UNIVERSITY OF OSLO
Department of Informatics

Implementation of an Adaptive Stream Scaling Media Proxy with the data path in the Linux Kernel

Master Thesis

Bjørnar Snoksrud

UNIVERSITAS OSLOENSIS • MDCCCXI •

Implementation of an Adaptive Stream Scaling Media Proxy with the data path in the Linux Kernel

Bjørnar Snoksrud

Acknowledgments

I would like to thank my supervisor, Alexander Eichhorn, for invaluable advice and good ideas.

I would also like to thank everyone at the Simula lab for entertainment, morale and helpful hints, in addition to thanking Simula and everyone working here for providing the best working environment any master student could ask for.

Preface

Streaming media is becoming increasingly common, and bandwidth demands are rising. Centralized distribution is becoming prohibitively hard to achieve and geographically aware hierarchical distribution has emerged as a viable alternative, and has brought with it an increased demand for CPU processing power.

There has also been an influx in an abundance of different devices on which to receive media — from HD plasma screens to 2" cell phone displays — and the need for media to be able to adapt to devices of differing capabilities are becoming obvious.

For this thesis we have implemented a media proxy using RTSP/RTP. It is implemented partially in kernelspace, which has drastically reduced the CPU-cost of relaying data from a server to multiple clients. This has been combined with a proof-of-concept implementaton for adaptively scaling a SVC media stream according to available bandwidth.

Contents

List of Figures

Chapter 1

Introduction

1.1 Problem statement

Our main goals for this thesis is to solve the problem of increasing processing demands for streaming media proxies, and to overcome the increasing diversity in devices used for consuming streamed media.

1.2 Motivation

An increasing amount of our daily media consumption is through the Internet, with streaming media on the way to outgrowing standard broadcasted transmissions like television. This is facilitated by a growing increase in network bandwidth, with most of Norway having access to bandwidths exceeding 2 megabit per second, which is sufficient for streaming aggressively compressed movies in Standard Definition (SD)-quality.

As these bandwidth demands are proving unmanageable, video streaming services have been moving away from the standard centralized structure towards a hierarchical approach. Technologies used to overcome this challenge are different kinds of pseudo-Video-on-Demand (VoD) in order to send the same data to many users at the same time. Proxies are also used to receive one copy of the media and resend it out to several consumers.

Contemporaneously, computer processing speeds still appear to be bound to Moore's law. As network bandwidth continues to grow, servers will encounter increasing processing demands.

The spread of video streaming is also in part due to a large influence of new devices with streaming support, like cell phones, portable gaming systems and similar. These devices have differing capabilities with regard to displaying media, like resolution, color depth, frame rate, and most importantly bandwidth. To provide an acceptable user experience to all consumers, a large number of different transcodings would need to be maintained of each video stream.

1.3 System overview

We will implement a split non-caching proxy for streaming media using Real Time Streaming Protocol (RTSP)/Real-time Transport Protocol (RTP). The control path, RTSP and RTP Control Protocol (RTCP), will be handled in userspace, while the datapath will be contained in kernelspace. The proxy will inform the kernel module of destinations for the RTP video stream. The control path will register connection metrics, and the data path will use this information to adaptively scale Scalable Video Coding (SVC) encoded streams.

1.4 Main Contributions

Our streaming proxy uses a lightweight kernel-module to achieve zero-copy re-sending contained in the kernel, without intervention from the application. This solution requires no modifications to the kernel codebase, and exists as a hot-pluggable module together with a userspace daemon to handle connections and signalling. This reduces CPU-consumption significantly.

We have integrated this solution with an adaptive media stream scaling approach which adapts the media stream according to available bandwidth without user intervention.

1.5 Outline

Chapter 2 will detail the main points of the technologies we are using, while Chapter 3 will show how the userspace proxy is designed and implemented. How the kernel-module works will be revealed in Chapter 4, and the adaptive stream scaling is documented in Chapter 5. The main findings will be provided in Chapter 6. Finally, some reflection over the overall project is to be found in Chapter 7.

Chapter 2

Background

This chapter will cover some background information necessary to follow the design and implementation of this thesis.

2.1 Scenario

The centralized approach to video-streaming, although prevalent, introduces several problems. One of which being bandwidth, with the computer serving out n identical copies of a each individual data packet. If the server has a 1 Gbps network connection, and is serving a stream of 1600 Kbps, which is an acceptable bit rate for SD-video with modern encodings[1], the server would be limited to $\sim$660 streams. By comparison, the Superbowl XLIII was viewed by almost 100 million people. If even one per cent of the viewer base should watch this event over the Internet, the total bandwidth required would amount to over 1500 Gbps. To get a perspective on the size of this number, the peak traffic over the Norway Internet eXchange (NIX) on an uneventful day is below 20 Gbps. People might also be interested in watching the Superbowl in high definition.

[1] A professionally encoded DVD averages at about 4.500 Kbps, but uses a less efficient encoding (MPEG-2).

High-definition as provided by a Blu-Ray disc is encoded with 25 Mbps, which is about 16 times higher.

To overcome the bandwidth limitation, a hierarchical distribution scheme is necessary, preferably with some geographical awareness. A simple example would be a master server in the US, with proxy servers located in Europe, Asia and Africa to handle the respective offshore customers.

With this approach, the number of servers required are proportional to the number of clients, and the bandwidth for each server is more-or-less constant. Now we have displaced the problem with live media streaming from being bandwidth-limited to being server-limited. This is a tried and true solution.

The current design of modern operating systems has made sending and receiving large numbers of network packets inefficient, due to kernel memory protection. While the CPU speed/network speed ratio is typically skewed heavily towards CPU, there is a growing concern regarding reducing power-consumption of server-farms. The typical direct power consumption due to the CPU and power supply is easily measurable, there is also an approximately equal [1] indirect cost from the required cooling to keep the servers from disintegrating into a heap of molten metal. Thus there is a great incentive in current system design to reduce power consumption.

Furthermore, the manner in which media is consumed is changing rapidly. The same live stream might be watched on everything from a Full-High Definition (HD) 70" video-projector via a 100 Mbps Internet connection to a cell phone with a small display and a 60 Kbps Internet connection. Keeping different streams for every possible configuration is clearly infeasible. By using scalable video streams, the proxies can adapt each stream to a diverse range of receivers based on a number of parameters, like bit rate, resolution and frame rate.

2.2 The RTP family

The RTP family consists of 3 different, but closely related protocols. We will give a small overview of each of these.

2.2.1 RTSP

RTSP [2] is the streaming setup protocol of this family, defined in RFC 2326. It is a stateless plain text-based Hypertext Transfer Protocol (HTTP)-like protocol, enabling clients to discover resources offered by servers, and negotiate connections to these. It carries information about media-encoding, port numbers, and other stream-setup related matters. It also implements "trick-play" functionality, allowing a client to fast-forward, rewind and skip in a stream, should the stream support it. RTSP is independent of the transport protocol used, and although RTP is by far the most used data carrier for RTSP, there are alternatives, e.g: RDT, RealMedia's proprietary stream transport protocol.

In order to better understand the requirements of the proxy, a short description of the RTSP commands and common requests and responses are needed.

An standard RTSP request follows this format:

```
COMMAND rtsp://example.com/forman.mp4 RTSP/1.0
CSeq: 1
User-Agent: A description of the useragent
```

With a response somewhat like:

```
RTSP/1.0 200 OK
CSeq: 1
Date: Mon, Apr 06 2009 11:11:02 GMT
A command-specific response, if applicable
```

The following is a small overview of the different RTSP commands and their effects.

OPTIONS

When the client sends **OPTIONS** to an RTSP server, the server will reply with a list of commands it supports.

DESCRIPTION

When a server gets a **DESCRIPTION** request for a resource, the client expects a response containing an (Session Description Protocol) (SDP) entry describing the media object, with information about different streams, media-types, formats and lengths, et cetera.

SETUP

SETUP is called to setup the transmission of a new stream. There can be many stream contained in an RTSP resource. The client sends additional information such as:

```
Transport: RTP/AVP;unicast;client_port=42450-42451
```

which tells the server which ports and by which mechanisms it wishes to receive data. The server responds, upon success, with

```
Transport: RTP/AVP;unicast;
destination=192.168.100.209; source=192.168.101.230;
client_port:42450-42451;server_port=49160-49161
```

informing the client of where the data will be coming from.

PLAY

This command is for telling the server to start transmitting the stream from a given time, and respond with RTP-time and sequence numbers corresponding to the time specified. The command can also be sent without specifying a time, to which the server will respond with information about the current location in the stream.

TEARDOWN

This command tells the server that the client is done with the stream, and lets the server close the connections cleanly and free associated resources.

Some RTSP commands are deemed out-of-scope for this thesis and will not be mentioned.

2.2.2 RTP

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|V=2|P|X|  CC   |M|     PT      |       sequence number         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           timestamp                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           synchronization source (SSRC) identifier            |
+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
|            contributing source (CSRC) identifiers             |
|                             ....                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2.1: The RTP header [3]

The main data-carrier used in this scheme is the Real-time Transport Protocol [3]. It is a protocol defined by RFC 3550 [3] to provide end-to-end transport of data with real-time properties. Other concerns such as stream setup, signalling and Quality of Service (QoS) gets taken care of by RTCP [3] and RTSP [2]. As such, real-time data sent with RTP usually travels over User Datagram Protocol (UDP), leaving flow control and congestion control up to the application. Additionally, one RTP session is limited to one stream, so an application wanting to stream both video and audio simultaneously — which has become increasingly popular since 1927 — would need to use two different streams, leaving the synchronization up to RTCP.

RTP and RTCP use dynamic port numbers, and RTP is recommended to use an even port, with RTCP using the next port.

2.2.3 RTCP

RTCP [3] is the out-of-band control protocol for use with RTP. While the RTP protocol only carries data, RTCP deals with flow control and QoS. The interesting parts of this protocol are the **Sender Reports** and **Receiver Reports**. The Sender Report informs the client about the servers time and last packet sent, allowing the client to properly synchronize several streams. The Receiver Report communicates packet loss and inter arrival jitter, thus allowing the server to adjust bandwidth and encoding to suit the current state of the network connection.

