence, the router will initialize the connection as either push or pull according to the context in which it is used. It is important to note that two elements communicating with each other needs to be paired using equal port types. Hence, a push output port needs to be connected to a push input port.

Configuration String

Configuration strings allows for a dynamic initializing and use of elements. This means that each element can take a set of input parameters and configure its behavior according to them. It can for example be the queue length which is used by the queue element. Then the queue length is taken as an input parameter when the queue element is initialized. An example is given in section 3.3.2. Next, we will introduce how the configuration files are written in Click.

3.3.2 Writing Click Configurations

As mentioned in the previous section, elements are the fundamental building blocks used when creating a router configuration. Human readable names are assigned to elements and can be compared to variables used in ordinary programming languages. The benefit of using human readable names for elements is that it makes it easier to identify the specific element handlers. These handlers are used to manipulate the behavior of elements at run-time. Element handlers will be outlined in the following subsection. First, we give an example for how we can define a source for packets and a queue using human readable names. Note that the double colon is the notation used for assigning an element a human readable name.

```
packetSource::FromDevice(eth0);

myQueue::Queue(10);
```

In both packetSource and myQueue the element is used with a corresponding configuration string which is contained inside the parentheses.

The flow of packets is then defined using arrows. An example using the previously defined elements is as follows:

```
packetSource->myQueue;
```

This example is not a complete Click configuration. This configuration is very basic and the only task these lines of code perform is handing a packet over from the designated Network Interface Card (NIC) and stores it in a queue. The queue is defined as myQueue with ten available queue positions. However, it is a good introduction to how the flow of packets is and how a Click configuration can be built. After building a Click configuration file it can be initialized using the following command:

```
click-install [configuration-filepath]
```

To uninstall a click configuration the following command is used:

```
click-uninstall
```

These commands are referred to as tools in the Click documentation and we only present the ones we frequently used. Other tools can be found on the click documentation website[32].

Up until now we have described how a Click configuration can be initialized at the startup of the Click router. In most cases, a click configuration is reinitialized whenever a new configuration file is written. However, there are two methods for interacting with elements and altering configuration files without restarting the router. These are handlers and hot swapping[1]. Both will be outlined in the following subsections.

Element Handlers

Element handlers are used by elements to support user interaction with elements in an already running configuration. They can also be used to retrieve statistics such as current or average queue size in a running queue element. Each element will create its own subfolder in the /proc path in a Linux system. In our installation of the Click framework all handlers where put in the following folder:

```
/click/
```

For example the element myQueue would have a unique folder, containing its designated handlers. myQueue could for example have a handler called "length", which is used to change the elements queue size. Then the folder structure would be as follows:

```
/click/myQueue
```

Inside the myQueue folder a text file named length would reside. The default value contained in the length file would be 10, because of the configuration string explained in section 3.3.1. This way the length of myQueue can be changed run time without having to click-uninstall and click-install the configuration file. It is important to note that elements which we want to interact with through handlers should have unique human readable name. Then it will be both easier and faster to identify each elements handler folder. This is because a Click configuration is built from several elements which are working together.

Configuration Hot Swap

Even though handlers give some interaction possibilities between the user and a running Click configuration, there is a need to sometimes add and remove elements without restarting the router. The issue with restarting the Click router every time we do changes to the Click configuration is that we lose the current router state. This can for example be the packets stored in a queue. Throwing away packets already stored in a queue is considered bad behavior and should be avoided. Because of this, there is a possibility to hot swap to a new configuration file without shutting down the router. The command for using hot swap is:

```
click-install -h [configuration-filepath]
```

The hot swap will be performed if the new configuration file does not contain any errors. If the new configuration contains errors, then the old configuration will be used.

3.3.3 Writing Click Elements

Click has several classes which can be used to create new elements. The description of these classes can be found on the click home page:

```
http://read.cs.ucla.edu/click/doxygen/classes.html
```

When writing a Click configuration, the framework may not contain all the functionality needed. Then we have to build a new Click element from a header and source file with the file extensions .hh and .cc accordingly. The header file contains class, method and variable declarations and can be viewed as an interface for other classes using the element. The source file is where the class, method and variables are implemented. Hence, it is where most of the complexity is placed. This is to hide complexity from the header file. Then a programmer can easily use the methods declared in the header file without being concerned about how they are implemented.

3.3.4 Compiling Click Elements

After a Click element is written we need to compile it into the running Click framework in order to use it when writing a configuration file. First, the element header and source files need to be put in a designated folder, where it can be read by the Click framework. Each type of elements has their own folder which is categorized based on their functionality. For example, in our Click installation, the checkipheader.cc and checkipheader.hh are placed under the /ip folder as follows:

```
/CLICKDIR/click-1.8.0/elements/ip
```

Because /ip is a folder which contains built-in elements it should be left as it is in order to maintain the Click frameworks structure. When developing new Click elements it is beneficial to keep the elements under development separated from the rest of the frameworks fully functioning elements. In a clean Click installation this folder is called /local and its path in our installation were as follows:

```
/CLICKDIR/click-1.8.0/elements/local
```

All our developed header and source files were put in this folder. This folder is not included in the Click frameworks build path as default. In order to add it, we first needed to navigate to where the Click framework was installed. Our installation path was as follows:

```
/CLICKDIR/click-1.8.0/
```

The /local folder was added to the Click build path using the following command:

```
./configure --enable-local
```

Next, we used the following command to update the list of elements which Click uses when it compiles the framework.

```
make elemlist
```

Finally, we compiled the Click framework using:

```
make install
```

It is important to note that all executing commands in the compiling section needs to be run in the super-user context.

3.4 Overview and Architecture

We have described our approach for retrieving information and the tools for altering traffic. In this section we would like to present how the entities will be connected and the information utilized. The description will focus both on the functional plane as well as the physical. The entities and their tasks will be described and the system architecture will be presented, showing how these entities are placed in our home network.

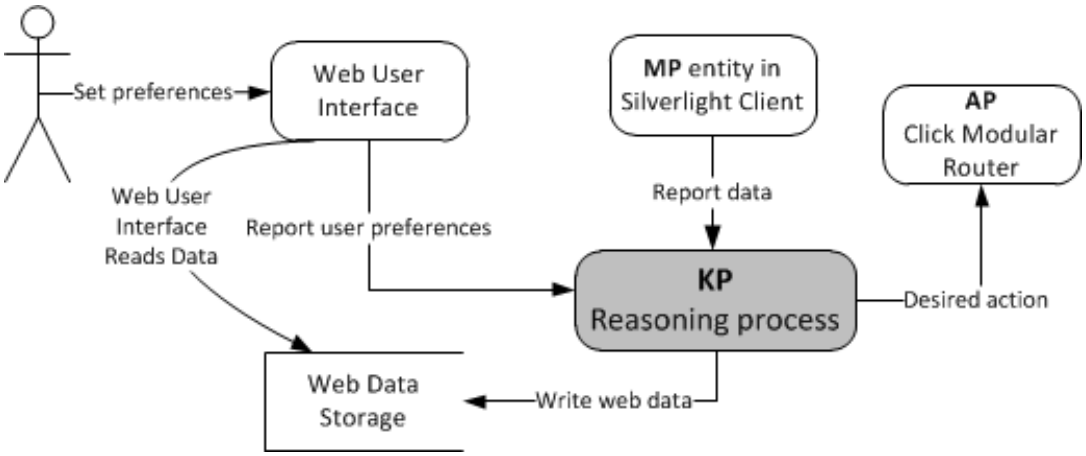

Figure 3.6: Illustrating the entities in our system and the communication between them. The MP entities will report the presence of media streams to the KP. The KP will in turn write this to the web data storage. This is where the web user interface reads data and presents them to the user. The user set its preferences which prompt the KP reasoning process to reconfigure the router.

The functional entities of our system are presented in figure 3.6. An addition to the already mentioned entities KP, MP and AP, is the concept of a web user interface. This interface is described in section 4.3 as a method of collecting user preferences. It is implemented to ease the reasoning process done by the KP and allowing the user to influence the decisions of the KP.

Our goal is to implement these entities in a system that will function as a complete home gateway. The placement of our system in a network setting is shown in figure 3.7. Routing tables are written in Click and forward regular IP traffic as an "off the shelves" home gateway would. Applications do not need to be aware of the enhanced functionality of our home router to get normal best effort service connection. This is a necessity as making all applications compatible is impossible. To utilize the enhanced functionality clients must contact the home gateway.

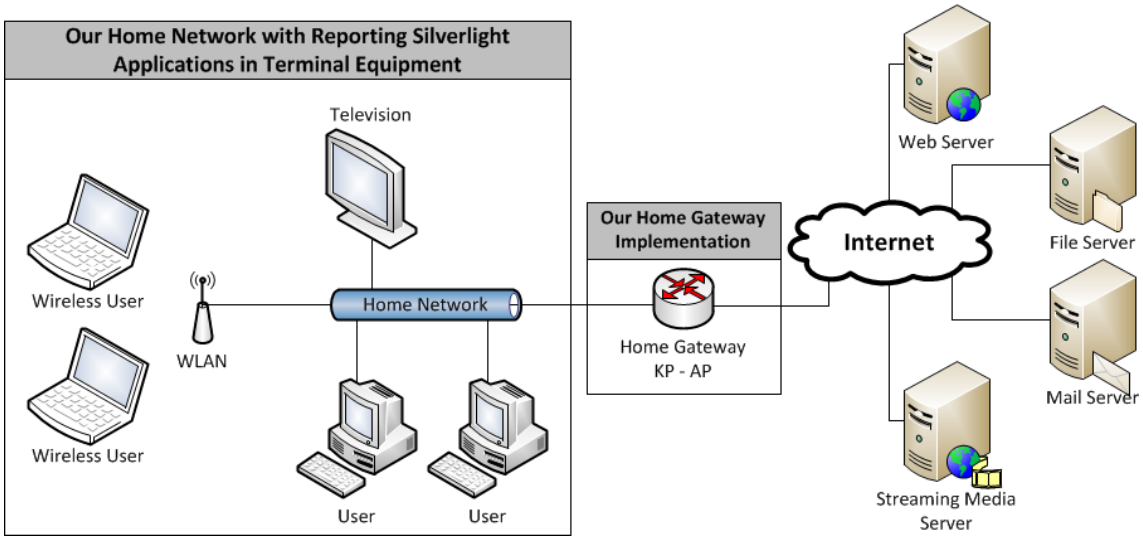

Figure 3.7: Here we illustrate where our system will be implemented in the Internet. Our Knowledge Plane and Action Plane components will reside in the home gateway. The Monitor Plane component will be loaded to the user terminal equipment upon starting a reporting video service.

Silverlight clients are downloaded from the service provider each time a web page is visited. This means that the service provider chooses if the user should get a video player with a reporting process. If the service provider has enabled the reporting functionality, as proposed in section 3.1.2, the Silverlight Application will try to contact the home gateway. This communication will be successful if the home gateway is enabled to process the information, if not the information goes to waste. The functionality of the Silverlight Application should not be affected by the reporting process. This should be true both when the application makes contact with a home gateway, and when the home gateway is non-compatible with our system.

The Silverlight applications are running at the users terminal equipment. The rest of our entities will run on a single physical entity and will be comparable to

a regular home gateway. A Silverlight application will report run time application specific parameters to the KP. Utilizing the information obtained, the KP will reason on the state of the network. If an action is found that would improve the overall Quality of Experience(QoE) for the users, it will be signaled to the Action Plane. With this communication between the entities, the system would be able to work as a collective system for quality optimization.

Chapter 4

Methods

In this chapter we will consider how we can control traffic in the home network. There is a variety of approaches that could be used to obtain control over the traffic flows. We will present some of these, and propose two Click elements based on the presented techniques. Current video streaming services, such as Silverlight Smooth Streaming, have deficits in their adaption to traffic in a home network because they only regard a limited set of local parameters. Using our architecture with its interaction between the MP, KP and AP we try to develop a home gateway that will be able resolve these deficits. After presenting the techniques for traffic control, we will present the web user interface. This is an interface where a user of the home gateway can monitor the traffic in the network and input his or her preferences.

4.1 Controlling TCP Traffic

In section 2.2 we presented video streaming and how this technology distributes both live and recorded video content to end users. Several protocols are used, but the state of the art technology is adaptive streaming which uses HTTP over TCP. In the home gateway proposed in our previous work, we presented a gateway which is able to distinguish between flows and prioritize among them. We did not further develop AP mechanisms to control TCP traffic. As discussed in section 2.1.6, we used built in functions with RED to control the traffic. We managed to control the traffic, but it's not the most effective solution when it comes to link utilization.

Traditional Downstream Limitation

This is the approach used in our previous solution. As explained in section 2.1.6, this can be done by dropping packets coming in to the receiver. We could also have delayed or corrupted incoming data to make the TCP receiver react to what it interprets as a lossy or erroneous connection. These forms for limiting TCP are effective at achieving their goal, but they introduce unwanted behavior like packet retransmission and TCP timeouts. We have therefore focused our work on more subtle forms of control. By this we mean that we want to manipulate the signaling used in TCP and imitate a poor link throughput or link congestion, leading TCP

to lower its transmission rate.

Controlling TCP by Altering Upstream Communication

In this section we present more sophisticated methods for controlling TCP traffic. First we begin by introducing main characteristics of how TCP adjusts its sending rate according to current network state.

TCP mainly implements two controlling features for the traffic flows[33]. These are as follows:

- Flow Control
- Congestion Control

TCP uses flow control to synchronize the transmission rate of the sender and the maximum receiving rate of the receiver. Some receivers have less processing power than an ordinary PC, and therefore needs a mean to advertise to the sender to slow down its sending rate. This is done by advertising the maximum window size, hence maximum receiving bit rate, in the ACK packets sent from the receiver to the sender. There is also another technique used for TCP to adopt its sending rate to the network environment. This is congestion control which is used to adapt TCPs sending rate to congested links and queues in the network. This is done by dropping packets, and then TCP will use congestion avoidance, slow start, fast retransmit and fast recovery to recover its sending rate. Another technique used is retransmission timeout (RTO), where a timer is used to decide for how long a sender should wait before an ACK from the receiver is considered lost. All of these features need to be taken into consideration when we are proposing a solution for controlling TCP flows traversing our home gateway. We begin with introducing TCP pacing.

4.1.1 TCP Pacing

The purpose of TCP pacing is to evenly distribute the packets sent from the streaming server to the client over the entire duration of the transportation RTT. The idea is that this will reduce the bursty behavior of TCP, where several packets are sent close to each other in a short period of time. Reducing burstiness will enhance transmission effectively in the network because an even distribution of packets will not lead to congested queues at network nodes in short time intervals. This evenly separation of packets will only lead to dropping from queues when the network is approaching its saturation point. Figure 4.1 illustrates a comparison of bursty TCP traffic with and without using TCP Pacing.

TCP pacing can be implemented at the sender or the receiver end. Normally, a sender using TCP will send the next packet in the sequence when the sender receives a TCP ACK of the previous packet from the receiver. When TCP pacing is activated, the sender will delay sending the next packet. The new sending rate is equal to the window size divided by the estimated RTT. All packets are then evenly distributed over the RTT[34].

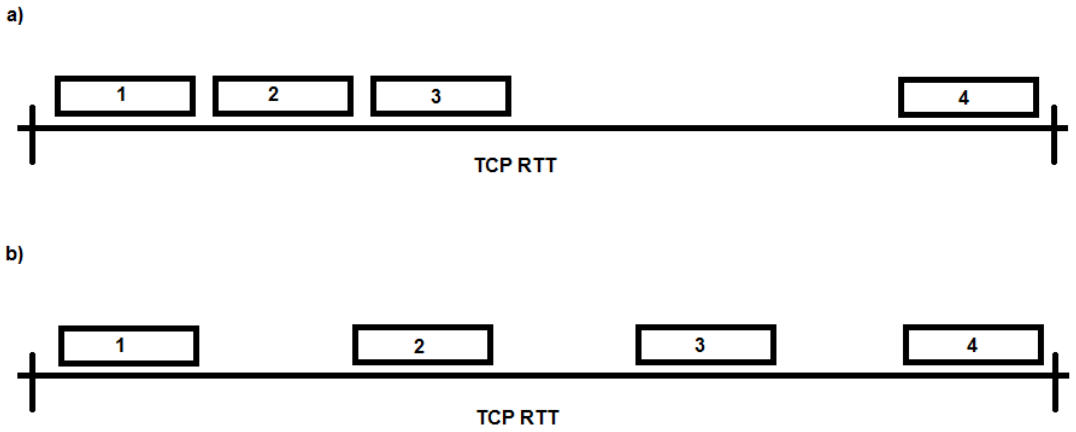

Figure 4.1: A comparison of an ordinary bursty TCP flow a) and TCP pacing b). We see that in ordinary TCP transfer then several packets are grouped together, while they are evenly distributed over the complete RTT when TCP pacing is activated.

TCP pacing could also be implemented at the receiver. This can be done by delaying the TCP ACK response. However, controlling a TCP flow at the receiver side is believed to be challenging. Some TCP versions use one ACK to acknowledge several packets, making control of TCP traffic at the receiver side hard. In this case we only accomplish delaying the sending of a burst consisting of all packets which belong to one ACK. However, it is interesting to see if this technique could be used to slow down the sending rate of the TCP sender. This will be elaborated in section 4.1.2.

It has been shown that evenly distributing TCP packets is not preferable in terms of increasing TCP throughput. This is because spreading the packets over a certain time interval will lead to synchronized losses between several TCP sources. Another argument for not implementing TCP pacing is that TCP will not adopt the sending rate before the network is actually saturated. This is because there is a higher probability for several TCP flows experiencing saturation at the same point in time. This is handled in for example Random Early Detection, which is performing random dropping of packets. Random dropping of packets will activate congestion avoidance at the sender side and therefore prevent the synchronized congestion avoidance behavior[34]. However, we will use this technique on separate flows at a gateway instead for using it on several flows and nodes simultaneously. The global synchronization should therefore not be a problem.

4.1.2 PostACK

PostACK is a technique which is used to delay the sending of ACKs from the receiver to the sender. This was discussed in section 4.1.1. It does however have high demands for adjusting the interval between ACKs sent from the receiver. This

is because there needs to be a close collaboration between the managing entity (KP) and the controlling entity (AP) that performs the adjustment of the flow sending rate. This is needed in order to make the traffic adjustment transient to the user, so that it does not influence the QoE. This is because the interval between ACKs needs to be adjusted according to both current traffic flows in the gateway and the user preferences.

Even though there can be some challenges in creating an element that takes all of these elements into account we will investigate how this can be implemented in Click and demonstrate the effects of it when it is implemented in our testbed.

4.1.3 Controlled Packet Drops

Controlled packet drops are the technique where dropping of packets is used to activate congestion avoidance at the sender. This is used in RED and other known active queue management (AQM) algorithms to control the sending rate of different TCP flows. This technique is implemented in routers today and is what we used in our previous gateway solution. It is not considered optimal because it will trigger retransmission of information which has already traversed the network. Because streaming of video with high quality (HD) uses large amounts of resources, it is advantageous to keep the amount of information retransmission to a minimum.

