UNIVERSITY OF OSLO
Department of Informatics

Evaluating the performance gains of specialization in a stream handler architecture.

Master Thesis

Kjell Andreas Solberg

Contents

1 Introduction **1**
- 1.1 Background and motivation . . . 1
- 1.2 Problem definition . . . 1
- 1.3 Outline . . . 2

2 Media streaming frameworks **3**
- 2.1 Introduction . . . 3
- 2.2 The stream handler architecture . . . 3
- 2.3 Existing frameworks . . . 5
  - 2.3.1 Infopipes . . . 5
  - 2.3.2 Komssys . . . 7
  - 2.3.3 Network-Integrated Multimedia Middleware . . . 10
  - 2.3.4 GStreamer . . . 12
- 2.4 Summary . . . 14

3 Media system **15**
- 3.1 Introduction . . . 15
- 3.2 MPlayer . . . 16
  - 3.2.1 Preliminary on the MPlayer code . . . 17
  - 3.2.2 Streaming with MPlayer . . . 17
  - 3.2.3 RTSP with Live555 . . . 20
  - 3.2.4 Changes made to MPlayer . . . 25
- 3.3 Komssys . . . 26
  - 3.3.1 The RTSP setup . . . 26
  - 3.3.2 From disk to net . . . 26
  - 3.3.3 Changes made to Komssys . . . 28
- 3.4 Linux . . . 29
  - 3.4.1 A new system call: sendfile_prefixed . . . 29
  - 3.4.2 Changes made to Linux . . . 31
- 3.5 Summary . . . 31

4 Time measuring tool **33**
- 4.1 Introduction . . . 33
- 4.2 Design . . . 33
  - 4.2.1 New system calls . . . 35
  - 4.2.2 Inserting time measure code into the kernel . . . 37

4.2.3 Retrieving a time stamp 40
4.2.4 The kernel control paths 41
4.3 Summary 42

5 Testing of proposed changes **43**
5.1 Introduction 43
5.2 Test setup 43
5.3 Test Layout 43
5.3.1 Testing three different graph managers 43
5.3.2 Testing cost of a context switch 44
5.3.3 Testing overhead of time measuring tool 44
5.4 Test results 45
5.4.1 Testing three different graph managers 45
5.4.2 Time used in kernel space 49
5.4.3 Estimating variability of MAD and mean 53
5.4.4 Overhead using time measure tool 54
5.5 Total performance 57
5.6 Evaluation 57
5.7 Summary 58

6 Conclusion and remaining challenges **59**
6.1 Conclusion 59
6.2 Remaining challenges 59

Appendices

A Concurrent background processes **60**
A.1 Processes in regular test 60
A.2 Processes in background noise tes 61

List of Figures

2.1 Data flow and terms in the stream handler architecture. 4
2.2 Principal interfaces of an Infopipe [1]. 6
2.3 Smart proxies used in Infopipes [1]. 7
2.4 The classes for making an object a stream handler in komssys [2]. 8
2.5 An example of connected stream handlers in Komssys [2]. 9
2.6 Synchronization in NMM [3]. 11
2.7 Session sharing in NMM. In (a), a query comes in to the registry service. In (b), the new session is running and is sharing two stream handlers with the other session [3]. 12

3.1 Streaming example using RTSP/RTP. 16
3.2 Call graph showing the functions used in the setup of the `stream_t` struct. . . 18
3.3 A call graph of the functions used in the setup of the `demuxer_t` struct. . . . 20
3.4 Collaboration diagram for MediaSession in Live555 [4]. 22
3.5 Back trace taken from Data Display Debugger (DDD) when streaming a mpg file from Live555 media server. Part A is related to the decoding of the video stream. Part B is related to decoding the audio stream. 25
3.6 RTSP Describe message. 28
3.7 User/kernel space interaction usingsendfile_prefixed() with TunedFilePrefixed-StreamerGM. 30
3.8 User/kernel space interaction using a combination of *read()* and *sendmsg()* system calls with FileStreamerGM. 30
3.9 User/kernel space interaction using a combination of *read()* and *sendmsg()* system calls with TunedFileStreamerGM. 31

4.1 Exceptions and interrupts both end up at the restore_all label [5]. 34
4.2 Execution of a Kprobe [6]. 38
4.3 Nested interrupts in kernel space [5]. 42

5.1 Plot of CPU cycles used with `FileStreamerGM` in user space. 46
5.2 Plot of CPU cycles used with `TunedFileStreamerGM` in user space. 46
5.3 Plot of CPU cycles used with `TunedFilePrefixedStreamerGM` in user space. 46
5.4 Plot of CPU cycles used with `FileStreamerGM` in kernel space. 50
5.5 Plot of CPU cycles used with `TunedFileStreamerGM` in kernel space. 50
5.6 Plot of CPU cycles used with `TunedFilePrefixedStreamerGM` in kernel space. 51
5.7 A histogram of the theta stars for the MAD. 54

5.8 The user space, kernel space and time measure tool overhead (in CPU cycles) for each measure for each graph manager. 56
5.9 The user space, kernel space and time measure tool overhead (in CPU cycles) for each measure added together in one column, for each graph manager. 56

List of Tables

3.1 Some important functions used to set up the stream_t struct. 18
3.2 Some important functions used when setting up the demuxer. 21
3.3 This table shows some important functions related to the interaction between Live555 and MPlayer. 24

4.1 Struct that is used to calculate statistics and time in a time measure. 36

5.1 Statistics of the number of CPU cycles used in user space from regular and noise test. 45
5.2 Statistics of the number of CPU cycles used in kernel space from regular and noise test. 49
5.3 95% confidence intervals for MAD and Mean measures in user space. 55
5.4 95% confidence intervals for MAD and mean measures in kernel space. 55
5.5 The medians of system calls, interrupts and page faults taken from the regular measures. 57

Chapter 1

Introduction

1.1 Background and motivation

As more and more media content is made available through the Internet, content such as radio, tv, movies and music, is streamed to users from media servers. The media servers must be able to serve a magnitude of users and therefore it is important that the media server application responsible of retrieving the content a user requests and sending it out to he or she, does this with as small processing effort possible. The less processing effort the server can use on each client it serves, the more clients it can serve. One possible architecture to build the media server application upon is module and component based called a stream handler architecture. This architecture relies heavily upon the use of virtual functions to allow new components be added without the need to rewrite existing code, and to plug arbitrary components together, old or new. But, with this ease of development of new components in the media server and the ease of changing existing compositions of components, there is an increased cost related to the use of virtual functions needed for components to communicate. If this architecture causes the server to support less concurrent users, then its use has to be considered and its ease of development has to be weighed against the cost of buying a new server.

1.2 Problem definition

A typical scenario is where a client, located on a different network than the media server, requests a video file located locally on the server. To serve this request, the media server, sequentially, reads the file from disk, packs the data into a transport protocol of some kind and sends it out on the net to the client. In the component based stream handler architecture, this process is split into three logical components; reading data, packing data and sending the data. The communication between these components are done using virtual functions. The problem we want to investigate is the cost of using the stream handler architecture and evaluate its performance against an alternative, stripped version, which uses no virtual functions. In the alternative version, the components that does the job of reading, packing and sending data to the client in the stream handler architecture, is merged together into a single component which then uses no virtual functions.

1.3 Outline

This thesis is organized as follows. In chapter 2 we present the stream handler architecture and four media frameworks that are based on the this architecture. In chapter 3, we take a look at a media system consisting of a media server and a media client. We present some changes and add-ons to this system and a new system call that is used to assemble network packets in kernel space and send it out on the network to a client. In chapter 4 we present a time measuring tool that we will use to test changes we have made to the media system, and in chapter 5 we will present and evaluate these results before we conclude our work in chapter 6.

Chapter 2

Media streaming frameworks

2.1 Introduction

The need for generic and reusable building blocks for building multimedia streaming applications has led to the idea of breaking down streaming applications into smaller units that can be reused. These units should be tasks that are typical and often used in such applications. This can be decoding, encoding, transcoding, presentation, analysis etc. Combined with interfaces that makes it easy to connect these processing units together, the application developer can write multimedia applications that adheres to a flow-based programming model, without the need of writing everything from scratch.

In this paper five kinds of such frameworks are presented: Infopipes [1], Komssys [2], Network-Integrated Multimedia Middleware [3] and GStreamer [7].

2.2 The stream handler architecture

The units introduced in the previous section are termed stream handlers or components. They are seen as objects and therefore built using an object oriented programming style. The stream handlers can usually be split into three categories which reflect their role in the application:

- A source stream handler that produces the data. This can be reading data from a file or from a live capture from a camera, or some other device.
- A sink stream handler which is responsible for sending the data stream to a file or sending the stream to a suitable device like a sound card and/ or a video card or sending data stream out on the net to a client.
- In between the source and the sink it is possible to have one or more processing or buffer stream handlers. The processing stream handler can be, as already mentioned, a decoder, encoder, transcoder, multiplexer, demultiplexer, filter etc. A buffer stream handler is only used for intermediate storage between two other stream handlers.

These three stream handler categories is presented in figure 2.1 together with some terms related to their interaction.

Since stream handlers are modeled as objects, most of them have properties/attributes that can be used to tune their behavior. This tuning can be done by the user to suit special needs, or it

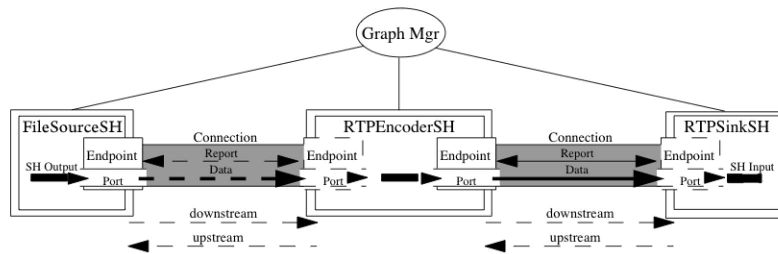


Figure 2.1: Data flow and terms in the stream handler architecture.

can be done dynamically by the stream handlers themselves while the application is streaming data. The latter is usually done by sending a report regarding the current state of the stream, and then the stream handlers that are affected is updated. This communication, combined with the media data flowing from source to sink, is the data flowing through the stream handlers. Unlike the media data that only flows in one direction, from source to sink, the communication between the stream handlers, called control information, can flow in both directions. The control information can be a notification to other stream handlers to change some of their attributes, or it can be information regarding the media stream, such as a notification to change the position in a video or audio stream.

To connect the stream handlers together, so the data can flow from one stream handler to the next, there has to be some kind of interface that dictates how data is delivered between two neighbouring stream handlers. This interface is called a port. The buffers and processing stream handlers have both an input and output port so the data can flow through them. It is also possible to have multiple input and/or output ports. The source and sink stream handlers have only one port, output port and input port respectively. For the data to flow from one stream handler to another, input and/or output ports must have mechanisms to deliver or retrieve data from a neighbouring stream handler. This is done with push and pull methods respectively. With these methods, media data can be put into a neighbouring downstream stream handler with a push method, or media data can be pulled out of a neighbouring upstream stream handler with a pull method.

The structure of all the connected stream handlers is called the composition and varies with the stream handlers used. If the stream handlers used only has one input port and/or output port, the composition is a pipe. When stream handlers with more than one input and output port are used, the structure is a graph or a tree, depending on how many sinks and sources are used:

- When the composition consists of one source and multiple sinks, the resulting structure is a tree with the source as root and the sinks as leaf nodes.
- When there are multiple sources and one sink, the structure is also a tree with the sink as root and the sources as leaf nodes.
- With only one source and one sink, the composition has a graph structure with branching in the middle and the branches meeting at the end. A demultiplexer, for example, is a stream handler that causes branching of the data flow.

The stream handlers are connected either at compile time, static connections, or at run time, dynamic connections.

2.3 Existing frameworks

In this section we will present four frameworks that are based on the stream handler architecture. These are Infopipes, Komssys, Network-Integrated Multimedia Middleware (NMM) and GStreamer.

2.3.1 Infopipes

Infopipes is a media streaming framework proposed by Koster et al. in [1]. In the following subsections we will see that this framework fits well with our generic stream handler architecture.

Infopipes architecture

Infopipes is a an abstraction, together with middleware and tools, for simplifying the task of constructing streaming applications. This is done by offering a rich set of stream handlers, which Koster et al. terms as infopipes, to the application developer. These stream handlers are the building blocks of the streaming application. Together they constitute a composition in which data and control information flow. Koster et al. uses infopipeline as the name for the composition. The setup of the composition is predefined and is done static at compile time.

Koster et al. use a plumbing analogy to simplify and capture the Infopipes vision: "just as a water-distribution system is built by connecting together pre-existing pipes, tees, valves and application-specific fixtures, so an information-flow system is built by connecting together predefined and application-specific Infopipes". For these stream handlers to communicate and data to flow through them, there is a communication interface between the components. Koster et al. uses the name port. There are two kinds of ports: outports in which data flows out of a stream handler, and inports where data flows into a stream handler. Each port is owned by exactly one stream handler, but a stream handler can have multiple input ports or output ports. Push and pull operations on these ports constitute the Infopipe's data interface. There is also a control interface which allows the stream handlers to exchange information and dynamically monitor and control their properties.

A vital property to the stream handler is that they are compositional. This means that the properties of a composition can be calculated from each individual stream handler constituting that composition. For example, the latency of a composition is computed by taking the sum of the latency for all stream handlers in that composition. Compositionality also requires that the connection between each stream handler has to be seamless. This means that the cost of connecting these objects together should be insignificant, and therefore the push and pull methods are seen to have no cost. In contrast, remote procedure calls or function calls that can block the caller may have high costs. But the Infopipe architecture does not allow such cost to be introduced automatically without encapsulating them into stream handlers. In this way their cost can be included in the composition cost calculation.

For an object to become a stream handler and be able join a composition, it has to implement the principal interfaces shown in figure 2.2. With these interfaces the user has the ability to create new compositional stream handlers from the already existing ones. To able these new complex stream handlers to be connected into the composition, they must have their own ports. These ports are called ForwardedPorts by Koster et al. and are in one-to-one correspondence,

Cloning	
clone	answers a disconnected copy of this Infopipe
Data	
pull	answers an item obtained from this Infopipe.
push: anItem	push an item into this pipe
Connection	
–>> aPortOrInfopipe	connect my Primary outport to aPortOrInfopipe
Port Access	
inPort	answers my Primary Inport
inPortAt: name	answers my named Inport
inPorts	answers a collection containg all of my inports
outPort	answers my primary Outport
outPortAt: name	answers my named Outport
outPorts	answers a collection containing all of my outports
nameOfInPort: anInPort	answers the name of anInPort
nameOfOutPort: anOutPort	answers the name of anOutPort
openInPorts	answers a collection containing all of my Inports that are not connected
openOutPorts	answers a collection containing all of my Outports that are not connected
Pipeline Access	
allConnectedInfoPipes	answers a collection containing all of the Infopipes in the same Infopipeline as myself.
inConnectedTo	answers a collection containing all of the Infopipes that are directly connected to my Inports
outConnectedTo	answers a collection containing all of the Infopipes that are directly connected to my Outports

Figure 2.2: Principal interfaces of an Infopipe [1].

but different from the open ports of the sub-stream handlers since ports can be owned by only one stream handler.

Control Interface

The control interface of a stream handler exposes and manages two sets of properties: properties regarding the stream handler itself and properties regarding the information flowing through the stream handler.

Information can pass from one stream handler to another in two ways: pull mode and push mode. In push mode the output port of the upstream component pushes data into an input port of the downstream component. In pull mode the downstream component invokes a pull method on the output port on the upstream component. The ports that invoke methods in other components are said to be positive, and the ones that execute the methods when invoked are said to be negative. For a composition to be well-formed, only ports with opposite polarities can be connected.

Netpipe

A netpipe is special kind of buffer that allows components in different address spaces connect to each other, but differs in the control interface as it reflects the properties of the network. The netpipe has an input port in one address space and an output port in another. This way the upstream neighbour is located in the same address space as the netpipe's input port and hence can do a seamless push of data into the netpipe. Equally the output port is located in the same address space as its downstream neighbour which then seamlessly pulls the data out of the netpipe. The Infopipe framework uses smart proxies (Koster and Kramp 2000) to implement netpipes. Smart proxies works as an extension of the server at the client side, allowing the server to control the network part of the composition, as shown in figure 2.3. But, instead of sending the complete code of the proxy for performing the task the server wants, it can instead take advantage of the fact that the client already has many primitive stream handlers that can do the job. It sends the blueprints for the proxy, and the client assembles it itself, or it sends a

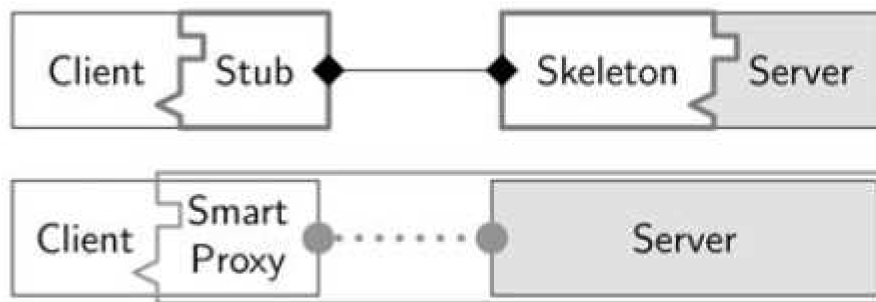


Figure 2.3: Smart proxies used in Infopipes [1].

small specialized stream handler if it is not already available from the stream handler library.

2.3.2 Komssys

Komssys is a media streaming framework that was proposed by Zink et al. in [2]. We will in the following section see that this framework fits well with our generic stream handler architecture.

Komssys architecture

The Komssys framework is targeted towards audio/video streaming over Real Time Streaming Protocol (RTSP) with Real-time Transport Protocol/Real-time Transport Control Protocol (RTP/RTCP) and the ability to handle multiple concurrent streams. It supports a wide range of distribution mechanisms such as combining multicast and unicast distribution and segmentation and reordering for efficient delivery.

The composition in Komssys is termed a graph by Zink et al. and is controlled by a graph manager. The stream handlers are connected through input and output ports which Zink et al. terms source endpoint and sink endpoint respectively. A stream handler can have multiple input and/or output ports, all depending on its purpose. The data that flows through these ports and their respective stream handlers are media data and report data, where the latter is used for in-band control of the stream handlers. Komssys also allows the stream handlers to use notifications to inform the composition manager that an event has occurred. This can for example be that a RTP packet arrives from an unknown source.

The creation of a composition in Komssys is made from well-known "blueprints" which connect the stream handlers that are needed for specific tasks. For each general task, Komssys has statically at compile time defined a composition manager that knows which stream handlers to use in the composition. For example, when Komssys shall stream a file from disk to a client, it creates a composition manager called `FileStreamerGM`. This connects three stream handlers, like in figure 2.1:

- One `FileSource` responsible for reading the file.
- One `RTPEncoderSH` responsible for creating data that shall be inserted into a Real-time Transport Protocol (RTP) packet.
- And one `RTPSinkSH` responsible for sending the RTP packet to the network.

The responsibilities for the composition manager is to set up and destroy the stream handlers, provide an interface towards the application and determining the interaction between stream handlers. It is able to split and merge the composition at run-time to handle new connections in response to, for example, user joins and leaves from multicast streams, or arrival of packets from unknown senders. This means that Komssys is able to dynamically connect new stream handlers into the composition at run time.

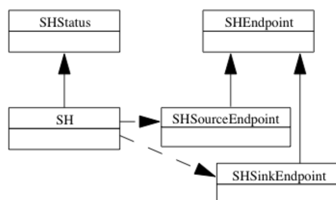


Figure 2.4: The classes for making an object a stream handler in komssys [2].