2.2.4 RTP mixers and translators

Somewhat relevant to the topic presented in this thesis is the work in RTP literature about **mixers** and **translators**. A mixer is an RTP-level relay, typically used when conferencing, which resynchronizes and possibly re-encodes several distinct audio or video stream into one stream, thus enabling conference-participants with low-speed network access to participate without having the entire conference settle on the lowest common denominator. An RTP-translator is effectively a tunneling device, e.g. for tunneling the RTP UDP-traffic over a firewall or subnet where UDP is blocked. The proxy described in Chapter 3 on page 19 does not fall into either of these categories,

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|V=2|P|    RC   |   PT=RR=201   |             length            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     SSRC of packet sender                     |
+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
|                 SSRC_1 (SSRC of first source)                 |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| fraction lost |       cumulative number of packets lost       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           extended highest sequence number received           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      interarrival jitter                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         last SR (LSR)                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   delay since last SR (DLSR)                  |
+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
```

Figure 2.2: The RTCP header for a receiver report packet [3]

but is a broadcaster, receiving data from one sender, and forwarding a subset of that same data to one or more recipients.

2.3 Scalable Video Coding

2.3.1 Video encoding

Video in its most basic form is a 3-dimensional matrix of color pixels. Two of the dimensions form the **spatial** directions of the moving images (width and height) while the third dimension is **temporal** (time). This is an immense amount of data. Revisiting the example of a Blu-Ray movie, a two hour feature in full HD-quality with 25 frames per second would amount to a matrix $1920 * 1080 * 180000$ elements big. Using 24 bits for each color pixel would leave us at a tad more than one terabyte (10^{12} bytes).

To overcome these immense amounts of data, all digital video is typically compressed. Without going too much in depth of the methods used, video is compressed both in spatial and temporal dimension. Spatial compression is achieved by taking into account the different levels of detail the eye can perceive, and temporal compressions is based on storing differences between frames in terms of motion. Digital video encoding has been a major field of research since its inception in 1984, with H.120 [4] being the first implementation.

2.3.2 Overview

The Scalable Video Coding is an extension to the H.264/MPEG4-Advanced Video Coding (AVC) standard, and its standard was approved in 2007. Its objective is to enable the encoding of a high quality video-stream consisting of several substreams of data, with the quality/bit rate ratio being comparable to that of H.264. One can then drop some of these substreams, ending up with a lower quality, but complete, video-stream. Scalable video bitstreams are also know as 'multi-layer bitstreams', while non-scalable video bitstreams are called 'single-layer bitstreams' [5].

This means that to 're-compress' a video stream into a lower quality version, all needed is to drop certain data from the video stream. This called 'scalability'.

There are several types of scalability modes available:

Temporal scalability Dropping full frames, effectively reducing the temporal resolution (frame rate) of the video-stream.

Spatial scalability Dropping spatial detail in each frame, thus reducing the spatial resolution. Each added substream further refines the resolution of the image.

Signal-to-Noise Ratio (SNR)/Quality/Fidelity The substreams provide the same spatio-temporal resolution, but at different levels of fidelity.

These modes can be combined arbitrarily. We are only using the spatial scalability mode in the implementation of this thesis.

2.3.3 Encoding

While the internal workings and details of SVC and H.264 are out of scope for this thesis, the resulting bit stream is of great importance. Encodings like WMV and MPEG-2 are ignorant of the underlying transport layers [6]. In order to stream video encoded using these encodings over the network, the content needs to be re-framed and re-adapted.

H.264, and by extension SVC, uses a concept named Network Abstraction Layer (NAL). The NAL facilitates the ability to map H.264/AVC data to different transport layers, both on-the-wire formats such as RTP, and filecontainers [5].

```
+---------------+
|0|1|2|3|4|5|6|7|
+-+-+-+-+-+-+-+-+
|F|NRI|  Type   |
+---------------+
```

Figure 2.3: The NAL unit header as defined by RFC 3984 [7]