4.1.4 Endpoint Window Size Adjustment

A solution proposed for controlling TCP traffic at the receiver side is one where the controlling entity rewrites the window field in the TCP header. The TCP header and its corresponding fields can be seen in figure 4.2.

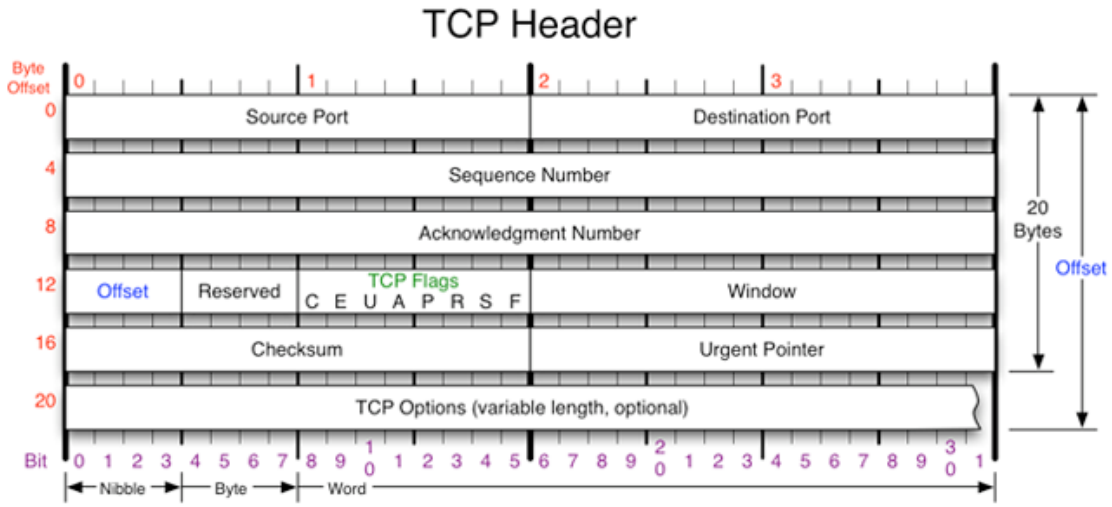

Figure 4.2: An overview of the TCP header and its fields.

The receiver window size indicates how much data in bytes which it can store in the receiving buffer. It is an indication of how much traffic that can be in transmission between the streaming server and the client at any time.

One solution discussed in other works is using this technique in combination with RED algorithm[33]. This way, saturation can be avoided at an early stage and the throughput of the network will be increased. Fairness between flows will be maintained, because dropping of packets is random instead of unfair models like tail dropping. Tail dropping is considered unfair because traffic sources transmitting large packet bursts will occupy most of the queuing capacity in network nodes, while senders using small bursts can then see the queues as full and will experience dropping of packets.

Using RED in combination with rewriting the TCP window field is referred to as RED-RWM and it has been shown in simulations that this technique is very beneficial when it comes to enhancing network throughput and effectively[33]. This is because it does not report to the sender with the use of packet drops. Instead the window field is changed to slow down or increase the sending rate.

One challenge with this technique is when encryption is used. Then the TCP header will be hidden and it will not be possible to read and update the header fields.

Despite problems if encryption is used, it is of great interest to see how this technique can be implemented using Click [1]. We will need to create an element which rewrites the TCP receiver window size in the ACKs sent from the receiver to the sender. Another important issue is that the TCP header checksum needs to be recalculated before the ACK is sent towards the sender. Checksums are used to control the integrity of the TCP headers. Hence, the senders dismiss packages with false checksums.

4.2 Click Element Proposals

This section provides an overview of the functionality provided by the Click elements we have developed. Code is not present in this section in order to ease the understanding of how these elements operate. Before our elements can manipulate the different TCP flows, we need to identify them.

4.2.1 TCP Flow Identification

All TCP ACKs traversing our home gateway upstream will be sent through the proposed element. A Click handler is used by the KP to set the source and destination IPs of the TCP stream the element is to manipulate. This handler is constructed using comma-separated values (CSV) and will be adapted to each proposed element. We assume that each user will view at most one video stream at any given instant in time. This assumption allows us to uniquely identifying a TCP session based on a source and destination IP pair. Another characteristic of the proposed elements is that all TCP ACKs which do not have the source and destination IP set in the handler, will traverse the element without modification.

4.2.2 ACK Pacing Element

The ACK pacing element is illustrated in figure 4.3. The basic idea is that every ACK is sent from the receiver to the server in order to confirm that each individual packet was received without any errors. The combined time it takes for a packet to be sent and the receiver to acknowledge a packet is denoted as RTT. The RTT and the receivers congestion window are used by the server side TCP algorithm to calculate the throughput at which the content should be transmitted. The formula used in these calculations is as follows:

$$Throughput = \frac{RWIN}{RTT}$$

We see from this formula that in order to affect the TCP sending rate there is a possibility to either increase the RTT or reduce the receivers RWIN. First, we propose a Click element for increasing the RTT by pacing ACK packets sent from the client to the server. Our element is placed in the receivers sending path. Every ACK sent from the receiver to the server will enter the ACK pacing element. As the ACK packets traverse the element we calculate the inter-arrival time between ACKs, denoted γ. Normally, γ would not be altered between intermediate nodes in the transmission path. But, in order to reduce the TCP throughput we increase the RTT by adding an extra time delay ϕ to the existing inter-arrival time.

Three phases become clear when we use this technique in an adaptive streaming environment. These are denoted as the initialization, learning and the operating phase. They are introduced in order to ease the understanding of some of the challenges in using the ACK pacing element. They will be outlined in the following subsections. An illustration on the KP/AP interaction can be found in figure 4.4

Initialization Phase

This state is where the KP decides that some streaming flows should be protected and others should reduce their transmission rate. The KP sets the parameters needed by the AP to identify the different active flows. Next, the ACK pacing element in the AP starts to measure the inter-arrival time between each incoming ACK and calculates an average inter-arrival time using these measurements. Meanwhile the KP calculates a metric from its collected parameters, this metric represent how much a flow should reduce its sending rate. The metric is used in combination with the average inter-arrival time to set how aggressive the increase in delay, ϕ, should be in the learning phase. The learning phase is explained in the following subsection.

Learning Phase

The learning phase is the time period when the element is used for the first time with a new streaming service. As described in section 2.2.5 an adaptive stream will operate at several distinct bit rates (QL). But the ACK pacing element has no means of knowing how much a ϕ increase in the time between each ACK is needed in order to reach a target bit rate. The element therefore needs to start by increasing

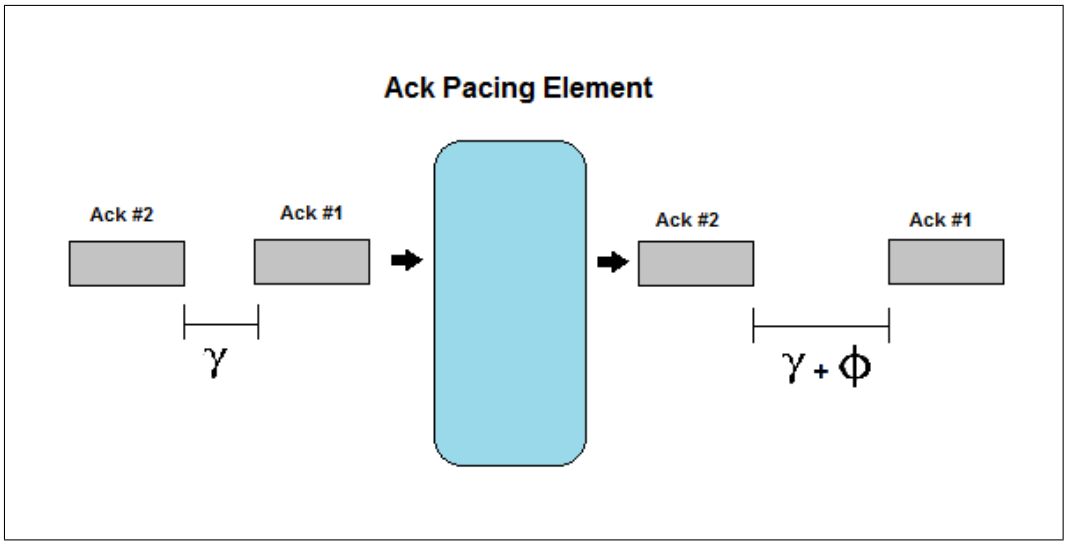

Figure 4.3: This is an illustration of the ACK pacing element which space the receivers ACK responses sent from the receiver to the streaming server. It can be seen that the ACKs enter the element with a time interval γ between them. After a calculation process the ACKs leave the element with a $\gamma + \phi$ increase in the time interval between them.

ϕ with for example 1 ms and wait for a response from KP. The aggressiveness of the increase in the added delay is controlled by the metric calculated by KP from the collected parameters.

$$\phi = \gamma * \frac{KPmetric}{10}, \qquad where \quad 0 \leq \text{KPmetric} \leq 100$$

The KP will report to the AP, i.e. the ACK pacing element, if there were any effect of the ϕ increase in delay. If the ϕ increase resulted in a flow reducing its sending rate (QL), the delay will be kept at this level. However, if the ϕ increase in delay had no effect, then the AP will continue to increase ϕ until the KP is satisfied and the optimal bit rate (QL) is achieved. When the optimal bit rates (QL) for all current flows are achieved, then the AP enters the operating phase which is described next.

Operating Phase

After an optimal receiving bit rate according to KPs reasoning process is reached, there should be a mechanism which stores current QL and the introduced delay ϕ, which was needed in order to achieve the corresponding QL. Then the ACK pacing element could faster reach the optimal QL next time it is needed to do so for the same service. Without this mechanism the time it takes for the AP to find a match between ϕ and the wanted QL can in some situations be time consuming and the adjustment in QL be too slow.

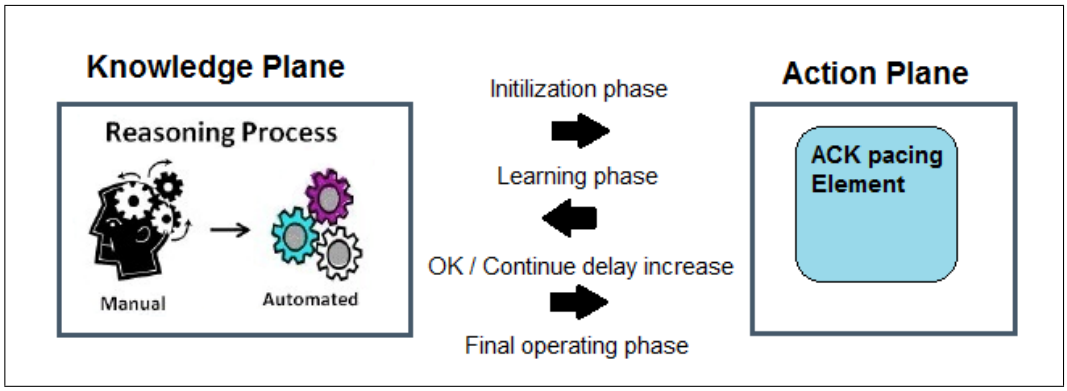

Figure 4.4: This figure illustrates the complete interaction sequence between the KP and AP. First the KP sets the initial parameters, then the AP enter the learning phase where the delay ϕ is increased and the KP reports back if the wanted effect is achieved. This cycle continues until the KP is satisfied and the ACK pacing element stores service name, ϕ and QL, which is stored for future reference. Finally, the element enters the operating phase.

4.2.3 TCP Receive Window Manipulation

This element is based upon the fundamental idea of how TCP senders throughput can be controlled by altering the clients advertised receive window. Theory behind this approach has been addressed in section 4.1.4. For explanatory reasons we divide the element functionality into three parts; TCP flow identification, new receive window calculation and TCP header checksum calculation. The elements basic behavior is illustrated in figure 4.5. First, we address how needed parameters are gathered from the KP through the use of a Click handler. Click handlers where introduced in section 3.3.2.

The Service Session Handler

The handler used by the receive window manipulating element is named serviceSession handler and is constructed from CSV as follows:

```
Source IP:Destination IP:RTT:Target bit rate, ...
```

The source and destination IP addresses are used for identification. All TCP headers are read and their source and destination IP address pairs are compared to the values found in the service session handler. If there is a match, a new receive window will be calculated. On the contrary, if there is no match, the packet is passed through the element without modification. The next parameter found in the handler is the target bit rate.

The RTT is self-explanatory and is used in combination with target video bit rate to calculate the advertised receive window. The last parameter is the target bit rate which represents the bit rate at which the KP wants the TCP flow to run at. Using this structure ease the complexity of the method needed in order to retrieve each

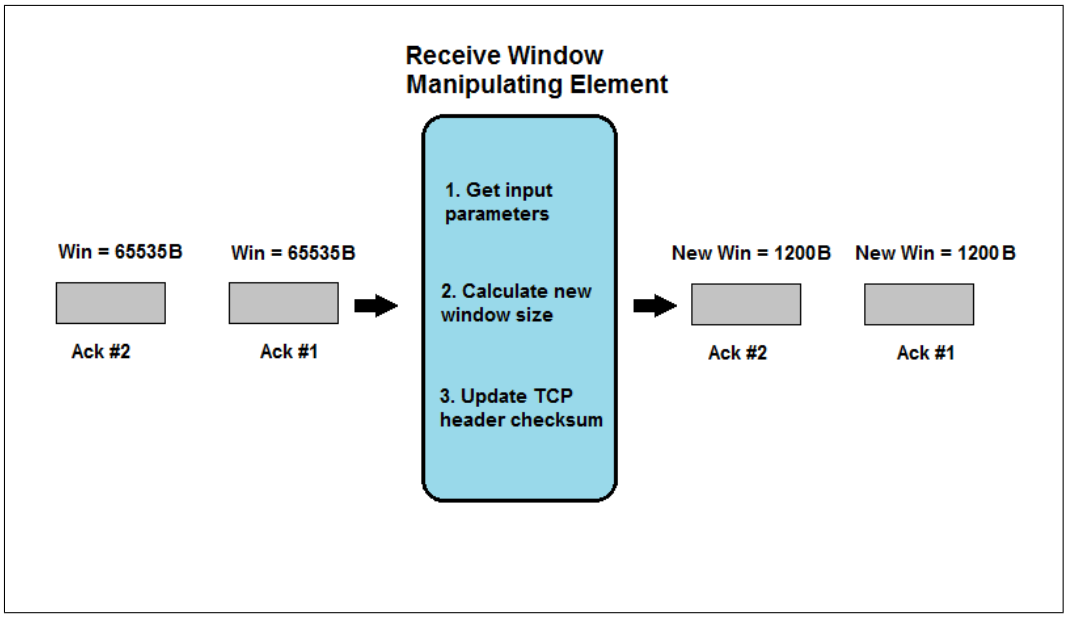

Figure 4.5: This is an illustration of the receive window manipulating element which rewrites the window field contained in the TCP header. The three main functions which this element performs are highlighted inside the element box.

parameter. Next, we explain how these parameters are used in order to correctly estimate the receive window size.

Receive Window Calculation

When calculating a TCP window which will control the bit rate of the TCP flow, there is a need for a general formula which describes the TCP senders behavior. Several different TCP versions was described in section 2.3 and the general throughput formula used by them is preferred to use for window size calculation. However, it is important to note that there is a difference in how TCP senders adjusts their sending rate according to the advertised receive window. This is because different senders can implement different versions of TCP. For this reason the element has a possibility to fail in achieving a target bit rate for the current TCP flow. However, the calculated window size can be viewed as a starting point. The KP can from this estimation adjust the window size according to feedback retrieved from the streaming clients. More information about this reasoning process is addressed in section 5.3.3.

Checksum Calculation

After a new window size for a TCP ACK is calculated it is important to note that the checksum which is located in the TCP header needs to be updated. This is because a receiver of a TCP ACK will use the checksum to control if the TCP header has been tampered with. The receiver of a TCP ACK will accept or dismiss

packets depending on if they pass the checksum control.

4.3 Web User Interface

The web user interface (webUI) is where the user can interact with our proposed home gateway. We implemented this interface in inspiration of how other home gateway producers do this today. Most home gateways have an interface where users can set security mechanisms, IP address allocation, etc. Our webUI can be considered as an addition to these standard user options, where the streaming flows and their corresponding parameters passing through the gateway are presented. It is also possible for the user to set its preferences toward media streams, in the form of individual bit rate control. In addition to the technical implementation of the webUI, we wanted to have a design that is appealing to the user. This is to enhance the experience of user interaction. The webUI is presented in figure 4.6.

In order for the webUI to run, we implemented a HTTP server which runs on port 80 in the home gateway. We chose the Apache HTTP Server because it is open source and is well documented. Another reason for choosing Apache is its Linux support, which is the operating system our home gateway runs on top of.

The web site presented to the user is written in PHP [35], AJAX [36] and is styled using CSS [37]. All technical functionality was developed using these technologies. The reason for using PHP in combination with AJAX is that AJAX makes it possible to use JavaScripts to update information presented in the webUI. All the information and the graphical presentation are then updated dynamically without having a disturbing effect on the user experience. Without AJAX, the user would need to refresh the webUI manually to get an updated view of the current traffic statistics. A styling template was used to improve the user interface appearance.

Figure 4.6: This figure demonstrates the webUI when there are two active flows traversing the home gateway. Parameters collected from each Silverlight client is presented to the user. A dropdown menu is also presented for the user where it is possible to set the bit rate at which the flow should be controlled to run at.

Chapter 5

Implementation

In this chapter we will discuss the implementation done in our project. We will explain the code written to enable Silverlight clients to send application specific parameters at run time. The Knowledge Plane implementation in the home gateway receiving and utilizing this information will also be explained, in addition to the statistics server that aggregates global service statistics. The chapter will be concluded by a section describing the element implementation process using the Click framework.

5.1 Information Gathering: Altering Client Code

In this section we will cover the basic functionality of our Silverlight client. Our client will be a standard Silverlight Smooth Streaming application with the addition of a reporting service. This reporting service will be tasked with sending application specific information describing the media stream. The information will be sent to the home gateway which will use it for network configuration. In addition, the information will be sent to a statistics server where it will be aggregated for post processing by the service provider.

5.1.1 Building a Silverlight Client

To test our custom code we needed a running Silverlight application. Using Visual Studio 2010 and a template, we built our Hello World application. When Silverlight was up and running the next addition to our application was to add a SmoothStreamingMediaElement. This is a media player from the Silverlight Media Framework. This player was then initialized to provide adaptive streaming. To simplify our client code and the need for video files for adaptive streaming, our client retrieves its video material from an already existing Smooth Streaming server. That way we could have control of the player and retrieve information from it without hosting our own material. Using this player we explored what parameters that was accessible directly from the Silverlight client.