To make Komssys usable for third party developers, there exist basic classes, templates and interface definitions, which are displayed in figure 2.4, that provide the developers with stream handler functionality. To create a stream handler in Komssys, one has to create a object that is a sub class of SH. And for the new stream handler to be able to communicate with other stream handlers, it has to have a `SHEndpoint` object, either a `SHSinkEndpoint`, `SHSourceEndoint` or both depending on the type of stream handler. These classes are used to push or pull data to and from, respectively, the stream handler.

Control interface

Komssys, by enforcing new stream handlers to implement a report interface, has the ability to provide direct feedback in the opposite direction of the data path. This way RTCP feedback can be sent to a RTP packetizer without the involvement of the graph manager.

Three types of stream handlers

When dealing with concurrency, clocks are essential to make things work. In the Komssys framework a stream handler can specify whether it has its own clock or not, and if it requires a clock or not. These properties constitute that the stream handlers has to be grouped into some operation modes, so that the graph manager can order them appropriately. The operation modes are active, passive and through. If a stream handler is active it implements its own timer. This can be either a local timer or external. The latter meaning that the stream handler is observing some external activity, like the arrival of network packets. When an active stream handler acts as a source it pushes data downstream actively by calling a push function on the neighbouring downstream stream handler. If it is a sink it pulls data from its upstream neighbour. An active stream handler may also combine these to properties; pulling from an upstream stream handler and pushing the data into a downstream stream handler. Combining these properties of an active stream handler, Komssys does not allow two active stream handlers to connect directly.

The second mode of a stream handler is passive. This stream handler does not implement its own clock. If it acts as a source, its downstream neighbour pulls data from it. If it is a sink, data gets pushed to it. If it implements both of these capabilities then it becomes a buffer, and therefore must provide some buffering capabilities that suit the needs for the graph it is likely

to be included in. Such buffer capabilities are for example a threshold that allows the stream handler to notify the graph manager of over- and under runs of the buffer. As with the active stream handlers, one can not connect two passive stream handlers because then no data would be exchanged.

The third kind of operation mode of a stream handler is called “through”. These kinds of stream handler are meant to do tasks like packet duplication, on-the-fly transcoding or filtering. They do not have an own timer, but should not introduce buffer capabilities, like the passive mode, beyond what is necessary for their operation. The stream handlers in this operation mode must implement both the source and sink interface, and unlike stream handlers in the two other modes stream handlers in this mode can be connected in an arbitrary number.

A combination of stream handlers from the three modes described may look like the one in figure 2.5. An active stream handler upstream should push data into a through stream handler,

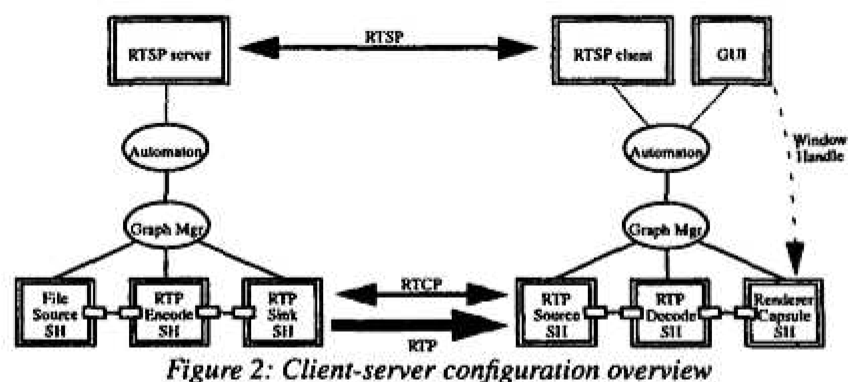


Figure 2.5: An example of connected stream handlers in Komssys [2].

which is followed by an arbitrary number of through stream handlers. The data should then flow into a passive stream handler, for example a buffer, which then a sink stream handler pulls data from.

Reconfiguration of the stream graph

Since the possibility of receiving several streams on one port is present when using RTP/RTCP over IP multicast, Komssys supports reconfiguration of the composition at run time. It may add or remove sub-compositions in response to users joining and leaving multicast streams.

A possible scenario where this feature of Komssys is useful is presented in [2]. Here, Komssys acts as proxy cache server using a write-through method and the user gets data from the origin server through the proxy cache. The proxy cache forwards the data to the client and writes it to disk as well. If at one point the client wishes to pause the stream, the trunk of the graph that sends data to the client has to be cut, while the trunk that stores the data has to be maintained. If at one point the client wishes to resume the now paused stream, a new graph has to be made which retrieves data from the cache.

2.3.3 Network-Integrated Multimedia Middleware

NMM is a media streaming framework targeted towards distributed multimedia devices. It is designed by Lohse et al. and is described in [3] and [8]. We will in the following sub-sections see that this framework fits well with our generic stream handler architecture.

NMM architecture

NMM is designed to access and control distributed multimedia devices. It uses proxies to allow distributed access to stream handlers. The stream handlers in NMM is called nodes by Lohse et al. and can be software components or hardware devices. For communications, the stream handlers have input and output ports, referred to by Lohse et al. as jacks, and are used to connect stream handlers into the composition which Lohse et al. calls the flow graph. The multimedia formats that are supported, the kind of data that can flow from source to sink, are specified by the ports. A stream handler is not restricted to having only one input and output port, it can have several, so each port is labeled with a "jack tag" which identifies it. Furthermore, ports can be dynamically duplicated to form a port group. This group then forwards outgoing messages from the stream handler to all of its ports.

The messaging system in NMM handles two types of messages. "Buffers" transport data and "events" transport arbitrary control information which can be used to control the stream handlers. NMM also include a registry service that allow stream handlers to be registered with its full description. This way suitable stream handlers can be queried for, instantiated if a match occur, and then returned. Queries are formatted as graph description, where the application describes the kind of composition it is looking for. Lohse et al. use the term flow graph to refer the our generic composition structure.

Control interface

The event messages in NMM can control the stream handlers in two ways. The stream handlers can create "in-stream" events that are sent between the stream handlers in the composition. Or the application that has instantiated the stream handlers, can send "out-of-band" events. This is mainly used to set parameters in the stream handlers.

The event system in NMM allows for dynamically adding and removing certain event handlers. It also allows listener objects to transparently register to be informed when certain events reaches a stream handler. Such events can for example be an "end-of-stream" event.

Synchronization

An important feature of NMM is the ability to have synchronized playback of several multimedia streams to a device. A handheld device can for example be streaming an audio and video stream from a DVD player. If the audio and video streams are separate streams, it is very important that those two streams are synchronized for the user to have a satisfactory playback experience. NMM achieves this by distinguishing between intra-stream and inter-stream synchronization. The former referring to the relations between data units in one stream, which can be frames in a video stream and the latter refers to the temporal relation among several streams, for example a video stream and an audio stream. The two kinds of synchronization can be seen in the figure 2.6.

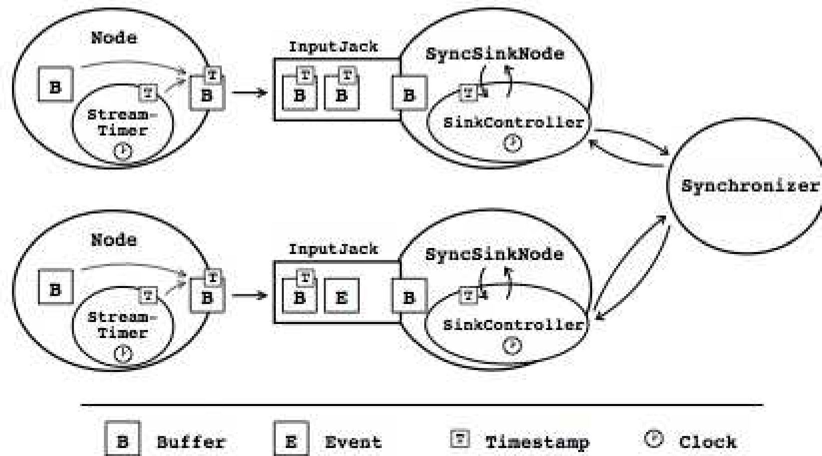


Figure 2.6: Synchronization in NMM [3].

The basis for the synchronization is a common source for timing information. A static clock object is located inside each address space, i.e. inside each source stream handler. Intra-stream synchronization is handled by time stamping each buffer, and then having the sink inspect each buffer that arrives to see if it should be presented, using a sinkController located inside the sink stream handler. Inter-stream synchronization is handled by a Synchronizer which each stream-sink, that should be synchronized, is connected to. The Synchronizer then tells all the connected sinks when to present their buffers in such way that the user experiences a synchronized playback. NMM does this by using what Lohse et al. refers to as theoretical latency. This is the highest latency of all the streams that should be synchronized.

Automatic session sharing and synchronized reconfiguration

In NMM there is a possibility to have streams, for example an audio and video stream, share parts of a composition. Therefore NMM uses a sharing policy on every stream handler that is registered in the registry service. Stream handlers can be marked as shared, exclusive and “exclusive and shared”, where the latter is to share a exclusively requested stream handler. There is also a possibility to combine, for example “exclusive or shared” where a shared reservation is chosen if an exclusive request fails.

When a composition of reserved and connected stream handlers is set up, it is stored as a session in the registry service. Shared stream handlers can then be reused in new compositions. For sessions, synchronization is achieved by having synchronizer object for each session. An example of session sharing can bee seen on the figure below. Here, the running session has chosen to share all the stream handlers, except those for rendering audio and video. If then another application wants to use a different audio stream, it queries the registry service for a setup as seen in figure 2.7 (a), and the composition is setup as seen in figure 2.7 (b).

Another interesting feature in NMM is the ability to migrate parts of an active composition to other devices in real time. And further, this is done seamlessly, i.e. the transition is not noticeable from the user’s perspective. One possible scenario is presented in [8]. Here, the user wants to play back media files stored on a mobile device. If there is no other system in a nearby proximity, the user’s mobile device does the decoding and playback itself. But as soon as a

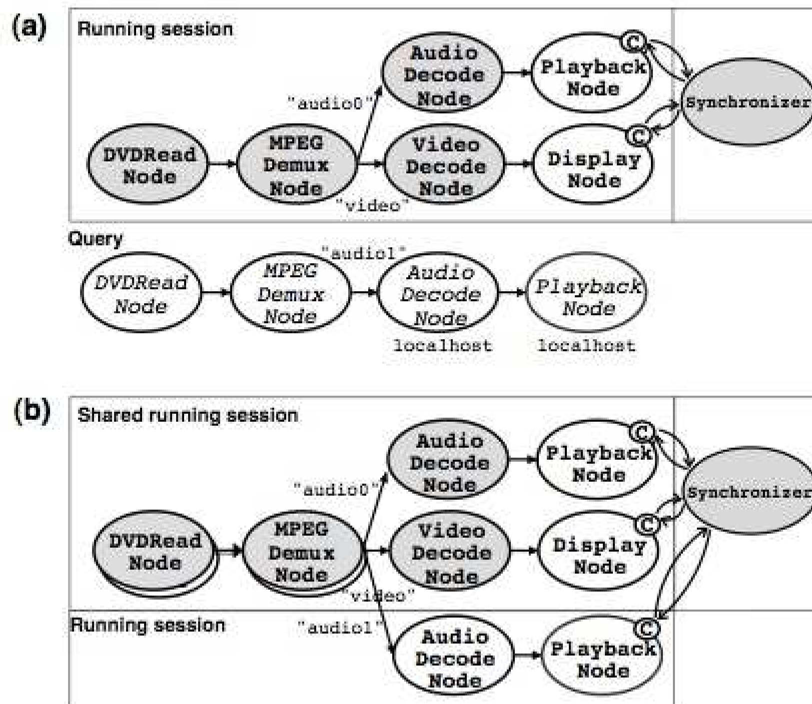


Figure 2.7: Session sharing in NMM. In (a), a query comes in to the registry service. In (b), the new session is running and is sharing two stream handlers with the other session [3].

stationary system with richer I/O capabilities (higher quality stereo output) comes into reach, it is desirable to have at least the audio playback directed to that system. If the stationary system also supports the process of decoding the selected media files, then it is also preferable to send that corresponding stream handler to the stationary system. This way the mobile device is able to save power by delegating some of the battery consuming processes. NMM is thus able to connect stream handlers dynamically at run time, but the initial composition is done at compile time.

The basic idea behind this migration of stream handlers is that the new parts of the current composition are configured alongside the running media processing. When the new parts have been synchronized, the old part of the composition is "cut loose". This seamless handover has to be performed in two steps: In step one the required parts from the current composition is instantiated and reconfigured. This newly created sub-composition is called a "slave graph". In the second step, data and control connections are transferred to the newly created "slave graph".

2.3.4 GStreamer

GStreamer is a streaming media framework project founded by Erik Walthinsen et al. In the following sections we will see that the framework described in [7] fits well with our own generic stream handler architecture.

GStreamer architecture

GStreamer is implemented in C, but uses the GObject programming model and therefore adheres to the object oriented programming paradigm. The stream handlers are referred to as elements and it uses ports to connect the stream handlers together. In GStreamer, the input and output port are referred to as sink and source pad respectively. Links and data flow are negotiated between the stream handlers ports before data can start to flow. It is the ports that describe the type of data that can flow or currently flows through the stream handler it is connected to, and links can only be made between stream handlers when their ports data type match. The composition is referred to as a pipeline in GStreamer.

GStreamer allows us to create new stream handlers by deriving the new classes from the stream handler class. But it also offers a way to create new stream handlers based on already existing stream handlers in the GStreamer library. The user is able to group together multiple stream handlers and mask it as a single stream handler using a container object which can hold a collection of stream handlers, called bin. This bin is a subclass of stream handler which makes it easy to mask the collection of stream handlers as just a single stream handler. The bin makes it easy to control the states of all its contained stream handlers; changing the state of the bin, causes all of its children to change state as well. The composition in GStreamer is a special type of the bin object, a generic bin called pipeline. This object allows for scheduling of its containing stream handlers using threads.

The data produced by the source stream handler and sent down the composition is wrapped in entities called buffers. They consist, amongst others, of: a pointer to a piece of memory, the size of memory, a time stamp for the buffer and a refcount that indicates how many stream handlers are using the buffer. The refcount is used to decide when to destroy the buffer, when it is zero it is destroyed.

Control Interface

To allow messages from the composition to be delivered to the application, GStreamer implements a system called a bus. The bus forwards messages from the composition threads to the application. Every composition contains a bus, so the user only have to attach a message handler to the bus to be able to listen for specific messages. All the messages have a source and type, so it is possible to identify which stream handler sent the message.

Flowing alongside with buffers in the pipeline is events which contains the control information, such as seeking information and end-of-stream notifiers. Events are sent both up- and downstream. Downstream events are used to notify stream handlers of stream states. Upstream events are used for application-stream handler interaction plus event-event interaction to request change in stream state, for example seeks.

Autoplugging

Autoplugging is a feature in GStreamer that allow compositions to be created in response to the type of media stream and available stream handlers in the GStreamer library. This means that GStreamer has the ability to create compositions dynamically at run time. To accomplish this, GStreamer forces all stream handlers to associate a Multipurpose Internet Mail Extensions-type (MIME-type) to its source and sink ports when it is loaded into the stream handler library. This way it can select the most suitable stream handlers that fit a given stream type it receives.

To find the MIME-type of an incoming stream, GStreamer uses a concept called typefinding which basically involves the pipeline object sending the incoming stream to specialized stream handlers called typefind which try to identify the MIME-type. If no type is found, it emits an error signal and the processing stops. If the type of the data stream is found, the GStreamer registry will be used to select the stream handlers that is suited for processing the media type.

2.4 Summary

We have looked at five media streaming frameworks in this chapter and have seen they all fit well with our generic stream handler architecture. The main difference lies in what these frameworks are used for and therefore they have implemented special features that are unique for that particular framework. GStreamer is a very general framework that can be used to create many types of applications. It can dynamically create a composition in response to a stream type, while the others have statically at compile created their composition. This makes GStreamer very easy to use. NMM has the ability to share stream handlers in a composition and to migrate parts of a composition to other devices, from for example a Personal Digital Assistant (PDA) to a desktop computer, which makes NMM suitable in mobile environments. Infopipes is also a general framework, but offers a specialized stream handler to take care of networking using smart proxies, which makes networking transparent.

Komssys is implemented as a media server that streams media content using RTSP, and can also be used as a proxy cache server. It differs from the other frameworks by being a server framework, while the others are frameworks that can be used to create general media streaming applications.

In the next chapter, we will look at how we can use Komssys together with a media player to stream media over a network using RTSP.

Chapter 3

Media system

3.1 Introduction

To watch a movie or listen to audio, stored locally or remote, on a computer, we have to have a media player. The purpose of the media player is to read the collection of zeroes and ones that make up the file we want to play, demultiplex the stream of data if it is multiplexed, decode the stream(s) that comes from an eventual demultiplexing and then send the decoded data to their respective output, video to screen and audio to the speakers.

When a data source is multiplexed it consist of two or more data sources, e.g. audio and video. The reason for multiplexing data together is to make it less complicated to save, for example, movies to disk, as we do not have to store the video data in one file and the audio data in another file. Demultiplexing this movie file in the media player retrieves the two data streams that have been interleaved on a single file when they were multiplexed.

When we want to watch a movie or listen to a song that is located on a different computer than the one we are sitting on, termed as a remote computer, the file is "streamed" from the remote computer. The media player has to support receiving data from the network and strip it from eventual protocol headers that have been added to it before starting the demultiplexing/decoding process described above.

The remote machine, from where we stream the data, has usually got a media server installed which is responsible for delivering media data to the user who requests it. A media server can support multiple connections to it, some may support transcoding files on the fly, which means re-coding e.g. a DivX encoded movie to Xvid encoding before sending it to the client, support different streaming protocols over which the data is delivered and so on. For each media file ,available through the media server, there is usually a Session Description Protocol (SDP) file associated with it. SDP is used for describing the media session initiation such as the name of the media, transport address, bandwidth information etc. [9].

An example of a media server streaming a movie over RTSP can be seen on figure 3.1. RTSP is used to set up and control the state of the media session between the server and the client, and RTP is used as a transport protocol to send the requested data to the client.

In this chapter we will take a closer look at the two components involved when playing streamed media content over a network, namely a media player and media server. In particular, we will look at MPlayer and Komssys, respectively. We have chosen to use Komssys since this is a server that follows the stream handler architecture, and we had access to the source code. We will look at the initialization of the data stream and demultiplexing structures in MPlayer,

and the initialization of a new file request in Komssys over the RTP/RTSP and till the first data packet has been sent to the client.

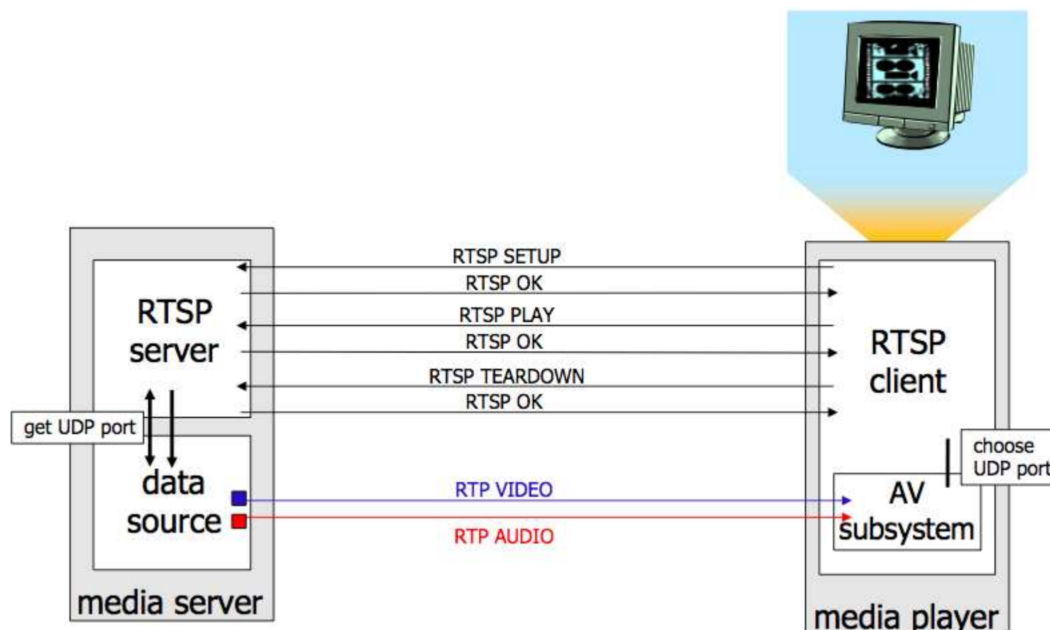


Figure 3.1: Streaming example using RTSP/RTP.