```
+---------------+---------------+---------------+---------------+
|0|1|2|3|4|5|6|7|0|1|2|3|4|5|6|7|0|1|2|3|4|5|6|7|0|1|2|3|4|5|6|7|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|F|NRI|  NUT    |R|I| |  PID    |N| DID |  QID  | TID |U|D|O|R2 |
+---------------+---------------+---------------+---------------+
```

Figure 2.4: The extended NAL unit header used by SVC [8]

Unlike other formats such as MPEG-2 and WMV, an H.264 bit-stream coming from the encoder is already packetized into NAL Units (NALUs). These are strings of bits of a certain length delimited with a start code prefix. This makes the format both flexible and easy to parse. Each NALU starts with a one byte header detailing the NALU type and number. SVC video has an additional 3 bytes [5] header which describe the substream layer dependencies.

This makes it easy to extract a sizable amount of information from the video bit stream by shallow inspection of incoming packets, and facilitates a lightweight stream scaling approach without having to decode or further analyze the incoming video stream.

A set of NALUs together form an access unit, the decoding of which results in a complete frame. The framer made decide to aggregate several NALUs into one 'virtual' NALU (STAP packetization mode), or split one NALU into many (FU fragmentation mode). This is done in order to respectively limit protocol overhead and ensure transmission of packets larger than the network's end-to-end Maximum Transmission Unit (MTU).

2.3.4 Dependencies

Like most common video encodings, AVC and by extension SVC, utilize inter-frame compression, meaning that a video stream is composed of several different frames.

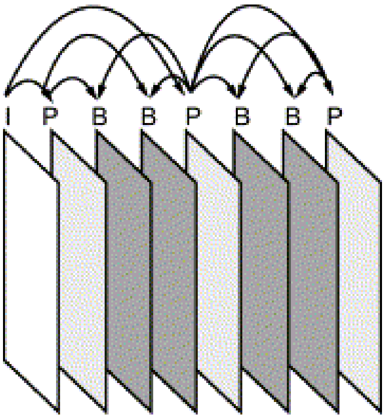


Figure 2.5: Figure showing common inter-frame dependencies in compressed video streams [9].

I-Frame Intra-coded frame. This frame has no dependencies, and consists of a self-contained picture frame. This requires more bits to store than the other frames. By being self-contained, it acts as a key frame, allowing the decoder to 'jump into' a stream.

P-Frame Predicted frame. This frame is dependent upon a number of previously decoded frames, and contains a mixture of picture data and motion vectors, which allows the decoder to reconstruct the frame cheaply.

B-Frame Bi-directionally predicted frame. This frame type is a specialized version of the P-frame, and may used a weighted average of motion vectors for different kinds of frames, and is seldom used for further prediction by following frames. This can be compressed very aggressively.

This implies a great deal of dependency between frames. While a missing I-frame nullifies all dependant frames, a missing B-frame has a far lesser impact, decreasing only the quality of the given frame. Additionally, different types of frames can be compressed with varying aggressiveness, and different frames have varying entropy. The leads to the resulting stream having a very variable bit rate, as witnessed by figure 2.6.

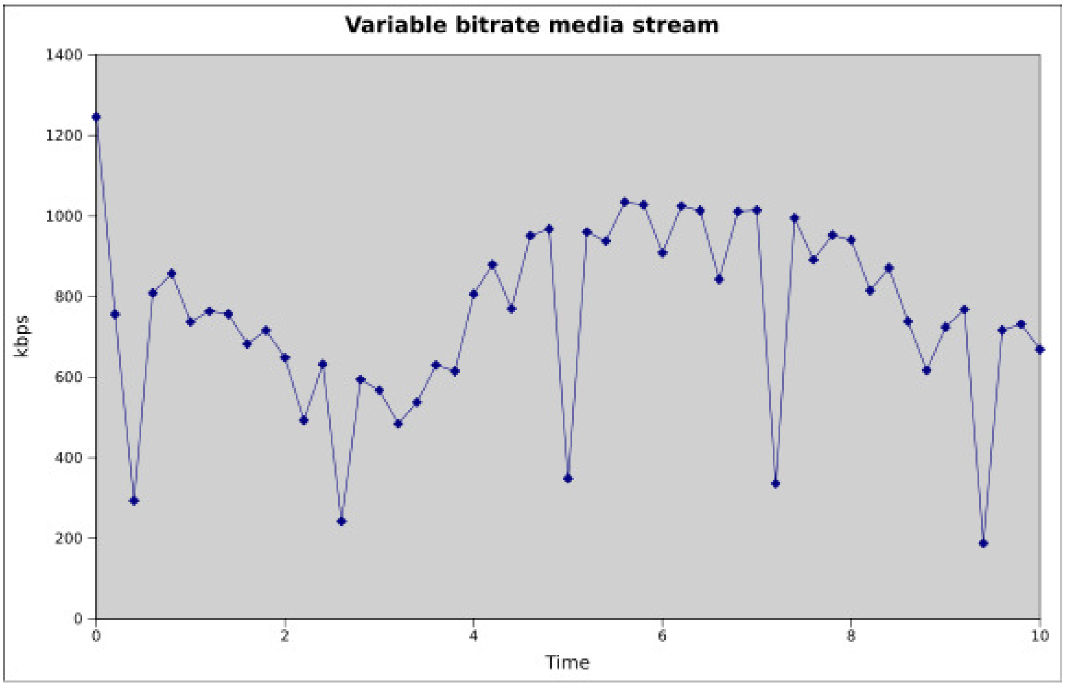


Figure 2.6: Bandwidth fluctuation in a variable bit rate stream

The dependencies between types of frames are shown in Fig. 2.5 on the facing page. B-frames depend on next and following reference frame, while I-frames are independent and can be decoded independently.

There are also dependencies between the different kinds of substreams. A lost temporal enhancement-layer only decreases the quality of the video stream in a predictable way, while a missing base layer for a few frames also makes the enhancement layer useless. Very specific dependency information is embedded in the extended SVC header, both in temporal and spatial dimensions.

2.4 Datapath

The obvious way of re-sending data to multiple receivers is shown in Figure 2.7. As a packet of data enters the network card, it is copied using Direct Memory Access (DMA) into the kernel's memoryspace. After the kernel has figured out which application the packets is destined for, it is placed in an application-specific queue. When the application in question then calls `read` or any variation thereof, the kernel copies the data into the userspace memory buffer specified by the application. Then, for each receiver, the application calls `write` (or one of its variations), which instructs the kernel to copy the data into its own memoryspace, and then send it to the network card using DMA.

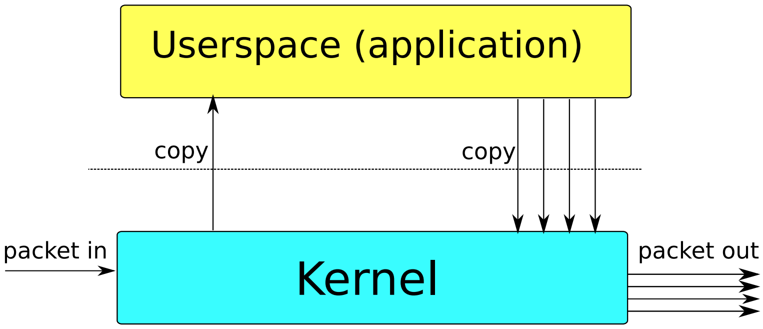


Figure 2.7: The typical approach to re-sending data.

Seeing as this is ineffective, some effort has gone into remedying this cost. One relevant concept is `sendfile`, a new Linux system call created for sending data from one file descriptor into another. While this removes the overhead of copying kernelspace ↔ userspace, it removes the possibility for the application to inspect the data before re-sending it, and it also incurs a cost of copying between buffers in kernelspace. Similar system call interfaces like `splice`,`vmsplice` [10] and `tee`, perform the same basic function. `mmap` is also widely used for disk-to-socket copy avoidance, but it is not applicable for reading data **from** a socket.

In addition to the cost of copying memory, using system calls are expensive. A system call effects a context switch in order to hand control over to the operation system, with a following context switch back to the application as the system call completes. Since a context switch changes the virtual address-space of the CPU, it also invalidates the Translation Lookaside Buffer (TLB) and CPU-cache. These are expensive operations, and they also bring about an indirect cost due to future cache-misses.

Although CPU-consumption is not a limiting factor in media streaming today, network bandwidths are outgrowing CPU speeds, and CPU consumption might emerge as a problem in the near future. Additionally, reducing computation costs will facilitate lower power usage and/or hardware requirements.

Efficient ways to stream media has been extensively researched, with differing emphasis:

Sending media disk-to-socket This is the most researched variation. **Kstreams** [11] focused on this with a heavy-weight in-kernel implementation, and Brustoloni investigated memory mapping of data [12].

Caching streams on disk The **Mocha** framework is a quality adaptive caching proxy, and the research on it deals with caching decisions and replacement policies [13]. Wang, et. al. researched an approach to caching proxy servers on a WAN [14].

Collaborative streams Collaborative streams such as video conferencing is a field with great commercial interest [15].

For our scenario, we have very specific needs. Improvements with regards to copy avoidance in the Linux kernel aim for generality [16]. Thus for our approach we will need a tailor-made solution to achieve efficient data re-sending without unneccessary cost of data-copying, while maintaining flexibility and allowing data to be inspected.

Chapter 3

Userspace signalling proxy

3.1 Introduction

We will implement a split streaming media proxy. It will receive **one** stream from the content server, and fan it out to a large number of clients. All data will be forwarded by the kernel module, but the main logic and connection handling will be handled in userspace. This is necessary both to obtain a clean design, and to keep the in-kernel code as simple as possible. This chapter will elaborate on the design and implementation of the userspace signalling proxy.

3.2 Architecture

The proxy will be implemented from scratch as a non-transparent proxy, that is, the clients connect to the proxy while believing they are connected to a real server. Likewise, the server thinks the proxy is just another client. Since host names and IP-addresses are embedded into RTSP requests and responses, we will need to translate between these.

Clients connect to the proxy using the RTSP protocol. The proxy handles signalling proxy↔client and proxy↔server, while the actual video stream data path goes through the kernel. The proxy needs to handle two kinds of communication, viz.: RTSP and RTCP. As detailed in sections 2.2.1 on page 7 and 2.2.3 on page 10, RTSP is used intermittently to set up and tear down the connection. RTCP is used periodically to report packet loss, jitter, and other QoS-related data, in addition to being used by the server to determine the liveliness of clients (in this case, the proxy). The main function of the proxy is to handle the fanning-out of one media stream to multiple clients — requesting one stream from the server regardless of the number of clients[1].

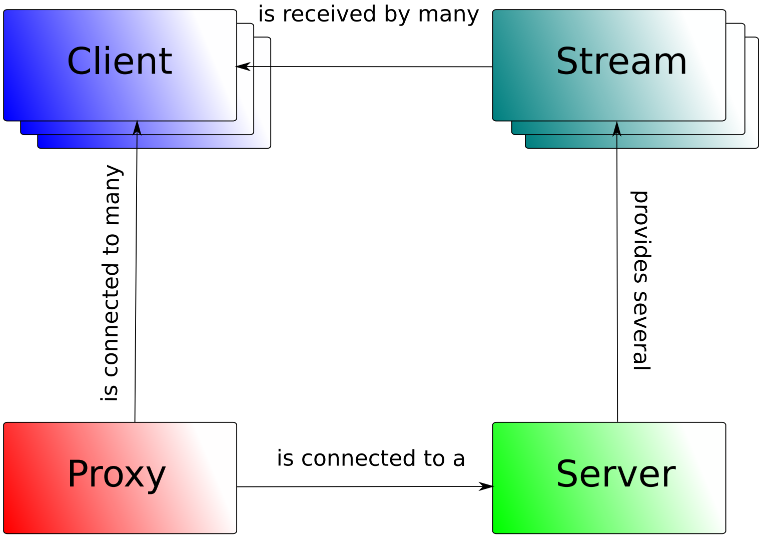


Figure 3.1: The proxy architecture

We have structured the proxy as a single-threaded event-driven program. Our reason for this approach is the limited amount of I/O the proxy will perform. Using `select` to multiplex the proxy's input and output, we can achieve timely handling of requests without having a thread per connected client.

The overall object-oriented design of the proxy is modelled as in figure 3.1. In this context, a **stream** refers to one of many possible streams encapsulated in a resource, as offered by RTSP. Typically there are two streams, one for each audio and video. More streams are possible, however, e.g. for multiple languages or viewing angles. At

[1] We do not request a stream with no clients, however.

start-up, the proxy connects to the media server. Upon receiving an RTSP-connection, it creates a client-object for the connecting clients. It then handles requests as follows:

RTSP request arriving from the client

The proxy will be listening for RTSP requests on a given port. As a client connects to the proxy server and issues an RTSP request, the proxy responds to the client, after possibly relaying the request to the server and getting a response. The handling of specific RTSP commands are detailed in section 3.3.

RTSP response arriving from the server

This is a response to a previous request by a client. The response does not indicate which request it corresponds to, so a queue of pending requests is maintained. We can even have several requests pending at a given time, and in this case we are guaranteed to receive responses in a First-in-First-out (FIFO) manner [2]. If the response should be relayed to clients, we forward the response to all clients registered as interested in the response.

RTCP report arriving from the client

The RTCP report contains information primarily regarding QoS. The loss statistics is used for estimating bandwidth, and is detailed further in Chapter 5 on page 37. Unlike text-based RTSP, RTCP is stored as binary data, and its somewhat complicated format needs to be parsed before accessing data.

The proxy also sends RTCP reports to the server at regular intervals, as the server uses these messages to determine liveliness. The proxy connects to one server and an arbitrary number of clients. The server can have any number of streams, although we only handle one resource from the server concurrently, a resource can be split into streams. The most common configuration is two streams, one for each audio and video, while any number of streams is possible.

The proxy will request at most one resource from the server at a time. If a client is the first to connect, the proxy will start the stream from the server; if a client connects while the proxy is already serving a resource, it will simply latch onto its streams. This is well suited for streaming of live media, with all clients watching the stream at the same point in time, but the main concept of this thesis is expandable to near-VoD solutions as well.

3.3 RTSP handling

All RTSP commands include a sequence number and date/time. These will all need to be maintained correctly when relaying commands, so we are keeping track each clients latest sequence number, and it is inserted into the relayed command when dispatched. Likewise, the RTSP answers contain a sequence number and occasionally a session number.

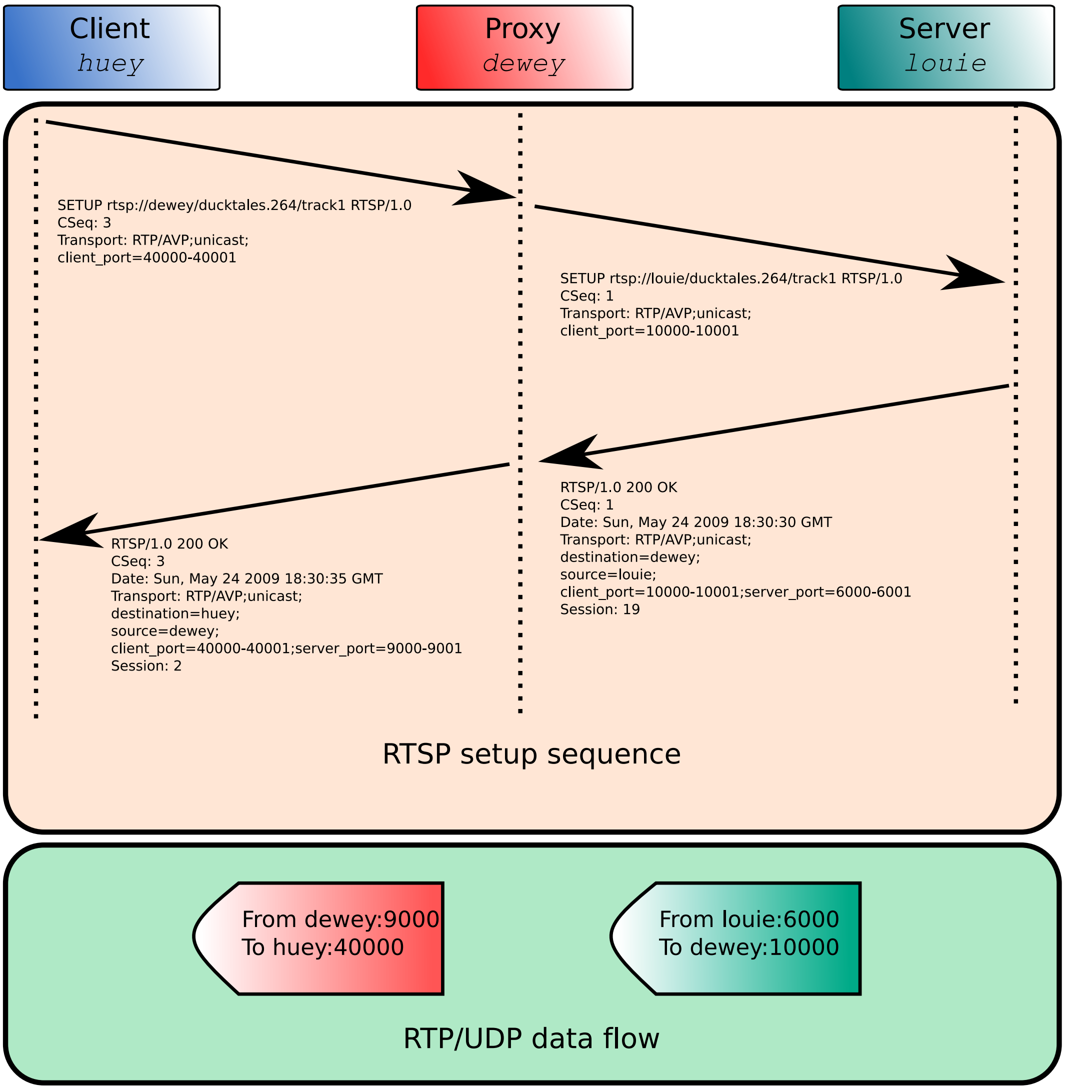
Additionally, since several of the RTSP commands and responses contain direct IP-address and port references, the proxy will need to translate between these addresses when relaying RTSP commands. This is shown in figure 3.2 on the facing page.

OPTIONS

Send back a list of commands we support, which are OPTIONS, DESCRIBE, SETUP, PLAY and TEARDOWN. The proxy can answer this without checking with the content server.

DESCRIBE

Since the proxy has no information about the media, we relay the command to the server, and relay the answer back to the client.