Table 5.1: Parameters collected from Silverlight client

event	The type of event triggered in Silverlight client
eventMessage	A message in relation to the event
playState	Current playstate of the Silverlight player
fullScreen	Boolean value indicating if player is full screen
clientId	A Silverlight client globally unique identifier
avgProcessorLoad	Processor load of the client computer
avgProcessLoad	Processor load caused by Silverlight
videoDownloadBitrate	Current rate of streaming
avaliableVideoBitrates	Video bit rates the media is offered in
myIp	IP address of the streaming client
ispAS	The AS number for the client IP, its ISP
sourceIpAndPort	IP and port of streaming server, as DNS name
timeIncrement	Time since application was loaded, in seconds

5.1.2 Retrieving Information from Silverlight

Silverlight pages have corresponding code behind files. In these files we can add code that should run with the page. From the code behind we can observe parameters pertaining to the video player. Using the Silverlight framework, we can also monitor terminal equipment parameters such as CPU usage. The main idea is that when we observe a change in these parameters, the information is then sent to the home gateway. This allows us to have instant updates from the application to the KP.

The Silverlight code uses a set of event handlers to notify if changes have occurred to the player or video stream. When such an event occurs, methods are invoked to retrieve a set of system parameters. The combined information from the Silverlight event and system parameters are passed in a HTTP POST message to the KP. The parameters retrieved are represented in table 5.1. An example of a running Silverlight client sending an HTTP POST message is shown in figure 5.1.

These parameters will be retrieved and sent when the video player triggers a PlayStateChange, FullScreenChange, BitRateChange, Initial- or PeriodicEvent.

The PlayStateChange is triggered by a change in playing state, i.e. playing/-paused/stopped/buffering and other possible player states. FullScreenChange is relatively self-explanatory, it occurs when the user enables or disables full-screen mode. BitRateChange event will be triggered when the Silverlight client decides to change its video download bit rate. This allows us to monitor what it is currently running and also be notified when it changes. This will alert us if unwanted competition for resources is present. Another use of this information is that it will tell us if our traffic manipulation is working.

In addition to these player and media specific events we have included some time activated events. InitialEvent is sent a few seconds after the player is loaded, to give notification to KP that it exists and what its status is. After the player is loaded, a PeriodicNotification will be sent every minute as a keep-alive message. This way the router is able to detect players that are closed and can remove them from its

registry when a PeriodicNotification is not received for some time.

Most of the parameters can be retrieved from Silverlight directly, either through event notifications or method invocations to retrieve variables. There are two exceptions though, namely myIp and ispAS. The Silverlight framework does not give access to information from the network or transport protocols. We have solved this by setting up a simple php script on an external server that we can connect to via HTTP and it will respond with our IP and the AS-number of our ISP. The script can be found in Appendix A. It first retrieves the IP address using php and then does a Linux shell execution of "whois" on the IP to find the ISP AS number.

Figure 5.1: Screenshot of our Silverlight client where the collected parameters are posted to screen. The parameters can be seen in larger scale in figure 6.14

The KP application has a separate web server in addition to the Apache web server running the webUI. This web server is coded in Java and will read the incoming HTTP POST messages and parse them to a SilverlightMeasurement object in the KP. These SilverlightMeasurement objects are stored in lists according to what client they originated from. This way, the KP will obtain and store all measurements it receives from the startup of a client until it stops.

5.1.3 Sending the Parameters

After retrieving the wanted information it must be sent somewhere. In our implementation we send the information to two separate servers. One of them is our home gateway where the KP resides. The other server is thought to be a global statistics

server. Silverlight clients will report information to a local server, the home gateway, for the purpose of traffic altering.

At the same time the information will be sent to a global server where information from all the clients is sent. This server can aggregate the information and use it in usage statistics or similar activities that the service provider deems useful.

Figure 5.2 depicts a simple overview of the information flow between the entities involved with service and content delivery as well as the statistics server, home gateway and end user clients.

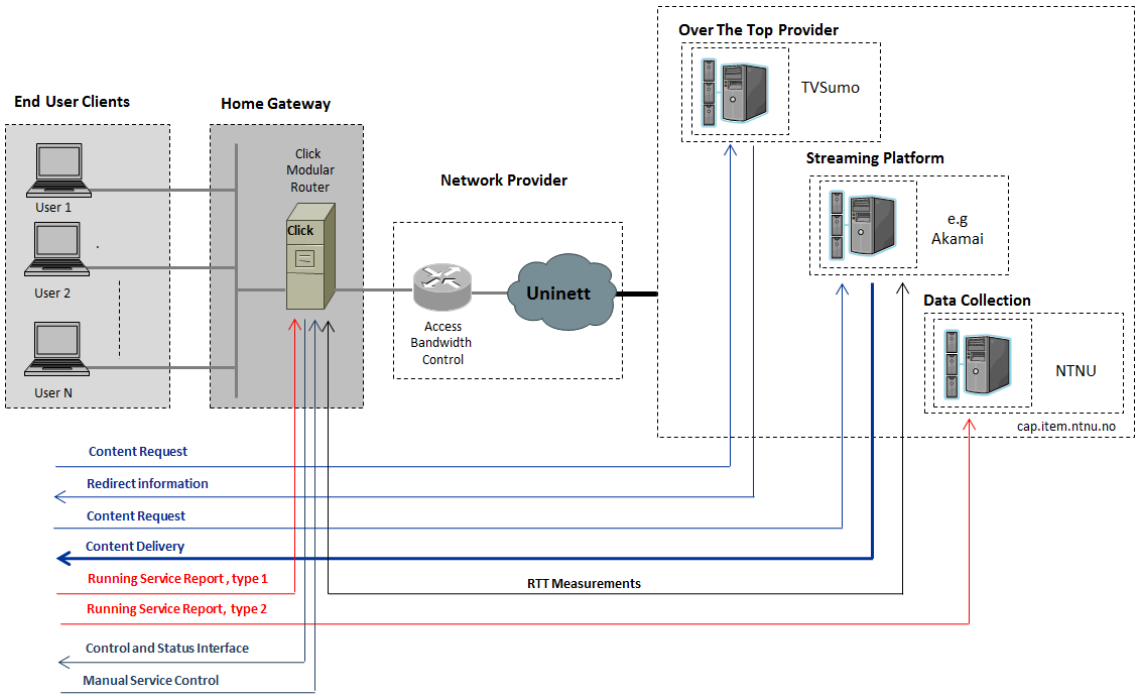

Figure 5.2: This is an overview of how the clients will access content and the information flow that follows. The service provider is contacted and it will redirect to a streaming server. After this the router will start RTT measurements and the clients will do service reports to the home gateway and a data collection. The information delivered to the home gateway will be the basis for the control and status interface presented to the user. Here the user can do manual service control.

To the Home Gateway

Every time an event occurs, this will trigger the client to retrieve lots of system parameters and put them in an information package. These packages are queued for transmission.

The transmission is done by a Silverlight Web Client. As long as there are information packages in the queue it will send them to the designated server. All the parameters will be written to a HTTP POST message in the form of

param1=data1&param2=data2&(...)paramN=dataN. It is then the responsibility of the KP to interpret this information. This is further discussed in section 5.3.

In order to know where to send these messages we have to obtain the IP address of the home gateway where KP resides. We have therefore assumed a static DNS entry in the home gateway. The Silverlight client will send the POST messages to a fixed address like click22.item.ntnu.no(as is the case in our testbed). This way we could instruct the router to resolve this address to itself. Then the Silverlight client would not have to do anything to obtain the address of the router.

To a Statistics Server

We have also included functionality to send a copy of the information package to one or several other IPs. Our thought is that this feature could be useful to aggregate information from all the clients in all the home networks. This could generate useful statistics for the service provider.

We have implemented an example statistics server at cap.item.ntnu.no where a copy of all measurements will be sent. The statistics server is described in section 5.4.

5.1.4 Issues and Imperfections

In implementing the Silverlight client we have met some issues. Silverlight is protective of the Internet layering. By this we mean that Silverlight allows us a bare minimum of information from the lower layers; the transport and network layer. We tried to retrieve the client IP address from Silverlight without success. That is the reason for the remote IP script, which returns the client IP address upon request.

We also tried to get the port numbers for the Silverlight media stream TCP connections. Without these port numbers we cannot differentiate between two sessions coming from the same client in the home network. This is because they will share a source and destination IP pair. We could not retrieve the port numbers, and our system is therefore limited to one streaming application from the same server per client. Having several applications runnig will result in them being classified as one stream by the home gateway. They will then share the same restrictions for bandwidth.

Before starting our implementation, the specifics of how our client should make itself known to the home gateway were unclear. One suggestion was to alter the second byte of the IP header, known as the Type Of Service (TOS) field. By setting this field to specific values, we could use them in a local context to recognize what IP packets were media packets. A coding scheme could be made so that our Silverlight application could communicate its presence and signal quality levels by piggybacking the IP packets TOS field. The Silverlight framework did not allow us this per packet control and this approach could not be used.

There is a lot more information to be collected from the Silverlight clients. This information is the information the service provider sends to the client in something called initParams. Information like the name of the service is sent as an information package. This is contained in initParams at the client side. The problem is that

retrieving this information would not be generic. A separate implementation would have to be done for each service provider to implement the functionality. Obtaining this information could prove useful, depending on the amount of information passed from the service provider. As a minimum, we should expect to find a service name. Naming the services would make the webUI more user-friendly.

5.2 Alternate Silverlight Client: Application Controlled Bit Rate

When developing the Silverlight client for reporting parameters to the KP we discovered a second form of rate control. This approach is strictly confined to the application layer of the services and does therefore not include any Click router functionality.

When Silverlight sends a POST message to the KP, it will respond with an "all is well" message and Silverlight will interpret this as a sign that the message was delivered successfully. One addition was made to the response from the KP. When a new measurement is sent to the KP it will respond with the user preference in addition to the "all is well" message. This message is illustrated in figure 5.3.

The Silverlight clients that does not read the message, but merely verify that it got a response will not know the difference. A second Silverlight client we built to test the concept will on the other hand read the response and check if the user has set a bit rate preference. If the user preferred rate is not 0, i.e. it is set to one of the available bit rates, the client will limit its download rate to the given rate.

```
Click22 server: PostInfoPOST /receiveParams2.php HTTP/1.1
Received message at time: 1305767660357
userRate=866000
```

Figure 5.3: Screenshot of the debug text block in our alternate Silverlight client. The Silverlight application is printing the response it gets from KP. In addition to a standard message from the server with a timestamp, the client is told what rate the user prefers

Silverlight Smooth Streaming has implemented the possibility to deny the client certain bit rates. It is useful when the content provider want to limit HD material to premium accounts or similar cases. By using this functionality we can read all available bit rates the video stream is offered in. Then we can make a custom list of rates that we approve of and call a function on the stream named SelectTracks. The player will then act as if our user defined level is the highest rate possible.

This approach offers high precision rate control. And as we only remove the rates higher than the user preference, the client will still be adaptive and decrease its rate if it experiences bandwidth limitations.

The Silverlight client is just that, a client. We can therefore not send a message to the client when the user sets a rate. The parameter has to be piggy-backed on the response from KP when a POST message is received from the client. In our implementation the client will send a PeriodicNotification every 60 seconds. This will be the worst case response time from a user preference is set until the client has decreased, or increased, its rate.

A possible solution to decrease this response time is to increase the frequency of PeriodicNotifications. The KP would then have opportunity to communicate to the Silverlight client more often.

5.3 Implementation of KP

Our KP instance is implemented in Java. Its responsibility is to aggregate the information coming from the Silverlight clients that are connected to the home gateway.

As previously explained, the home gateway will use a customized DNS entry to direct all the client measurements to the KP. At the receiving side, the KP needs to interpret the incoming information and structure it so that it can be used to determine how to alter the traffic in order to improve the traffic situation.

In this section we will describe the most important aspects of the KP implementation. The web server allows both the Silverlight clients and the webUI to communicate with the KP and is described in section 5.3.1. The web server will relay the incoming information to the ServiceSessionBank which serves as a registry of all active Silverlight clients. The ServiceSessionBank is the central entity for keeping state and history of the ongoing video streams; we will discuss this in section 5.3.2. Finally we will describe the ActionReasoner which is the entity responsible for communication with Click and the AP actions. The ActionReasoner will be discussed in section 5.3.3.

5.3.1 The Web Server

In addition to the Apache web server for the webUI, described in section 4.3, the KP process itself runs a Java web server. The Apache server is running at port 80 which is the default HTTP port. The Apache server will be the one you reach if you contact the home gateway at its address using a web browser.

At port 23500, the KP web server is running. The reason for this extra web server is that it is run by the KP and we can therefore make a HTTP POST message to result in a method call in the KP. This will make the router more responsive than in our previous work. We had a system where a call to the KP was processed and written to file by an Apache web server and this file was read by the KP periodically. Now we will have real time processing of incoming messages.

The KP web server has two functions

- Receive Silverlight Information
- Receive User Preferences

To fully accomplish these it also needs a third function. When called for, it has to respond with a clientaccesspolicy.xml file which Silverlight requires for cross-domain access.

The Silverlight parameters are structured as described in section 5.1.3. The web server should obtain these parameters and respond with a HTTP 200 OK message.

After successfully receiving and responding to the Silverlight client, the parameters are passed on to the ServiceSessionBank instance which obtains lists of all active Silverlight clients. This is described in section 5.3.2. This entity is responsible for storing and organizing clients and measurements.

The second source of information is the webUI. The user is presented with a collection of streaming information in a graphical representation of the services that is currently running. When a user makes a decision pertaining to the quality of one of the services this will also be reported to the KP using HTTP POST messages.

The KP then needs to identify the service the user has marked and what changes the user wants. This information is written to the ServiceSession identified by the Silverlight client id.

This way the KP web server allows for Silverlight clients and end users to report directly to the running service. This has greatly improved the dynamic processing of events. Instead of periodically reading input and calculating changes, we now handle each event separately and dynamically.

5.3.2 The ServiceSessionBank

We mentioned that this entity is a central registry of what ServiceSessions are currently streaming to the end users. In the Java application a ServiceSession instance is the equivalent of a Silverlight client run by one of the end users.

Each Silverlight client generates its own guid (Globally unique identifier) which we also use in the Java code to separate ServiceSessions. This way we can match the incoming Silverlight measurements to a ServiceSession by using the client id (the guid).

A ServiceSessionBank has its own Parser, GuiXmlWriter and ActionPlaneSessions objects. These entities simplify the work of controlling ServiceSessions and receiving new measurements as well as reporting to the webUI.

Figure 5.4 graphically describes how the ServiceSessionBank organizes its data. When a new measurement arrives from the web server, it is sent to the parser. The parser will read the incoming string and parse this to a SilverlightMeasurement object. This is a domain object that is made up of variables and get-methods for reading the information. They are instantiated by the parser and cannot be changed, they just represent a measurement.

The ServiceSessionBank contains a list of ServiceSession objects. Each of these ServiceSession objects contains a list of SilverlightMeasurements that originated from this ServiceSession. When we have gotten the parsed SilverlightMeasurement, we simply iterate through the list of ServiceSessions in the ServiceSessionBank until we find a match by client id. Then we add the measurement to this ServiceSession list of measurements.

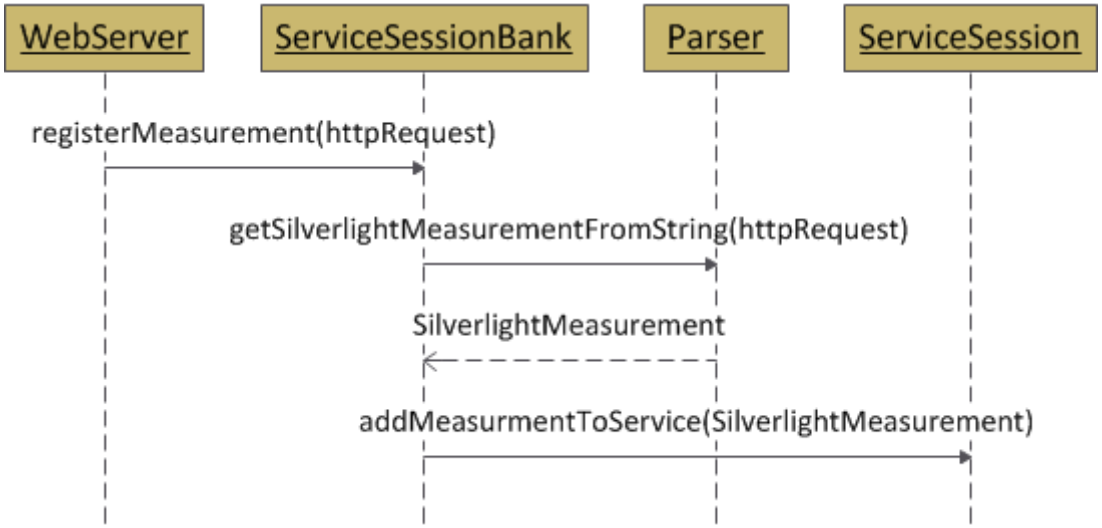

Figure 5.4: This is a high-level representation of how a HTTP POST message is handled after being received by the web server. The request body will be forwarded to the ServiceSessionBank which parses it into a SilverlightMeasurement using the Parser class. This SilverlightMeasurement will then be sent to the correct ServiceSession and added in a list of measurements from that session.

This way all the information coming from the Silverlight clients will be stored under the ServiceSessionBank in lists of measurements. One list of measurements per ServiceSession; where the measurements in the list share client id.

When a new SilverlightMeasurement is added we need to update the webUI, this measurement will contain the newest information from the Silverlight client. The ServiceSessionBank then calls a method on the GuiXmlWriter that will write a status report from all the ServiceSessions to an XML file located in the Apache web server directory, see figure 5.5. This information is then read from the web interface and presented to the user. The webUI was described in section 4.3.

With new information we also need to check if this will affect our AP settings. This is where the ActionPlaneSessions come into play. The ActionPlaneSessions entity can be considered as a ServiceSessionBank for the ServiceSessions that are marked for traffic alteration in the AP.

It is therefore a subset of the ServiceSessions contained in ServiceSessionBank. Every time a change in the lists occurs, we call the ActionPlaneSession entity. It will iterate the ServiceSessionBank list of ServiceSessions and make a list of the ones that the user has marked for special treatment. If the list differs from the one it already has, it will call the ActionReasoner with an alert that changes has occurred in the list of ServiceSessions that the ActionReasoner should be aware of. Once again, we refer to figure 5.5 for a flow chart representation of these events.

The ActionReasoner is, as its name implies, the entity that reads the user preferences and calculates changes we could do to the traffic. This entity will be explained in section 5.3.3.