3.2 MPlayer

In this section we will take a closer look at MPlayer [10]. MPlayer is a free and open source media player available to most major operating systems today such as Unix-like systems, Microsoft Windows and OS X.

It supports many popular video and audio formats used today which includes video formats like H.263, Theora, H.264/Moving Picture Experts Group version 4 (MPEG-4) Advanced Video Coding (AVC), MPEG-1, MPEG-2, Windows Media Video (WMV), and audio formats like Advanced Audio Coding (AAC), Dolby Digital (AC3), MPEG-1 Audio Layer 3 (MP3), Vorbis and Windows Media Audio (WMA) [10]. MPlayer also supports streaming via Hyper Text Transfer Protocol/File Transfer Protocol (HTTP/FTP), RTP/RTSP, Microsoft Media Services/Microsoft Media Services Over Tcp (MMS/MMST), Message Processing Subscriber Terminal (MPST), Session Description Protocol (SDP). It supports data sources such as tv, radio, dvb, local file etc. [10].

In the following, we will take a closer look on how MPlayer initializes a data stream that shall be played, some of the data structures it uses and which files are used. Then we will describe how MPlayer accomplishes the task of streaming via RTP/RTSP using the open source library Live555 [4].

3.2.1 Preliminary on the MPlayer code

The main file in MPlayer, mplayer.c is divided into logical modules, code blocks, that describe in a concise way what is being done. The current module is printed out as debug information if MPlayer crashes, making it easier to locate which code block that caused the crash. It also makes it easier to refer to MPlayer code when trying to explain what goes on, and is how the code in mplayer.c is referred to in the following sections.

The data is termed "stream" by MPlayer and is encapsulated in a struct called `stream_t`. This struct is defined in stream/stream.h. It is one of two major structures used in MPlayer that we will look at. The other one is the `demuxer_t` struct defined in libmpdemux/demuxer.h. Together with some helper structs defined in the same files as `stream_t` and `demuxer_t`, these structs help MPlayer retrieve data, do seeks, resets and other data stream related operations (functions defined in stream/stream.c), plus header processing for different video and audio formats and demultiplexing before sending the data on to a decoder (functionality that concerns the demuxer).

Two helper structs which MPlayer use when trying to find suitable stream and demuxer structs are `stream_info_t` and `demuxer_desc_t`, respectively. The `stream_info_t` struct holds information about a corresponding stream type, the protocol names that stream can handle (a stream can handle more than one protocol) and a pointer to a function that sets up the correct `stream_t` struct. The `demuxer_desc_t` struct has the same purpose when MPlayer is trying to find a suitable demuxer for a given file or streaming protocol. What differs is that `demuxer_desc_t` has more function pointers that help the demuxer to check if the file can be demuxed with the current demuxer and fill the data buffer, plus some optional functions that can be set.

Each `stream_info_t` struct is set up at the end of the file that handles the stream handling for a specific stream type. The `stream_info_t` struct for a Digital Video Broadcasting (DVB) for example is set up in stream_dvb.c. All the `stream_info_t` structs are then made available through an array called `auto_open_streams` which is initialized at the top of stream/stream.c. The same code structure is also applied to the `demuxer_desc_t` structs which are set up in their corresponding demuxer files (located in libmpdemux) and then made available in libmpdemux/demuxer.c through an array called `demuxer_list`.

3.2.2 Streaming with MPlayer

MPlayer starts at the main function located in mplayer.c. It starts by initializing variables and properties, checking if a Graphical User Interface (GUI) is present and other init-related procedures before we come to the "open_stream" module. Here, the demux and stream structs, type `demuxer_t` and `stream_t` respectively, is initialized to NULL. Then, *open_stream()* (see table 3.1) is called and the return value is put into stream.

Initializing the stream struct

open_stream() sets the `file_format` variable, sent in as an argument, to UNKNOWN-STREAM_TYPE if we are not trying to open a play list. It then calls *open_stream_full()* (see table 3.1) which does the stream setup, and is implemented in stream/stream.c. This function runs through all the `stream_info_t` structs in `auto_open_streams`, looking for a stream type that fits the kind of medium the user tries to open. When a stream type that

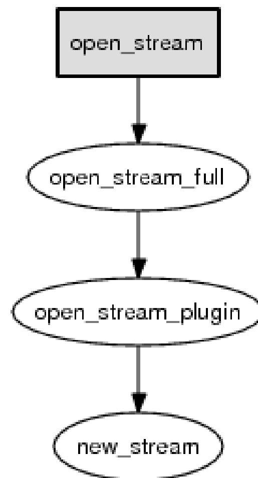


Figure 3.2: Call graph showing the functions used in the setup of the `stream_t` struct.

Function:	*Parameters:*	*Source file:*
stream_t * **open_stream()**	char *filename, char **options, int *file_format	stream/open.c
stream_t * **open_stream_full()**	char *filename, int mode, char **options, int *file_format	stream/stream.c
stream_t * **open_stream_plugin()**	stream_info_t *sinfo, char *filename, int mode, char **options, int *file_format, int *ret	stream/stream.c

Table 3.1: Some important functions used to set up the stream_t struct.

can handle the medium is found, *open_stream_plugin()* (see table 3.1) is called. This function handles the setup of a new `stream_t` struct by first allocating space for it and then sending it into the open function pointer in the `stream_info_t` struct. When the `stream_t` struct has been set up it is returned to the caller, all the way back to mplayer.c. If no supported stream type is found to match the medium the user tries to open, MPlayer terminates. A small call graph of the functions used and who calls who can be seen in figure 3.2.

Initializing the demuxer struct

After the `stream_t` struct has been set up, we eventually enter the "demux_open" module which handles the setup of the demuxer. The most important functions used to set up the `demuxer_t` struct, are listed in table 3.2. *demux_open()* is responsible for initializing the `demuxer_t` struct. The setup follows the number ordering on figure 3.3, and is described in

the following.

1 The first thing that is done, is to check whether the user has provided a demuxer name as a command line argument. If this is true, the demuxer type to use is retrieved by calling *get_demuxer_type_from_name()*. If the user has not specified which demuxer to use or the given demuxer name is not valid, the function returns DEMUXER_TYPE_UNKNOWN which is assigned to the variable `demuxer_type`.

2 After these checks have been made for an audio, video and sub[1] demuxer, the presence of separate audio and sub streams are checked, and if existent, *open_stream()* is called for each of these.

3 The next function that is called is *demux_open_stream()*. This function returns a new `demuxer_t` struct. The type of `demuxer_t` struct returned depends on the `demuxer_type` variable and the `file_format` parameter (which was set in the initialization of the `stream_t` struct). `demuxer_type` takes precedence over the `file_format`, so if this variable is different from DEMUXER_TYPE_UNKNOWN this kind of `demuxer_t` is created. If it not, the demuxer is found using the `file_format`.

4 Either way, the `demuxer_desc_t` struct is retrieved by calling *get_demuxer_-desc_from_type()*.

5 If it is found, a new `demuxer_t` struct is created by calling *new_demuxer()*. This function allocates space for the new struct, initializes the fields, sets the correct, recently created, `demuxer_desc_t` struct for the demuxer and creates a pointer to the `stream_t` struct, allowing the demuxer to retrieve data from the incoming data stream. The latter is done by setting the stream field in the demuxer to the `stream_t` struct created in the “open_stream” module.

6 When we return from *new_demuxer()*, the `demuxer_desc_t`'s *check_file()* function pointer is called if it has been set in definition of the `demuxer_desc_t` struct, to check if the functions found in the `demuxer_desc_t` struct can demux the file provided by the user. If so, a call to `demuxer_desc_t`'s *open()* function pointer with the new demuxer as argument is executed. This open function sets up the newly created demuxer. If this call succeeds, we jump to the label `dmx_open`, located at the bottom of *demux_open_stream()*, where some video specific parameters are set in the demuxer struct and then the demuxer is returned.

7 It is also a possibility that the `file_format` parameter does not give any useful information, in the case it has been set to STREAM_UNSUPPORTED. This happens for example when one tries to stream media using RTSP with MPlayer. *demux_open_stream()* is still called but the `demuxer_desc_t` struct is found using a different method. One of these methods is to check the file name of the file we are trying to open with *demuxer_type_by_filename()*

[1] A short for subtext.

8 When we return from the call to *demux_open_stream()*, the returned demuxer is checked for validity. If it is not NULL various if-tests are performed to see if we also have to create an audio demuxer and sub demuxer. Then the function returns either the video demuxer we got from *demux_open_stream()* or it returns a new demuxer, created with a call to *new_demuxers_demuxer()*. This function encapsulates three `demuxer_t` structs inside a locally defined struct and puts this inside the priv field of a new allocated `demuxer_t` struct. Among the other variables that are set are the type and `file_format` fields which are both set to DEMUXER_TYPE_DEMUXERS, and the`demuxer_desc_t` field which is set to a `demuxer_desc_t` struct defined in demux_demuxers.c called `demuxer_desc_demuxers`.

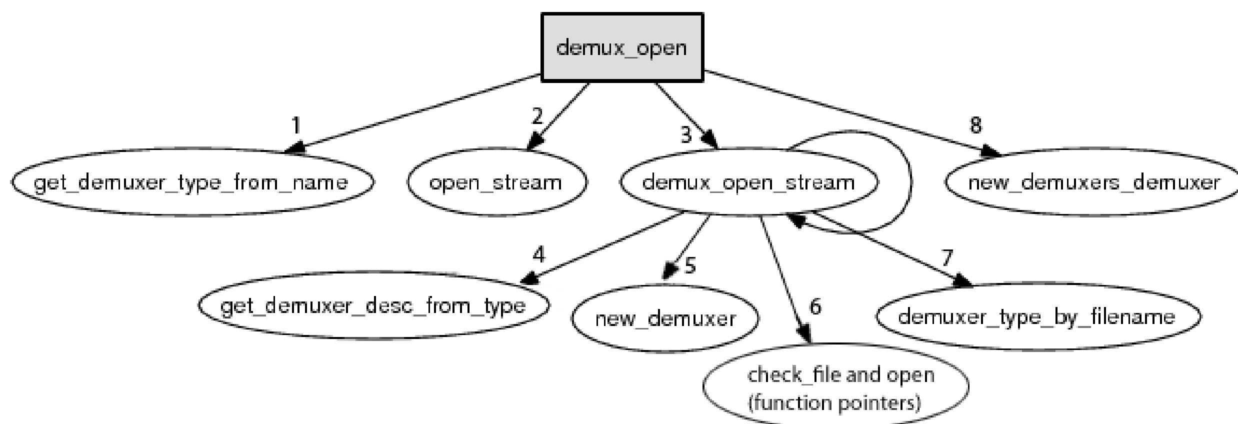


Figure 3.3: A call graph of the functions used in the setup of the `demuxer_t` struct.

3.2.3 RTSP with Live555

When MPlayer tries to stream videos using the RTSP protocol, it uses the Live555 open source library [4]. From the libmpdemux/demux_rtp.cpp file in the MPlayer source tree, Live555 code gets linked in to handle the RTSP setup and tear down together with the delivering of the payload of each incoming RTP packet to MPlayer (omitting the libmpdemux/ directory prefix from here on).

Live555 setup in MPlayer

The initialization of Live555's RTSP and RTP handling code is done in demux_rtp.cpp. This is the only C++ file in the MPlayer source together with demux_rtp_codecs.rtp, which contains helper functions for demux_rtp.cpp. demux_rtp.cpp acts as a communication link between Live555's C++ code and MPlayer's C code. *demux_open_rtp()* (see table 3.3) is linked to the Live555 `demuxer_desc_t`'s (`demuxer_desc_rtp`) *open()* function pointer and therefore executed when the `demuxer_t` struct is set up.

When *demux_open_rtp()* is called from *demux_open_stream()*, a TaskScheduler and an UsageEnvrionment object from the Live555 library is created. "The 'UsageEnvironment' and 'TaskScheduler classes are used for scheduling deferred events, for assigning handlers for

Function:	*Parameters:*	*Source file:*
demuxer_t * **demux_open()**	stream_t *vs, int file_format, int audio_id, int video_id, int dvdsub_id, char *filename	libmpdemux/demuxer.c
static demuxer_t * **demux_open_stream()**	stream_t *vs, int file_format, int audio_id, int video_id, int dvdsub_id, char *filename	libmpdemux/demuxer.c
int **get_demuxer_ type_from_name()**	char *demuxer_name, int *force	libmpdemux/demuxer.c
static demuxer_desc_t * **get_demuxer_ desc_from_type()**	int file_format	libmpdemux/demuxer.c
demuxer_t * **new_demuxer()**	stream_t *stream, int type, int a_id, int v_id, int s_id, char *filename	libmpdemux/demuxer.c
demuxer_t * **new_demuxers_demuxer()**	demuxer_t *vd, demuxer_t *ad, demuxer_t *sd	libmpdemux/demux_demuxers.c

Table 3.2: Some important functions used when setting up the demuxer.

asynchronous read events, and for outputting error/warning messages" [4]. All the Live555 files mentioned here is located in the liveMedia/ directory of the Live555 source tree if not a different location is mentioned.

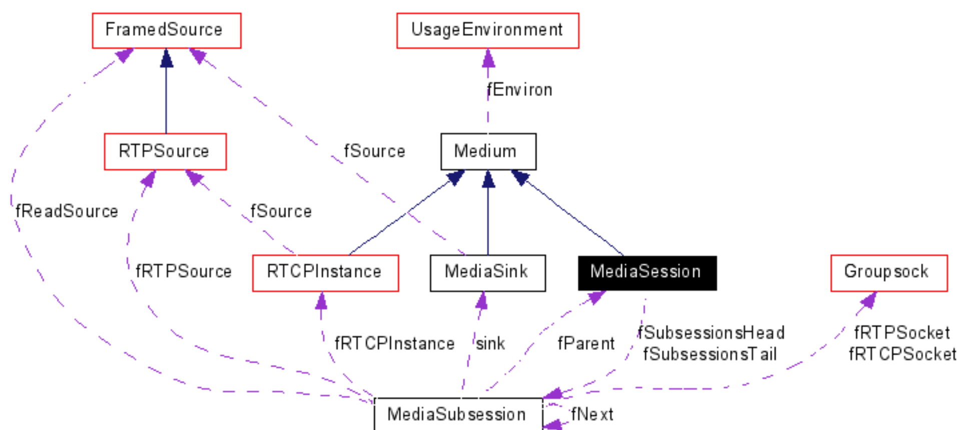


Figure 3.4: Collaboration diagram for MediaSession in Live555 [4].

For handling sending and receiving RTSP messages to and from the server, Live555 offers a `RTSPClient` object, implemented in RTSPClient.cpp. This object is created in *demux_open_rtp()* when the user only has provided MPlayer with a RTSP Uniform Resource Locator (URL). The URL together with the `RTSPClient` object is sent into *openURL_rtsp()*, a local function in demux_rtp.cpp. Here, the SDP description is retrieved from the server by calling either the function *inviteWithPassword()* if a network username is provided, or *invite()* if not. Both functions are implemented in RTSPClient.cpp. The information about the session taken from the SDP description, is used to create a `MediaSession` object which holds general information about the session, mainly things found in the SDP description. This MediaSession is created by calling *MediaSession::createNew()* which then instantiates a new `Mediasession` object and calls *MediaSession::initializeWithSDP()* with the new `MediaSession` object as argument. Here, the SDP description file is parsed and the information extracted and put into member variables. For every "m=" option, which describe a stream name and transport address [9], a `MediaSubsession` object is created and appended to a list accessed with the variables `fSubsessionsTail` and `fSubsessionsHead` defined in include/MediaSubsession.hh. Each `MediaSubsession` object encapsulates the stream and its properties. A collaboration diagram of `MediaSession` and `MediaSubSessoin` can be seen on figure 3.4. We also see members and inheritance of the two classes.

When the *MediaSession::initializeWithSDP()* function is done parsing the SDP description, we return to *MediaSession::createNew()* which return the initialized `MediaSession` object. Back in demux_rtp.cpp, a `RTPState` struct defined in demux_rtp.cpp is created to store the RTSP state just created. This struct stores the SDP description, `MediaSession` object and `RTSPClient` object plus initializes other fields. This struct is then stored in a void pointer field, in the `demuxer_t` struct that is being set up, called `priv`. The next thing that is done is to initiate every `MediaSubsession` contained in the `MediaSession` object by iterating through them. For each `MediaSubsession`:

- It is checked whether the sub-session encapsulates an audio or video stream. This is

so a `desiredReceiveBufferSize` variable can be set correctly. If it is video, the `desiredReceiveBufferSize` is set higher.

- The `MediaSubsession`'s client RTSP port number is set to a pre-defined number, defined in the MPlayer source code in stream/libRTSP/RTSP_rtp.c. In current status it is always 0, so it is the kernel's task to find a port number to use.

- *MediaSubsession::initiate()* is called. To keep it short, this function creates one socket for RTP and one socket for RTCP data. For reading data out of the RTP socket and process it, `MediaSubsession` has a `fReadSource` member of type `FramedSource` which is an abstract class. For each stream type, e.g., MPEG-2 transport, QuickTime, H.261 etc. there are subclasses of the `FramedSource` class that handle these special formats, with respect to their headers. The RTP headers is also stripped from the data and analyzed. For handling RTCP messages, `MediaSubsession` has a `RTCPInstance` member, `fRTCPInstance`. `RTCPInstance` is implemented in RTCP.cpp. This class handles the RTCP messages and RTP statistics for the RTSP session.

- The receive buffer size is increased for the RTP socket by calling *increaseReceive-BufferTo()* with the `desiredReceiveBufferSize` variable as one of the arguments. *increaseReceiveBuffer()* is located in groupSock/GroupSockHelper.cc (from the root directory in Live555).

- The last thing that is done is to send a RTSP setup message to the server to initiate the RTSP session. This is done with the *RTSPClient::setupMediaSubsession()*.

When all the MediaSubsessions have been iterated through, the RTSP play message is sent for the whole session by calling *RTSPClient::playMediaSession()*. This sends the RTSP "PLAY" message for all sub-sessions to the server, which after sending an "OK" response starts streaming the requested media. After the `MediaSession` and `MediaSubsession(s)` have been set up, a `ReadBufferQueue` object is created for each `MediaSubsession`. The `ReadBufferQueue` is implemented in demux_rtp.cpp and represents input data for each stream. If the `MediaSubsession` holds an audio stream, *rtpCodecInitialize_audio()* is called. This function is implemented in demux_rtp_codec.cpp and sets audio codec-specific parameters. *rtpCodecInitialize_video()* is called if the stream is a video stream and found in the same file as the audio version. This function also sets a flag if the stream is a multiplexed video and audio stream, such as a MPEG system stream.