```
SETUP rtsp://dewey/ducktales.264/track1 RTSP/1.0
CSeq: 3
Transport: RTP/AVP;unicast;
client_port=40000-40001
```

```
SETUP rtsp://louie/ducktales.264/track1 RTSP/1.0
CSeq: 1
Transport: RTP/AVP;unicast;
client_port=10000-10001
```

```
RTSP/1.0 200 OK
CSeq: 1
Date: Sun, May 24 2009 18:30:30 GMT
Transport: RTP/AVP;unicast;
destination=dewey;
source=louie;
client_port=10000-10001;server_port=6000-6001
Session: 19
```

```
RTSP/1.0 200 OK
CSeq: 3
Date: Sun, May 24 2009 18:30:35 GMT
Transport: RTP/AVP;unicast;
destination=huey;
source=dewey;
client_port=40000-40001;server_port=9000-9001
Session: 2
```

Figure 3.2: RTSP hostname and port translation by the proxy

SETUP

Register the client as interested in the stream identified in the request, and send back an 'OK'-response. If this is the first client interested in the stream, the stream needs to be initiated, and we relay the request to the server.

PLAY

This command tells the server that it should start sending data corresponding to the stream identified in the request. We intercept this message and inform the kernelmodule to start relay data to this client. This command elicits a response containing RTP sequence numbers et cetera, so we cannot answer this ourselves. However, the RTSP standard [2] allows a client in playing mode to send the PLAY command to the server, and receive the response we are after without altering the play-state, so we relay the command to the server, and relay the answer back to the client.

TEARDOWN

Remove the client from the stream identified in the request, and tell the kernel module to stop relaying data to this client. If this client was last client receiving the stream, we relay the TEARDOWN request to the server in order to close down stream, and to free resources on the proxy.

Figures 3.3 on the next page and 3.4 on page 26 show a sample of the RTSP interaction for both common scenarios, viz.: a client requesting a 'new' stream, and a client requesting an already active stream.

3.4 Summary

This chapter has outlined the design and implementation of the control path of our split proxy. The design and implementation of the data path, along with the interaction between the parts will be found in the next chapter.

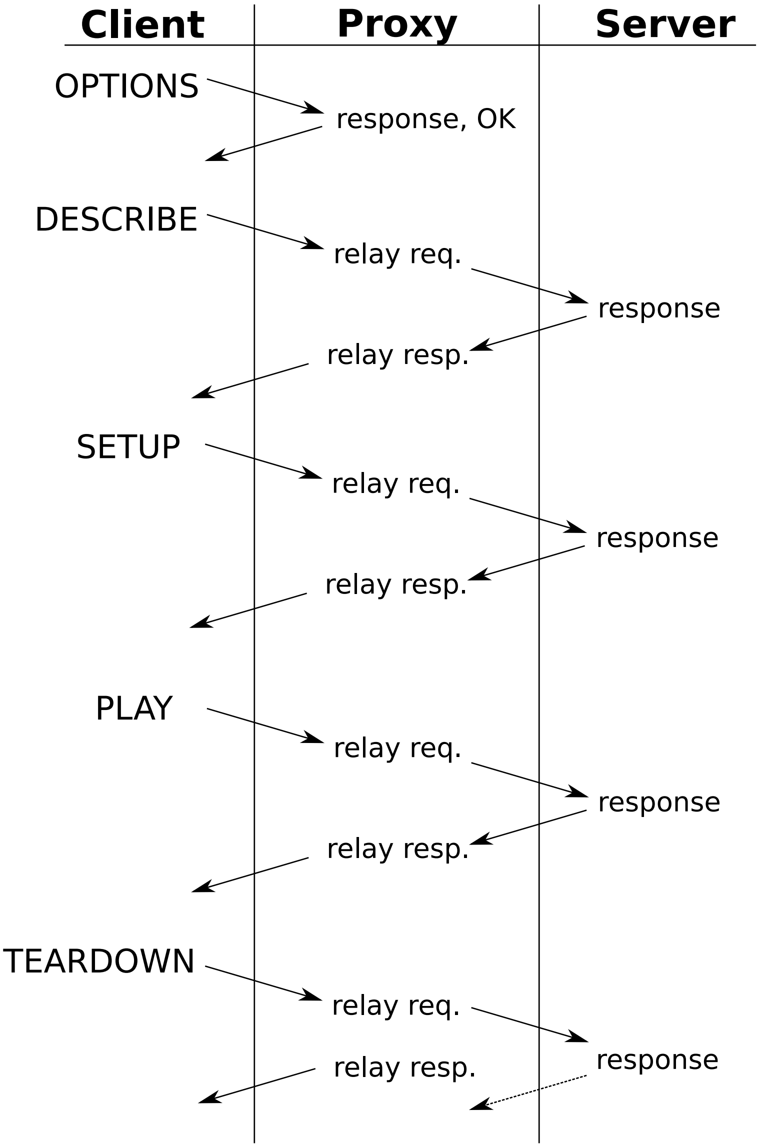


Figure 3.3: Interaction between proxy, client and server for a client starting a new stream from the server

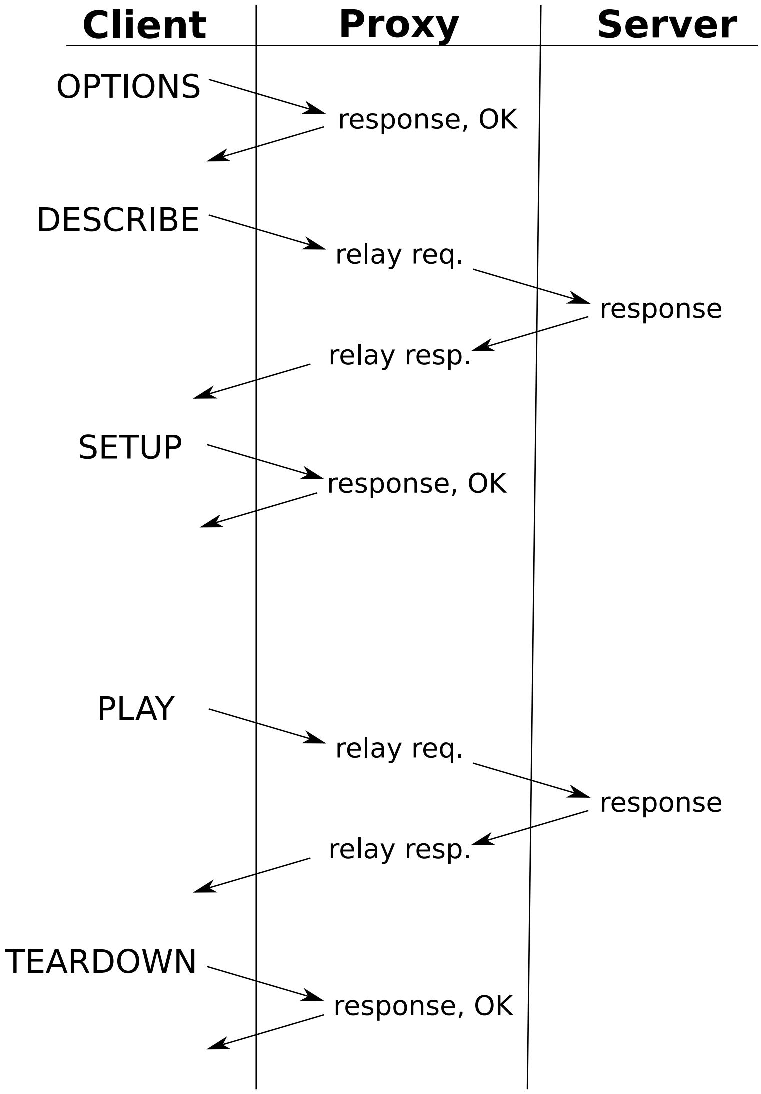


Figure 3.4: Interaction between proxy, client and server for a client requesting an active stream

Chapter 4

The kernelspace packet re-sending

The traditional userspace data path for receiving and re-sending data over the network is CPU-ineffective, as data transitioning back and forth over the userspace↔kernelspace barrier is subject to copying. We aim to be able to resend data in kernelspace, thus avoiding the excessive copying operations.

4.1 Rationale for doing re-sending in-kernel

A straight forward [1] userspace solution to data re-sending over the network would require a data copy and a system call per receiver per packet.

An alternative is a solution in which the kernel actively resends packets received according to a maintained list of receivers, without the userspace program's intervention. This would result in no userspace ↔ copies and no system calls — apart from the system-calls used for signalling, the cost of which are insignificant by comparison.

As shown by figure 4.1, this will remove the application's role in re-sending data, but the data still requires a copy operation per received packet. However, the cost of the

[1] A solution using standard POSIX system-calls such as `read` and `write`.

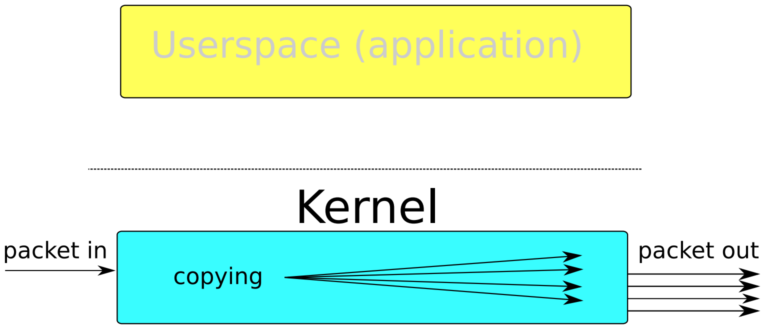


Figure 4.1: Re-sending data contained in kernelspace

copying is greatly reduced by exploiting some zero-copy features of the Linux kernel and common network interfaces, as further detailed in section 4.4 on page 31.

4.2 Netfilter

There two ways to get code to run in kernelspace: either as a kernel module or as code compiled into the kernel. Adding code into the kernel by means of compiling it statically into the kernel gives by far the biggest flexibility. We could add new system calls, modify behavior of existing calls, et cetera. But the freedom comes with a cost; this approach has the possibility of becoming very intrusive and is additionally very cumbersome, both in development and in use. A potential user would need to install a custom kernel, which would need to kept up to date with regular Linux kernel releases.

Writing code as a kernel module which is dynamically linked into the kernel at run-time makes for a very comfortable work cycle, and a non-intrusive installation process — loading a module can be done at run-time, while a kernel change would require a system restart. As for drawbacks, there are limits to what you can do with a kernel module. It cannot change data-structures defined elsewhere, and you rely on registration functions to get your code called.

It so happens that the Netfilter framework offers registration functions which lets your code be called per packet received. By giving the correct rules to Iptables when setting up Netfilter, the framework does the filtering for you — so it is perfect for our needs. The limitations imposed by being a kernel module do not affect us, since we do not need to change any kernel data structures.

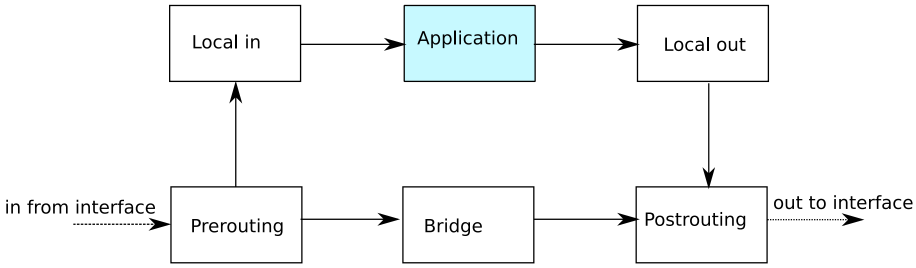


Figure 4.2: An overview of the placement of the different Netfilter hooks

With deciding on using a Netfilter module, we can use the Iptables-framework to register a function as a call-back for received packets. We are interested in intercepting packets as soon as they come in to the system, to avoid any excess protocol processing. There are no applications listening for UDP packets, so the system would then reject packets destined for the machine after routing. Hence we chose to latch onto the pre-routing hook (See Fig 4.2). The only downside to this is that hook would trigger on packets not destined to the local machine, in case of the machine being used as a router with packet forwarding enabled. This is an edge-case and should not prove a problem in practice, so using pre-routing should be a good choice.

The netfilter rule is inserted through Iptables as follows:
`iptables -t mangle -A PREROUTING -p udp --dport 41966 -j PROXY`, with one rule required for each port we want to receive data on.

Upon reception of a UDP packet to the given port, the kernel module performs the following actions:

1. For each entry in the list of clients:

   (a) Create a duplicate of the packet

   (b) Change the source IP, destination IP, source UDP port, destination UDP port of the new packet.

(c) Inject the packet into the networking stack at the IP-output level.

Using this approach verbatim can lead to severe fairness problems. Imagine the scenario of a constantly *almost* saturated network interface, and a list of 90 receivers. The first receiver is guaranteed to get his data, while the last receiver would never get any data at all, since the network interface would be overloaded with data before his turn. Our simple solution to this would be to rotate the list of receivers by one each time we service a packet. This should even out the fairness disparities in the long term.

All this is done synchronously as the received packet gets handled by the IP-protocol layer. There is a trade-off regarding system response time, and memory use and latency. Having the kernel do immense amounts of work while in network processing code will preempt userspace processes, and might provoke the network interface to drop incoming data if it's backlog becomes too long. An alternative could be to defer the work to a kernel thread or similar, but this will require increased memory usage by buffering incoming data.

4.3 Sending a packet

Sending an UDP packet is typically done in userspace, via the `sendmsg` system call, while providing a UDP socket, a message and an address. As shown in 4.3 on page 32, its path through the kernel goes through many different interfaces and layers, and we need to figure out where in this call-graph we should inject our own packet when sending it from inside the kernel.

The system call triggers `sys_sendmsg`. which looks up the file descriptor and matches it to a socket. The call is then relayed to socket's `sendmsg` function, `udp_sendmsg`. It calls `ip_append_data` which copies the data from userspace to a kernelspace socket buffer (skb), which is queued onto the sockets write queue. Next, it calls udp_push_pending_frames which adds the UDP header to the skb. This hands the control over to the IP layer with ip_push_pending_frames which in turn adds the IP header to the soon finished packet.

Next, the skb is handed to ip_local_out. This function simply check with Netfilter's **local out** (see figure 4.2 on page 29) rules if this packet is allowed to pass. If this turns out OK, ip_output is called, which in turn calls ip_finish_output, which sole purpose is to check with Netfilter's **post routing** hook if everything is OK with the packet.

After this, the packet is handed off to the interface, and we move on to the link layer, where Ethernet (in most cases) sends the packet off.

Since the kernelmodule is synthesizing packets without sockets corresponding to sender and receiver, we deemed it necessary to insert our packets after ip_push_pending_frames. We also did not see a need to subject our packets to the Netfilter framework on the way out, so we injected our packets in the Transmission Control Protocol (TCP)/IP stack right above the link layer with ip_finish_output2[2].

4.4 Zero-copy packet forwarding

We cannot simply relay a verbatim copy of the packet we received. The data buffer includes IP, UDP and RTP headers, and the IP and UDP header will need changing in order for the outbound packet to arrive at its destination. The obvious solution here is to make a complete copy of the packet, and then change the relevant fields in the IP and UDP header. Unfortunately, this involves copying $n * m$ bytes per packet received, with n being the packet size and m being the number of receivers. The video stream used

[2] Actually, we chose to use a modified version of the ip_finish_output2 function, bypassing some checks regarding multicast and broadcasting.

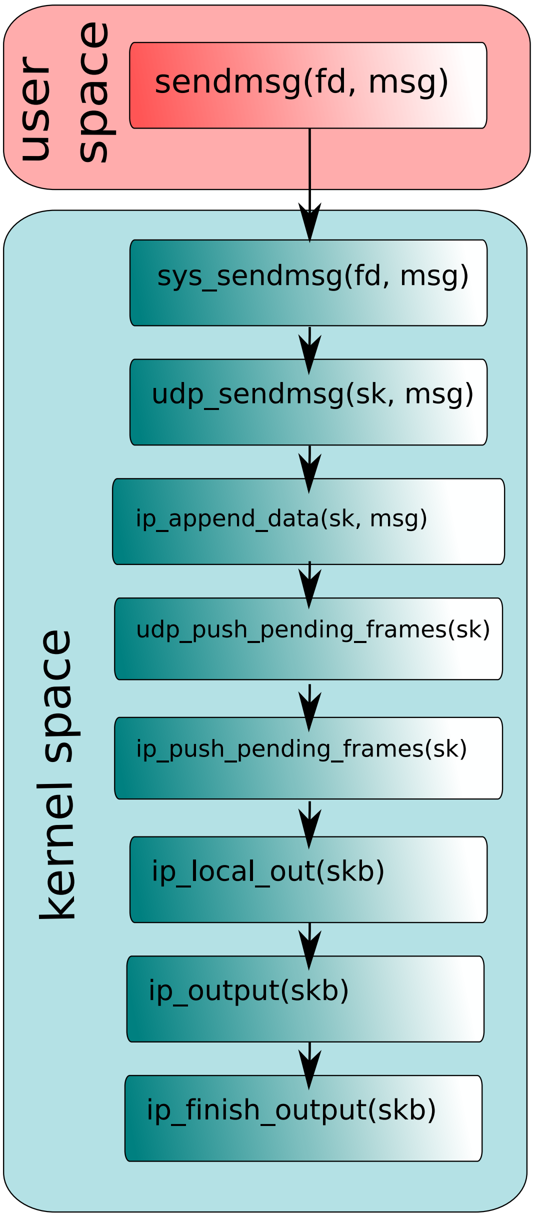


Figure 4.3: A simplified flow graph of an UDP packet's egress path through the Linux Kernel

for testing had an average packet length of 466 bytes and an average of 271 packets per second, which for 100 users amounts to 27100 copies per second of 466 bytes each, for 11 megabytes of copying each second. While this might seem inconsequential (the testing system, see Chapter 6 on page 43, is rated at an effective memory bandwidth of 6400 MB/s [17]), it provokes hard-to-predict performance degradation due to cache-misses et cetera.

Our goal is to be able to supply a new set of IP and UDP headers (a total of 28 bytes), bundle it together with the data — but without copying — and have the network card send it off. This is possible through what is known as Gather Scatter I/O (GSO), where the network card independently copies all the data fragments into its buffer using DMA. This is detailed in figure 4.4.

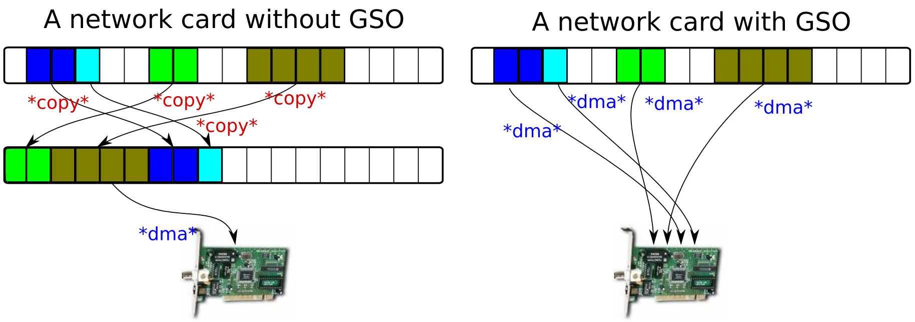


Figure 4.4: Operation of network card with and without GSO support

When using a network card with GSO support, we can make a `sk_buff` structure containing the headers, and provide tuples of pages, offsets and lengths to be copied into memory by using DMA. These are stored in a list of `skb_frag_structs` (see figure 4.5) in the `sk_buff`.