The ServiceSessionBank also has a ServiceSessionBankCleaner. The responsi-

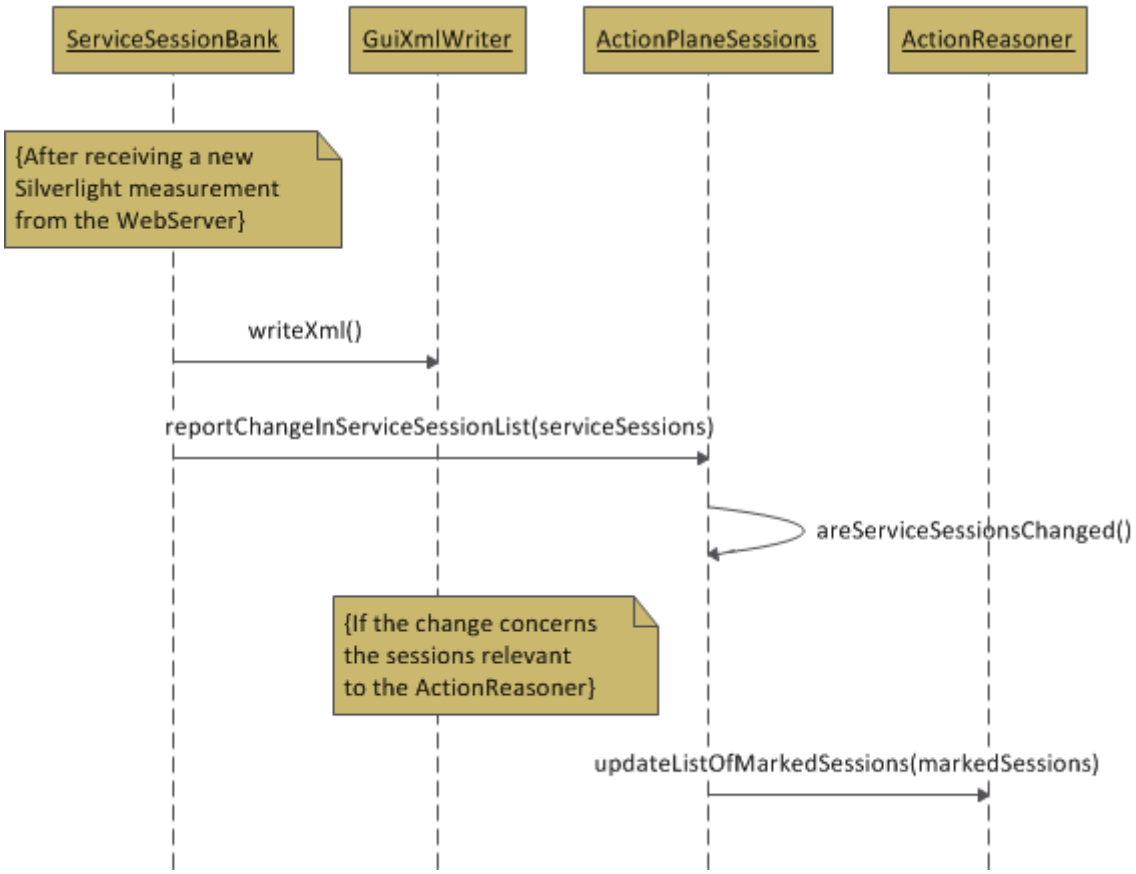

Figure 5.5: This figure shows post processing in ServiceSessionBank after receiving a new SilverlightMeasurement. We need to update the WebUI and alert the ActionPlaneSessions that a change may have occurred to a session that is relevant to the AP.

bility of the cleaner is to periodically iterate all the ServiceSessions and calculate the time since last received measurement. If this time exceeds a given threshold, SESSION_INACTIVE_LIMIT specified in the Constants class, the session will be purged. This way we make sure that sessions that are inactive for too long will be removed.

5.3.3 The Action Reasoner

This is the entity that does the reasoning and writes configuration data to Click. As previously described, the ServiceSessionBank will check to see if the list of services has changed. If it has, the ActionReasoner will be notified.

This way, the ActionReasoner will always contain a list of the services that are currently marked for special treatment by the user. This list will be iterated and a Click element handler, described in section 3.3.2 will be written. This is the way the KP communicates with the Click which is our implementation of AP in the KP-MP-AP approach.

The handler will be written to include the source and destination IP of all the services that should be altered. In addition we will include a RTT measurement and a target rate to aid the Click element calculations. The Click element we have implemented will change the window field in the TCP headers of the ACK packets generated from the video stream. Using the source and destination IPs, Click will be able to isolate the packets pertaining to the video stream TCP connection. Then it will calculate a desired window size using the RTT measurements and target video rate.

This handler information will be periodically updated. If the rate wanted by the user is changed we will instruct Click to set target rate in between the rate the user wants and the next rate offered by the content server. Let's look at an example; if the stream is offered in 0.5Mbps, 1.5Mbps, 2.5Mbps and 3.5Mbps and the user selects 2.5Mbps as the target rate. Then we will calculate a target rate between 2.5Mbps (the user specified) and 3.5Mbps (the next available rate), i.e. 3Mbps.

The window adjustment is a effective way to control the throughput, but we have experienced that there is a large margin of error. The reason we instruct Click to set the target rate in the middle of two levels is that we then expect it to fall to the lower one of the two. In our previous example, trying to limit the throughput of the TCP connection to 3Mbps, we hope to avoid the Silverlight client trying to obtain the 3.5Mbps quality level. At the same time we try to allow the Silverlight client to obtain the 2.5Mbps quality level.

After starting in the middle the KP will adjust the window size to allow slightly more or slightly less throughput depending on observed traffic. When a traffic limit is set the KP will continuously wait for a new Silverlight measurement to arrive. If it doesn't, this means that our efforts are not having effect. The KP expect to get a DownloadBitrateChanged event with a message that the Silverlight client has decreased its rate. If this is not the case, then it needs to decrease the target rate written to the Click element handler. It will do iterative decreases until the wanted effect is achieved. We have implemented a limit for how often the Click handler is allowed to be rewritten. This is defined in the Constants class as CLICK_CHANGE_INTERVAL.

As a default we have tested with a CLICK_CHANGE_INTERVAL set to 10 seconds. We feel that the video player should get 10 seconds to alter its behavior before getting further punished by bit rate changes. We need to let it converge.

The same procedure is started if the opposite should prove to be the case. That is, we try to decrease the rate from 3.5Mbps to 2.5Mbps by setting target rate 3Mbps, but the video stream falls below 2.5Mbps. If this is the case, that the video is streaming at 1.5Mbps while we want it to be 2.5Mbps, then the target rate will be incrementally increased every CLICK_CHANGE_INTERVAL.

This approach is most effective when the rate difference between the quality levels is large, as in our example. Then we are allowed larger margins, in this case 0.5Mbps in each direction to closest quality level. Then we will often obtain good results quickly and be able to stabilize. Some adaptive streams have lots of quality levels that are very close to each other. We have done testing on a stream that is offered in 300Kbps and 427Kbps, see figure 5.6 Differentiating between them is

tricky, and we end up incrementing and decrementing the target rate without the stream becoming stable.

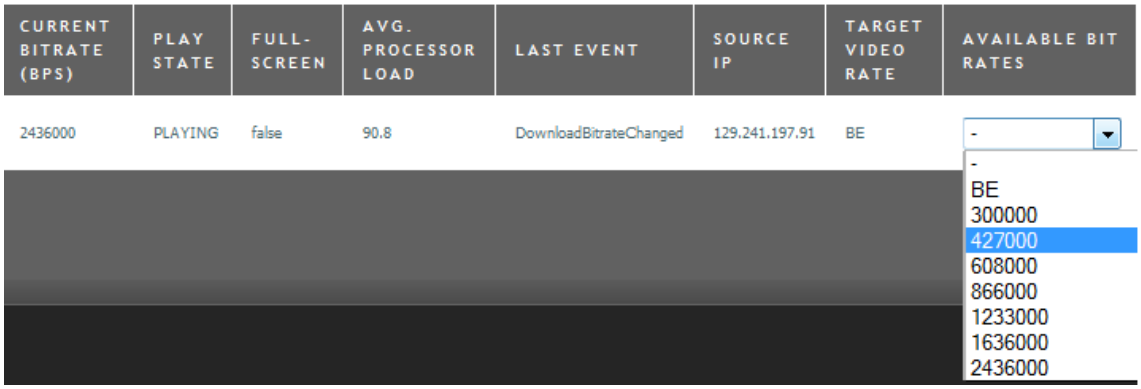

Figure 5.6: This adaptive stream has four quality levels below 866Kbps. Differentiating between them and obtaining a stable rate is difficult.

5.4 Statistics Server

As mentioned previously we propose that the Silverlight application in addition to sending information to the home gateway also sends a copy to a statistics server.

The reason for sending a copy of the information to a common server can be as varied as the service providers are many. We believe that this information could serve as an important source of QoS/QoE measurement from the clients. Using this information the service providers can aggregate data from all of their clients. The addition of more parameters is relatively straight forward in the Silverlight code, should there be more parameters of interest to the service provider.

The service provider would be able to get statistics of what its users are watching, for how long and at what quality. As the service provider would have the bit rate of the client and AS-number of the ISP, one could investigate if users from certain ISPs experience lower QoE. This is valuable information to a content provider and could point to a bottle neck in the content delivery network.

We have set up a simple statistics server for proof of concept purposes. In this section a brief introduction to this server is given.

5.4.1 Receiving Information

As previously stated, we send data from Silverlight by HTTP POST messages. The message coming from Silverlight to the statistics server should therefore be read and stored. We use a php script, see Appendix B for source code, to extract the POST parameters and write them to a log file. The log files will have a daily rotation to avoid growing too large.

In addition to the parameters passed through the POST message, see table 5.1 for a complete list, we add some server side parameters. As shown in the source code we

append sequence numbering, a couple of time stamps and a client id representation without the hyphens as it is easier to sort by.

5.4.2 Organizing Information

To be able to use the data collected by the statistics server, it needs to be structured. As both MATLAB[38] and Mathematica[39] have MySQL connectors, storing the data in a MySQL database became the solution. This way we could allow direct access to the data from programs that are to analyze the data.

Instead of making the php script to do SQL INSERTs directly to the database, mainly for performance reasons, we went for a batch insert approach. The log files written by the log script can be read and inserted as a daily batch. We made an example application that will read a log file using our syntax and insert all log entries to a MySQL database. This application was written in Java and is attached to this report (BatchSqlInsert.jar).

This MySQL database was set up on the same server and all log files were inserted. This way, the statistics server will write incoming data into log files by using the batch insert application periodically.

Our master thesis does not address the actual analysis of the data. We have simply just made the data available. This is in essence a "value-added service" from our point of view, as our primary objective is the traffic monitoring and manipulation.

5.5 Implementation of the Click Elements

This section contains a detailed description of the implemented Click elements and how they were developed in C++. These were proposed and discussed on a high level in section 4.2. The reader should have some programming experience as we do not explain the code line by line. The focus will lie on the most important functionality built into the element. Methods are described functionally; we refer to the code attached to this report for further details. We start by explaining how the ACK Pacer was developed using C++.

5.5.1 Implementing the ACK Pacer

To avoid confusion while reading, the ACK Pacing Element is referred to as PacketPacer. This is a generic name for its purpose.

5.5.2 Basic Click Element Properties

The first step in implementing a Click element is to make the header and source file as described in section 3.3.3. These were named PacketPacer.hh and PacketPacer.cc accordingly. The header file was declared as follows:

```
#include <click/element.hh>
```

```
class PacketPacer: public Element { public:

PacketPacer();
~PacketPacer();

const char *class_name() const { return "PacketPacer";}
const char *port_count() const { return "1/1";}
const char *processing() const { return "h/lh";}

int initialize(ErrorHandler*);
void push(int port, Packet *p);
Packet * pull(int);
void add_handlers();
```

It can be seen that the PacketPacer class extends already built in methods from the Element class. Then the methods from the Element class can be used. The element.hh file has to be included above the class declaration.

After the class declaration, a constructor and destructor were declared. This element does not have a configuration string, and the constructor is therefore empty. A destructor in C++ is usually used to clean memory for a class object and its contained class members.

After the destructor, mandatory methods for every element in the Click framework are declared. These are addressed stepwise as follows:

class_name

Used by Click tools to retrieve the name of the element.

port_count

Sets and returns the port definition of Click elements. Here, "1/1" means that this element has exactly one input and one output port. Several other settings can be used, and these are found in the Click documentation.

processing

Defines element port settings. Here, "h/l" sets the input port to PUSH and the output port to PULL. Several other options can be used, and examples of these are found in the Click documentation.

initialize

This method is called just before the router is put online. Hence, it is where all error checking should be placed in order to avoid logical errors.

push

A method which implements the logic for processing a packet that is sent to this element. It is defined with a port number, and a pointer to the packet class. The packet class contains several useful methods for packet processing.

pull

Defines the pull port of this element. The pull port is where other elements ask for packets. This method takes a parameter used for numbering the port. As with PUSH, this method also has a pointer to the packet class.

add_handlers

This method initialize all handlers written within the element code.

5.5.3 Our Functionality

During implementation it is useful to print debug text to check the functionality of the program. This is offered by the Click framework through the StringAccum class. It is included and instantiated in the header file. The StringAccum object can therefore be used in the source file as follows.

```
* In packetPacer.hh:

#include <click/straccum.hh>

StringAccum sa;

* In packetPacer.cc:

sa << "Printing some debug text" << "\n";
```

The « operator can be viewed as + for string handling in programming languages like Java and C#. The debug text ends with a new line operator. Then all text written after this line will start on a new line. The debug text can be viewed in Ubuntu by using the dmesg command. Running dmesg prints the message buffer of the kernel to terminal.

Another important building block of Click elements are the handlers. They are defined within the add_handlers method in the source file. The main handler used by the PacketPacer is sessionInfo. It is defined as follows:

```
add_data_handlers("sessionInfo",
Handler::OP_READ | Handler::OP_WRITE, &_sessionInfo);
```

The handler options are set within the parentheses. First, it is given a name. Next, the read and write options are set. Then a user can access and alter the handler using a text-editor. Last, the address of the variable which the element code uses is taken as input. _sessionInfo is declared as a string in the header file and is used accordingly. The structure of the sessionInfo handler is presented next.

```
Source IP:Destination IP:Pacing Value, ... , ... ,
```

Other elements which send packets to the PacketPacer will not know if the Packet-Pacer is idle or not. A queue and a queue manager are therefore needed to handle packets that arrive when it is busy.

```
#include <click/standard/storage.hh>

Storage _primaryStorage;
Storage _secondaryStorage;
```

The storage class is not described in the standard Click documentation. But the header file is contained in the Click framework. Hence, an include of the storage class is placed in the top of the PackedPacers header. The storage class can be viewed as a queue manager and have variables maintaining the queue state. These are queue capacity in addition to references for head and tail positions of the queue. Useful methods for queue management are also contained in the storage class. These will not be discussed further in this report, but can be found in the storage header file.

After the queues are created, there is a need for methods that store and extract packets from them. How packet enters the PacketPacing element is illustrated in figure 5.7 and the enqueue method is defined as follows:

```
inline bool enqueue(Packet*);
```

In the enqueue method, the incoming packets source and destination IP address is checked against the source and destination address written in the sessionInfo handler. If there is a match, then the packet is placed in the primary queue. If there is no match, then the packet is placed in the secondary queue. When a packet is stored in either queue, then the tail variable in the storage class is updated. The tail variable is a reference to the queue position of the last packet.

If either the primary or secondary queue is full, then the packet is dropped and the _drops variable is updated. _drops is the total number of dropped packets from both queues. The _drops variable has a read handler which enables the element to being monitored when it is running. We can then check if the element drops a lot of packets, or if it is performing well with no packet drops. The enqueue method returns a boolean value to indicate if a packet was stored successfully or not.

When a packet is stored in the primary queue, a method is used to set the pacing between two packets. This method is defined as follows.

```
inline void setLeaveTimeForPacket(int);
```

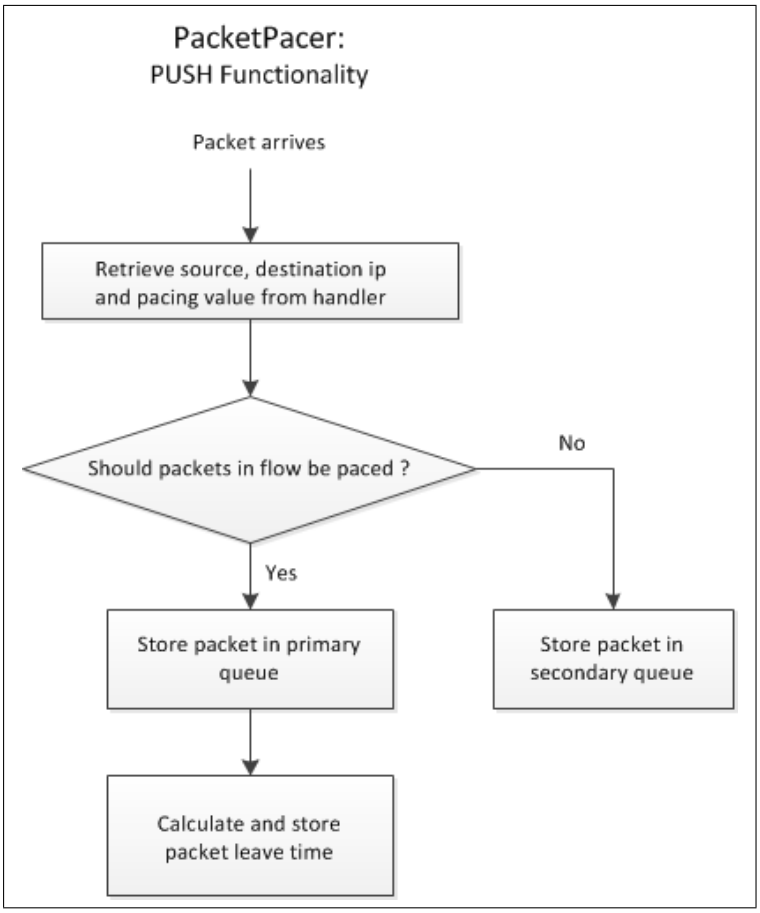

Figure 5.7: Flowchart which illustrates how incoming packets are handled. Information used to identify packets is retrieved from the serviceInfo handler. If the packet is identified, then it is stored in the primary queue and its leave time is calculated. If the packet is not identified, it is stored in the secondary queue.

This method uses the pacing value set in the serviceInfo handler to calculate the timestamp of when a packet should leave the element. The pacing value is the amount of time between packets in milliseconds. If the pacing value is set to 5 in the serviceInfo handler, then all packets belonging to this flow should leave the element 5 milliseconds apart. The pacing between packets is controlled as follows. The timestamp of when a packet leaves the element is stored. The next packet in the corresponding flow is then given a leave time of 5 milliseconds + the leave time of the previous packet. Hence, all packets belonging to this flow will have a leave time which is 5 milliseconds apart.

Another feature added to the leave time calculation was that we had to consider if a packet arrives at the element later than the pacing value set in the handler. Then this packet would pass the element without experiencing any constraints. Because of the bursty nature of Smooth Streaming, we discovered the need for a solution to also constrain these packets. Hence, we add a delay equal to the pacing value to packets which arrive later that the leave time of the previous packet + the pacing

value.

The calculated leave times for incoming packets are stored in a separate array called packetLeaveTime. The packetLeaveTime array uses the same indexing as the primary queue. When a packet is stored at position 5 in the packet queue, then the calculated leave time for this packet is placed at position 5 in the array which contain the leave times.