The last thing that is checked, if the streams were set up correctly, is, if we are receiving a single stream, if it is a multiplexed audio and video stream. When it is multiplexed, a new demuxer is created with the *demux_open()* function described in section 3.2.2. This new demuxer is then wrapped inside a new demuxer using the *new_demuxer_demuxers()* function, also mentioned in 3.2.2, and then returned. The "old" demuxer just set up as described above is preserved by creating a new stream with the *new_ds_stream()* function in stream/stream.c. This function creates a new stream and sets the stream type to STREAMTYPE_DS. This means that the stream is coming from a demuxer, namely the RTP demuxer. It also saves the video `demux_stream_t` struct from the old `demuxer_t` struct that has been set up. This way the "old" demuxer is not lost when a new is created since the `demux_stream_t` struct has a reference to the parent demuxer. We will take a closer look at this structure in section 3.2.3.

Function:	*Parameters:*	*Source file:*
static char * **openURL_RTSP()**	RTSPClient *client, char const *url	libmpdemux/demux_rtp.cpp
demuxer_t * **demux_open_rtp()**	demuxer_t *demuxer	libmpdemux/demux_rtp.cpp
static demux_packet_t * **getBuffer()**	demuxer_t *demuxer, demux_stream_t *ds, Boolean mustGetNewData, float &ptsBehind	libmpdemux/demux_rtp.cpp
int **demux_rtp_fill_buffer()**	demuxer_t *demuxer, demux_stream_t *ds	
void **demux_close_rtp()**	demuxer_t *demuxer	libmpdemux/demux_rtp.cpp

Table 3.3: This table shows some important functions related to the interaction between Live555 and MPlayer.

Streaming with Live555 in MPlayer

After the RTP demuxer has set up the `MediaSession`, Live555 starts receiving data from the media server and MPlayer goes through the rest of its initialization modules, before it starts playing the data stream(s). The functions called before entering the interface functions towards Live555 in demux_rtp.cpp vary in response to the format of the stream. We will use examples from streaming a MPEG file through Live555 media server [4]. The file is split into two separate streams, one for audio and one for video, and therefore there are two different call sequences, but the functions used to retrieve data from Live555 are the same. A trace back of the functions called are shown in figure 3.5.

From the fourth call (#3 in both figures) and down the function calls are the same. The *ds_fill_buffer()* function uses a struct not discussed in previous sections, the `demux_stream_t` structure. While the `demuxer_t` struct holds information like demuxer type, file name, general information in the form of a `demuxer_info_t` struct and link the `stream_t` struct into the demuxer, the `demux_stream_t` struct holds information regarding the actual stream. More detailed information for each stream; video, audio or sub stream. It holds pointers to packets, wrapped in a struct called `demux_packet_t`, flags, the position in the stream, the pointer to the buffer where the data is stored etc. The `demux_stream_t` struct is accessed from the `demuxer_t` struct which holds a pointer to it.

ds_fill_buffer() checks if there is a data packet in the `demux_stream_t` struct's buffer. If so, extracts the packet, updates the fields in the `demux_stream_t` struct and returns. If there are no packets in the `demux_stream_t` buffer, a call to *demux_fill_buffer()* is made. This function calls the function pointed to by the `fill_buffer` pointer in the `demuxer_t`'s `demuxer_desc_t` struct. This points to *demux_rtp_fill_buffer()* (see table 3.3) in demux_rtp.cpp. Here, a loop is entered and is not exited until a valid demux packet, with respect to its presentation time, is retrieved. To get a new demux packet, the *getBuffer()* (see table 3.3) function is called, and blocks until a packet is available from Live555. In *getBuffer()*, the `ReadBufferQueue` object is referenced from the `RTPState` struct saved in the `demux_t` struct's priv field. *MediaSubsession::readSource::getNextFrame()*, which is stored in `ReadBufferQueue::readSource`, is then invoked to get a RTP packet from the

socket, strip it from its header and deliver the payload into the `demux_packet_t`'s buffer. We are then blocked until the data is available, and when it arrives the `demux_packet_t` is returned. The data retrieved is then processed and encoded by the respective functions shown

A: Reading and retrieving video:

```
#7  0x08056653 in main () at mplayer.c:4144
#6  0x08115be1 in video_read_frame () at video.c:438
#5  0x08117118 in read_video_packet () at parse_es.c:65
#4  0x0811abd5 in demux_pattern_3 () at demuxer.c:449
#3  0x0811a335 in ds_fill_buffer () at demuxer.c:336
#2  0x0817290e in demux_rtp_fill_buffer () at demux_rtp.cpp:277
#1  0x08172442 in getBuffer () at demux_rtp.cpp:497
#0  MultiFramedRTPSource::doGetNextFrame1 () at MultiFramedRTPSource.cpp:134
```

B: Reading and retrieving audio:

```
#8  0x080547e1 in main () at mplayer.c:4075
#7  0x0809cfbf in decode_audio () at dec_audio.c:387
#6  0x08624479 in MP3_DecodeFrame () at sr1.c:60
#5  0x0809f563 in mplayer_audio_read () at ad_mp3lib.c:28
#4  0x0811aa58 in demux_read_data () at demuxer.c:404
#3  0x0811a335 in ds_fill_buffer () at demuxer.c:336
#2  0x0817290e in demux_rtp_fill_buffer () at demux_rtp.cpp:277
#1  0x08172442 in getBuffer () at demux_rtp.cpp:497
#0  MultiFramedRTPSource::doGetNextFrame1 () at MultiFramedRTPSource.cpp:134
```

Figure 3.5: Back trace taken from Data Display Debugger (DDD) when streaming a mpg file from Live555 media server. Part A is related to the decoding of the video stream. Part B is related to decoding the audio stream.

in figure 3.5, until more data have to be retrieved and we enter the same call flow just described. This is done until the stream is ended.

3.2.4 Changes made to MPlayer

MPlayer version 1.0rc1 does not support receiving a MPEG 1 system stream over RTSP. It fails to find the correct demuxer when it discovers that the MPEG stream it receives is a multiplexed stream with audio and video. This is easily fixed by inserting an if-test where we check the codec name string Live555 has received from the streaming server. If it is MPEG 1 system stream (MP1S)[2] we create a new demuxer of type DEMUXER_TYPE_MPEG_PES, if the codec name is something different, like for example MPEG-1 or 2 video (MPV), we make MPlayer search for a suitable demuxer itself, as it did before we changed it.

To close the stream in a proper way, we have to make sure that a TEARDOWN message is sent to the server when the user decides to abort the streaming. This is done in demux_mpg.c. Here, we check the stream type and if it is of type STREAMTYPE_DS, we know that the RTP demuxer is used to deliver the data. A call to the *close()* function in demux_rtp.cpp is then all it takes to make Live555 initiate a tear down. The *close()* function is accessible from the `demuxer_desc_t` struct contained in the RTP demuxer.

[2] We have only tested this with MP1S in Komssys

3.3 Komssys

Komssys is a media server built upon the idea of stream handlers (section 2.3.2). In this section we will look at unicast delivery over RTP/RTSP and see which components Komssys uses to send RTP packets to the client when it streams a MPEG file from disk.

As Komssys is a stream handler framework it relies on the use of base classes to make it easy to plug different types of stream handlers together. We will not describe the base class pointers Komssys uses but only use the names of the sub classes that are being used in this particular scenario.

3.3.1 The RTSP setup

Before Komssys can start sending out RTP packets to the client, the RTSP session has to be set up. The RTSP setup starts when the client sends a SETUP message to Komssys. But this might not always be the first message the client sends. Live555, for example, sends a DESCRIBE message before it decides that it can send a SETUP message. The DESCRIBE message tells Komssys to send a description, in SDP format, of the requested media file back to the client. This message then helps the client decide if it supports the way the server will stream the requested media. If the client does not support the way Komssys will stream the requested media, it will not start a RTSP session. If there are no problems, the client will send the SETUP message and thereby initiates the setup procedure that can be seen on figure 3.1. When Komssys receives the SETUP message, in short, it does this:

- Creates a `MNSocket` object which is defined in os/net/MNSocket.h. This class encapsulates socket functionality for the connection and has member functions that takes care of sending and receiving data to the client and other socket related functionality.

- Parses the SETUP message with a `SDPParser` object. This is a class defined in sdp/SDPParser.h.

- Creates a graph manager object, in this case a `FileStreamerGM` which we saw in section 2.3.2 creates the three stream handlers `FileSourceSH`, `RTPEncoderSH` and `RTPSinkSH`. The graph manager opens the file the client requested, sets RTP and RTCP port numbers that it shall use and initializes the stream handlers. All the graph managers and stream handlers are defined in the rtp/gm/ and rtp/sh/ directories respectively.

- Creates a `MNRTCP` and `MNRTP` object which are responsible for sending out RTCP and RTP packets respectively. `MNRTP` builds up the RTP packet in memory before sending it to the `MNSocket` for delivery. They are defined in rtp/rtp/MNRTCP.h and rtp/rtp/MNRTP.h respectively.

When the PLAY message arrives from the client Komssys starts the streaming process by calling *FileStreamerGM::play()*.

3.3.2 From disk to net

When *FileStreamerGM::play()* gets called, it “starts” the stream handlers by calling *start-Streaming()* on each of them. This prepares the stream handlers to start streaming, especially it

starts a timer in RTPEncoderSH which calls *RTPEncoderSH::send_timer_callback()* at every thread clock tick and makes sure that data is pulled from FileSourceSH and pushed to RTPSinkSH.

A smart pointer that points to a Data object is used to transfer data between the three stream handlers. The smart pointer named DataPtr, using typedef declaration in C++. Using DataPtr, Komssys only sends a pointer between the stream handles, eliminating the need to send all the data, and thus saving a lot of CPU cycles.

Pulling data from FileSourceSH

Pulling data from FileSourceSH starts with a call to *RTPEncoderSH::pull_encode()* with a DataPtr as argument. This function checks if we are streaming and that a member variable of type NoRetrans is not NULL. If it is set, a call to *NoRetrans::get_next_data()* is made. NoRetrans is defined in rtp/codec/rt-profiles/NoRetrans.h and holds the retransmission functionality, which is a RTP extension. As the name imply, this class offers no retransmission, but only helps RTPEncoderSH get the data from FileSourceSH.

In *NoRetrans::get_next_data()* we get the encoder for the file format we are streaming, which helps read out the MPEG file correctly from disk, called PacketizerMPEG. Then, *PacketizerMPEG::get_next_data()* is called. *PacketizerMPEG::get_next_data()* uses a helper function in RTPEncoderSH's SinkEndpoint object called *pull_from_peer()* to pull data from the neighboring upstream stream handler FileSourceSH. This function asserts that RTPEncoderSH and FileSourceSH are connected before calling FileSourceSH's *SourceEndpoint::pull()*. The arguments to both of these functions are a DataPtr to store the data, and two arguments specifying how much to data pull. *SourceEndpoint::pull()* calls *FileSourceSH::pull_from_file()* which reads a given amount of data from disk and puts it into the DataPtr and returns it. When we return to *PacketizerMPEG::get_next_data()*, information regarding how much we have read and current byte position are updated, together with the calculation of the time stamp for the RTP packet. Then a rope [2] to the data is returned to *MNRTPEncoderSH::pull_encode()*. Here, a *RPtr<MNRTPPacket>*, MNRTPPacket being a sub class of class Data, defined in rtp/rtp/MNRTPPacket.h, object is created to hold the pulled MPEG data, and to hold parameters related to the RTP packet, like the time stamp and RTP marker bit [11]. The DataPtr which was sent in as argument to *pull_encode()* is then assigned to the pointer to the newly created RPtr<MNRTPPacket> object, and returned.

When Komssys uses NoRetrans to handle the retransmission, which then means no retransmission, an extension header is added to the RTP packet. This extension header holds the difference between the absolute byte position and how many bytes we have pulled.

Pushing data to RTPSinkSH

When we return from the call to *MNRTPEncoderSH::pull_encode()*, the data and RTP related information are stored in a DataPtr object. If the pulling from the FileSourceSH went fine, we send it downstream, to the neighboring stream handler RTPSinkSH, using RTPEncoderSH's *SourceEndpoint::push_to_peer()* with the DataPtr object as argument. This function asserts that RTPEncoderSH and RTPSinkSH are connected, and calls RTPSinkSH's *SinkEndpoint::push()*. This function then calls *RPTSinkSH::push_to_net()*. Here, the data we pulled from FileSourceSH is extracted from the DataPtr object, together with the time stamp, marker bit and the extension header length. If the RTP

packet should be sent without an extension header, *MNRTP::rtp_send()* is called, otherwise *MNRTP::rtp_send_ext()*. The two functions differs only in the way the RTP packet is constructed. In short, both functions builds the RTP packet, sends a pointer to it to *MNSocket::send()* which then sends the packet out on the net using a *sendmsg()* system call. The amount of data that was sent is then returned from *MNSocket::send()* and is, in *MNRTP::rtp_send()* and *MNRTP::rtp_send_ext()*, used to update statistics. Then if there were no errors while sending the packet, the sequence number of the RTP packet sent is returned to *RTPSinkSH::push_to_net()*.

3.3.3 Changes made to Komssys

We have made some changes and added some new stream handlers and graph managers to Komssys. The changes have made it possible for MPlayer to communicate with Komssys, and let us use a new system call that we have implemented in Linux, which is described in section 3.4.1 and called *sendfile_prefixed()* [12]. The new stream handlers and graph managers are created so we can test different implementations of Komssys.

Streaming MPEG using Komssys and MPlayer

As of today, MPlayer can not communicate with Komssys when streaming a MPEG file over RTSP/RTP. The reason for this lies in the setup of the RTSP session, more precisely in the response to the DESCRIBE message sent to Komssys from MPlayer. As we can see from figure 3.6, Komssys sends the character string “X-PN-MPG” together with the codec code 32. This is an unknown codec name to Live555, which causes it not to start the setup of the RTSP session, and thereby causing MPlayer to quit. A small change in sdp/PayloadTypes.h fixes this. By making Komssys associate the codec code 32 with “MP1S” instead of “X-PN-MPG”, Live555 is able to set up the RTSP session.

```
DESCRIBE rtsp://localhost:9070/full_1.mpg RTSP/1.0
CSeq: 1
Accept: application/sdp
User-Agent: MPlayer (LIVE555 Streaming Media v2007.08.03)

RTSP/1.0 200 OK
CSeq: 1
Content-Type: application/sdp
Content-Length: 141
Content-Base: rtsp://localhost/full_1.mpg/

v=0
o=mzink
s=Erricson IP Docu
c=IN IP4 0.0.0.0
b=AS:1965
t=0:15:0:2
a=framerate: 45.000000
m=video 0 RTP/AVP 32
a=rtpmap:32 x-pn-mpg/90000
```

Figure 3.6: RTSP Describe message.

New stream handlers and graph managers

We have added two new stream handlers and graph managers to Komssys. The stream handlers are called `TunedFileSH` and `TunedFilePrefixedSH`. `TunedFileSH` is a stream handler that combines the functionality of the three stream handlers we looked at in section 3.3.2 into one stream handler. This new stream handler has removed any use of virtual functions that are used in the regular stream handlers, which are functions that are implemented in a base class but can be overridden in a sub class to behave differently.

The second stream handler, `TunedFilePrefixSH`, is based on `TunedFileSH`, but uses the new system call *sendfile_prefixed()*. We have removed read-from-disk functionality from `TunedFilePrefixedSH`, instead it uses *sendfile_prefixed()* to do this.

The new graph managers are `TunedFileStreamerGM` and `TunedFilePrefixed-StreamerGM`. They use `TunedFileSH` and `TunedFilePrefixedSH` respectively as stream handlers. The rest of their functionality is the same as the `FileStreamerGM` we have looked at in section 2.3.2.

Adding support for a new system call

In section we describe a new system call that we have implemented. To have Komssys support this, we have changed rtp/rtp/MNRTP.h and net/os/MNSocket.h. In MNRTP.h we have added two new functions, one for sending regular RTP packets and one for sending RTP packets with extension headers, where both are using the new system call. They do not use the system call directly but use a new function we have added to MNSocket.h called *sendfile_prefixed()* which calls it for them.

3.4 Linux

We have used Linux kernel 2.6.20.15 as the test bed for the experiments we have done. Most of the changes made to the kernel is described in chapter 4, but we have implemented a system call that is used by Komssys and therefore we describe it here.

3.4.1 A new system call: sendfile_prefixed

sendfile_prefixed() is a modification to the already existing system call *sendfile()*. *sendfile_prefixed()* allows us to send header data down to the kernel, which can be prepended to the data we wish to send using *sendfile()*. By using this new system call we avoid using CPU cycles on context switching which Komssys does in `FileSourceSH` when it issues a *read()* system call to read from file, and on copy operations during sending [12]. Instead, we can discard of the *read()* system call by creating the RTP header in Komssys and send it down to the kernel with *sendfile_prefixed()*. In the kernel, *sendfile_prefixed()* will send the RTP header to the network buffer and cork it, which prevents it from being sent out to the client, where it waits for a chunk of the file the client has requested. When this arrives, the chunk of data and the RTP header will be uncorked and sent as a RTP packet to the client. The data path, using `TunedFilePrefixedStreamerGM` as graph manager in Komssys, is illustrated in figure 3.7. For a comparison, the data paths for the graph managers `FileStreamerGM` and `TunedFileStreamerGM` is illustrated in figure 3.8 and 3.9 respectively.

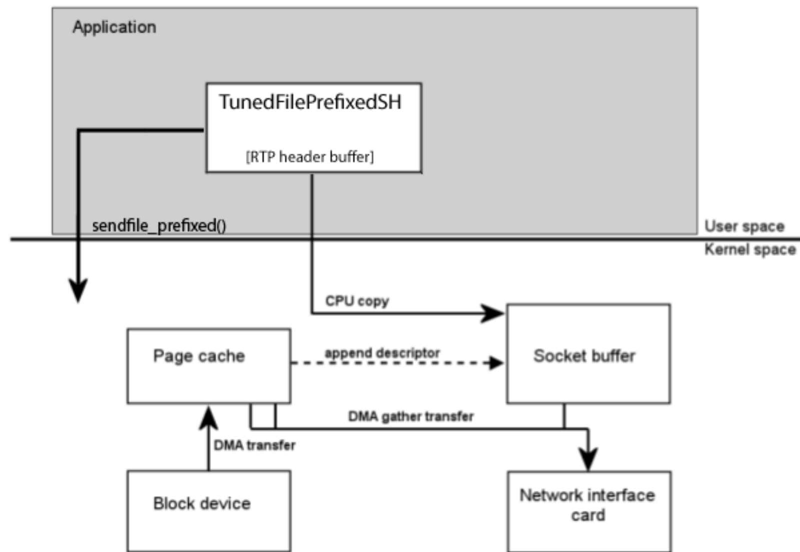


Figure 3.7: User/kernel space interaction usingsendfile_prefixed() with TunedFilePrefixed-StreamerGM.

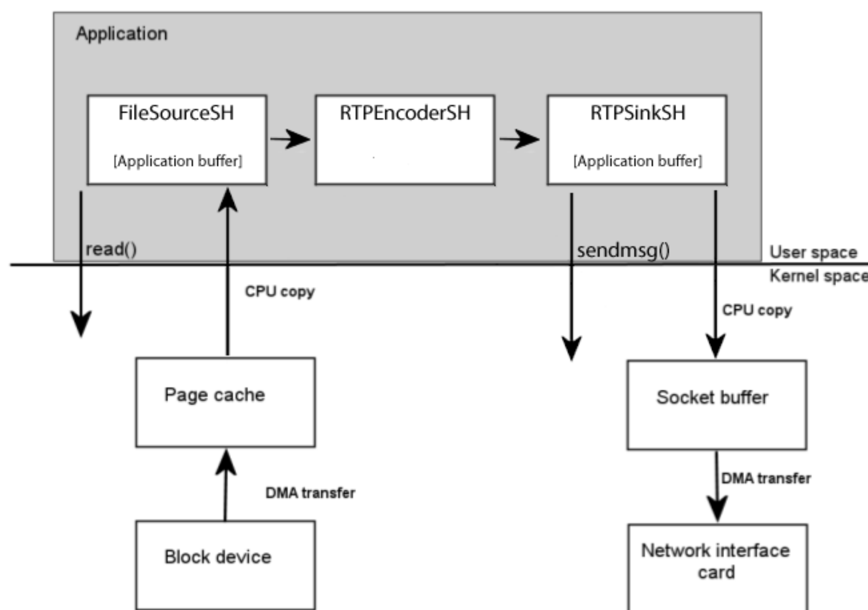


Figure 3.8: User/kernel space interaction using a combination of *read()* and *sendmsg()* system calls with FileStreamerGM.