```
struct skb_frag_struct {
        struct page *page;
        __u32 page_offset;
        __u32 size;
};
```

Figure 4.5: The `skb_frag_struct` data structure.

We can only DMA from one page in one entry, so we need to take special care that `page_offset` + `size` <= *PAGE_SIZE*. Incidentally, the ubiquitous Ethernet LAN technology enforces a MTU of 1522 bytes, well under the page size on x86-64 architectures of 4096 bytes.

4.5 Reporting

The kernel can resend data, but has no knowledge of where to send data. Thus we need a way of communicating state about receivers between the kernel module and the proxy. Communicating between two user processes is easily done using standard Inter-Process Communication (IPC) mechanisms, but these are useless in establishing userspace ↔ kernelspace-communication.

To overcome this problem, we need specific OS support, and fortunately the Linux kernel offers several alternatives. These include:

- Ioctl and sysctl. These are both based on a single system call to the kernel. Different requests are multiplexed through assigned code numbers. To implement either of these would require non-transparent changes to the Linux kernel, which is too heavy-weight for our needs.
- The virtual file system based. These abstract the reading and writing away in files, and are very transparent to use from userspace.
  - Procfs. Mainly used for information about processes and kernel subsystems,
  - Sysfs. Used for miscellaneous kernel internals.
- set/getsockopt. Is used for setting options on sockets (again, hence the name), but requires changes to the Linux kernel.
- Netlink. This is a socket interface which acts as a bi-directional message-passing interface between the kernel and a userspace processes. It uses a protocol number which is unique per application.

Of all these alternatives, we decided upon using Netlink. It has the cleanest interface, and its semantics are very suitable for this purpose. We will need to commandeer a number to identify the protocol we are using.

Using Netlink, we will need to create a light-weight protocol. For the kernel module's operation we will need four commands:

Add To add a recipient to the data-path.

Delete To remove a recipient from the data-path.

Flush To remove all recipients from the data-path, in case of proxy-restart or crash.

Loss To inform the kernel module about packet loss as reported to the proxy by the recipient via RTCP. This will be utilized for adaptive stream scaling, and is described in depth in Chapter 5 on page 37.

As shown in figure 3.2 on page 23, the packet needs to be sent out to the correct IP-address, with both source and destination port sat correctly. Hence we will need to store IP-address and both ports for each receiver. When the proxy adds a receiver to the kernel module via Netlink, it will be stored in a link list of `xt_proxy_data` items, which are defined in our program as:

```
struct xt_proxy_data {
    char cmd;
    unsigned int pid;
    unsigned int ip;
    unsigned short srcport;
    unsigned short dstport;
    unsigned int data;
};
```

The userspace proxy sends these commands to the kernel module when necessary. The kernel module has the possibility of sending an answer to the proxy, but we have as of yet not had a use for that functionality.

Chapter 5

Adaptive stream scaling

5.1 Overview

Different clients may have different demands of a stream. Differing factors include bandwidth, resolution and frame rate. Using a video encoding with a multi-layer bit stream such as SVC, the proxy can selectively drop packets belonging to a specific layer of the stream, thus reducing quality of a stream in benefit of reduced bandwidth.

Our goal is to drop layers in the stream based on each clients available bandwidth.

5.2 Analyzing the stream

In addition to being a Variable Bitrate (VBR) encoding, SVC has several layers in a single stream, which also vary, as graphed in figure 5.1 on the following page. These layers are distinguishable by several ID specifiers in the SVC-header. The three types of layers are **dependency layers**, **quality layers** and **temporal layers** [8].

We will only use the Dependency ID (DID) which separates the stream into 8 layers. DID0 is the base-layer, DID1 is an enhancement layer that depends on and refines

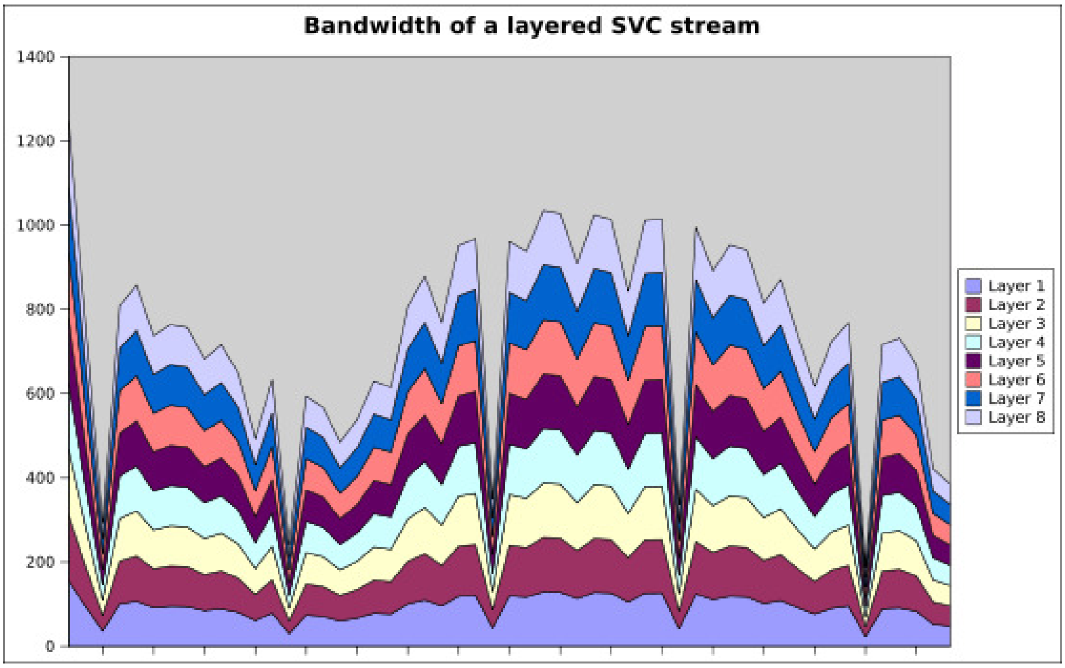


Figure 5.1: Possible bandwidth distribution for a multi-layered stream

DID0, and so on, with DID7 being the less important layer. This gives a coarse grained scalability in the spatial dimension [8].

In order to estimate how much bandwidth the stream needs, we decided to keep a moving average of bandwidth for each layer. Every 200ms we count up the bytes received in the last 200ms period, and incorporate it in our running average as follows

$$Average_{new} = \frac{Average_{old} * 15 + \frac{bytes\_received}{time_{delta}}}{16}.$$

This calculation is repeated for all 8 layers. The weights and timespans used in the calculation are arbitrary, but work nicely. This will provide us with a fair average of the bandwidth over time. There are also some non-SVC specific NALUs in the data stream. These are typically parameter sets and other meta-data, and is treated as if it had a DID of 0, since the data is important.

5.3 Analyzing the clients

The kernel module keeps track of the amount of data sent to each client. As mentioned in Chapter 3 on page 19, when the userspace proxy receives an RTCP-report from the client, it sends loss statistics to the kernel module. We must keep in mind that the receiver is oblivious to the difference between a deliberately dropped packet, and a packet lost in transmission. Upon receiving the information, we estimate the effective bandwidth of the clients as

$$Bandwidth = \frac{bytes\_sent}{time} * \frac{packets\_sent - (packets\_lost - packets\_dropped)}{packets\_sent}.$$

As an example, let us assume we have a video stream at 8 Mbps (which is equivalent to 1 MB per second), split over 1000 packets per second. We proxied a reduced 800 Kbps version of this stream to a client over 1 second, voluntarily dropping 200 packets. RTCP returns a loss of 300 out of 1000 expected packets. This gives us an estimated bandwidth of $800000B/1s * \frac{1000-(300-200)}{1000}$ which equals $720000Bps$. In effect, the client received 700 packets out of 1000, for an estimated 700000 bps, but our calculation is sufficiently accurate.

The RTP packets are unevenly sized, so an exact number of bytes lost is impossible to obtain, save for an extension of the RTCP packets — but this would require a modification of the media client and is out of scope for this thesis. Another error source is also present; the count of bytes sent to a client is maintained in 'real-time' in the kernel module, but the RTCP report have no information about data sent from the server later than two round-trip times ago. The two metrics are essentially out of sync. This might lead to some bandwidth oscillation, depending on the bandwidth development of the stream.

5.4 Dropping packets

With information in hand about both the available bandwidth to the client, and the bandwidth needed for each layer, determining the correct number of layers to each client becomes a trivial task.

DID	bandwidth requirement	cumulative bandwidth requirement
0	300 bps	300 bps
1	100 bps	400 bps
2	100 bps	500 bps
3	100 bps	600 bps
4	200 bps	800 bps
5	100 bps	900 bps
6	100 bps	1000 bps
7	100 bps	1100 bps

Given the above bandwidth analysis of the stream, and an estimated bandwidth to the client of $850bps$, we find the first layer of data that cumulatively overshoots our bandwidth estimation, in this case DID5 for a total of $900bps$. We always attempt to over-saturate the bandwidth in order to provoke loss for the next round of RTCP reports — we are unable to estimate bandwidth with any certainty if the client receives all packets. The number of the highest layer is stored per client, to reduce computation needed. For each packet received, we simply find he packet Dependency ID and compare it turn to every client's layer 'allowance'. If the DID is lower or equal, we create and send a packet to the client.

To avoid sporadic behavior due to sudden changes, we only allow a client to raise or lower its stored layer allowance by one layer each round. This means that we need to wait 7 RTCP-packet in order to reduce a client from level 7 (the full quality stream) to 0 (only the base-layer and meta data). For our test-stream we receive RTCP-packets approximately every 5 seconds, so for a new stream to adapt to a very low bandwidth client we need to wait about 35 seconds for optimal behavior. This avoided a few possible pitfalls regarding variable bandwidth in the stream and other unpredictable events.