Next, we present the method which select and send packets from the queues.

```
inline Packet* decreaseQueue(Storage*, Packet**);
```

This method is used by the pull method to retrieve packets from either the primary or secondary queue. It takes the storage class and an array of packets as input. The storage class is the queue manager, while the queue is where the actual packets are stored. A flowchart of how a received pull is handled can be seen in figure 5.8.

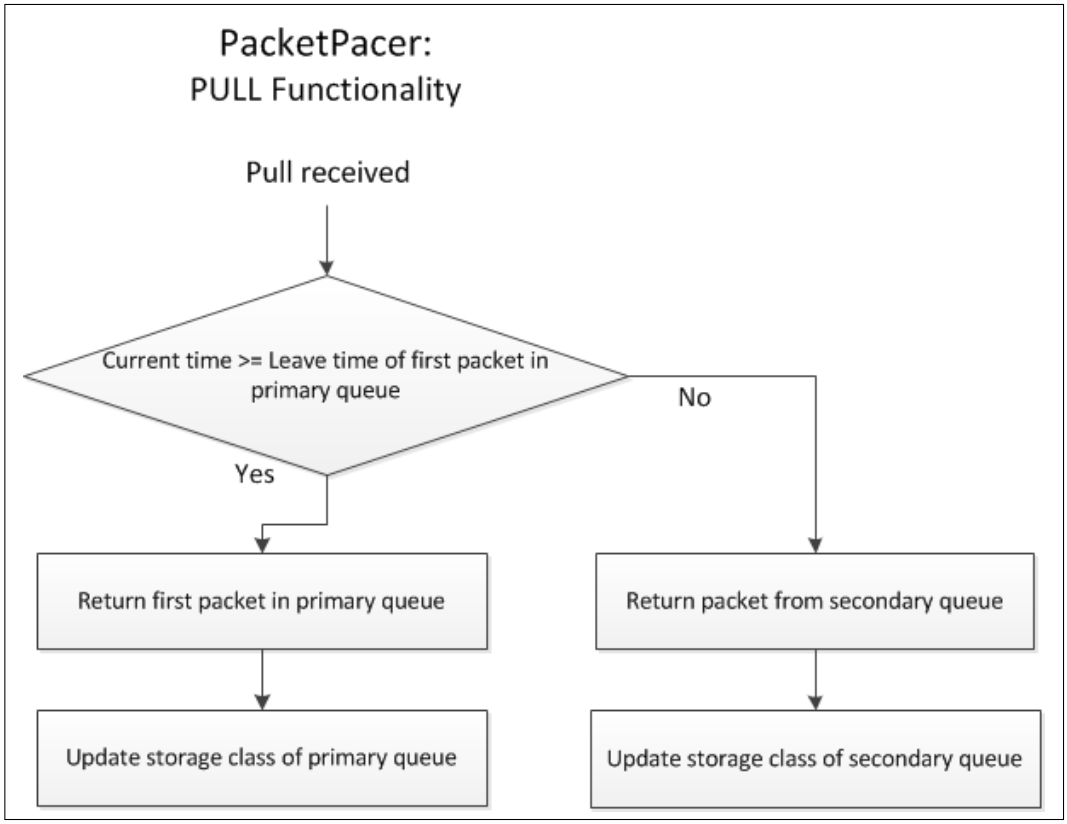

Figure 5.8: Flowchart which illustrates how packets leave the PacketPacer. Every time a pull is received, the current time is compared to the leave time of the first packet in the primary queue. The first packet in the primary queue is sent if the current time is larger than the packet leave time. If not, then a packet is sent from the secondary queue.

Controlling which packet to send from each queue is done by the storage class. A head variable is maintained as a reference to the first packet in the queue. First,

the packet leave time is retrieved from the packetLeaveTime array. This is done by using the packet queue position as index in the packetLeaveTime array. Next, the packet leave time is compared to current time. The packet is sent if the leave time has expired. However, if the leave time has not expired, then a packet is sent from the secondary queue. After a packet is sent from either of the queues, then the head variable is updated using the storage class.

Issues with the ACK Pacer Implementation

The implementation of the ACK Pacer has not been done in complete compliance with the proposal in section 3.3.3. First, the time between incoming packets was not measured. Instead, we used the leave time of the previously sent packet and added a delay in milliseconds which decided when the next packet was sent. This way we kept the time between each packet sent stable and we could adjust it through the Click handler.

Another issue with implementing this element was the support for altering the sending rate of several simultaneous TCP flows. We implemented quicksort in order to sort the array where the packets are stored based on each packet leave time. This approach failed. This is probably because of how the storage class keeps track of the first and last packet in the queue. By sorting the queue, we unknowingly altered some head or tail pointers that made the storage class fail. In our current implementation, if several TCP flows are altered we risk that they block each other. A packet from one flow might get a low leave time while another packet is in front of the queue with a larger leave time. The packet with the low leave time will not be sent before the blocking packet is sent. This will cause some unwanted behavior when multiple streams are altered simultaneously. It should be noted that all other traffic will pass this element through a secondary queue. This implementation deficit will only affect packets from the streams selected for ACK pacing.

5.5.4 Implementing the Receive Window Manipulator

The mandatory methods required by the Click framework described in section 5.5.2 are equal to the ACK pacing element. The main difference is that this element has both PUSH input and output ports. The basic functionality of this element is as follows. If the packet is marked for receive window manipulation, we rewrite the window field in the TCP header. The new value of the window field is calculated from parameters given by the KP. Then the packet is sent to the next element in the forwarding graph.

Because of the push input and output ports there is no need for a queue to store the incoming packets. This reduces the complexity of implementing the element as we can just rewrite the header of an incoming packet and then push it to the next element. We do not wait for an element to pull for a packet, eliminating the need for queues. A flowchart of the element functionality can be seen in figure 5.9.

As in the section presenting the ACK Pacer, we will only outline main methods when describing the element code. The method used for handling packets is declared in the header file as follows:

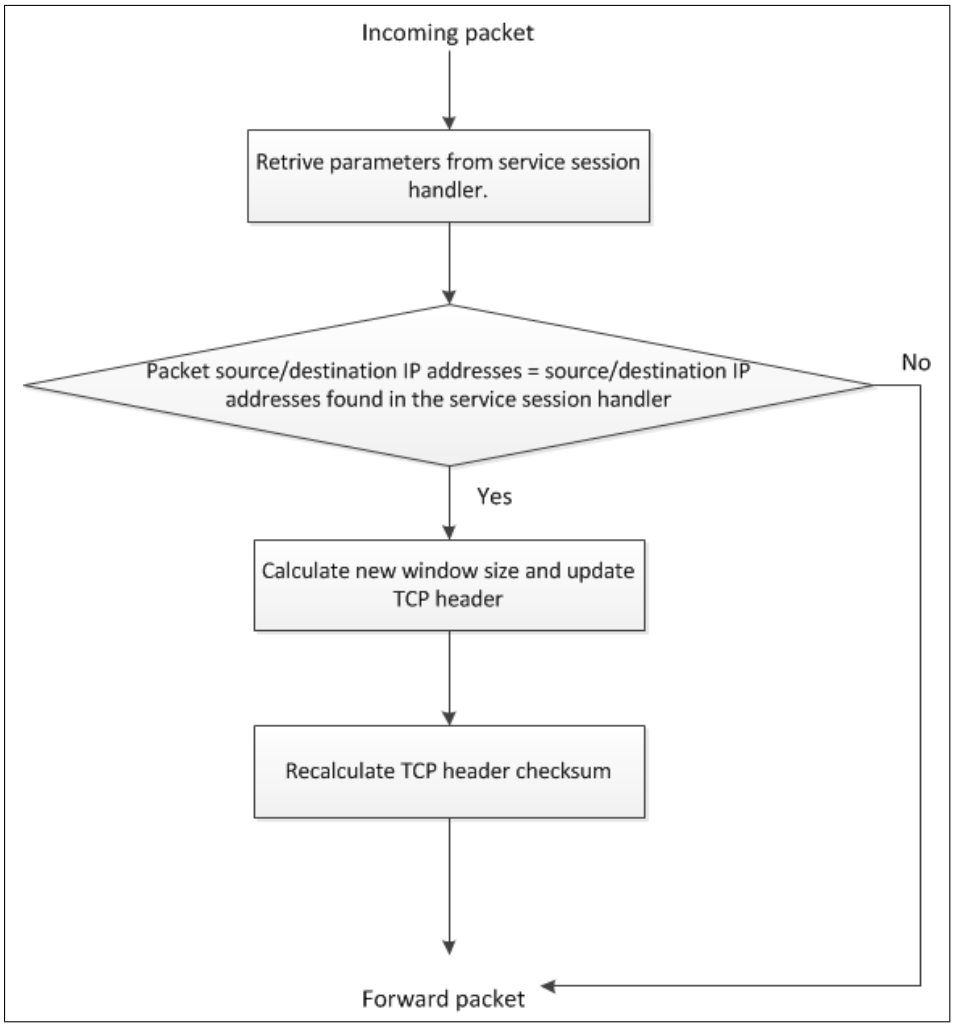

Figure 5.9: Flowchart describing the receive window manipulating element.

```
Packet * TcpWindowFieldReWriter::simple_action(Packet *p);
```

The class name used for the Receive Window Manipulator is TcpWindowFieldReWriter. This was again to have a generalized name for the element. The simple_action command is used when we have a simple element which only handles an incoming packet and sends it to the next element. All functionality regarding packet handling is therefore placed in this method. Next, a handler is used to retrieve parameters needed by this element to identify packets. The handler is declared as follows in the source file:

```
add_data_handlers("serviceSessions",
Handler::OP_READ | Handler::OP_WRITE, &serviceSessions);
```

The handler options where explained in section 5.5.3. It can be seen that this handler is named serviceSessions and its syntax is as follows:

```
Source IP:Destination IP:RTT,Target Bit Rate, ... , ... ,
```

The RTT is the distance between the source and destination measured in milliseconds. When a packet is identified, a new window size for this packets TCP header is calculated. This is done by using the following method which is declared in the header file:

```
inline long TcpWindowFieldReWriter::calculateWindowSize(long,int);
```

This is where the new window size of the packet is calculated. It takes the RTT and target bit rate as inputs. These are then used to calculate the new window size. We have used the general TCP throughput formula for this calculation as described in section 2.3.2.

Before a new calculated window size can be written in the header, we need to extract the old TCP header from the packet. This is done using the following built in Click method.

```
click_tcp *tcpHeader = p->tcp_header();
```

Here a pointer is instantiated in the source file which points to the TCP header of this packet. There are several methods which can be used to retrieve information contained in the TCP header. These can be found in the Click documentation. The most important methods used were the method which set the new window size and the one which updates the checksum of the TCP header. Both are used in the source file and are presented as follows:

```
tcpHeader->th_win = ntohs(_newWindowSize);

click_in_cksum((unsigned char *)tcpHeader, plen);
```

The second method takes the TCP header and packet length as inputs for calculating a new header checksum. Finally, the checksum is set as follows in the source file.

```
tcpHeader->th_sum =
click_in_cksum_pseudohdr(packetCheckSum, ipHeader, plen);
```

Here, the calculated checksum, IP header and the packet length is taken as input. The packet is now updated with a new window size and checksum. Then it is pushed out of the element.

Chapter 6

Testing and Results

In this chapter we will test our proposed solutions in order to outline their characteristics. Individual tests and analysis of our implemented Click elements will be presented. This is followed by the Silverlight client and KP components.

6.1 Testing the Proposed Click Elements

We will in this section present tests which illustrate how our proposed and implemented Click elements can manipulate the bit rate of a TCP session. First, we present the results from tests with the ACK Pacing element. Then tests of the TCP Receive Window element will follow. Tests of both elements are divided into two sub categories. These are an ordinary FTP file download, and an actual Smooth Streaming session. It is important to note that all tests are run over the Internet and not only in a restricted lab environment. The FTP server used is part of the FreeBSD community distribution network. It was chosen because of its capacity and its distance, measured in milliseconds, from our testbed. The server IP address is as follows:

```
FTP server IP address: 128.205.32.24
```

The Smooth Streaming service is tested using TV2 Sumo. This is to test how our proposals will behave in a commercial streaming setting.

The measurements performed in this section are samples and are only performed once. These tests therefore bare no statistical validity as they are not replicated. However, they are done in order to illustrate how the elements perform and show that they can be used to manipulate TCP traffic. To obtain statistical data on how effective these solutions are, further study and comprehensive testing should be done.

6.1.1 Test of the ACK Pacer

This section contains tests of the TCP ACK Pacing element developed and presented in section 5.5.1. The element assumes a close collaboration between the AP and the

KP. This is because the KP is the entity that regulates the amount of pacing between each TCP ACK. The pacing value is incremented when the KP is trying to reduce the TCP senders throughput and oppositely decremented when the throughput should be increased. In this section we will do the KP reasoning process manually by inserting the parameters into this elements Click handler. This was done using nano which is a Linux command-line text editor. Note that this operation needs to be run in the super-user context.

ACK Pacer: FTP Download Test

We conducted two tests where we incremented the pacing of the TCP ACKs linearly. In the first test we increased the pacing with 3 ms every 30th second. This was to see the effect in the transmission throughput when the pacing was gradually increased. The reason for keeping the increment small is to prevent TCP timeout from occurring. TCP timeout occurs when a server waits for a segment to be acknowledged. The TCP session has a timer to determine when it regards the segment as lost. When a timeout occurs, this triggers a retransmission. We try to minimize the amount of retransmissions because this is considered as wasted bandwidth. However, consider a scenario where there is a rapid increase in the number of TCP sessions flowing through the home gateway. Then there could occur a need for limiting some of these flows fast in order to protect the ones with high priority. A TCP timeout could then be an acceptable method, in worst case scenarios, to slow down TCP sessions.

The second test presented in this section will use an aggressive increase in the pacing value in order to decrease the bit rate of the TCP session. We increase the pacing value with 10 ms every 20 seconds. By this aggressive increase we will risk getting TCP timeouts, but we assume that risk in order to effectively reduce the bit rate. After reaching our maximum of 50 ms pacing, we gradually decrease the pacing to observe how the flow restores.

FTP Download With a Linear Packet Pacing Increase

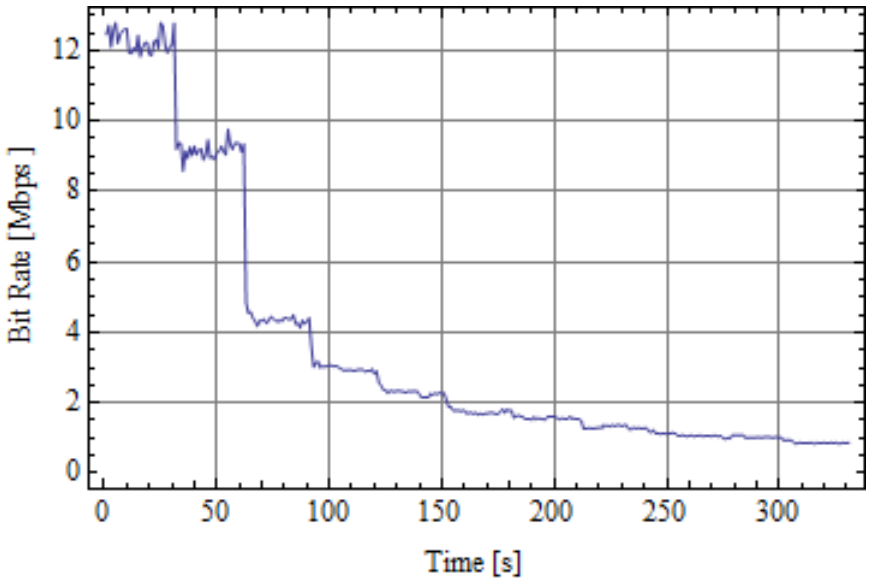

Figure 6.1: This figure shows the measured bit rate when the pacing between packets of this flow is increased. It can be seen that the bit rate is gradually decreased as the pacing is increased. The pacing value can be seen in figure 6.2.

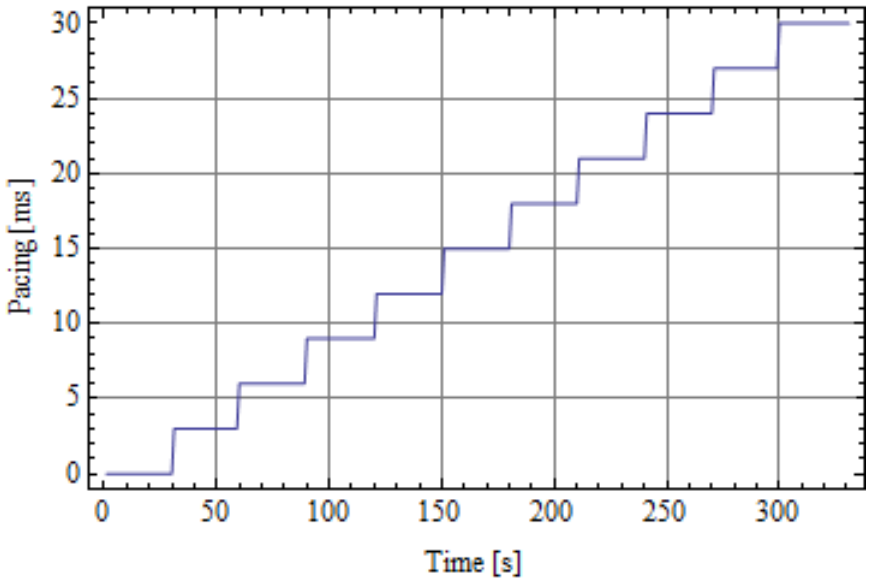

Figure 6.2: This figure illustrates the pacing value between packets leaving the element.

FTP Download With a Aggressive Packet Pacing Increase

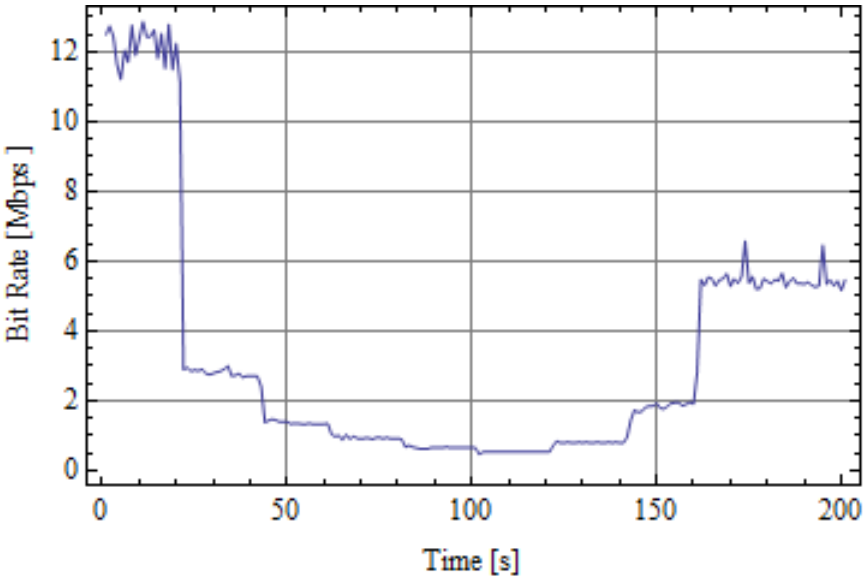

Figure 6.3: This figure illustrates the bit rate of an FTP download when increasing the pacing between outgoing packets more aggressively than in the previous test. It can be seen that the throughput is reduced faster. Another feature this figure outlines is that the bit rate increases when the pacing value is reduced. The corresponding pacing value is presented in figure 6.4.