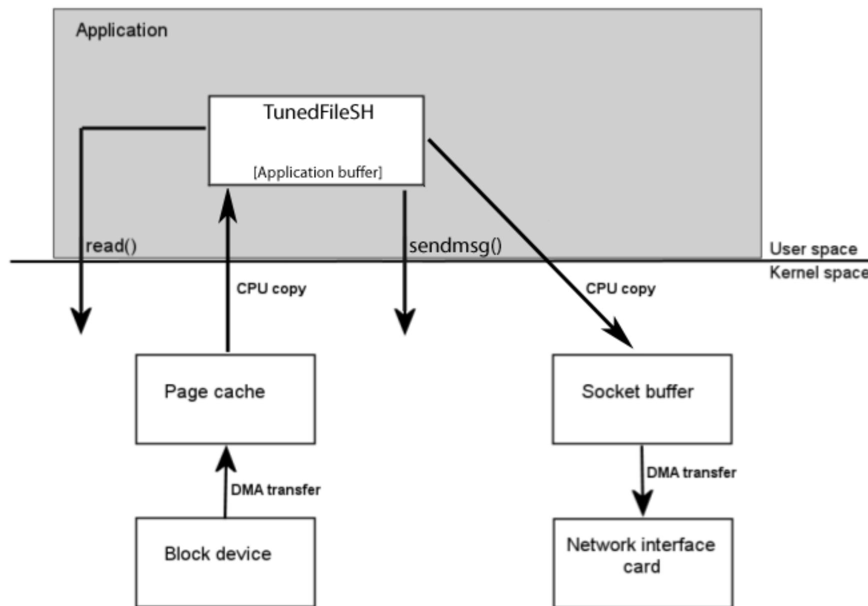


Figure 3.9: User/kernel space interaction using a combination of *read()* and *sendmsg()* system calls with TunedFileStreamerGM.

sendfile_prefixed() uses two helper functions, one that corks the header data at the network buffer, and one that uncorks it when we have called *sendfile()* to send a chunk of data from disk to the network buffer. These helper functions are implemented for both User Datagram Protocol (UDP) and Transmission Control Protocol (TCP).

3.4.2 Changes made to Linux

To implement a new system call in the 2.6 Linux kernel, there are three files that have been modified: arch/i386/kernel/syscall_table.S, include/linux/syscalls.h, include/asm-i386/unistd.h. The changes in these files are needed to make Linux aware of the new system call.

The system call itself is implemented as a module so it can easily be modified without re-compiling the kernel. As already mentioned, *sendfile_prefixed()* uses two functions that helps it cork and uncork data at the network buffer. Both of them are implemented for UDP and TCP in net/ipv4/upd.c and net/ipv4/tcp.c respectively.

3.5 Summary

We have looked at how MPlayer initializes two important structures, namely the demuxer_t and stream_t structs. These help MPlayer retrieve the data from a source and demultiplex it. We have also seen how MPlayer makes use of Live555 to handle streaming with RTSP as this is currently unsupported in MPlayer's own code. In section 3.2.4 we looked at the changes made to MPlayer to make it work when streaming MPEG with multiplex audio and video over RTSP.

In section 3.3 we looked at the data path in Komssys, which functions and classes that are used, when streaming a MPEG file from disk to network over RTSP and using RTP/RTCP as

transport protocol. Then, we went through some changes and add-ons, to make Komssys work with MPlayer and some new stream handlers and graph mangers to Komssys, that we will make use of in the chapter 5. We also added support for using the new system call, *sendfile_prefixed()*, in Komssys. The last we covered was a new system call we have added to Linux. This was a modified version of the *sendfile()* system call, that we will use with Komssys when performing experiments.

In the next chapter we will look at a time measurement tool that we we will use in chapter 5 to do benchmarks of Komssys using the proposed changes.

Chapter 4

Time measuring tool

4.1 Introduction

Measuring how much time used from point A to point B in program code can easily be done with existing functions in Linux. It involves retrieving a time stamp both at point A and B then calculating the difference to find out how much time was used. This works fine when measuring small pieces of code. But if we want to do time measures when A and B is far apart in terms of code lines, the method just outlined may be inadequate for several reasons: the program may be scheduled out, it may get interrupted to handle hardware interrupts, page faults might occur if the program code is not residing in memory and different kinds system calls may be called. All these factors affect a time measure, some more than others. If the program is scheduled out this usually will affect the time measure more than if it is interrupted by a single interrupt and then later resumed. System calls that offer reads and writes to disk, for example, may have variable execution time depending on the state of the disk. So to get a good approximation on how much time the CPU spends on program code alone, when doing time measure over many lines of code, one subtract time spent in kernel space doing interrupts, system calls and exceptions from the total time measured, starting at A and stopping at B. Subtracting all these factors makes it easier to compare different implementations of a program, for example with or without the use of virtual functions when written in C++, with respect to time spent on the CPU.

It is an essential condition that the data path in the code we are time measuring is single threaded, if this was not the case, several time measures could start before the first measure stops, making it hard to estimate how much time used from A to B.

4.2 Design

Identifying how much time is used for user space processing and kernel space processing, on behalf of the program, and time spent on other processes, we first have to identify where the entry points to the kernel resides, start a timer at these points and when the program is to resume, stop the timer at the exit point out of the kernel. The file that holds all entry points to the Linux kernel can be found is entry.S and is located in arch/i386/kernel/ for the i386 architecture[1]. It also holds the exit point which is used by interrupts, exceptions and system calls that use the int $0x80 instruction labeled restore_all (figure 4.1), and the entry and exit point for system calls

[1]The measure tool has been developed for the i386 architecture using kernel 2.6.20.15.

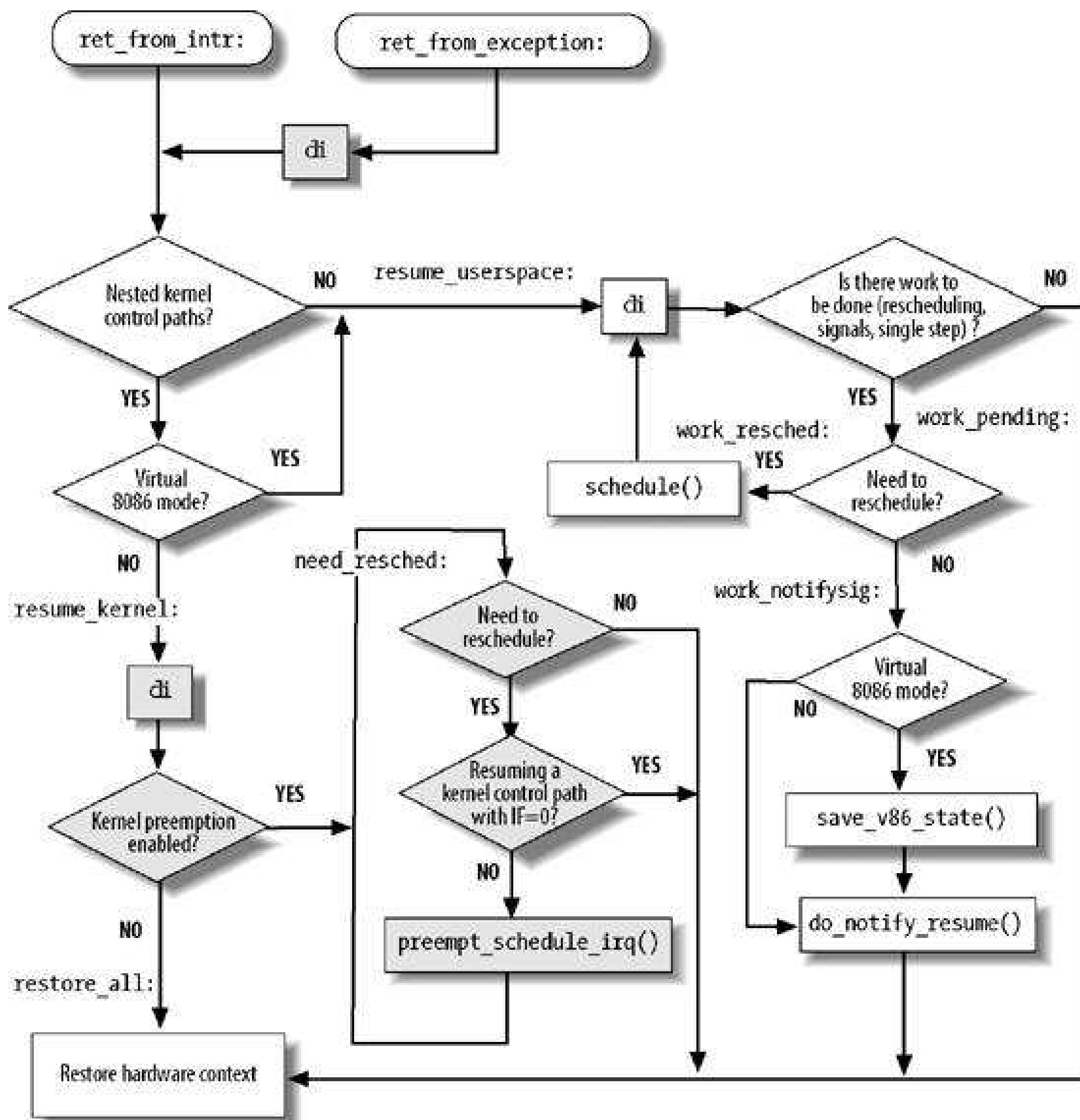


Figure 4.1: Exceptions and interrupts both end up at the restore_all label [5].

that use the sysenter instruction. The exit point for the sysenter instruction is at the bottom of the sysenter entry point. It is at these two exit points, restore_all and sysenter exit point, that we stop the timer.

All interrupts are set up to push the IRQ number, associated with the interrupt minus 256, on the stack and then jump to a label called common_interrupt [5]. So a suitable location for interrupt detection will be there. System calls that use the int $0x80 instruction enters at ENTRY(system_call) and those that use the sysenter instruction enter at ENTRY(sysenter_enter). Placing code here will be a good location to detect when these system calls are issued.

Exceptions are the third possibility an user space processes may end up in kernel space. Most of the exceptions are used to signal when errors have occurred in the program, like for example divide by zero and overflow. These relates to errors in the program and usually makes the program that caused the exception to terminate. The only useful exception, for time measuring, which does not make the program terminate is a page fault, assuming that the program has access to the given page. Its entry point is KPROBE_ENTRY(page_fault) and is the last point where code is to be inserted to detect when the program enters kernel space.

For the user to be able control when to start and stop the time measure, the program has to communicate with the kernel. This is done with two system calls, one for starting the measure and one for stopping it. These system calls take a struct as argument to be filled with total time used in both kernel space and user space and statistics, when the time measure stops.

In the following sub sections we will look at how to use system calls to start and stop the time measure, present some design issues related to how to insert code into the kernel to detect when one of the above mentioned entry/exit points are entered/exited, how to retrieve a time stamp and what possible kernel control paths the time measure code has to take into account an.

4.2.1 New system calls

Since we start the time measures in user space and the code that calculates time stamps and time used is in kernel space, we have to create two system calls, *start_time_measure()* and *stop_time_measure()*, so that we can communicate with the code located in the kernel. The system calls are implemented in such way that we can send a struct "down" to kernel space. This struct is then filled time information. Information regarding when we started the time measure when *start_time_measure()* is called, and total time spent in kernel space and user space and statistics when *stop_time_measure()* is called. The members of the struct, and an explanation of what the meaning of each, is put into table 4.1

The total time used in kernel space and user space is then calculated as follows:

$$\begin{aligned} total_time_kernel_space = & \ kernel_space_total_time \\ & - \ umeasure.kernel_space_time_base \end{aligned}$$

$$\begin{aligned} total_time_user_space = & \ stop_time_stamp \\ & - \ umeasure.start_time_stamp \\ & - \ total_time_kernel \end{aligned}$$

Here, umeasure is the name of the struct we send down to kernel space. stop_time_stamp is created when we stop the time measure, and umeasure.start_time_stamp when we started. kernel_space_total_time is the total time used in kernel space for all measures taken for a single

Data type:	*Name:*	*Comment:*
uint64_t	kernel_space_time_base	We set this to the time used in the kernel when we start a time measure.
uint64_t	start_time	Time stamp from when we start the measure.
uint64_t	total_time_user_space	The total time used on the time measure in user space.
uint64_t	total_time_kernel_space	The total time used on the time measure in kernel space.
uint16_t	syscall_count	Number of syscalls issued in a time measure.
uint16_t	page_fault_count	Number of page fault happened in a time measure.
uint16_t	interrupt_count	Number of interrupts hap-pend in a time measure.
uint16_t	sched_count	Number of times the CPU has been used by other processes while in between a time measure.
uint16_t	sched_count_between	Number of times the CPU has been used by other processes while in between two time measures.

Table 4.1: Struct that is used to calculate statistics and time in a time measure.

process, and umeasure.kernel_space_time_base is what kernel_space_total_time was when we started the time measure.

System calls implementation

The implementation of the system calls are actually made in a module which we insert when we want to perform a time measure. The default implementation, which resides in the kernel, only return -ENOSYS to signal that the module is not present. The reason we have chosen to do it like this is because it makes the development much easier. If the implementation is located in the kernel we have to recompile and restart Linux every time a change to the system calls are made. Writing the implementation in a module saves us a lot of time as we only have to recompile the module.

Implementing the system calls in a module is done by "hijacking" the default implementations in the kernel by inserting new ones in the syscall_table and saving the old ones. The new ones are inserted when the module is inserted into the kernel and removed when the module is removed. The syscall_table must be made available from arch/i386/kernel/i386_ksyms.c by first declaring it as an extern void pointer to an array and then exporting it using EXPORT_SYMBOL. Doing this makes it possible to alter the syscall_table inside our module.

4.2.2 Inserting time measure code into the kernel

Inserting code into the Linux kernel can be done by either statically, modifying some existing source files, or it can be done dynamically at run time. The latter is a safer and a more time friendly method. It involves the use of kernel modules [13] and kprobes [14] which are inserted into the kernel at run time. Because the kernel is not modified there is no need to re-compile the kernel and reboot which is a time consuming business, and has to be done when inserting code statically. Kprobes is originally intended to be used as a debugging tool. But it may also make a very suitable tool when we want to insert code into the kernel to record time stamp information as described above.

Kprobes

Kprobes is a mechanism that allows users to dynamically insert non-disruptively break points, probes, at almost any instruction in the kernel. The exceptions are entry points defined with KPROBE_ENTRY in entry.S and functions declared with the keyword __kprobes. These definitions prohibit kprobes to be inserted[2]. Kprobes are usually packaged inside a kernel module. It gets registered when the module is initialized and unregistered when the module is removed. The register process involves inserting the break point at a given instruction address in the kernel and associate handler functions to be called when the break point is encountered. There are two handler functions that can be associated: a pre-handler and a post-handler function. The pre-handler gets executed just before the probed instruction, where the break point is inserted, and the post handler right after the probed instruction has been executed. One does not have to specify both handler functions, it suffices to specify one of them. When control reaches one of the handler functions, we have access to a copy of all of the CPU registers that

[2]Defined in include/linux/linkage.h and include/linux/kprobes.c respectively.

were present when the break point was encountered. A schematic figure of these steps is shown in figure 4.2

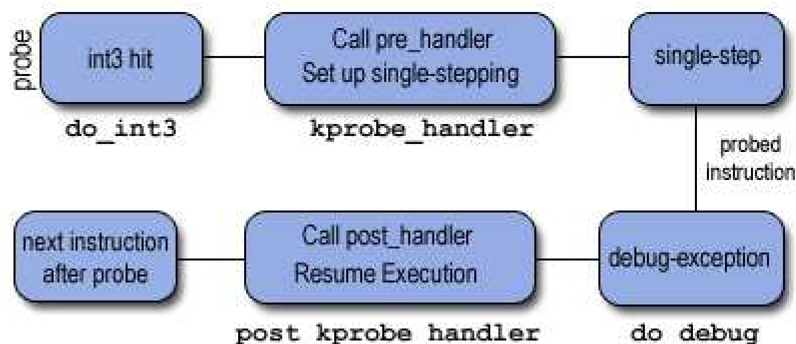


Figure 4.2: Execution of a Kprobe [6].

How kprobes work is architecture dependent [14]. We will take a short look at what is done on the i386 architecture. Here, kprobes works by making a copy of the instruction where the break point is inserted, and replaces the first byte(s) of the probed instruction with the break point instruction int3 [14]. This is done when the Kprobe is registered. When the break point is encountered a trap occurs, the registers are saved and control is passed to kprobes which executes the pre-handler (if existing). Then the saved instruction is single stepped and the post handler is executed (if existing). When the post handler exits, the instruction coming after the probed instruction is executed and code execution continues as before.

Using kprobes to insert measure code

Using kprobes to insert time measure code into the identified entry points in section 4.2 involves creating a kernel module which will hold the kprobes, and then register six kprobes, one for each entry point and one for each of the exit points. The first hinder we meet is that the page fault entry is defined with KPROBE_ENTRY, meaning that we are not allowed to insert a Kprobe at this entry. We could argue that the time spent doing page faults are a diminishing factor in the time measure results if we are doing repetitive measurement on the same code, i.e. in a while loop. In a loop, page faults will only happen the first time, and then the code is in memory and no more page fault will occur if not those pages are swapped out. But if the time measuring is to be done one time only the time spent retrieving missing pages might have an impact on the time measure. The different code lines we are measuring might have different disk and memory size and therefore may have different page faults times. So, it can be a good idea to add the over all time time spent doing page faults to the total kernel space time.

The second hinder and the most difficult to pass is the consequences that occur when inserting a Kprobe at the restore_all label in entry.S. Without code here we are not able to find out how much time was spent in kernel space. And the reason why it does not work using kprobes is that kprobes uses the int3 instruction which causes an exception. As all other exceptions, when the int3 exception is done with its processing, when the probe has been processed, it makes a jump to a label called ret_from_exception which eventually ends up at the restore_all label. By inserting a probe at the restore_all label we will be inducing an infinite loop in the Linux kernel which is, to say at least, something we not want to be doing.

In conclusion we can say that kprobes is not suitable for these kinds of time measures, i.e. where we wish to find the time spent in kernel space and subtract this from the total time used from start to stop. This is because it is impossible to detect when we are done with kernel space processing and re-enter user space. On the other hand, we can and will use kprobes to count how many times the measuring process is scheduled out, to be able to see the effect of cache misses. We want to count the number of processes, not counting the one we are measuring, that is run on the CPU during and in between time measures. We achieve this by registering a Kprobe, in the same module as the two time measure system calls resides on entry to *__switch_to()*, and associates a pre-handler with this probe. Then, in the pre-handler we check if the process we are measuring is being switched out or in. If it is switched out, we increment a variable called sched_count_total and set the variable scheduled_out to TRUE. If the process is being switched in we set scheduled_out to FALSE. When scheduled_out is TRUE, sched_count_total is incremented every time a process switch occurs. Thus we will get the total number of processes that have used the CPU when the measuring process has been scheduled out.