Chapter 6

Results and evaluation

6.1 Test setup

Our testing used three machines.

The proxy

Ubuntu 8.10 Intel Core 2 Duo 2.66 GHz

2 GB RAM

Marvell Technology Group Ltd. 88E8056 PCI-E Gigabit Ethernet Controller

The server

Ubuntu 8.10 Intel Pentium 4 2.4 GHz

1 GB RAM

Intel Corporation 82540EM Gigabit Ethernet Controller

The client(s)

Fedora 9 Intel Pentium 4 3.0 GHz

2 GB RAM

Broadcom Corporation NetXtreme BCM5751 Gigabit Ethernet PCI Express

The machines were connected through a 100 Mbps switched network. The video stream was served by a customized Live555-server with support for AVC/SVC.

All instances of the clients were ran on the same machine, using the `openRTSP` tool provided by Live555. *n* instances of these were started up concurrently, and we started gathering test-data after all clients were started.

The measuring tool used were `dstat` [18], which is an umbrella program gathering up statistics regarding network throughput, CPU usage, interrupts, context-switches, et cetera. Loss was measured as an average ratio over the clients life as reported by RTCP from the clients.

The test stream used for these tests were a ∼1 Mbps video clip encoded with SVC. The clip was 8 seconds long, and was looped indefinitely.

6.2 CPU-usage

Implementation	context switches per second
kernelspace	108
userspace	502

Table 6.1: Context switches per seconds at 90 clients

As seen in figures 6.1 on the facing page and 6.2 on the next page, the kernel packet-relaying is very CPU-conserving, both from avoiding numerous systemcalls and the following context-switches, and avoiding unnecessary data copying. There is some overhead imposed on the userspace implementation, as it is implemented in a high-level language (Python).

CPU usage was measured as a percentage of system total CPU usage, on an idle system. Nevertheless, some services and programs are running in the background, and are a likely source of the outliers in CPU usage. With this in mind, the kernel-resending version appears to scale beyond 40 times higher than our tests, for 4800 users at a total

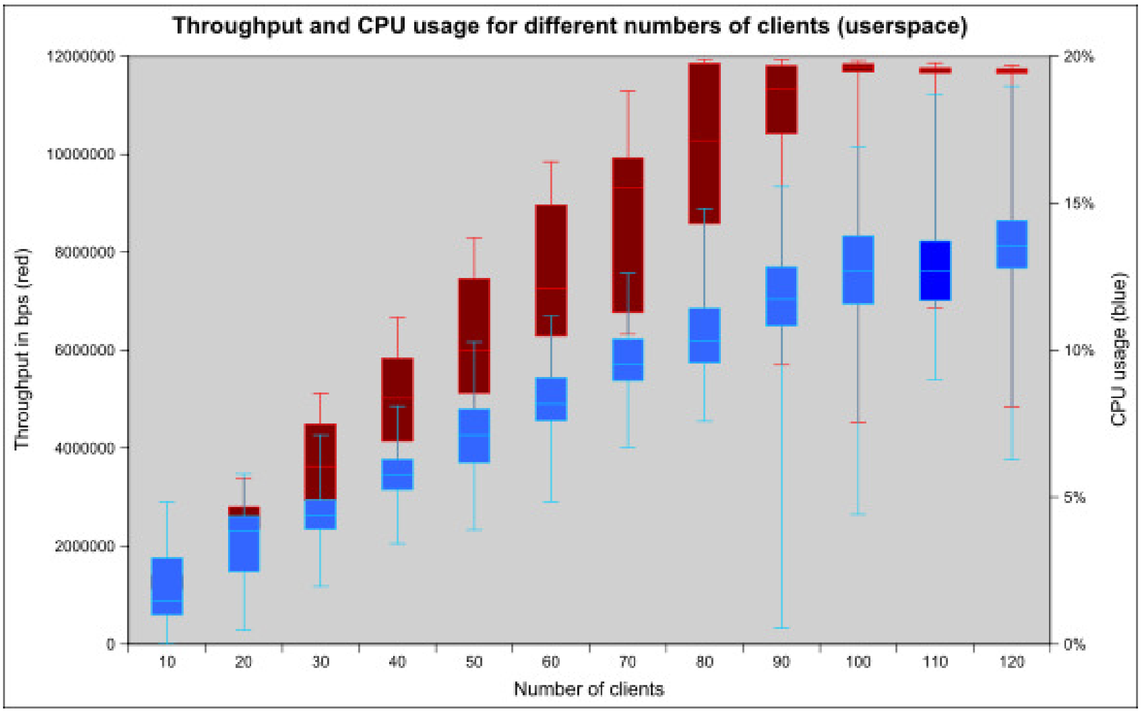


Figure 6.1: Throughput for different numbers of client versus CPU usage (userspace)

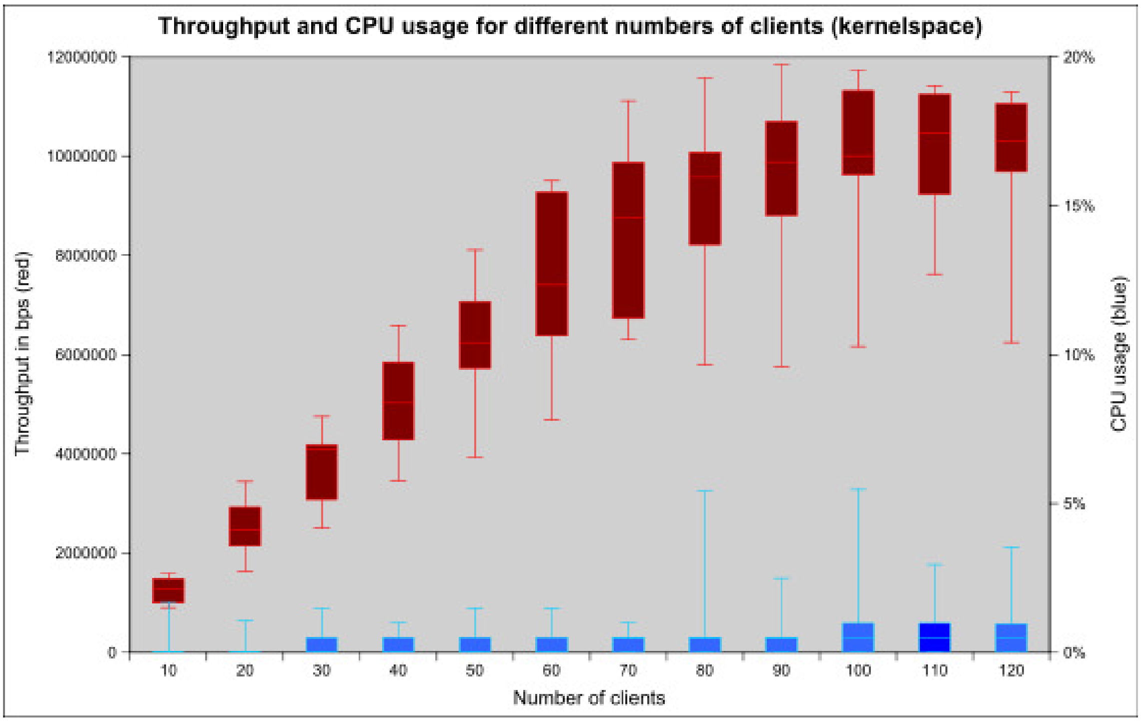


Figure 6.2: Throughput for different numbers of client versus CPU usage (kernelspace)

of 4.8 Gbps. The userspace version written in Python would however become CPU-bound at 700%, around 840 users.

This test was performed the stream scaling engine disabled.

6.3 Packet loss

Figures 6.3 and 6.4 indicate that the kernelspace implementation suffers from excessive packet loss. A likely reason for this behavior is the bursty nature of media streams with regard to bit rate.

When the userspace implementation receives a packet, it makes a system call for each client. These systemcalls cause context switches, and the scheduler lets other processes use the CPU between each call. Effectively, this spreads out the outgoing packets and prevents the network card buffers from overflowing. The kernelspace implementation on the other hand, pushes out up to n copies of the packet at the same time, almost guaranteeing to overflow network card buffers when proxying a 'busy' period of the stream.

As seen by the extremely low variance in loss numbers reported by the clients in the kernelspace test, we can deduce that our efforts to gain fairness by rotating the list of receivers works nicely. This test was also performed the stream scaling engine disabled.

6.4 Stream scaling

Figure 6.5 on page 49 shows the amount of data we attempted to send to each client, for different numbers of clients. As expected, amount of data delivered is reduced as the number of clients increase. This is due to the stream scaling engine adapting the

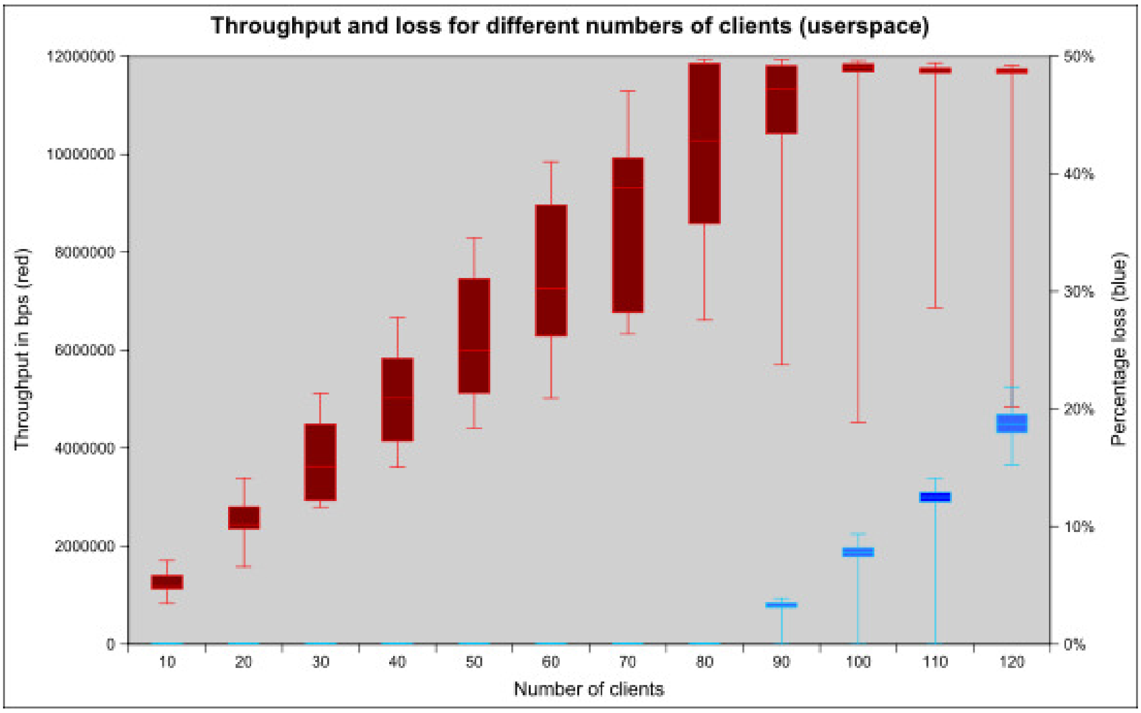


Figure 6.3: Throughput for different numbers of client versus packet loss (userspace)

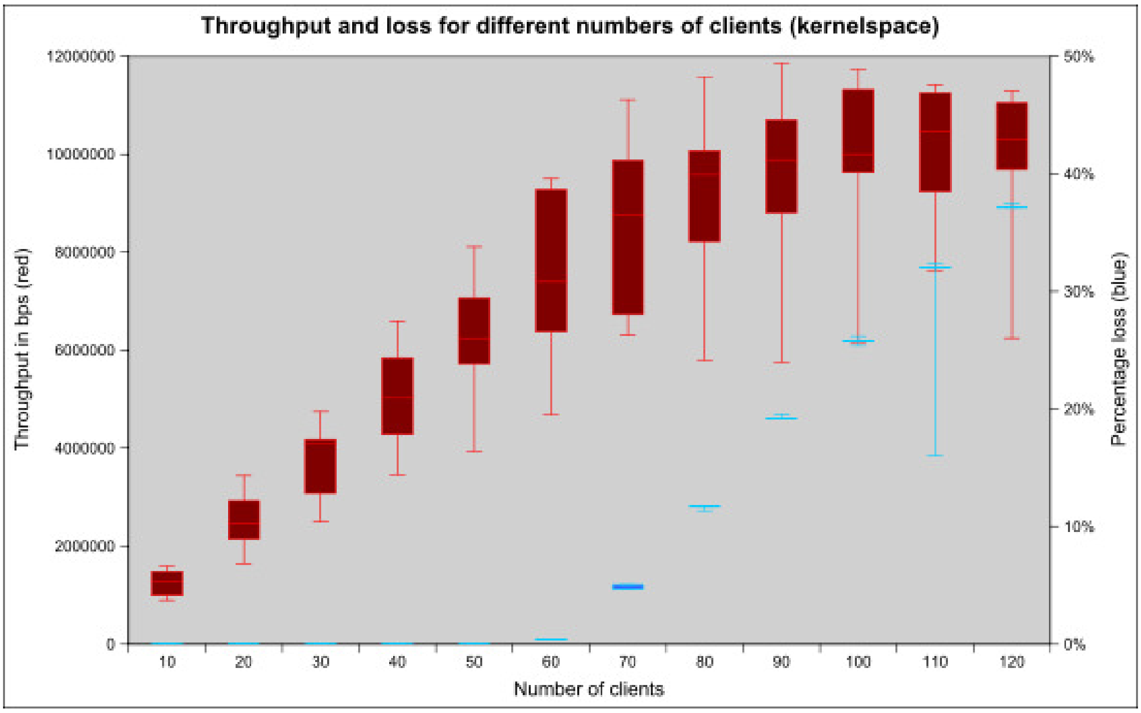


Figure 6.4: Throughput for different numbers of client versus packet loss (kernelspace)

stream according to loss, as reported to the client.

Figure 6.6 on the next page shows the **total** amount of data attempted sent. The pink line shows the linear development we could expect with no stream scaling measures involved, and the yellow line is the network saturation limit for our experiments[1]. One can clearly see how the total amount of data sent is affected by a congested or saturated network connection. However, the optimal result in this case would be a far lower than overshooting the available bandwidth by 40%, as seen for 160 users. Our approach to bandwidth estimation and stream adaption, while simple and cheap to implement, appears to be insufficient to solve this problem.

[1] Keeping in mind that the kernelspace implementation encounters packet loss at a far lesser bandwidth.

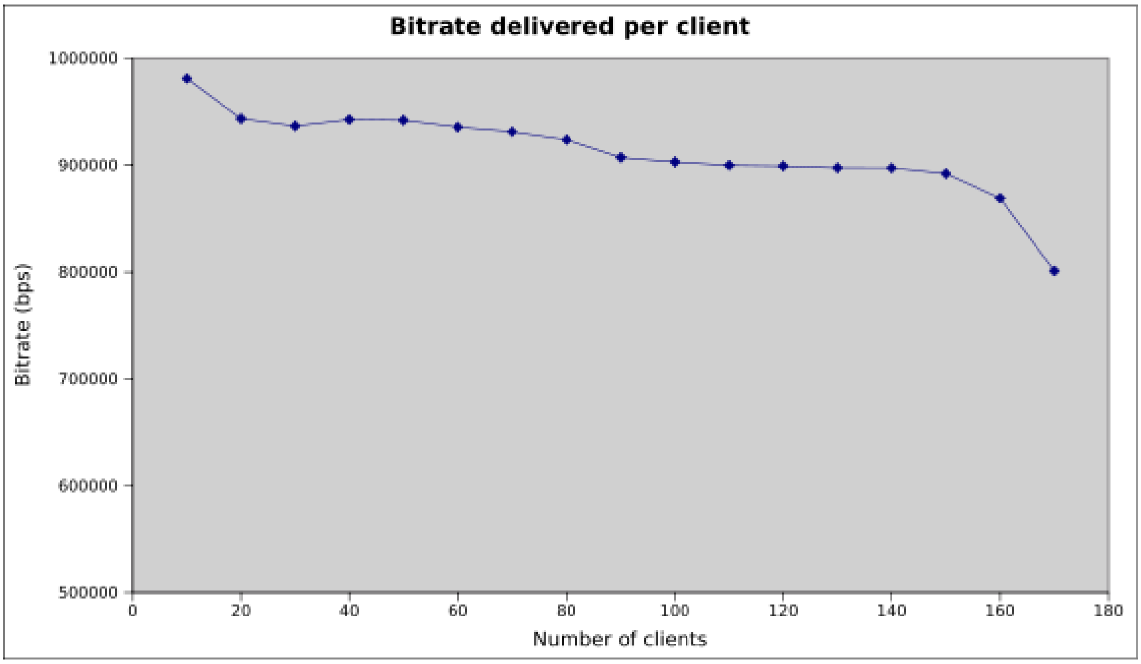


Figure 6.5: Bandwidth delivered per client

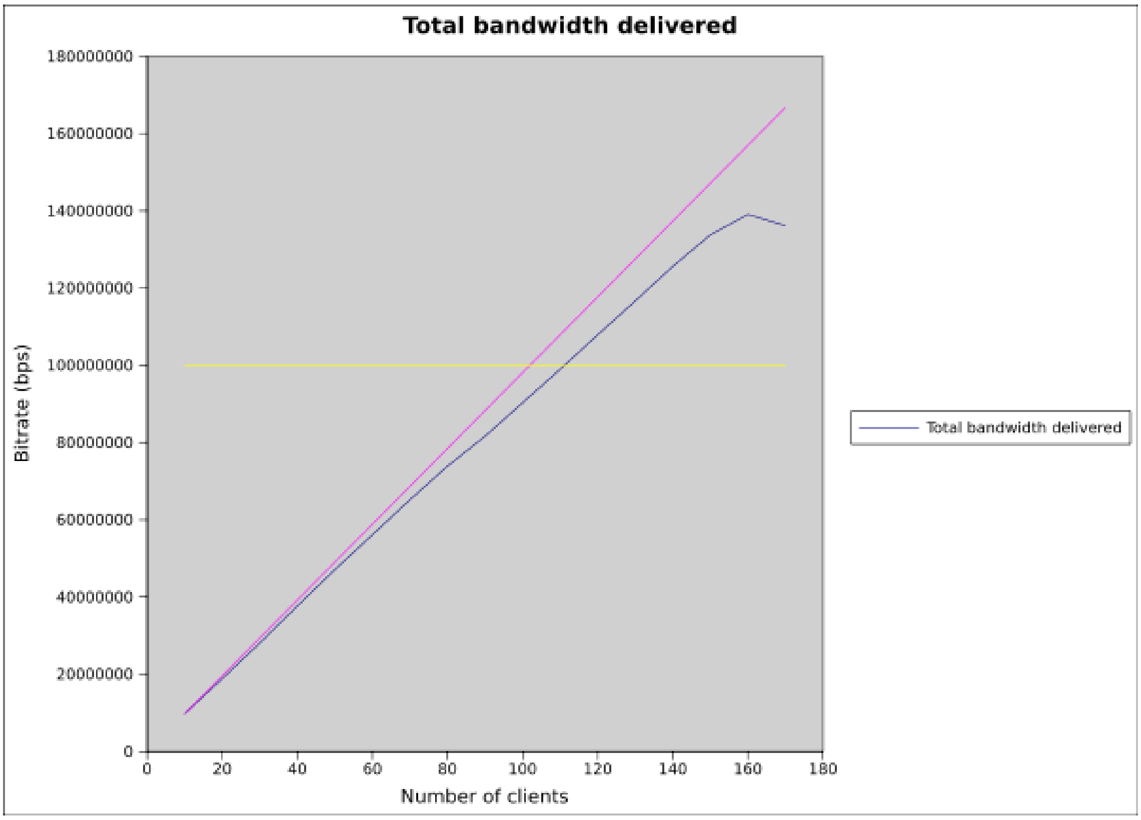


Figure 6.6: Bandwidth delivered total

Chapter 7

Conclusion

7.1 Summary

In the course of this thesis we have designed and implemented a split media streaming proxy, the main objective of which was to relay data contained in kernelspace in order to obtain reduced CPU usage. In addition, we designed and implemented an ad hoc adaptive stream scaling mechanism. Here we will briefly summarize the results of our work and the most important contributions it has brought forth. We will end this thesis with some ideas for further research.