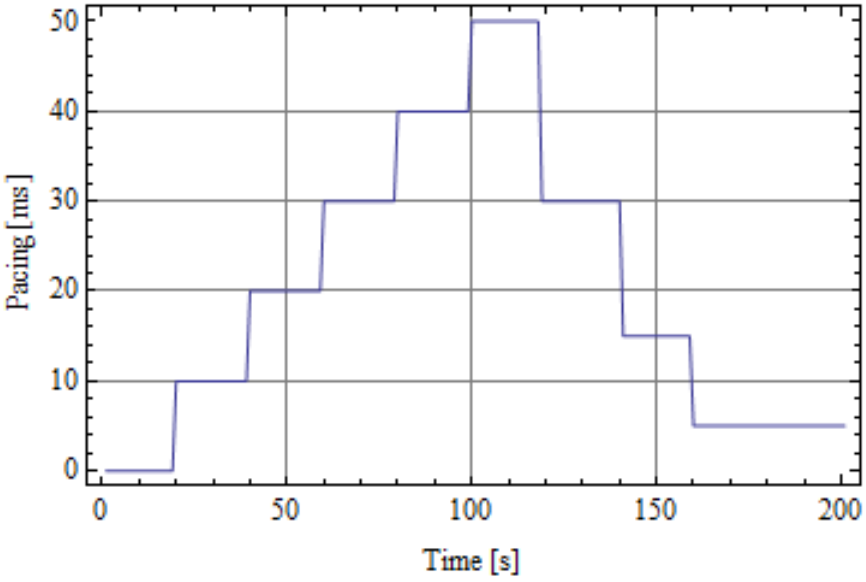

Figure 6.4: This figure illustrates the pacing value between packets leaving the element.

ACK Pacer: Smooth Streaming Test

This test is done using the TCP pacing element in combination with TV2s Smooth Streaming service. The goal of this test was to constrain the streaming session to 1.5 Mbps. The pacing between ACK packets were in this test found through thorough testing in order to find the pacing value that resulted in the bit rate wanted. This can be regarded as the elements learning phase presented in section 4.2.2. The pacing value which gave 1.5 Mbps was approximately 19 milliseconds. The test will first let the TCP flow run unaltered for 60 seconds. Then we will set the pacing to 19 ms for 180 seconds. Next, we will set the pacing value to 13 ms to constrain the stream to 2.5 Mbps for 120 seconds. Last, we will remove the constrain on the stream to let it flow at maximum bit rate again. The total run time of the test was 480 seconds. The resulting bit rate from this test is presented in figure 6.5.

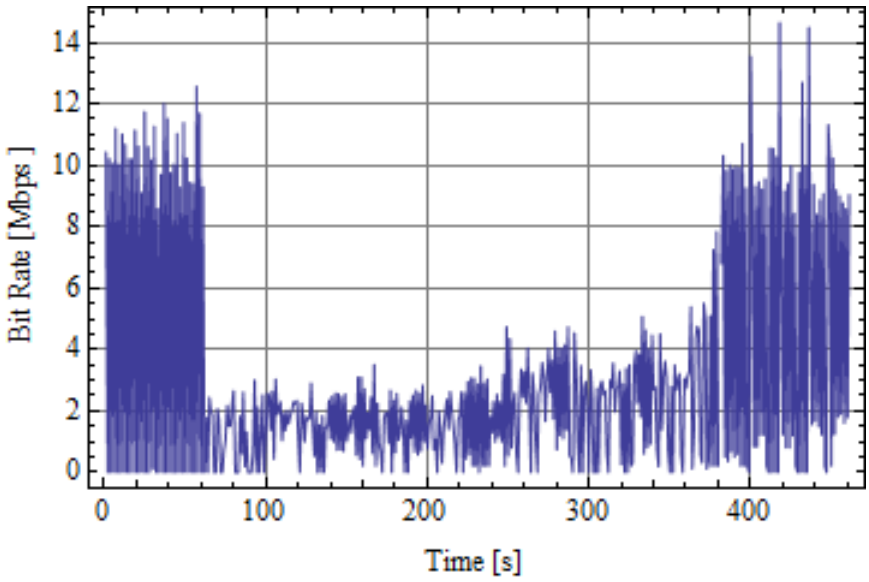

Figure 6.5: This figure illustrates the bit rate when testing the TCP ACK Pacer. ACK Pacing was activated after 60 seconds. From here the pacing value was set as illustrated in figure 6.6. The stream constraint was released at 360 seconds.

Discussion of TCP ACK Pacer Test Results

The tests performed in this section show that the ACK pacing element can be used for decreasing TCP session throughput. However, there is a need for a learning period for this element where the controlling entity (KP) acquire some knowledge about which pacing value that give the wanted throughput. This can be time consuming and could result in poor element performance if the relation between throughput and pacing value is not found. Another issue is the TCP timeout, but this can be considered acceptable and effective as a last resort controlling mechanism. This is because a TCP session that needs to be limited fast, to protect other sessions, must

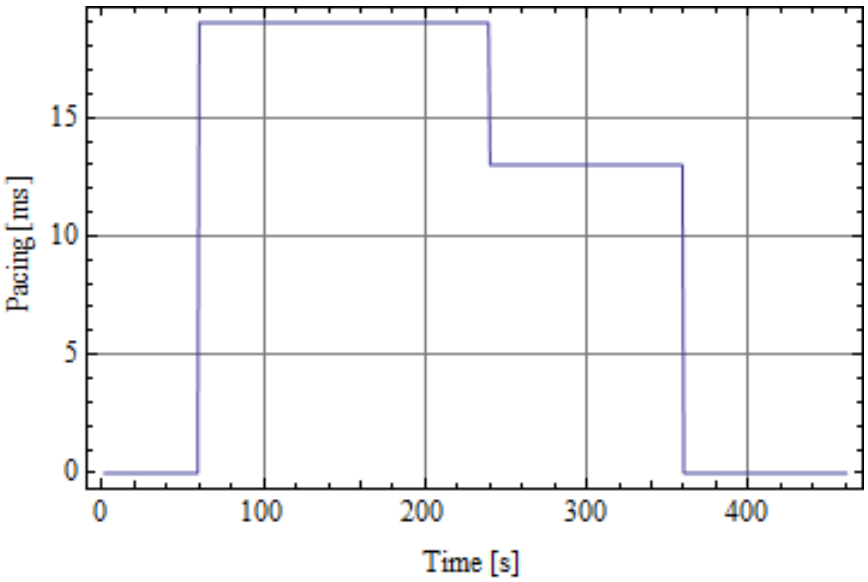

Figure 6.6: This figure illustrates the pacing value between packets leaving the element.

in worst case scenarios experience a harsh treatment. This will let high priority TCP sessions maintain an acceptable quality level.

One observation made when we illustrated the bit rate in the bit rate graph was how the bit rate flucutates between high and low measured throughput. We have used the same method for measuring the throughput in all tests, so there is a clear difference in the behavior of the TCP flows used in FTP downloads and Smooth Streaming. One reason for this behavior in Smooth Streaming can be that the client repeatively ask for chuncks of two seconds each. If the time it takes to download one chunk is shorter than 2 seconds, then the client will be idle for a small time interval. Hence, the result will be a graph which flucuates like the one in presented in figure 6.5.

6.1.2 Testing the Receive Window Manipulator

This section contains documentation of tests performed using the receive window manipulating Click element that were proposed in section 4.2.3. At first we used the general TCP throughput formula to calculate the senders throughput and corresponding advertised receive window. The result was no change in bit rate. Hence, we conducted several tests at different window sizes to find when the sender bit rate was affected. Therefore, the FTP download test section will first present the results from using the general TCP throughput formula, followed by a test when the window size was small enough to give expected decrease in sender throughput. Both tests start with the Click element deactivated to let the TCP flow run at maximum throughput. Next, five different window sizes were set for 60 seconds in order to check the stability of this approach. Hence, each test total run-time were 360

seconds.

```
Reply from 128.205.32.24: bytes=32 time=123ms TTL=46
Reply from 128.205.32.24: bytes=32 time=123ms TTL=46
Reply from 128.205.32.24: bytes=32 time=123ms TTL=46
Reply from 128.205.32.24: bytes=32 time=123ms TTL=46
Reply from 128.205.32.24: bytes=32 time=123ms TTL=46
Reply from 128.205.32.24: bytes=32 time=123ms TTL=46
Reply from 128.205.32.24: bytes=32 time=123ms TTL=46
Reply from 128.205.32.24: bytes=32 time=123ms TTL=46
Reply from 128.205.32.24: bytes=32 time=123ms TTL=46

Ping statistics for 128.205.32.24:
    Packets: Sent = 750, Received = 750, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 123ms, Maximum = 172ms, Average = 123ms
```

Figure 6.7: The Ping statistics are retrieved from 750 consecutive echo replies.

An important parameter when calculating the correct window size in order to obtain a certain bit rate level is the RTT measurement. As elaborated in section 2.4, there are small differences in using a passive or active approach. Hence, to reduce complexity we used the built-in Windows ping tool to run 750 consecutive ping echoes. The summary from this can be seen in figure 6.7. In order to use the result from this measurement, we assume stable network conditions, which mean that the RTT will not change during the test period.

All collected parameters, including the clients IP address, were inserted manually into the serviceSession handler. This was done using nano as explained in section 6.1.1. The general structure of the service session Click handler were presented in section 4.2.3.

Results from FTP Download Without Change in Senders Throughput

This test was started with a calculated maximum throughput of 2.5 Mbps. From here a gradual decrease of 0.5 Mbps in throughput every 60 seconds was performed. The window field in the TCP header was checked in order to confirm that the window field was set to the calculated values.

At first we present the RTT measurement, the summary of the ping measurement is presented in figure 6.7. After this a summary from this test scenario is presented in table 6.1. The resulting bit rates are plotted against time and are illustrated in figure 6.8 and the corresponding calculated window size can be seen in figure 6.9. It can be seen from the results presented that there were no clear effects of altering the window size, despite that the general TCP throughput formula was used.

These results provoked testing smaller window sizes in order to discover if this approach to control TCP traffic could be completely discarded. We therefore conducted several tests to find the window size which gave a resulting throughput of 2.5 Mbps. From this window size we performed a linear decrease to check if the linear behavior of the general throughput formula was maintained. This is elaborated in the following subsection.

Table 6.1: Parameters and achieved bit rate levels from testing the receive window manipulating element. The window size were calculated using the general TCP throughput formula, it can be seen there were no change in the achieved senders bit rate.

Target (Mbps)	RTT (ms)	Old Win (B)	New Win (B)	Achieved (Mb/s)
Unlimited	123	32768	32768	12
2.5	123	32768	38437	12
2.0	123	32768	30750	12
1.5	123	32768	23063	12
1.0	123	32768	15375	12
0.5	123	32768	7687	12

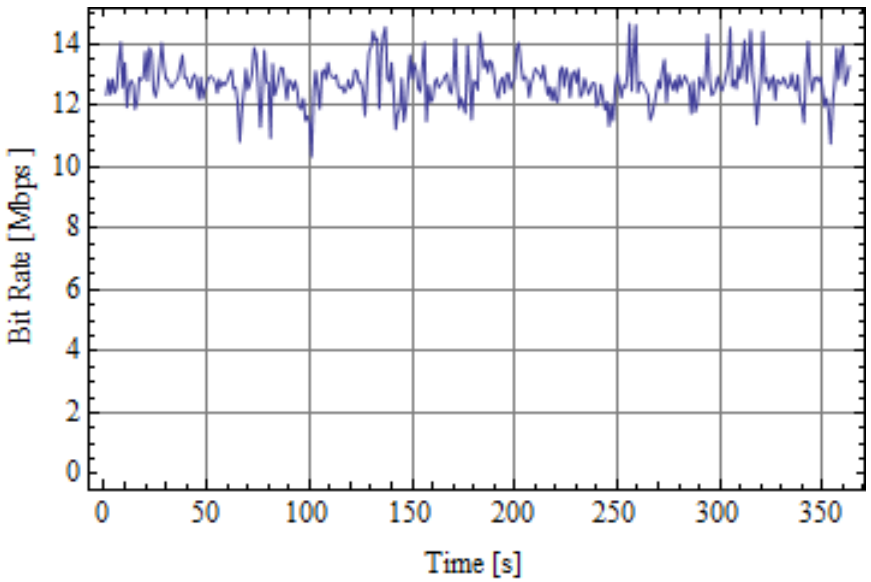

Figure 6.8: This figure shows the bit rate when we calculated the target bit rate with the general TCP throughput formula.

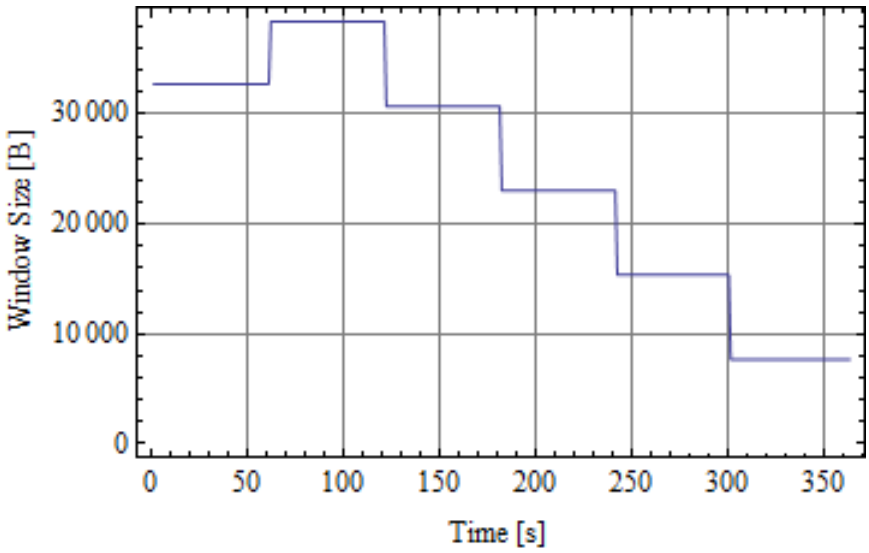

Figure 6.9: This figure shows the calculated window size when running the test.

Results from FTP Download With Effect on Throughput

After using the general TCP throughput formula to calculate the different bit rate levels, we performed extended testing in order to find if it was possible to influence the TCP senders throughput by altering the advertised receive window. After several tests we discovered a relation between window size and senders throughput. However, the effect did not become evident before we approached a receive window field of some hundred bytes. A summary of a test which gave a change in senders throughput is presented in table 6.2. The bit rate and receivers window size of the FTP download is illustrated in figure 6.10 and 6.11 accordingly.

Table 6.2: Parameters and achieved bit rate from testing the receive window manipulating element without using the general TCP throughput formula. In this test we used parameters which were found through extended testing which gave us the expected results.

Target (Mbps)	RTT (ms)	Old Win (B)	New Win (B)	Achieved (Mb/s)
Unlimited	1	32768	32768	12.00
2.5	1	32768	312	2.55
2.0	1	32768	250	2.1
1.5	1	32768	187	1.6
1.0	1	32768	125	1.0
0.5	1	32768	62	0.5

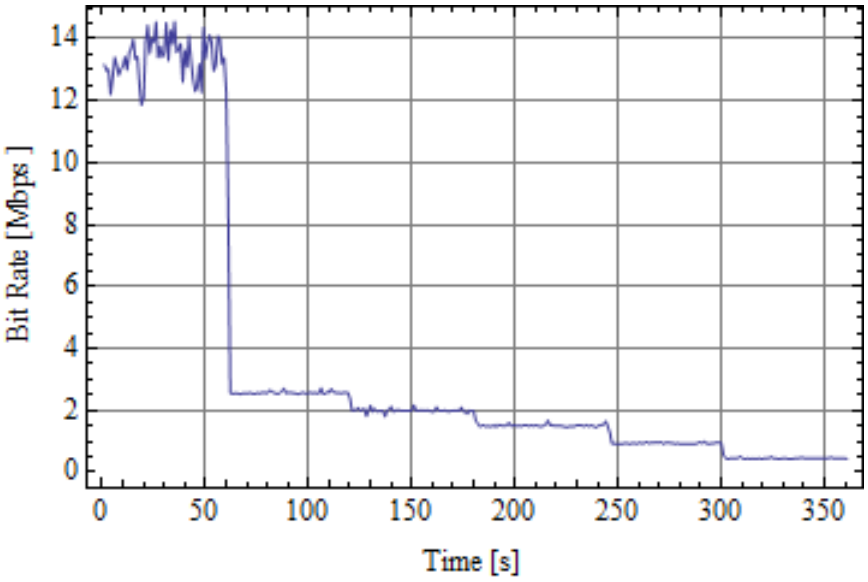

Figure 6.10: This figure shows the bit rate when the window size was set to a level which gave a change in senders bit rate.

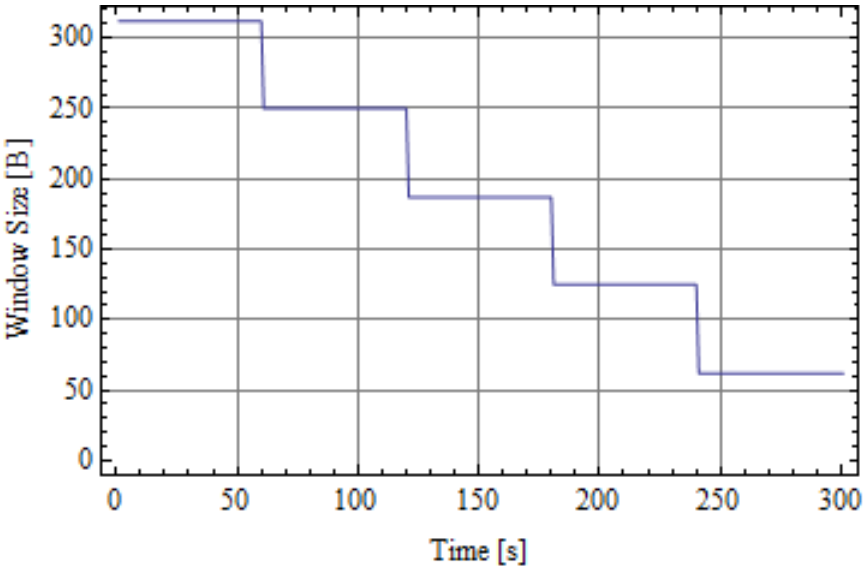

Figure 6.11: This figure shows the calculated window size when running the test. The first 60 seconds are not included because the window was above 30000, scaling for this would make the graph useless as the other values are below 300.