For example, let us say we are doing time measure in code that is repeated, e.g. in a while loop, and the process is interrupted after one time measure has ended. The kernel will check if there are other processes that are waiting to use the CPU and if there are, and they have higher priority than the process we are measuring, call *__switch_to()*. On entry to this function, our kprobe will intercept and take us to the pre-handler. Here, we check the pid of the process that is beeing switched out, and if it is the process we are measuring, we increment schedule count. We then set the flag variable indicating that we are scheduled out to true, and return to normal execution. Now, every time a process switch occurs that does not involve the process we are measuring, the kprobe intercepts the *__switch_to()* function and increments the schedule count. When our process is switched back in, we set the schedule out flag to false, and return.

Statically inserting measure code

Statically inserting code at each of the identified entry/exit points to the kernel in section 4.2 involves writing assembly code in entry.S. At each entry point we have to create a start time stamp and at the return to user space, in the restore_all label and exit from sysenter, create a stop time stamp. To ease the development these functions can be written in C and called from entry.S. At each point, we have to save the registers that will be clobbered and then restore them when we return from the call. This has to be done so everything run as if the inserted code is not there. If we were to forget this it can result in unexpected behavior of the programs affected by clobbered registers.

We define the functions to call in a C-file. In this file we define the functions to be used and some global variables that shall hold the time stamps and some statistics regarding page faults, system calls and interrupts. Four functions are defined, one for each of the entry points and one for each of the exit points. To have only the process we are measuring trigger the creation of time stamps, each function checks the pid of the process entering kernel space. This is done by comparing the current[3] process pid with the process pid of the process we are measuring. The latter pid is saved when we start the time measure and will be described in section 4.2.1. If the two pids match we start or stop the time measure, which depends on if we are at an entry or exit point respectively.

[3]The current process refers to the process that has execution time on the CPU.

To recap, we create a time stamp in each of the functions that are to be placed at the entry points and we create a time stamp at the exit points. At the exit points we subtract the time stamp we created when we entered kernel space from the stop time stamp. The difference is then how much time spent processing in kernel space and possible on other user processes if the program has been scheduled out. This time is then added to a global variable that holds how much time is spent in kernel space altogether. When the measuring is stopped, this variable, indicating how much time is spent in kernel space and possibly on other processes, is subtracted from the over all total time it took from A to B. The difference is how much time was used by the user process.

4.2.3 Retrieving a time stamp

Retrieving current date and time could have been done with several system calls in Linux [5]. Two of them is *time()* and *gettimeofday()*. The former returns the number of elapsed seconds since midnight at the start of January 1, 1970 (UTC), the latter does the same but returns it in a struct named timeval, that also holds the number of elapsed microseconds in the last second.

Doing time measures with second resolution is not very useful. Most time measures will often complete in much less time and secondly, this resolution is not very useful when comparing two time measures against each other. Using *gettimeofday()* gives us an increase in resolution with a factor of one million which is acceptable in many situations. But the fact that today's CPUs operates in terms of Ghz, microsecond resolution is far away from giving us the most detailed timing information. To get this we have to use nanoseconds resolution. This is only achievable on platforms that has a Time Stamp Counter[4] (TSC) and offers an assembly instruction to read this called rdtsc. On x86 processors this instruction reads the value from the TSC and stores it in edx:eax [15]. One is free to use both registers or just one of them. Using only the eax register gives us a cycle count of 2^{32} before it overflows. How much time before the eax register overflows depends on the processor speed. If we are running on a 1.6 GHz processor, the time between overflows is equal to:

$$2^{32} cycles * (1\ second/1,600,000,000\ cycles) = 2.68\ seconds$$

The exactness and reliability of the time stamp counter relies on some factors that we will take a closer look at in the following section.

Issues related to TSC

Using the rdtsc will under optimal conditions give us the number of CPU cycles the CPU has executed between start point A and stop point B when we subtract A from B. From this return value we can deduce how many nanoseconds have elapsed by dividing it by the CPU frequency. If our CPU is running at exact 2.4 Ghz we divide the return value from rdtsc by 2,4.

There are some issues that have to be taken into account that are related to exactness and reliability. Exactness is affected by the processor speed. We have to know the exact processor speed to calculate how many nano seconds used. This can be found via BIOS information or /proc/cpuinfo/.

As mentioned, we need optimal conditions for the rdstc to be reliable. There are two important properties that must be fulfilled in order for this to happen. The first property is

[4]Time Stamp Counter is a 64-bit register and holds the number of CPU cycles since start up.

that the value received from rdtsc must be strictly increasing: the rdtsc value from stop point B must be larger than the rdtsc value from start point A. The second property is that the CPU frequency where the rdtsc value is taken from must be constant between A and B. The former property secures that we get a positive difference when we subtract the rdtsc value A from the rdtsc value B. The latter property ensures that we can convert the computed difference into nanoseconds. But, there is no guarantee that these properties are met and so the rdtsc values retrieved are seen to have no reliable value and can not be used. This happens when we are on a multiprocessor or multicore system. These systems do not offer guarantees that their cycle counter is synchronized between cores [15]. Since there is no guarantee that the executing process will not be scheduled on a different core than the one it started on, the first property will be broken. The second property is often broken in laptop systems where the CPU is sometimes tuned to save as much battery as possible. If this happens we can not be sure that the CPU frequency is constant in between the A and B.

There is actually a third property that we have to take in to account when we are doing time measures on small pieces of code; which is out-of-order execution. This can happen when the CPU reorders the instructions it is about to execute. To overcome this, we can issue a cpuid instruction before we start a time measure and right before we stop the time measure. But on time measures where the code distance is big, the effect of out-order-execution is diminishing and therefore no need to execute cpuid.

Choosing between gettimeofday and rdtsc

gettimeofday() is not affected by any of the issues described about the rdtsc in section 4.2.3. The downside is the resolution of the timer. But if we can not be sure that the two properties described in section 4.2.3, then *gettimeofday()* is the best we can do in terms of timerr esolution. Under Linux though we have the ability, firstly, to make sure it only uses one core by disabling Symmetric Multiprocessing (SMP) support and secondly that it does not use different CPU frequencies by disabling support for CPU frequency shifting. Doing this is an easy task when configuring the Linux kernel and we have therefore decided to use the TSC, by using rdtsc assembly instruction, for creating the time stamps.

4.2.4 The kernel control paths

Every interrupt or exception creates a kernel control path that execute in kernel space on behalf of the current process [5]. Since interrupts can occur at any time, except when interrupts are turned off, the control paths can be nested, as shown in figure 4.3; an interrupt or exception handler can be interrupted which causes nesting levels greater than 1. Exceptions can only give rise to at most two kernel control paths since bug free kernels do not cause exceptions. The only time an exception occurs in kernel space is when the user invokes a system call which then causes a page fault.

The nested nature of kernel control paths must be taken into account when calculating how much time a process spends in kernel space. It is only the top level control path where we have to create a time stamp and when other nested interrupts occur we check if we are already in a nested control path. If we are, we do nothing and return. When the topmost control path is about to exit to user space again, we calculate the time spent in kernel space and clear the variables that indicates that we are in a kernel control path.

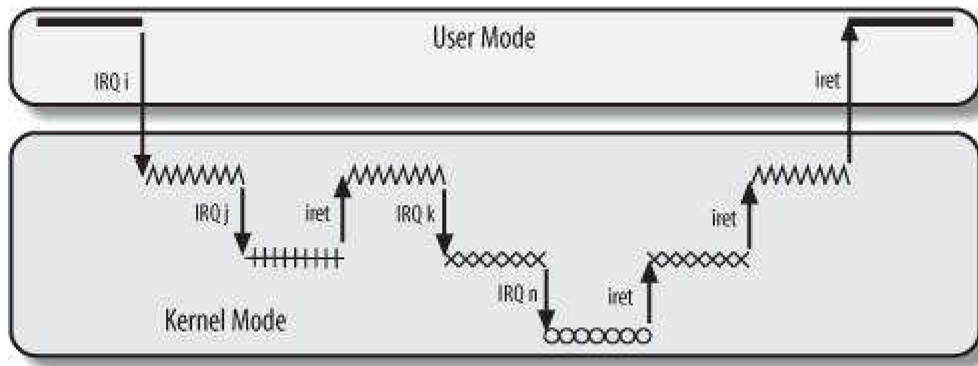


Figure 4.3: Nested interrupts in kernel space [5].

4.3 Summary

In this chapter we have looked at a time measurement tool that splits up the total time used on a time measure into kernel space time and user space time, and at the same time give us information about kernel events and if the process has been scheduled out. Doing this makes it possible to do time measures over a greater code distance without having to worry about the process being scheduled out or affected by other kernel variabilities as this does not affect the time used in user space. We discussed how this could be done using kprobes or statically inserting code, and looked at what time taking instruments Linux and the x86 platform had to offer. To start and stop the time measure, we introduced two system calls, and looked at how we calculate the time spent in kernel space for a process. For the results from the time measure to be valid, we saw that the measuring tool is only usable in code that is single threaded, if this was not the case, several time measures can be started in parallel.

In the next chapter we will use the tool presented in this chapter together with the standard time measurement procedure to investigate the effects of the proposed changes to Komssys we looked at in chapter 3.

Chapter 5

Testing of proposed changes

5.1 Introduction

In this chapter, we will test and evaluate the proposed changes to Komssys and look at their effect on the number CPU cycles Komssys have to use in order to create a RTP packet and send it out on the net, to a client. We will also evaluate the proposed time measuring tool described in chapter 4.

The proposed changes to Komssys and the time measuring tool have been developed so that we can measure the penalty of using virtual functions in Komssys, and to see the effects of using *sendfile_prefixed()*. We want to measure the effect, with respect to the number of CPU cycles used, of removing the use of virtual functions and further find out whether there are any improvements if we use *sendfile_prefixed()* instead of the combination of *read()* and *sendmsg()* system calls, used by `FileStreamerGM`.

5.2 Test setup

For the testing we run Komssys on a desktop computer running Linux 2.6.20.15 kernel with CPU frequency scaling turned off. It has a 1.6 GHz AMD processor with 512 MB ram. The client, which runs MPlayer and displays the video sent from Komssys, is a Macbook Pro 2.4 Ghz dual core processor.

5.3 Test Layout

5.3.1 Testing three different graph managers

We will test three different graph managers: `FileStreamerGM` (FSGM), `TunedFile-StreamerGM` (TFSGM) and `TunedFilePrefixedStreamerGM` (TFPSGM). Each time measure is started right before the call to *pull_encode()* in `RTPEncoderSH`, `TunedFileSH` and `TunedFilePrefixedSH`, and stopped right after we have sent the RTP packet to net by calling *MNRTP::rtp_sendfile_prefixed_ext()* or *MNRTP::rtp_send_ext()*, depending on if we are using *sendfile_prefixed()*. After the time measure is stopped we pack gathered statistics of the measure into a class called `TMeasure` and put this into a vector. We stop the time

measuring when Komssys has sent out 20000 RTP packets to the client, and write the gathered statistics out to file. These measurements are done with the time measuring tool.

To see effects of cache over-writes, we re-run the these tests with some background noise. This noise is background processes running as daemons and a cron job. The cron job is triggered execute every minute and update the desktop background. There are some daemons running on the regular test also, but these are system specific daemons that we did not turn of. These processes will often over-write the cache used by Komssys and this will affect the time measures as Komssys has to go to the main memory to get instructions and data, which is much slower than using the cache. The processes running on the regular and background noise test are put into the appendix.

We will look at the time used in user space and kernel space. We will also construct confidence intervals for some statistical measures we create from the test data we gather, and try do a meaningful interpretation of these results.

5.3.2 Testing cost of a context switch

Using *sendfile_prefixed()*, we are able to remove *read()* system call that is used in `FileStreamerGM` and `TunedFileStreamerGM`. This saves the CPU from copying data from disk into a buffer in user space. It also saves CPU cycles by not having to do a context switch.

To see the effect of removing a context switch, we wish to measure how many CPU cycles a context switch takes from when we call a system call and till we are back in user space again. We use an empty implementation of the *sendfile_prefixed()* system call, which just returns -ENOFAULT, and use the rdtsc instruction right before and after the call, using the guidelines found in [15]. The time measure is repeated 20000 times.

The *sendfile_prefixed()* system call takes six parameters. To see how much the number of parameters affects the context switch time, we also run the same test using an empty implementation of *start_time_measure()* which only takes one parameter.

The minimum value we got from the measures using *sendfile_prefixed()* was 278 CPU cycles. Using *start_time_measure()* instead gave us a minimum value of 268 CPU cycles. There is a difference of 10 CPU cycles between using 6 and 1 parameters, in other words 2 CPU cycles per parameter. After the system call we issue a cpuid instruction to remove out-of-order execution problems. We have to remove the time used to issue this instruction, together with the time used to issue rdtsc when we end the time measure to find the time used on a context switch. The rdtsc instructions was found to take 11 cpu cycles. The cpuid instruction on the other hand has a variable execution time, but its minimum value was found to be 58 cpu cycles, and therefore a context switch takes $(268 - 2 - 69) = 197$ CPU cycles to complete, from user space to kernel space and back again. We have to keep in mind that this number also includes the call to the system call after we have entered kernel space.

5.3.3 Testing overhead of time measuring tool

To see how much the time measuring tool affects each time measure, we set up a test to find the overhead using the time measure tool. We also used the statistics gathered from a regular time measure test using `FileStreamerGM`, to find, in average, how many CPU cycles are used by the measuring tool between the measure points A and B.

5.4 Test results

The test results we gathered were put into Matlab [16] and R [17], programs used for scientific and statistical programming respectively, to make plots and produce statistical information. Plots and statistics will be presented in the following sections.

5.4.1 Testing three different graph managers

		Min. [cycles]	1st Qu. [cycles]	Median [cycles]	Mean [cycles]	3rd Qu. [cycles]	Max. [cycles]
FSGM	**Regular:**	27840	28830	29000	29580	29280	209500
	Noise:	27420	29010	29220	33260	29530	216900
TFSGM	**Regular:**	26440	27480	27670	28080	27940	187900
	Noise:	25910	27040	27240	30750	27560	232000
TFPSGM	**Regular:**	21080	22290	22440	22790	22620	176000
	Noise:	20570	22040	22200	25340	22420	234800

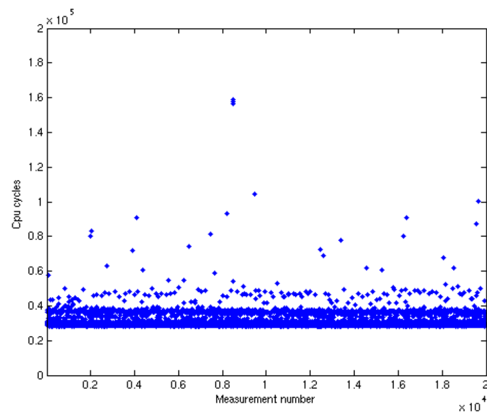
Table 5.1: Statistics of the number of CPU cycles used in user space from regular and noise test.

To see the improvements, if any, using the proposed changes, we did our first test on `FileStreamerGM`. This is the graph manager Komssys uses when streaming a file from disk. We will use the results we get from this test and compare it with the test of the proposed changes.

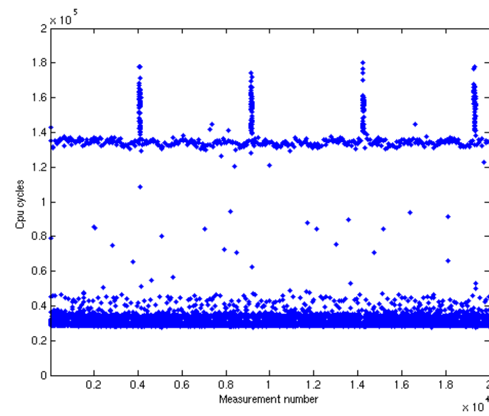
In table 5.1 and 5.2 we have created some statistics for the three test cases. Table 5.1 gives statistics for the number of CPU cycles used in user space for the regular and background noise test, and table 5.2 presents the same information but for the number of CPU cycles used in kernel space. An explanation of the column values, for both tables, are given in the following:

- **Min.**: The minimum observed value of all the 20000 measures.
- **1st Qu.**: This measure gives us where the 25% quantile lies, i.e. it tells us that 25% of all the measures lie between, including, this and the minimum value.
- **Median**: The middle value when arranging all the observations from low to high.
- **Mean**: The sum of all observed values divided by the number of observed values.
- **3st Qu.**: The same as the 1st Qu., but this marks the 75% quantile.
- **Max.**: The maximum observed value for all the 20000 measures.

Data in both tables have been rounded to make it easier to read. These tables together with plots of the data will be used to discuss the results.

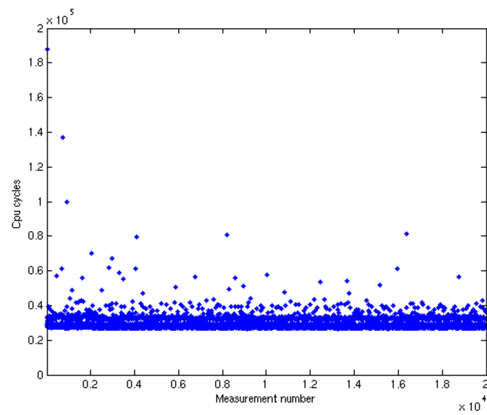


(a) Regular

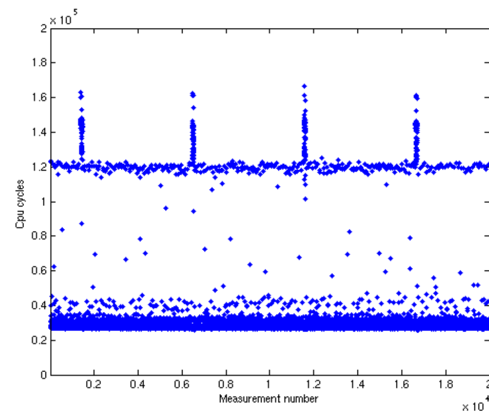


(b) Background noise

Figure 5.1: Plot of CPU cycles used with `FileStreamerGM` in user space.

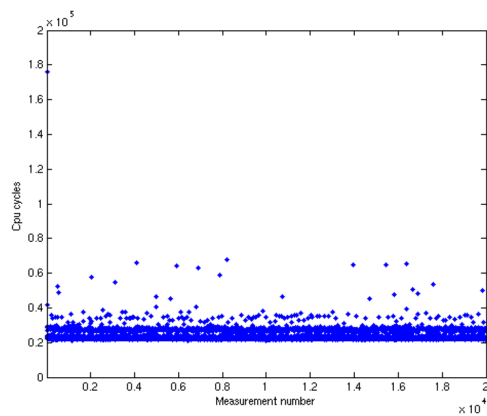


(a) Regular: kernel processing

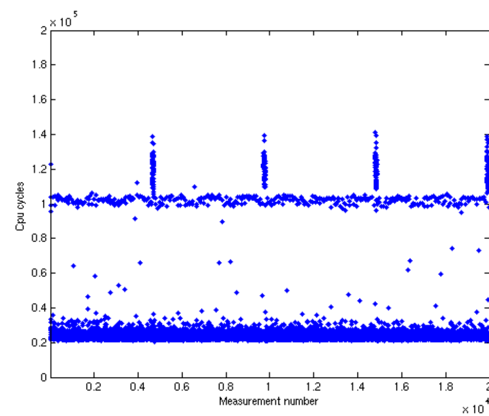


(b) Background noise: kernel processing

Figure 5.2: Plot of CPU cycles used with `TunedFileStreamerGM` in user space.



(a) Regular



(b) Background noise

Figure 5.3: Plot of CPU cycles used with `TunedFilePrefixedStreamerGM` in user space.