7.2 Summary and contributions

As streaming media grows in both quality and viewer base, increasing bandwidth and CPU use follows. The centralized server architecture is giving ground to a hierarchical distribution with numerous proxies. Common media streaming proxies are implemented in userspace, and is subject to excessive data copying over the userspace↔kernelspace barrier, and this is causing unnecessary high CPU loads. We have implemented a split proxy for streaming media over RTSP/RTP for Linux, with the control path in userspace and the data path in kernel space as a Netfilter module. This has radically cut down on CPU consumption by removing superfluous buffer copy operations. This is a novel approach, with literature focusing primarily on caching media proxies and applications related to teleconferencing.

Furthermore, the wealth of devices that can receive streamed media calls customized versions of a media stream in order to cater to differing capabilities, be it bandwidth, frame rate, or resolution. We set out to create a simple adaptive stream scaling engine for shaping a stream in accordance with available bandwidth. It uses the layered bit streams created by SVC, and drops higher quality layers in response to packet loss as reported by RTCP. It does this without changing either client or server, and is therefore a very non-intrusive solution.

7.3 Future Work

While our approach shows great promise, we have observed several shortcomings that shows that the ideas presented in this thesis require some refinement.

Work deferring and/or internal buffering seems to be a requirement for the data path in kernelspace. As shown by testing, the straight forward approach provokes excessive packet loss and is unable to saturate the bandwidth of the provided network link. It would also be of great interest to investigate the impact of the zero-copy mechanism, and determine the relative impact of the zero-copy mechanism versus system call avoidance.

The stream scaling algorithm used is a very primitive one. As such, it has no regard to fairness for other traffic over the network link — and being UDP it is certain to smother any competing TCP stream if given the chance. This is, however, not a problem with the implementation but an intrinsic feature of UDP. While the stream scaling algorithm effectively reduces the amount of data the proxy attempts to send on a congested network, it will in these cases still experience random loss, and this will more than likely bring about a reduced experience for the media consumer. Ideas for further development include using a more sophisticated **Additive increase/multiplicative decrease** based rate-controlling scheme, in order to avoid over-saturating the network too much.

Bibliography

[1] Mehta. Behold the server farm (accessed 11/05/09). `http://money.cnn.com/2006/07/26/magazines/fortune/futureoftech_serverfarm.fortune/index.htm`, 2009.

[2] Schulzrinne et. al. Rfc 2326: Real time streaming protocol (rtsp), 1998.

[3] Schulzrinne et. al. Rfc 3550: Rtp: A transport protocol for real-time applications, 2003.

[4] div. H.120: Codecs for videoconferencing using primary digital group transmission. `http://www.itu.int/rec/T-REC-H.120-199303-I/en`, 2009.

[5] Schwartz et. al. Overview of the scalable video coding extension of the h.264/avc standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 17(9), 2007.

[6] Klemets et. al. Rfc 4425: Rtp payload format for video codec 1, 2006.

[7] Wenger et. al. Rfc 3984: Rtp payload format for h.264 video, 2005.

[8] Wang et. al. System and transport interface of svc. *IEEE Transactions on Circuits and Systems for Video Technology*, 17(9), 2007.

[9] Balk et. al. Adaptive video streaming: pre-encoded mpeg-4 with bandwidth scaling. *Computer Networks*, 2004.

[10] Pasquale Fall. Exploiting in-kernel data paths to improve i/o throughput and cpu availability. *USENIX*, 1993.

[11] Schwan Kong. Kstreams: Kernel support for efficient data streaming in proxy servers. *NOSSDAV*, 2005.

[12] Brustoloni. Interoperation of copy avoidance in network and file i/o. *INFOCOM*, 1999.

[13] Kangasharju Rejaie. Mocha: A quality adaptive multimedia proxy cache for internet streaming. *NOSSDAV*, 2001.

[14] Wang et. al. A network-conscious approach to end-to-end video delivery over wide area networks using proxy servers. *INFOCOM*, 1998.

[15] Wolf Kahmann. A proxy architecture for collaborative media streaming. *International Multimedia Conference*, 2001.

[16] Griwodz Halvorsen, Dalseng. Assessment of data path implementations for download and streaming. *DMS*, 2005.

[17] Tomshardware. Desktop cpu charts, q3/2008 (accessed 26/05/09). `http://www.tomshardware.com/charts/desktop-cpu-charts-q3-2008/Sandra-2008-Memory-Bandwidth,806.html`, 2009.

[18] Dag Wieers. dstat. `http://dag.wieers.com/home-made/dstat/`, 2009.

Appendix A

Acronyms

AVC	Advanced Video Coding
DID	Dependency ID
DMA	Direct Memory Access
FIFO	First-in-First-out
GSO	Gather Scatter I/O
HD	High Definition
HTTP	Hypertext Transfer Protocol
IPC	Inter-Process Communication
MTU	Maximum Transmission Unit
NAL	Network Abstraction Layer
NALU	NAL Unit
NIX	Norway Internet eXchange
QoS	Quality of Service

RTCP	RTP Control Protocol
RTP	Real-time Transport Protocol
RTSP	Real Time Streaming Protocol
SDP	(Session Description Protocol)
SD	Standard Definition
SNR	Signal-to-Noise Ratio
SVC	Scalable Video Coding
TLB	Translation Lookaside Buffer
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VBR	Variable Bitrate
VoD	Video-on-Demand

Appendix B

Source code

Complete source code is available at `http://bjornars.at.ifi.uio.no/thesis`.