Smooth Streaming Test

In this section we present a test to show how the window manipulating element performs in a Smooth Streaming environment. It was done using the Silverlight client we developed and presented in section 5.1.1. We select one QL which we will try to manipulate the sender to distribute the video content at. The parameters used for this test is presented in table 6.2 and the resulting bit rate in figure 6.13. The RTT was measured using the built-in Windows Ping tool, and the summary from this is presented in figure 6.12.

Table 6.3: Parameters used for testing the receive window manipulating element running TV2s Smooth Streaming service.

Time (s)	Target QL (Mbps)	RTT (ms)	Win Size (B)
0-59	Unlimited	1	55480
60-359	1.5	1	187
360-480	Unlimited	1	16425

```
Reply from 158.38.14.136: bytes=32 time=1ms TTL=57
Reply from 158.38.14.136: bytes=32 time=1ms TTL=57
Reply from 158.38.14.136: bytes=32 time=1ms TTL=57
Reply from 158.38.14.136: bytes=32 time=1ms TTL=57
Reply from 158.38.14.136: bytes=32 time=1ms TTL=57
Reply from 158.38.14.136: bytes=32 time=1ms TTL=57

Ping statistics for 158.38.14.136:
    Packets: Sent = 750, Received = 750, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 1ms, Maximum = 6ms, Average = 1ms
```

Figure 6.12: This figure shows Ping statistics when running 750 consecutive Ping echoes towards the Smooth Streaming Server.

Discussion of TCP Window Manipulator Test Results

The reason for the fluctuating measurements between 0 and 10 Mbps is the bursty behavior of the Smooth Streaming technology as discussed earlier.

An issue when using the receive window manipulator is the time it takes for the client to respond to a receive window adjustment. A Silverlight client is constructed to maintain maximum bit rate whenever possible. Hence, the streaming client will be less responsive compared to a FTP download. The FTP download responded almost instantly after the window size was adjusted. Another reason for the slow response in the Silverlight client is its buffer, which is used to prevent some bit rate fluctuations.

The general TCP throughput formula used to calculate receivers advertised window size and corresponding bit rate needs to be inspected further. This is because we got a result in the FTP download test where we had to find the relation between

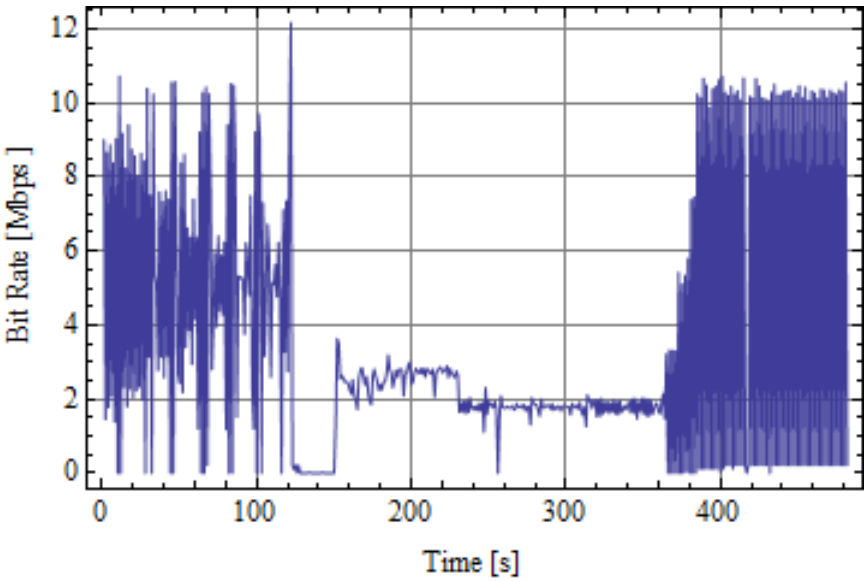

Figure 6.13: This figure illustrates the bit rate of the streaming client. Receive window manipulation started at 60 seconds. We see results at 120 seconds. Then the window is restriced further and we see more immediate effect. At 360 seconds control is released.

the throughput and window size through testing. This was because the theoretical approach failed. We have not found any theoretical explanation for this deviation between our calculated and measured throughput. Neither have we been able to define exactly when the theoretical approach will fail. For some servers our calculated throughput is correct, for others it's not. We therefore expect that this has to do with the server side implementation of TCP.

Even though there are some issues concerned with fine tuning the receive window manipulator, we have shown that it is possible to control the bit rate at which a TCP session runs at through this approach. However, improvements towards tuning the element should be studied further.

6.2 Testing the Silverlight Client

The first form of testing done on the Silverlight client was printing information from the media player to the user interface. When a parameter was successfully retrieved from the Silverlight framework it was presented below the video player in a text box. After successfully retrieving and printing a parameter, a new parameter was chosen for implementation. We then implemented a method of retrieving it, and so on. The Silverlight client was therefore incrementally implemented with focus on the wanted parameters.

When we had retrieved all the wanted parameters we started implementing a web client. This contained methods for sending the information to the router and statistics server. The first server side functionality was to respond to the Silverlight

client with the parameters received. The response from the server was then displayed at the Silverlight application page. Figure 6.14 shows the response.

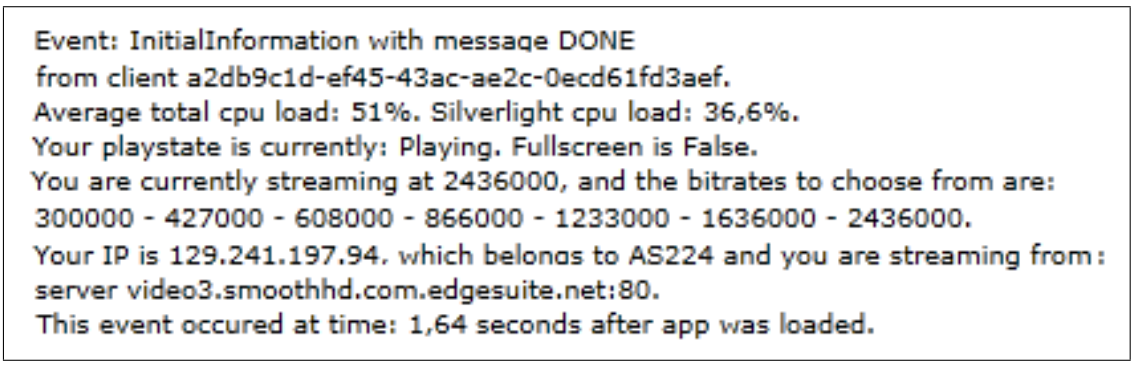

Figure 6.14: This figure illustrates the response message from the first server side parameter reception script. It sends a response informing the user of the parameters it received

During the testing of the client, we tried to make sure that the client runs error free and that the messages are sent correctly and received server side. Another focus has been on error control. If things go wrong, we want to control the outcome so that the video streaming does not stop and we want to resume our reporting if possible.

The message shown in figure 6.14 shows that all parameters are sent and received successfully. And from the tens of thousands of measurements sent we only got one error message that came repeatedly. We had a problem when the Silverlight client could not reach the servers. This is valid both for the home gateway and the statistics server.

When we tried to send messages to a server that was not running we got an exception and the further reporting became unstable. If the home gateway was not responding, a message was sent to the reporting server and from there anything could happen. We could have continued success in transmitting to the statistics server or it might crash from the exception. This variation in behavior prompted us to build in a fail-safe mechanism. If the web client gets a sending error, a new web client will be created and start with a blank slate. The message that was in the progress of sending will not be sent again to the server that failed. The next message will however be sent to all recipients, given that they do not fail again. This way, all the messages are attempted sent to all recipients. If the server recover to a running state, messages will be delivered successfully again.

If the Silverlight client receives an HTTP sending error it will notify the user with a message; “error occurred, restarting web client”. We have tested this by taking down the home gateway web server which makes the Silverlight web client send POST messages to a non-existing server. We get our error message displayed, and the reporting to the statistics server is not affected. All messages will be sent to the statistics server while the home gateway is down. When we later start up the home gateway web server, messages are received and normal operation is resumed.

We believe that our Silverlight client is relatively stable under normal conditions.

Figure 6.15: This screen capture presents the TV2 Sumo service running our information gathering code in the background. We can see the KP response to the client saying that it's the Click22 server and that the message was received at a given time stamp. The userRate parameter is used in the application controlled client described in section 5.2

To test further, in a live environment, we got our client code to be implemented in TV2 Sumo. This was possible due to cooperation through the R2D2 project that our thesis is written under. Our client code is like a regular Silverlight application except for an additional class that does the information gathering. To implement our functionality in TV2 Sumo, they simply needed to add our class and instantiate it during startup of their service. Figure 6.15 shows TV2 Sumo running with our added functionality and is reporting state information to the KP.

After adding our code to the TV2 Sumo player we were able to monitor its state through the webUI. This also allowed us to do better tests of other aspects of this project. As TV2 Sumo supports HD quality up to 5Mbps, our scenario was more realistic when it is supposed to emulate a home network with a television. With TV2 Sumo as the testing platform, our integration tests also performed better. This is because the distance between quality levels is larger at the Sumo service than with the test movie we had used. It is much easier to control bandwidth levels when the

level below 5Mbps is 3.5Mbps. This allowed us a window of 1.5Mbps between the two quality levels, which make the job of separating them easier.

6.3 Testing the Knowledge Plane Component

Here we will briefly explain how we have tested that the KP exhibits the wanted behavior when it comes to receiving information and writing it to the webUI. In addition we will take a look at the action reasoner, and how it behaves when a bit rate target is set.

6.3.1 The Data Structure

To test if the KP correctly handles information we will run a test where we start a Silverlight client and generates a couple of messages by pressing PLAY/PAUSE. Then we will start a new client and the KP should successfully recognize this as a new service. The responding output to the webUI should be correct, showing two clients and their current state. In figure 6.16 we can see that we have started the KP control application and it is running. The KP then detects a new Silverlight session by its unique session ID. This session then sends some measurements like InitialInformation and a few PLAY/PAUSE messages to the KP. We then start a new client and it is successfully recognized by the KP. Further measurements received are matched up to the correct session. This is an example of how the KP process looks from the server side. This process has been running error free for weeks, and we believe that the data structure for storing and receiving measurements is stable.

```
Router Control Java Application is running
Web server running at click22.item.ntnu.no:23500
New session with Id: 7178391a-8bd7-4cfc-ae61-c00ad221b4e6
New measurement from 7178391a-8bd7-4cfc-ae61-c00ad221b4e6
New measurement from 7178391a-8bd7-4cfc-ae61-c00ad221b4e6
New measurement from 7178391a-8bd7-4cfc-ae61-c00ad221b4e6
New measurement from 7178391a-8bd7-4cfc-ae61-c00ad221b4e6
New session with Id: 41060e20-5be4-4dc5-b162-a2c2dc802ee6
New measurement from 41060e20-5be4-4dc5-b162-a2c2dc802ee6
New measurement from 7178391a-8bd7-4cfc-ae61-c00ad221b4e6
New measurement from 41060e20-5be4-4dc5-b162-a2c2dc802ee6
New measurement from 41060e20-5be4-4dc5-b162-a2c2dc802ee6
```

Figure 6.16: This is a cropped view of a Linux terminal where the KP control application is running. We can see that text is printed every time a new measurement arrives.

The KP takes a lot of information as input and it remembers it for use in statistic graphs in the webUI and other usage areas that might come up in future development. As important as retrieving information correctly, the KP also needs to output some information to the webUI. This information should at any time be a representation of each known session with information that it has gotten from the sessions.

Especially the last measurement is of importance, this is the newest state of the player.

Having used the KP component extensively in testing traffic manipulation while observing Silverlight client responses, we are satisfied that this is working as intended.

6.3.2 Action Reasoner

We want to be sure that the action reasoner is behaving as expected. When a user provides a preference for a download bit rate, the action reasoner should react and try to limit the bit rate for the given stream. This will be done by rewriting a Click handler that is linked to the window size manipulation element presented in section 4.2.2. In this test we want to see that the handler is correctly written and that it is rewritten according to CLICK_CHANGE_INTERVAL. This means that the handler should not be rewritten more frequently than given by the change interval constant. In our testing this interval is 10 seconds. We allow this amount of time to check if our changes have effect and if they don't, the operation is repeated.

We will start a Silverlight video stream and let it run for 5 minutes. After this it will run at 5Mbps and we will set the user preference of 3.5Mbps and observe how the action reasoner is writing the handler. We have a debug-text occurring every second saying either OK or NO which tells us if the target rate is met or not. Figure 6.17 shows the terminal output of a running action reasoner.

We see that the action reasoner writes a new handler which includes IP addresses, delay measurements and a target bit rate of 4250000. This rate is chosen because it is in the middle of 5Mbps and 3.5Mbps. This handler is then kept for 10 seconds, when the stream actually drops to 3.5Mbps and the handler is not changed. The rate stays at 3.5Mbps for the next 8 seconds. In the 9th second, the rate drops to 2.5Mbps. This is too low, and since it is over 10 seconds since the handler was last written, the handler is instantly rewritten with a bit rate increment. We conclude that this is the wanted behavior and is satisfied with the reasoning at this time.

```
NEW HANDLER: 129.241.197.93:158.38.14.136:0.71:1:4250000
NO! Target rate: 3500000, DL rate: 5000000
NO! Target rate: 3500000, DL rate: 5000000
NO! Target rate: 3500000, DL rate: 5000000
NO! Target rate: 3500000, DL rate: 5000000
NO! Target rate: 3500000, DL rate: 5000000
NO! Target rate: 3500000, DL rate: 5000000
New measurement from 74815ca5-2762-4d95-9df9-8342a6955be5
NO! Target rate: 3500000, DL rate: 5000000
NO! Target rate: 3500000, DL rate: 5000000
New measurement from 74815ca5-2762-4d95-9df9-8342a6955be5
OK Target rate: 3500000, DL rate: 3500000
New measurement from 74815ca5-2762-4d95-9df9-8342a6955be5
OK Target rate: 3500000, DL rate: 3500000
OK Target rate: 3500000, DL rate: 3500000
OK Target rate: 3500000, DL rate: 3500000
OK Target rate: 3500000, DL rate: 3500000
OK Target rate: 3500000, DL rate: 3500000
OK Target rate: 3500000, DL rate: 3500000
OK Target rate: 3500000, DL rate: 3500000
OK Target rate: 3500000, DL rate: 3500000
New measurement from 74815ca5-2762-4d95-9df9-8342a6955be5
NO! Target rate: 3500000, DL rate: 2500000
NEW HANDLER: 129.241.197.93:158.38.14.136:0.75:1:4270000
```

Figure 6.17: The action reasoner is set to limit the bit rate from 5Mbps to 3.5Mbps. We see that it rewrites the Click handler, supplying ip addresses, delay measurements and a target rate.

Chapter 7

Future Work

In this chapter we will propose and discuss improvements and new features that would take our solution further.

7.1 Improving KP Action Reasoner

The KP is functioning well with regards to information gathering from Silverlight clients and the webUI. Parsing of input data has not been a concern and the ServiceSessionBank entity has until now provided the needed functionality for storing Silverlight measurements. However, the process of determining actions to improve network traffic conditions is at the moment extremely simple. This is probably where the KP is most in need of further development.

Developing a good action reasoner can be a very time consuming task. In the literature describing autonomic access networks, the KP is envisioned as a learning instance. It should be able to recognize traffic patterns and act based on experience. This form of artificial intelligence is of course a vision for the architecture as a whole. It is not a realistic goal for this implementation in the home gateway during a master thesis.

It should however be mentioned that improving the current state of the action reasoner is not a formidable task. At the moment it is only capable of reducing the rate of a stream. This would allow another stream a higher rate, given that it is the internet access at the end user side that is the bottle neck. If the reduction of a stream is imposed to allow higher rate to a competing stream, this is an example of human reasoning and not the KP itself. There should therefore be implemented functionality for selecting bit rates above the currently streaming rate. Such a selection should prompt the action reasoner to automatically lower the rate of competing video streams and it could also impose some sort of limitation for other traffic.

The action reasoner should also be configured to use a diversity of tools to affect the network traffic. The current version only supports rewriting the TCP window fields. As described in section 4.2.2, we have also implemented an element for ACK pacing. The use of this element in addition to or as a substitute for the window size manipulation could prove useful and should be tested. In addition, some simple BandwidthRatedUnqueue elements could be used to limit other traffic

if this was necessary to obtain an acceptable video QoE. BandwidthRatedUnqueue is a standard Click element that will empty a queue at a given rate. If the traffic situation in the home network got bad we could sort all non-video traffic into a queue that we applied a BandwidthRatedUnqueue element to. This way, other traffic would be forced to stay below a given bit rate. All traffic exceeding the given bit rate would be dropped.

The latter is not a very effective way of handling traffic as it results in packet loss. The lost traffic will probably be retransmitted, as most traffic is sent over TCP. Even so, having a diverse set of traffic controlling tools available could improve the total effect achieved by our system.

Given that the rate calculation of the TCP window manipulation has proven to be less accurate than expected, improvements could be done to increase the accuracy. As mentioned in section 6.1.2, tests have shown a linear trend between the window size and the actual throughput. Even if the calculated rate does not correspond with the real rate, testing show that when the receive window reach a certain level and the window size is halved. Then the actual rate is halved to.

Let's say that we set a target rate of 3Mbps and calculate a window size that should give this rate. If the actual throughput is 5Mbps instead of 3Mbps, testing shows that a reduction of the window size will give linear results. By setting the window size to half its original size, we will obtain a bit rate of 2.5Mbps. In our current system, slowly decrementing to obtain the correct bit rate will be very time consuming and the user will not see an immediate effect. The throughput calculation formula does not always give the correct rate. It is still very useful in determining how to get there.

If we could, by some mechanism, actually know the peak rate allowed by the manipulated TCP streams we could utilize the linearity to quickly scale down the window size. Given that we actually want a throughput of 3Mbps, but our calculated window obtains a bit rate of 5Mbps, if we knew this in run time we could adjust for it.

Smooth Streaming downloads the video in chunks. During the download of a chunk, Silverlight measures the bit rate it experiences and uses this information to decide if it should increase its video bit rate. If we could get access to Silverlights bandwidth measurements, we could use this to estimate the actual TCP rate. Then we could make a linear adjustment and calculate our new window size. The easiest place to access this bandwidth measurement would of course be in the Silverlight client. We believe that the retrieval and utilization such buffering information could greatly improve the TCP window size manipulation scheme. How this could be done is proposed in section 7.2.2.

7.2 Future Work for Silverlight Client

The extra functionality we have built into a Silverlight Smooth Streaming client has some main functions. These are retrieving relevant application-specific information and transmitting it to anyone who might be able to utilize it. In addition to the requirements in what to do, we also have some requirements for how to do it. By

this we mean that we should be able to expect that the client will obtain and report information without interrupting the media delivery in any way that harms it. We do not want it to fail and block or crash the actual video streaming application. In addition to having a polite way of failing, leaving other processes intact, we want failure to occur as infrequent as possible. These are, in short, the only goals for functionality of the Silverlight client we aim to build.