FileStreamerGM

We have plotted the test results from the regular test in figure 5.1 (a). In table 5.1 we can see that the maximum value for the regular time measure test, using `FileStreamerGM`, is more than 7 times higher than the minimum value, i.e. in worst case it takes 7 times as much time to complete the given task compared to the best case. But luckily these high values only happen on very few occasions. The mean is only 1740 CPU cycles higher than the minimum value. The dispersion around the median is equal to 308 CPU cycles, using a dispersion measure called **median absolute deviation from the median** (MAD). MAD is defined to be the median of the numbers $|x_i - \tilde{x}|$, where $\tilde{x}$ is the median for the data points $x_1, ..., x_n$ [18]. Using MAD instead of the standard deviation (SD), gives us statistical measures that are more robust to high values. The same property is true for the median of the data set compared to the mean, but for this data set there are no extreme outliers that affect the mean enough to not use it. There is only one value over 200000 CPU cycles, and this value is associated with the initial 4 page faults that are needed to bring the program code, from measure point A to B, into memory. As we can see, the mean for `FileStreamerGM` is not far off from the median in table 5.1. The SD, on the other hand, is very sensitive to even small outliers as we can see by computing it, making SD equal to 3314 CPU cycles.

When we introduced background noise to the time measure test, we got a result that is plotted in figure 5.1 (b) and with statistical measures in row 1 in table 5.1. We can see that this test has a higher frequency of bigger values, compared to the regular test, if we look at the plots in figure 5.1. In table 5.1, we see that the test done with background noise has higher values on almost every statistical measure. The only one that is lower is the minimum value. This is just a coincidence and could easily have been the other way around since the lower values in both tests are not affected by any cache over-writes. The statistical measure that reveals there are much more higher observed values in the background noise test, is the mean which is 11.1% (3680 CPU cycles) higher than that of the regular test. The MAD for this data set is 366 CPU cycles, indicating, together with the mean, that the data set is more dispersed. Looking at figure 5.1 (b), we see how the background processes have affected the time measurements. The measurements where the background and cron processes have been using the processor, when we are not measuring from A to B, lies over 90000 CPU cycles above most of the other measures. The difference gets even higher when the cron job starts executing, which we can see from the spikes. These measures are probably a result of the cache being replaced by data and instructions from other processes. When Komssys is allowed to run on the CPU again, the CPU has to fetch instructions and data from memory which is costly, with respect to CPU cycles, compared to fetching them from the cache. There are still some high values in the regular time measure test in figure 5.1 (a). Although we removed some daemons and a cron job, there will always be some background processes running, and as we can see, they will on some occasions affect the running time of the process we are measuring by tampering with the cache.

TunedFileStreamerGM

`TunedFileStreamerGM` offers the same functionality as `FileStreamerGM`, but it does so without the use of virtual function pointers. Row 2 of table 5.1 gives us statistical measures of the CPU cycles used in user space from the regular and background noise test. Figure 5.2 (a) and (b) shows plots of the observed CPU cycles from the regular and background noise test, respectively.

If we compare the regular test results we got from using `FileStreamerGM` with the results from this test, we can see from table 5.1 that we, in average, have saved 1500 CPU cycles by not using virtual functions, which is a 5% gain in performance. Comparing the plots from the two tests, figure 5.1 (a) and 5.2 (a), it is not easy to see any differences, but it looks like the observed values from this test are more compact. The MAD and the width of the two data sets tell us the opposite though. MAD is calculated to be 328 CPU cycles and width of 75% of observed values is 1500 CPU cycles, which is 6.5% and 4.1% (20 and 60 CPU cycles) higher than of the results from `FileStreamerGM`, respectively. This is not shown on the plots because of the scale of Y-axis. If we were to calculate the SD for both data sets, it would coincide with what we observe in the plots because the time measure test using `FileStreamerGM` has a higher frequency of high values. To be precise, the SD for the time measure test using `TunedFileStreamerGM` and `FileStreamerGM`, would be 2392 and 3314 CPU cycles respectively. The reason why the dispersion of this time measure test is higher than the time measure using `FileStreamerGM` is most probably due to variability in the time measures. As we will see in section 5.4.3, the true value for the MAD and mean measures will vary from one test to another, so to get a good estimate of the true mean and MAD we need more and longer tests using the same graph manager. Doing this, we can achieve a good confidence interval.

When we added background noise to the measurements, the results we got followed a similar pattern as the results we got from `FileStreamerGM`, as can be seen in figure 5.2 (b), for this test, and 5.1 (b) for the test using `FileStreamerGM`. From row 2 in table 5.1 we see that the mean for the background noise test lies 9.5%, (2670 CPU cycles) above the regular test, which is 1.6% lower than the same difference when we use `FileStreamerGM`. If we look at the small belt of values that lies around and over 100000 CPU cycles, including the spikes, in both figure 5.2 (b) and 5.1 (b), we see that the high values from this test have sunk by far more than the mean of the whole data set, compared to the test using `FileStreamerGM`. Taking the mean for these high data values, for this and the test using `FileStreamerGM`, we get 127502 and 142422 respectively. In other words, these high values have sunk 10.5% (14920 CPU cycles) in average when we removed the use of virtual functions. This result indicates that virtual functions have an increasing cost related to how much of the cache that is over-written by other processes. If much of the cache is over-written, it takes many CPU cycles to use the virtual functions pointers.

The dispersion for the background noise test is calculated to be, using MAD, 353 CPU cycles. This is 3.6% (13 CPU cycles) better than the same test using `FileStreamerGM`. The spread of the data has improved a bit when there are many processes competing for the processor.

TunedFilePrefixedStreamerGM

Together with removing the use of virtual functions, the stream handler used by `TunedFile-PrefixedStreamerGM` has removed the code that was associated with the reading of data from disk into user space. By doing this, we have also removed a context switch associated with the read system call. The result of using this graph manager is plotted in figure 5.3 (a) and (b), and in row 3 in table 5.1.

Looking at table 5.1 and comparing row 1 and 3 for the regular test, we can see that we in average have saved 23% (6570 CPU cycles). This is a combination of removing virtual function,

a context switch and code that handled reading the file from disk in Komssys. The performance gain from just removing the code and a context switch is found by comparing this test with the test done with `TunedFileStreamerGM`, since that graph manager also has removed the use of virtual functions. The difference between the mean in row 2 and 3 is 18.8% (5290 CPU cycles). Most of this due to the removal of some code in Komssys, but some also from avoiding a context switch.

In figure 5.3 (a), there is a noticeable difference to the dispersion compared to figure 5.1 (a). We can see that figure 5.3 (a), is more compact, where 95% of the observed data is actually centered in a belt with a width of 3520 CPU cycles and where the minimum value is 21080. This causes the MAD to be quite low compared to the other test we have run, only 239 CPU cycles. Compared to the test using `FileStreamerGM`, this is an improvement of 26.8% (1290 CPU cycles) on the width and 22.4% (69 CPU cycles) on the MAD.

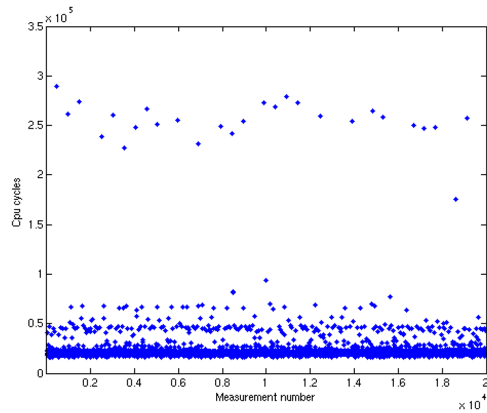
Introducing background noise into the test did not give us any surprises. We see the same patterns in figure 5.3 (b) as we have with the other background noise tests. But we can see that the high observed values around and over 100000 CPU cycles have decreased even more compared to the test using `TunedFileStreamerGM`. Taking the mean for these values gives us 110010 CPU cycles, over 22.8% (32412 CPU cycles) lower than `FileStreamerGM` used in average, when instruction and data cache gets over-written. This comes from the fact that we have trimmed down the code plus the removal of virtual functions. The dispersion for the whole data set from the background noise test is calculated to be 261 using the MAD, an improvement of 28.6% (105 CPU cycles) compared to the test using `FileStreamerGM`.

5.4.2 Time used in kernel space

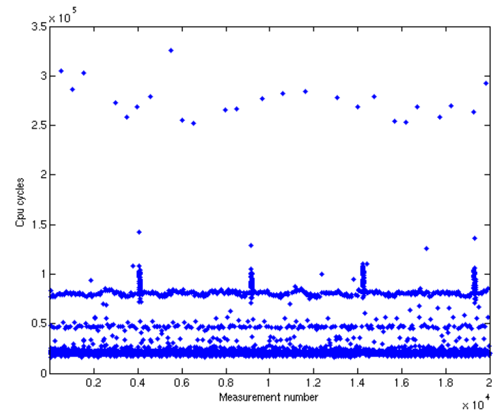
		Min. [cycles]	1st Qu. [cycles]	Median [cycles]	Mean [cycles]	3rd Qu. [cycles]	Max. [cycles]
FSGM	**Regular**	15880	19280	19640	20450	20080	289500
	Noise:	16420	19540	19880	22720	20360	325400
TFSGM	**Regular:**	15850	18920	19250	20140	19710	1123000
	Noise:	16490	19590	19940	22730	20390	326300
TFPSGM	**Regular:**	17120	19960	20320	21490	21730	280200
	Noise:	17030	20250	20730	24070	22300	352300

Table 5.2: Statistics of the number of CPU cycles used in kernel space from regular and noise test.

When we used the time measure tool described in chapter 4, to do the tests in section 5.4, we also got hold of how much time spent in kernel space for each of the tests. Statistical measures of the results have been put into table 5.2.

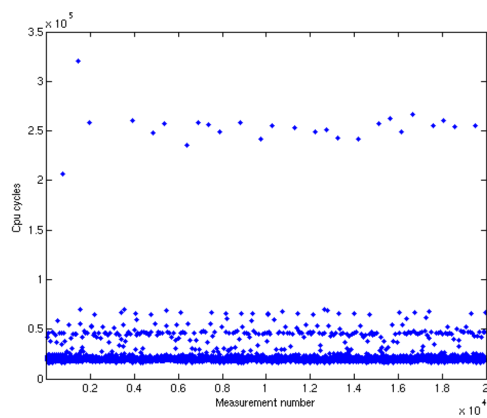


(a) Regular

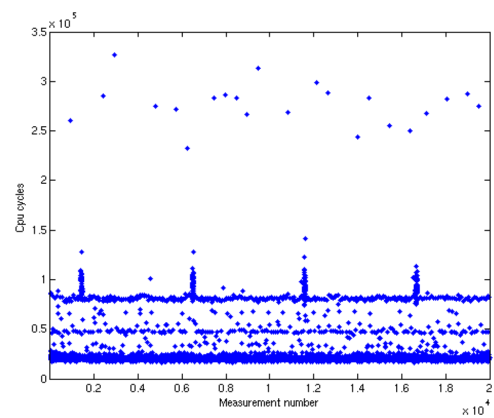


(b) Background noise

Figure 5.4: Plot of CPU cycles used with `FileStreamerGM` in kernel space.

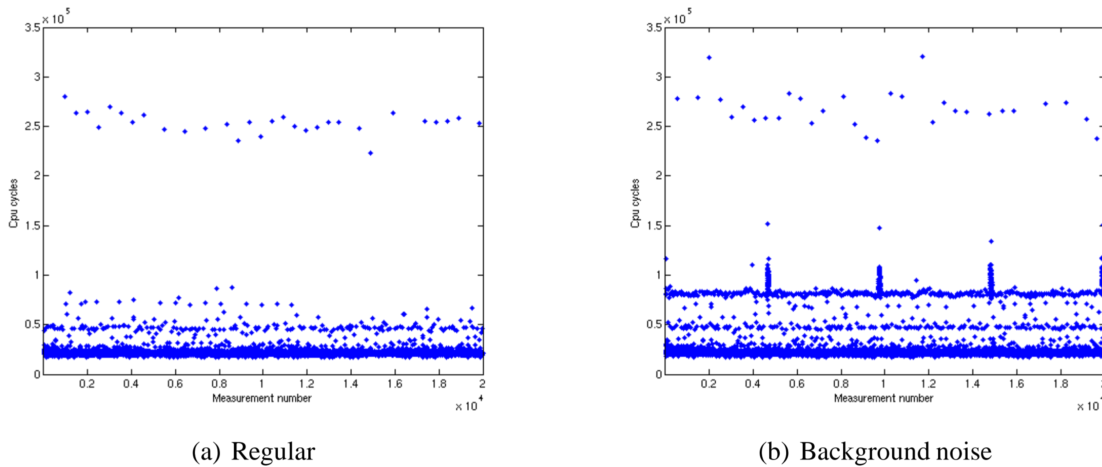


(a) Regular: kernel processing



(b) Background noise: kernel processing

Figure 5.5: Plot of CPU cycles used with `TunedFileStreamerGM` in kernel space.



(a) Regular (b) Background noise

Figure 5.6: Plot of CPU cycles used with `TunedFilePrefixedStreamerGM` in kernel space.

FileStreamerGM

Figure 5.4 shows the test results from kernel space using `FileStreamerGM` for a regular test (a) and a background noise test (b). We see that the measures where the process has been scheduled out, stick out from the rest on both plots by being much higher. We can also see the characteristic of cache over-writes in figure 5.4 (b). Though in kernel space, the distance between the belt of high values, including the spikes, is much closer to the rest of the observed values which might indicate that the kernel code is not so sensitive to cache changes. We can also see that there is a second belt of values on all the measures, with all the graph managers actually. This belt lies between the the spikes and the rest of the observed values at approximately 50000 CPU cyles. This might be from smaller cache over-writes, or it might come from disk reading.

Summing the time used in kernel space and user space for this test, with the median as a measure for the typical value of the two data sets, we find that `FileStreamerGM` spends 40.3% and 40.5% of its time in kernel space, for the regular and background noise test respectively. The reason for using the median instead of the mean in this case is because the kernel space measures have extreme outliers related to when the process has been scheduled out.

The data dispersion for both measures taken in kernel space are higher, compared to the dispersion from user space, looking at plots in figures 5.1 and 5.4. The MAD is calculated to be 572 CPU cycles for the regular test and 586 for the background noise test. This high dispersion of the data is due to the fact that the time used reading from disk has a greater variability than only reading from memory and cache.

TunedFileStreamerGM

The result of the kernel time measurements for regular and background noise test with `TunedFileStreamerGM`, is depicted in figure 5.5. As before, there are no real surprises here, the observed values are visually similar distributed as the results we got from the kernel

space times using FileStreamerGM. This is only natural as the use of system calls in both graph managers are the same. But, can we say that the observed values from the two time measures test, taken in kernel space, are identically distributed? Table 5.2 indicates that this is not likely for the regular tests, but might be true for the background noise tests. In a perfect world where the test run has had identical conditions, then perhaps we could conclude that two data sets follow the same probability distribution with probability p, for every test done with these graph managers. But the measures are run under different conditions, even though they are small, and the test results may be affected by this. Differences that may affect the measure is when we start the measure, memory access times, the degree of cache over-writes and the code for running FileStreamerGM is different from running TunedFileStreamerGM.

Going a bit deeper, we decided to test the hypotheses H_0 that the observed values, in kernel space, for the two graph managers come from the same distribution. To check this, we run a Mann-Whitney test [18] in R by sampling with replacement, 1000 values from the data set created using FileStreamerGM and from TunedFileStreamerGM. We tested the regular tests against each other and the background noise test against each other. To exclude any lucky sampling, we run a Monte Carlo simulation with 10000 iterations, and then calculated the mean of all the gathered p-values. For the regular tests, the data sets are too different and we can conclude that the values do not come from the same distribution. For the tests done with background noise, the p-value was found to be 0.25, and so can we accept H_0. These results may change if we gather new time measure data because the conditions under which we take the time measures are so variable.

The time spent in kernel space for this graph manager is 41% and 42.3%, for the regular and background noise test respectively, when we use the median as an estimation to find the most typical value. This is 0.7% and 1.8% more than FileStreamerGM uses, a consequence of that we have removed the use of virtual functions and therefore spend less time in user space.

TunedFilePrefixedStreamerGM

The kernel space measures for TunedFileStreamerGM have been plotted, for regular and background noise test, in figure 5.3. Both plots are similar to the other kernel space measures we have done. If we look at table 5.2, the measures done with TunedFilePrefixedStreamerGM are a bit higher than the two other tests, for all statistical measures except the maximum value. The difference between this graph manager and FileStreamerGM, looking at the median, is 3.5% (680 CPU cycles) and 4.3%(850 CPU cycles) for regular and background noise test respectively, in favour of FileStreamerGM. This difference comes from the fact we probably do more work in the kernel with this graph manager than in the other two graph managers. With *sendfile_prefixed()* we have to issue *cork()* and *uncork()* to assemble the RTP packet before it is sent out on the net. In total, when we add up the time spent in kernel space and user space for FileStreamerGM and TunedFilePrefixedStreamerGM, we save 11.8% ($6570 - 680 = 5890$ CPU cycles) in total on a regular test, using TunedFilePrefixedStreamerGM instead of FileStreamerGM.

Calculating the MAD for both regular and background noise test, 1131 and 832 CPU cycles, shows that there is high variability in the the *sendfile_prefixed()* code.

5.4.3 Estimating variability of MAD and mean

Doing time measures can give, as we have seen, variable result from time to time, when we measure over a large distance of code. As such, it is interesting to find the variability of the location estimates we have looked at, which are mainly the MAD and mean. Finding the variability allows us to set up confidence intervals for the the location estimates and thus we can see the interval where the true value of the two estimates lie.

One way of finding the variability of the location estimates is with a method called bootstrapping [18]. With this method we view the empirical cumulative distribution function (cdf) F_n of the data set as an approximation of the real underlying cdf F. To find, for example the SD for a location estimator $\hat{\theta}$, we sample B samples of size n uniformly from F_n with replacement, giving each observed value $x_1, x_2, ..., x_n$ a probability of $1/n$ for being drawn. For each n sized sample we calculate the location estimator, giving us $\theta_1^*, \theta_2^*, ..., \theta_n^*$,, for which we then can calculate the SD as follows:

$$s_{\hat{\theta}} = \sqrt{\frac{1}{B}\sum_{i=1}^{n}(\theta_i^* - \bar{\theta}^*)^2}$$

When B is big enough, and if we assume that $\theta_1^*, \theta_2^*, ..., \theta_n^*$, are random, independent and identical distributed, we can use the result from the Central Limit Theorem [18] which tells us that the distribution of $\theta_1^*, \theta_2^*, ..., \theta_n^*$, tends to the normal distribution, and use $s_{\hat{\theta}}$ to create a confidence interval. In figure 5.7 we have created a histogram of all the theta stars we created for the MAD, with data from a regular test using `FileStreamerGM`, and as we can the characteristic shape of the normal distribution is present. A more thorough explanation is found in [18].

In table 5.4.3 and 5.4.3 we have put in 95% confidence intervals for the MAD and mean for all the test we have done, in user space and kernel space respectively, with $B = 1000$. These intervals tells us with probability 0.95 that the real value of MAD or mean will lie in this interval. It is important to have in mind though, that these intervals are not valid for every time measure test we run. There is too much variability in the conditions under which we have run the tests, to claim that all the measures will have MAD and mean in these intervals. The data from different tests, using the same graph manager, might come from the same distribution, but the parameters of the distribution vary from time to time. In order to find intervals where the MAD and mean will lie with probability 0.95, for all time measure tests using the same graph manager, we would have to run, for example, 1000 time measure tests for each graph manager. Then, for each measure calculate the MAD and mean. When we have gathered 1000 such location estimators, we run the bootstrap method on those values. But this would probably take weeks if we should have done it for all the tests we have done, and that is a luxury we do not have.