Possible improvements when it comes to retrieving and transmitting the information will probably surface as usage of the reporting function progress. There could be other types of information that could prove useful, that we have yet to consider. We believe that our application is easy to modify if another parameter is wanted. The transmission of data could also be changed or complemented by another form of communication if needed. The .NET framework is no stranger to web services as a form of machine to machine interaction. As of now, we are satisfied with the information retrieved from the client and the transmission of the collected data.

In this section we will comment on how far we have come and which parts of the client code that could use a fine tuning touch. We will first attend to the question of reliability experienced and what could be done to improve it. Then we will look at pure functional aspects of what information could be retrieved in a future version that are not present now. During our testing we have found certain flaws in controlling the TCP streams. It is our belief that these could be corrected by the KP if our Silverlight application would supply some additional information.

7.2.1 Reliability of the Additional Code

For our intent and purpose of the code we feel that its reliability is well within the accepted range. We have mostly experienced uninterrupted service as long as the code is used as intended. When the code is inserted into other players it might behave unintentional. By this we refer to the TV2 Sumo client where our code is implemented where we have some known bugs. The TV2 Sumo player has implemented a control panel that allows the user to change the video source without reloading the client. As our client have some of the functions tied to the loading of the player, this is not correctly reloaded when the source is changed. Also, we have only implemented support for reporting information when the video is coming from a Smooth Streaming source. When the stream is not an adaptive stream, we have not seen a reason to report information from it. For the purpose of controlling the flow, it would be irrelevant as the user chooses bit rate in the player when it is not self-adaptive.

Changing the channel from the Silverlight client will cause the code to malfunction either by not reloading or actually failing. We have implemented some checks so that a change in media source from Smooth Streaming to regular streaming will stop the reporting function. This is to avoid unwanted side effects of the client crashing. It will stop the client in a controlled manner instead of letting it crash. In a complete system our client should address the changes and adapt.

These compatibility problems we have seen when implementing our code into the TV2 Sumo player are the result of us not knowing the TV2 code during the

implementation. We implemented a standalone Silverlight streaming application that we added our code to. Dynamically changing the source of the stream did not occur to us. We believe that knowing that such functionality is used by service providers, one should rewrite the client code to more explicitly say what scenarios are tolerated and what is not. The support for changing media source should be added and maybe support for other types of changes. The most important part is however to code the Silverlight client so that it knows where it will work and not. If we know that the client will work in a certain environment, then we should check that that is the environment we're in before starting. By checking all resources we plan to use in the code we could prevent unknown failures by simply not loading the statistics process. Then the code might be updated to support new environments if needed, instead of things partly working.

7.2.2 Estimating Avaliable Bandwidth

> "The client can choose between chunks of different sizes. Because Web servers usually deliver data as fast as network bandwidth allows them to, the client can easily estimate user bandwidth and decide to download larger or smaller chunks ahead of time."

This is stated in [17], as they describe how Smooth Streaming video is encoded into different bit rates and the client choosing between them. As previously discussed, our TCP traffic control parameters have not been easy to calculate exactly. While testing our solution we have discovered that if we knew what bit rates Silverlight is experiencing, this could greatly improve our estimations. As we try to estimate how to manipulate TCP window size, if we knew how Silverlight experienced our tampering we could do better.

In section 6.1.2 we see that the receive window manipulation doesn't always give the correct rate. It is however shown that if we reduce the window size we will eventually achieve the desired effect. What's even more useful from this testing is that the experienced bandwidth, even when we cannot get our calculations to be correct, varies linearly with the window size. This inclines us to think of what could be achieved if we were to obtain Silverlights experienced bandwidth.

Say we wanted to reduce the TCP connection to a maximum throughput of 3Mbps. If we do our calculations and initiate the window manipulation we might not get any results. If we were to learn that Silverlight experiences a throughput of 6Mbps, we could use the linearity of our results to simply scale for the imperfections. We could calculate a window size based on the target maximum throughput and then correct it by comparing it to the reported bandwidth from Silverlight.

Due to the advantage this would give us, we believe that this should be included in future versions. As far as we can see, the Silverlight framework does not allow us to read this value directly. There might be a way to retrieve it directly; we have not done extensive research. If this was possible, it could simply be read and sent to the KP. If it cannot be directly read, we need to calculate it. The Silverlight SmoothStreamingMediaElement has a property named TotalBytesDownloaded. From the Silverlight documentation we can read that this property is a read-only value that

is “The number of bytes downloaded”. Using this metric together with a timer, we should be able to calculate the download rate. Silverlight downloads data in chunks, and we should take this into consideration. Calculating every 10 seconds would not give a correct answer because the traffic is bursty and we want the peak load. The time interval should not be too small either, it may result in sporadic measurements that are hard to interpret.

Having stated a general idea of how to achieve this, in addition to a few considerations with regards to measurement intervals, we believe that this should point in the right direction for future development. Silverlight reports its bit rate, the KP tries to reduce it and Silverlight can continuously inform KP if the reductions are having an effect. This addition will greatly improve the accuracy of any mechanism trying to control the video stream.

Chapter 8

Conclusion

Our task was to investigate and implement new components for the Knowledge and Action Plane. The work has resulted in a complete system for a content-aware home gateway with Action Plane components for controlling traffic. The work is divided into four main areas of work. We will discuss them in order and finally address the system as a whole.

A Reporting Silverlight Application

To make the home gateway content aware we chose to implement extra functionality for Silverlight Smooth Streaming video players. The functions are packed in a way that makes it simple for already existing Silverlight applications to add them. When added, information relevant to the home gateway will be collected during video playback. This information includes, but is not limited to, client and server IP addresses and available bit rates offered by the server.

Our client will report this application specific information to the home gateway for processing. In addition to sending the information to the home gateway, it also sends the information to a statistics server. This is thought to represent a value to the service provider, as they will be able to collect usage statistics from their service.

A Knowledge Plane Implementation

We implemented a minimalistic Knowledge Plane entity that receives information from the Silverlight clients and processes it. The information is stored and used to keep track of each video stream originating from behind the home gateway. From an interface, the user can give its preferences for the videos being streamed to the network. When a user sets a bit rate target for one of the active streams, the Knowledge Plane will through a simple reasoning process determine a course of action. The goal of the Knowledge Plane is to optimize the resource allocation of the network.

Web Interface

The web interface will serve as a graphical representation of the information contained in the Knowledge Plane. The currently streaming media will be presented

along with key parameters. This allows the user to set bit rate targets for a given video stream, making the system optimization user-specific. A user preference will be communicated to the Knowledge Plane, which in turn will act upon this preference. The Knowledge Plane will, in addition to key parameters, give the web interface a list of previous measurements to graph the historical bandwidth on a per-stream basis.

Click Elements and the Action Plane

To provide the Knowledge Plane with tools to do service differentiation we have developed two mechanisms for traffic control. Contrary to our previous work, we now try to avoid hard limits and sectioning of resources. We want to affect video streaming quality without reserving bandwidth, which is not optimal in regards to other traffic. One Click element is built to identify and alter the ACK packets coming from a video stream. We do receive window manipulation to make the streaming server believe that the client cannot receive data in the current rate. This will force the server to decrease its sending rate, freeing resources for other streams. The other Click element tries to obtain the same results by clocking ACK packets back to the server with greater inter-arrival time than the original. Since TCP tries to limit the amount of data packets that are not ACK'ed, we will force it to wait for our ACKs before sending more data. Since we control the rate of the ACK packets, we can also control the sending rate data packets.

As a Complete System

Our testing and demonstrations have shown effect on the bit rate when using the Click elements to control video streaming. In addition to this, the Silverlight client has proven to be a good way of obtaining application-specific information that can be useful to the Knowledge Plane. TV2 has shown interest in the information gathering from the Silverlight clients. This has allowed us to implement our reporting function in their Internet video service TV2 Sumo. Our testing has shown promise for this kind of application control. The Knowledge Plane, complemented by a user interface is working well in regards to information gathering and reasoning. We feel that further complexity should be implemented to the action reasoner to improve the outcome of the Action Plane component. A challenge for further work is to make the Knowledge Plane correctly reason how to use Click functionality of clocking out ACK packets.

With some future development we believe that this form of architecture can prove useful in a home gateway. It will give the user control of the streaming media. In addition, it's a good tool for visualization of traffic. This could improve the users understanding of video services and available resources in the home network.

References

[1] Morris, Kohler, Jannotti, Kaashoek. “The Click Modular Router”. 1999

[2] Gascón, Díez, et al. “Designing a Broadband Residential Gateway using Click! Modular Router”. 2005

[3] Official Microsoft Silverlight Site. 26.05.2011. <http://www.silverlight.net/>

[4] “Google buys YouTube for $1.65 billion”. US business msnbc.com. 10.10.2006. 21.05.2011 <http://www.msnbc.msn.com/id/15196982/>

[5] Cheng, Dale, Liu. “Understanding the Characteristics of Internet Short Video Sharing: YouTube as a Case Study”. 2007

[6] “YouTube at five- 2 bn views a day”. BBC News. 17.05.2010. 08.06.2011. <http://news.bbc.co.uk/2/hi/8676380.stm>

[7] Karim, Hovell. “Everything over IP — an overview of the strategic change in voice and data networks”. 1999

[8] Official Akamai Site. 22.05.2011. <www.akamai.com>

[9] Akhshabi, Begen, Dovrolis. “An Experimental Evaluation of Rate-Adaption Algorithms in Adaptive Streaming over HTTP”. 2011

[10] Haugene, Jacobsen. “QoE/QoS Mechanisms for Content Aware Broadband Home Gateways”. 2010

[11] Latré, Simoens, Vleeschauwer, et al. “An autonomic architecture for optimizing QoE in multimedia access networks”. 2008

[12] Clark, Partridge, Ramming, Wroclawski. “A Knowledge Plane for the Internet”. 2003

[13] Latré, Verstichel, et al. “On the Design of an Architecture for Partitioned Knowledge Management in Autonomic Multimedia Access and Aggregate Networks”. 2009

[14] Latré, Simoens, Vleeschauwer, et al. “Design for a generic knowledge base for autonomic QoE optimization in multimedia access networks”. 2008

[15] Begen, Akgul, Baugher. “Watching Video over the Web - Part 1 - Streaming Protocols’. 2011

[16] Krasic, Li, Walpole. “The Case for Streaming Multimedia with TCP”. 2001

[17] Zambelli. “IIS Smooth Streaming Technical Overview”. 2009

[18] Timmerer, Muller. “HTTP Streaming of MPEG Media”. 2010

[19] Krasic. “A Framework for Quality-Adaptive Media Streaming: Encode Once - Stream Anywhere”. 2004

[20] Pantos, May. “HTTP Live Streaming”. IETF draft November 2010. Visited: 16.02.2011 <http://tools.ietf.org/html/draft-pantos-http-live-streaming-05>

[21] Chen, Wang, Zhang, Shen and Wee. “Segment-Based Proxy Caching for Internet Streaming Media Delivery”. 2005

[22] Daoust, Hoschka et al. “Towards Video on the Web with HTML5”. 2010

[23] Pilgrim, Mark. “No 5. Video on the Web - Diving In”. Visited: 15.02.2011. <http://diveintohtml5.org/video.html>

[24] Vaughan-Nichols. “Will HTML 5 Restandardize the Web?”. 2010

[25] “Immediate Release: W3C Confirms May 2011 for HTML5 Last Call, Targets 2014 for HTML5 Standard”. The World Wide Web Consortium. 14.02.2011. 31.05.2011. <http://www.w3.org/2011/02/htmlwg-pr.html.en>

[26] Jamal, Sultan. “Performance Analysis of TCP Congestion Control Algorithms”. 2008

[27] Song, Tan, Zhang, Sridharan. “A Compound TCP Approach for High-speed and Long Distance Networks”. 2006

[28] Li, Zhang, Xie, Yang. “TCP and ICMP in Network Measurement: An Experimental Evaluation”. 2005

[29] Dovrolis, Jiang. “Passive Estimation of TCP RoundTrip Times”. 2002

[30] Li, Lowenthal, Veal. “New Methods for Passive Estimation of TCP Round-Trip Times”. 2005

[31] The Click Modular Router Project - Element List. 28.02.2010. 02.03.2011 <http://read.cs.ucla.edu/click/elements/>

[32] The Click Modular Router Project - Click Driver and Tool Documentation. 02.03.2006. 28.02.2011. <http://read.cs.ucla.edu/click/docs/>

[33] Govindaswamy, Zaruba, Balasekaran. “Complementing Current Active Queue Management Schemes with Receiver-Window Modification (RWM)”. 2005

[34] Aggarwal, Savage, Anderson. “Understanding the Performance of TCP Pacing”. 2000

[35] Brooks, David R. Introduction to PHP for Scientists and Engineers Beyond JavaScript. London: Springer-Verlag, 2008

[36] Babin, Lee. Beginning Ajax with PHP. New York: Springer-Verlag, 2007

[37] Collison, Simon. Beginning CSS Web Development. New York: Springer-Verlag, 2006

[38] MATLAB - The Language Of Technical Computing. Visited 19.05.2011. <http://www.mathworks.com/products/matlab/>

[39] Wolfram Mathematica: Technical Computing Software. Visited 19.05.2011. <http://www.wolfram.com/mathematica/>

Appendix A

IP and ISP AS number retrieval script

```
<html>
<body>
<?php
        // Determine Client IP-address
    if (getenv(HTTP_X_FORWARDED_FOR)) {
        $pipaddress = getenv(HTTP_X_FORWARDED_FOR);
        $ipaddress = getenv(REMOTE_ADDR);
echo $pipaddress. "(via $ipaddress)" ;
    } else {
        $ipaddress = getenv(REMOTE_ADDR);
        echo "$ipaddress";
    }

        // Separate the return values
        echo ";";

        // Run whois on the IP-address of the client
        $isp = shell_exec('whois '.$ipaddress);

        // Need to substring until we only have the AS
        $start = "origin:";
        $isp = get_after($isp, $start);

        // $isp starts at AS-number but has a lot more
        $isp = trim($isp);

        // Now we don't have any whitespaces in front of the
            AS, split at white spaces
        $array = explode(" ", $isp);
```

```
        // The first element is now the AS
        $isp = $array[0];

        // The last char of $isp is supposed to be a digit,
           remove the last char until we see a digit
        while(!is_numeric(substr($isp, strlen($isp)-1)))
        {
                $isp = substr($isp,0,-1);
        }

        // Now, we should have isolated the AS number
        echo "$isp";


        // Function for returning the part of $input after
           the pos og $start
        function get_after($input, $start)
        {
                $substr = substr($input, strlen($start)+
                   strpos($input, $start), strlen($input));
                return $substr;
        }

?></body></html>
```

Appendix B

Statistics Server Log Script

```
<html><body>
<?php

        // Get todays date in ddmmyy format
        $date = date(dmy);

        //Variable to hold the path of the file
        $myFile = '/var/www/logger/logs/log'.$date.'.txt';

        // Need a separate file to maintain information of
           logs sequence numbering
        $seqFile = '/var/www/logger/seqNumbering.txt';

        // Make the log string for this entry
        $logEntry = "";

        // Append all server side parameters
        $logEntry = $logEntry.get_seqNumber($seqFile, $date)
           .";";
        $logEntry = $logEntry.get_hexClientId($_POST["
           clientId"]).";";

        list($totalSeconds, $extraMilliseconds) =
           timeAndMilliseconds();
        $logEntry = $logEntry.
           get_humanReadableTimeWithMillis($totalSeconds,
           $extraMilliseconds).";";
        $logEntry = $logEntry.get_unixTimeWithMillis(
           $totalSeconds, $extraMilliseconds).";";

        // Then the ones passed through POST
        $logEntry = $logEntry.$_POST["event"].";";
```

```
    $logEntry = $logEntry.$_POST["eventMessage"].";";
    $logEntry = $logEntry.$_POST["playState"].";";
    $logEntry = $logEntry.$_POST["fullScreen"].";";
    $logEntry = $logEntry.$_POST["clientId"].";";
    $logEntry = $logEntry.$_POST["avgProcessorLoad
        "].";";
    $logEntry = $logEntry.$_POST["avgProcessLoad"].";";
    $logEntry = $logEntry.$_POST["videoDownloadBitrate
        "].";";
    $logEntry = $logEntry.$_POST["avaliableVideoBitrates
        "].";";
    $logEntry = $logEntry.$_POST["myIp"].";";
    $logEntry = $logEntry.$_POST["ispAS"].";";
    $logEntry = $logEntry.$_POST["sourceIpAndPort"].";";
    $logEntry = $logEntry.$_POST["timeIncrement"]."\n";

    // Open file, write it, close it
    $fh = fopen($myFile, 'a') or die("can't open file");
    fwrite($fh, $logEntry);
    fclose($fh);


// --- FINISHED ---

// Method declarations:

    function get_humanReadableTimeWithMillis($numSeconds
        , $numMillisec)
            {
                    return date("H:i:s", $numSeconds) .
                        ",$numMillisec";
            }

    function get_unixTimeWithMillis($numSeconds,
        $numMillisec)
            {
                    return $numSeconds.",$numMillisec";
            }

    // Returns an array with seconds and milliseconds in
        that order
    function timeAndMilliseconds()
            {
                    $m = explode(' ',microtime());
                    return array($m[1], (int)round($m
```

```
                [0]*1000,3));
        }

// To be more sort-friendly we get a plain hex
   clientId without '-' chars
function get_hexClientId($originalClientId)
        {
                // return string after removing -
                return str_replace("-", "",
                   $originalClientId);
        }

// Function to get the correct sequence number of
   the day
function get_seqNumber($numberFile, $dateToday)
{
        // The sequence number and date to return
        $returnDate = "";
        $returnSeq = "";

        // Read the current seqNumber info
        $fh = fopen($numberFile, 'r');
        $currentData = fread($fh, filesize(
           $numberFile));
        fclose($fh);

        // Split the input data into day and
           sequence number
        $dataArray = explode(";", $currentData);
        $writtenDate = $dataArray[0];
        $writtenSeq = $dataArray[1];

        // Check if the date is correct. If not, we
           need to start at 1 again
        if( strcmp ($writtenDate, $dateToday) == 0)
                {
                        // It's the same date,
                           update seq number
                        $writtenSeqInt = (int)
                           $writtenSeq;
                        $returnDate = $writtenDate;
                        $returnSeq = $writtenSeqInt
                           +1;

                } else {
```

```
                                    // The date has changed,
                                       update it and start over
                                    $returnDate = $dateToday;
                                    $returnSeq = 1;
                                }

                        // Write the new information to file
                        $fh = fopen($numberFile, 'w') or die("can't
                           open file");
                        $stringData = "$returnDate".";"."$returnSeq
                           ";
                        fwrite($fh, $stringData);
                        fclose($fh);

                        // Print debug info to see that date and seq
                            numbering is OK
                        echo "Read date "."$writtenDate"." and seq
                           "."$writtenSeq"."\n";
                        echo "Wrote date "."$returnDate"." and seq
                           "."$returnSeq";

                        // And return the seqNumber
                        return $returnSeq;
        }
?>
        <p>The file was run</p>
</body></html>
```