The confidence intervals

We can see from the confidence intervals in table 5.4.3 and 5.4.3 that the width of the intervals are bigger in the measures taken in kernel space, than in user space. This is because the variability of the measures in kernel space is bigger since the hard disk is used. In user space the intervals are bigger when we have introduced background noise into the measures because then there are more variability in the measures. This is partly true in kernel space also, but not

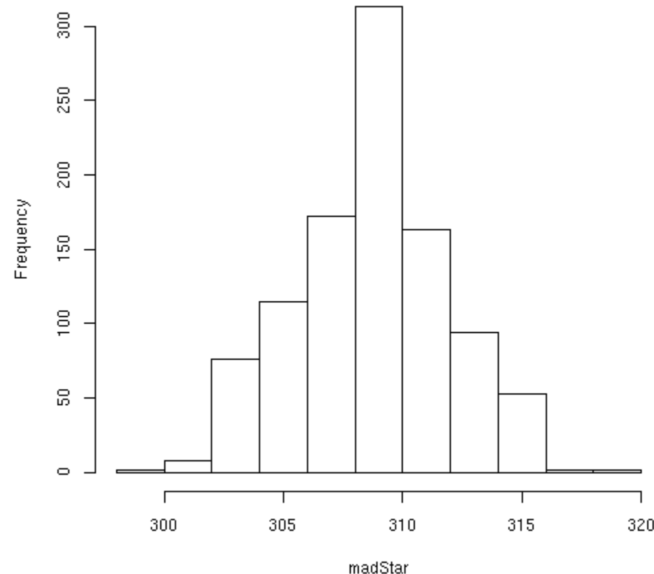


Figure 5.7: A histogram of the theta stars for the MAD.

for the MAD. This is probably because, as we know, the MAD is a robust measure which is not much affected by the high measures we encounter in kernel space when the process is scheduled out. The mean on the other hand is more sensitive to high measures and therefore the interval is wider.

5.4.4 Overhead using time measure tool

It is important to know how much the measure tool itself afflicts the time measure. The overhead should be very small compared to the result of the time measure. To find this overhead we need to know how many CPU cycles are used for the time measure; saving and restoring registers, calling the functions responsible for creating time stamps and time stamp creation itself.

We calculate how many CPU cycles are used by running a test similar to the one we used to calculate the time used on a context switch. We start the time measure with a call to *start_time_measure()* before we enter a loop which issues an empty implementation of *sendfile_prefixed()* 20000 times. We use rdtsc and cpuid as we did with the context switch time measure. For each iteration the call to *sendfile_prefixed()* will be intercepted by the time measure code in the kernel. It will also intercept when the system call is done and is about to return to user space. The time we get using rdtsc in the for loop gives us the time it takes to call an empty system call when we have a time measure enabled kernel.

If we now repeat this test without the call to *start_time_measure()* and without a time measure enabled kernel, subtract the minimum values from both measures, we will have the number CPU cycles the time measure code in the kernel uses when intercepting a system call. The code that intercepts interrupts and page faults are very similar to the system call intercept code, so the number of CPU cycles to intercept these kernel entries are thus approximately the

		MAD	Mean.
FSGM	**Regular**	[302, 314]	[29530, 29620]
	Noise:	[359, 374]	[32980, 33530]
TFSGM	**Regular:**	[322, 333]	[28050, 28120]
	Noise:	[346, 360]	[30510, 30990]
TFPSGM	**Regular:**	[235, 243]	[22760, 22820]
	Noise:	[255, 267]	[25130, 25560]

Table 5.3: 95% confidence intervals for MAD and Mean measures in user space.

		MAD	Mean.
FSGM	**Regular**	[562, 583]	[20330, 20580]
	Noise:	[575, 596]	[22510, 22930]
TFSGM	**Regular:**	[558, 578]	[19950, 20320]
	Noise:	[569, 588]	[22520, 22950]
TFPSGM	**Regular:**	[799, 865]	[21360, 21620]
	Noise:	[1083, 1158]	[23860, 24290]

Table 5.4: 95% confidence intervals for MAD and mean measures in kernel space.

same. The interception when returning from one of these kernel entries is handled by the same function, and thus also approximately the same. The result for the time used by system call interception kernel code was found to take 86 CPU cycles.

Time measure tool overhead in Komssys measures

When we tested the different graph managers and the proposed changes to Komssys, we gathered some statistics which are listed in table 4.1.

We will use the statistics to show that the overhead from the measure tool code is very small compared to the CPU cycles used on the measure itself. We have put the median of system calls, interrupts and page fault into table 5.5, taken from the regular time measure test using `FileStreamerGM`. We can see that it is only system calls that in the long run affect the measure. In average, using the median as measure, the time measure only incur an overhead of 430 CPU cycles, which is very small compared to the measure results in table 5.1 and 5.2.

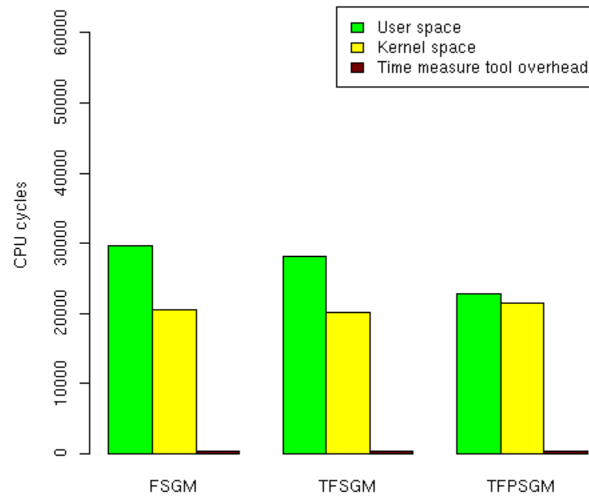


Figure 5.8: The user space, kernel space and time measure tool overhead (in CPU cycles) for each measure for each graph manager.

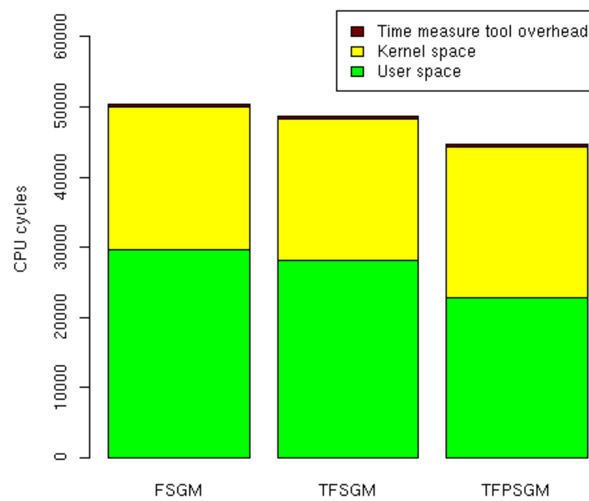


Figure 5.9: The user space, kernel space and time measure tool overhead (in CPU cycles) for each measure added together in one column, for each graph manager.

	FSGM	TFSGM	TFPSGM
system calls:	5	5	4
page faults:	0	0	0
interrupts:	0	0	0
overhead (cycles):	430	430	344

Table 5.5: The medians of system calls, interrupts and page faults taken from the regular measures.

5.5 Total performance

In figure 5.8 and 5.9 we have added the time used in user space and kernel space and the overhead of the time measure tool for the three graph managers. On figure 5.8 we see that `TunedFilePrefixedStreamerGM` uses more time in the kernel than the other two graph managers, but uses far less in user space. In figure 5.9 we see that the total time used for all the graph managers. There is a considerable performance gain using `TunedFilePrefixedStreamerGM` compared to `FileStreamerGM`, but we see that the performance gain using `TunedFileStreamerGM` is not so high. The total CPU cycles used for the three graph managers are 50460 (FSGM), 48650 (TFSGM) and 44620 (TFPSGM), which include the overhead from the time measure.

5.6 Evaluation

We have been running several tests that demonstrates the effect of the proposed changes to Komssys. We have seen that the removal of virtual functions do save some CPU cycles, but far from enough to say that the stream handler architecture has a bottleneck when it comes to performance by using virtual functions when the CPU load is low. When the CPU load gets high on the other hand, we saw that the cost of using virtual functions increase. So, in a heavily loaded system, we can save many CPU cycles by not using virtual functions, but then again a media server should not be heavily loaded by other processes.

By using *sendfile_prefixed()* in `TunedFilePrefixedSH` we were able to show that assembling the RTP packet in the kernel is done more efficiently than doing it in user space. This increase in efficiency is a combination of the changes we made in `TunedFilePrefixedSH` to remove code related to reading data from disk, and the fact that *sendfile_prefixed()* does the reading of data from disk more efficiently by not copying it into an user space buffer. Since the effect of introducing *sendfile_prefixed()* into Komssys was so good, it should be considered using it instead of the ordinary data path that is used today when streaming a file from disk using `FileStreamerGM`. To keep the generality we get using virtual functions, a good solution would be to use virtual functions as `FileStreamerGM` does, and use *sendfile_prefixed()* to send the packets to the client, since it was by introducing *sendfile_prefixed()* we saved most CPU cycles. `FileStreamerGM` will then be reduced to two stream handlers, one for creating the RTP data and a sink. The source stream handler will be removed since *sendfile_prefixed()* is responsible for reading the file from disk. This breaks the stream handler architecture which stipulates that data is generated, or comes, from a source stream handler and exits through a sink stream handler. A fix to this small problem would be to have a dummy source stream

handler that just creates an empty RTP packet which then is filled with RTP information in `RTPEncoderSH`.

We saw that the context switch did not take so many CPU cycles, only 197, if we compare it to the time spent on the time measures. This means that the removal of a context switch, by using *sendfile_prefixed()*, does not reduce the amount of CPU cycles used much, compared to removal of code related to the *read()* system call and by not having to copy the data from the hard disk into an user space buffer.

5.7 Summary

In this chapter we have tested the proposed changes to Komssys and discussed the result. We saw that the combination of removing virtual functions and introducing *sendfile_prefixed()* helped us save, in average on a regular test, 5890 CPU cycles on reading a chunk of data from disk, packing it in a RTP packet and sending it out on the net. We have also seen that these results vary and that the real value of a location estimator is located in an interval for a given time measure, but to find the interval of a location estimator for all time measures, we would have to do many more tests. We looked at the time it takes to perform a context switch to see how many CPU cycles we save when using *sendfile_prefixed()*, and last we looked at the overhead of using the time measure tool.

Chapter 6

Conclusion and remaining challenges

In this report we have looked at the cost of using virtual functions in the stream handler architecture, and the effect of using *sendfile_prefixed()*.

6.1 Conclusion

Virtual functions are used in the stream handler architecture to achieve a generality that allow us to easily create new stream handlers and plug them into the composition. To find the cost of this generality, we created a new stream handler in Komssys that is a fusion of three stream handlers, removing the use of virtual functions. This new combined stream handler was then used by a new graph manager we created, time measured and compared with the original, three stream handler version with virtual functions. The tests showed us that under small CPU loads, the virtual functions is not costly enough to not use them. But, as the CPU load increases, the cost of virtual functions increase also, so on a heavily used server, the virtual functions should be avoided.

Assembling the RTP packet in kernel space with *sendfile_prefixed()* showed to be very effective compared to doing it in user space. We saw that we actually increaded performance by 11.8% (5890 CPU cycles) in average on each time measure using *sendfile_prefixed()* and removing virtual functions. This is such a good improvement that we should use it, and abandon the standard way of assembling the RTP packet in user space. To keep the generality we get from the use of virtual functions and to conform with the stream handler architecture, we suggest that we should still use virtual functions and have a dummy source stream handler that just creates an empty RTP packet.

6.2 Remaining challenges

It would be interesting to test how well a variation of `FileStreamerGM`'s stream handlers do, where we use virtual functions, *sendfile_prefixed()* and a dummy source stream handler, compared to the test result we have already gathered. It would also be interesting to repeat the test we have run maybe 100 times, to get enough data to create a interval where the mean CPU cycles lies, for every test run.

Appendix A

Concurrent background processes

A.1 Processes in regular test

These are the processes that was running alongside Komssys when we were doing time measures on a regular test.

```
  PID TTY      STAT   TIME COMMAND
    1 ?        Ss     0:00 /sbin/init splash
    2 ?        SN     0:00 [ksoftirqd/0]
    3 ?        S      0:00 [watchdog/0]
    4 ?        S<     0:00 [events/0]
    5 ?        S<     0:00 [khelper]
    6 ?        S<     0:00 [kthread]
   26 ?        S<     0:00  _ [kblockd/0]
   42 ?        S<     0:00  _ [kseriod]
   79 ?        S      0:00  _ [pdflush]
   80 ?        S      0:00  _ [pdflush]
   81 ?        S<     0:00  _ [kswapd0]
   82 ?        S<     0:00  _ [aio/0]
 1594 ?        S<     0:00  _ [khubd]
 1676 ?        D<     0:00  _ [kjournald]
 2629 ?        S<     0:00  _ [kpsmoused]
 2672 ?        S<     0:00  _ [kgameportd]
 1746 ?        Ss     0:00 //sbin/logd
 1882 ?        S<s    0:00 /sbin/udevd --daemon
 3440 tty1     Ss     0:00 /bin/login --
 4598 tty1     S      0:00  _ -bash
 4727 tty1     T      0:00      _ emacs -nw ../../bin/medianode2.xml
 4868 tty1     R+     0:00      _ ps fax
 3441 tty2     Ss+    0:00 /sbin/getty 38400 tty2
 3443 tty3     Ss+    0:00 /sbin/getty 38400 tty3
 3444 tty4     Ss+    0:00 /sbin/getty 38400 tty4
 3445 tty5     Ss+    0:00 /sbin/getty 38400 tty5
 3446 tty6     Ss+    0:00 /sbin/getty 38400 tty6
 3522 ?        Ss     0:00 /sbin/syslogd
```

```
3548 ?        Ss     0:00 /bin/dd bs 1 if /proc/kmsg of kmsg[1]
3550 ?        Ss     0:00 /sbin/klogd -P /var/run/klogd/kmsg
3662 ?        Ss     0:00 /usr/sbin/cupsd
3693 ?        Ss     0:00 /usr/sbin/hpiod
3932 ?        Ss     0:00 /usr/bin/dbus-daemon --system
3947 ?        Ss     0:02 /usr/sbin/hald
3948 ?        S      0:00  _ hald-runner
3957 ?        S      0:00      _ /usr/lib/hal/hald-addon-keyboard
3960 ?        S      0:00      _ /usr/lib/hal/hald-addon-keyboard
3972 ?        S      0:00      _ /usr/lib/hal/hald-addon-storage
3974 ?        S      0:00      _ /usr/lib/hal/hald-addon-storage
3995 ?        S      0:00 perl SystemToolsBackends.pl[2]
4126 ?        Ss     0:00 /usr/lib/postfix/master
4145 ?        S      0:00  _ pickup -l -t fifo -u -c
4146 ?        S      0:00  _ qmgr -l -t fifo -u
4317 ?        Ss     0:00 /usr/sbin/atd
4330 ?        Ss     0:00 /usr/sbin/cron
```

A.2 Processes in background noise tes

These are the processes that was running alongside Komssys when we were doing time measures on a background noise test.

```
  PID TTY      STAT   TIME COMMAND
    1 ?        Ss     0:00 /sbin/init splash
    2 ?        SN     0:00 [ksoftirqd/0]
    3 ?        S      0:00 [watchdog/0]
    4 ?        S<     0:00 [events/0]
    5 ?        S<     0:00 [khelper]
    6 ?        S<     0:00 [kthread]
   26 ?        S<     0:00  _ [kblockd/0]
   42 ?        S<     0:00  _ [kseriod]
   79 ?        S      0:00  _ [pdflush]
   80 ?        S      0:00  _ [pdflush]
   81 ?        S<     0:00  _ [kswapd0]
   82 ?        S<     0:00  _ [aio/0]
 1594 ?        S<     0:00  _ [khubd]
 1676 ?        S<     0:00  _ [kjournald]
 2638 ?        S<     0:00  _ [kpsmoused]
 2863 ?        S<     0:00  _ [kgameportd]
 1746 ?        Ss     0:00 //sbin/logd
 1882 ?        S<s    0:00 /sbin/udevd --daemon
 3445 tty1     Ss+    0:00 /sbin/getty 38400 tty1
 3446 tty2     Ss+    0:00 /sbin/getty 38400 tty2
```

[1] /var/run/klogd/kmsg

[2] /usr/share/system-tools-backends-2.0/scripts/SystemToolsBackends.pl

```
3447 tty3     Ss+    0:00 /sbin/getty 38400 tty3
3448 tty4     Ss+    0:00 /sbin/getty 38400 tty4
3449 tty5     Ss+    0:00 /sbin/getty 38400 tty5
3450 tty6     Ss+    0:00 /sbin/getty 38400 tty6
3525 ?        Ss     0:00 /sbin/syslogd
3551 ?        Ss     0:00 /bin/dd bs 1 if /proc/kmsg of kmsg
3553 ?        Ss     0:00 /sbin/klogd -P /var/run/klogd/kmsg
3625 ?        Ss     0:00 /usr/sbin/gdm
3632 ?        S      0:00  _ /usr/sbin/gdm
3635 tty7     Ss+    0:00      _ /usr/X11R6/bin/X :0
4128 ?        Ss     0:00      _ /bin/sh /home/kjellso/.xsession
4335 ?        Ss     0:00          _ /usr/bin/ssh-agent
4418 ?        S      0:00          _ fvwm
4436 ?        S      0:00              _ /usr/X11R6/bin/xterm
4437 pts/1    Ss     0:00                  _ bash
3665 ?        Ss     0:00 /usr/sbin/cupsd
3696 ?        Ss     0:00 /usr/sbin/hpiod
3699 ?        S      0:00 python /usr/sbin/hpssd
3752 ?        S      0:00 /bin/sh /usr/bin/mysqld_safe
3816 ?        Sl     0:00  _ /usr/sbin/mysqld
3817 ?        S      0:00  _ logger -p daemon.err -t mysqld_safe
3934 ?        Ss     0:00 /usr/bin/dbus-daemon --system
3949 ?        Ss     0:02 /usr/sbin/hald
3950 ?        S      0:00  _ hald-runner
3959 ?        S      0:00      _ /usr/lib/hal/hald-addon-keyboard
3962 ?        S      0:00      _ /usr/lib/hal/hald-addon-keyboard
3974 ?        S      0:00      _ /usr/lib/hal/hald-addon-storage
3976 ?        S      0:00      _ /usr/lib/hal/hald-addon-storage
3997 ?        S      0:00 perl SystemToolsBackends.pl
4008 ?        Ss     0:00 /usr/bin/SCREEN -S hellanzb
4018 pts/0    Ssl+   0:00  _ /usr/bin/python /usr/bin/hellanzb.py
4049 ?        Ss     0:00 /opt/usit/wpa_supplicant/wpa_supplicant
4132 ?        Ss     0:00 /usr/lib/postfix/master
4151 ?        S      0:00  _ pickup -l -t fifo -u -c
4152 ?        S      0:00  _ qmgr -l -t fifo -u
4278 ?        Ss     0:00 /usr/sbin/anacron -s
4297 ?        Ss     0:00 /usr/sbin/atd
4311 ?        Ss     0:00 /usr/sbin/cron
4338 ?        S      0:00 /usr/bin/dbus-launch
4370 ?        Ss     0:00 /usr/bin/dbus-daemon
4429 ?        Ss     0:00 /sbin/dhclient eth0
```

Bibliography

[1] Andrew P. Black, Jie Huang, Rainer Koster, Jonathan Walpole, and Calton Pu. Infopipes: An abstraction for multimedia streaming. *Multimedia Systems*, pages 406–419, 2004.

[2] Carsten Griwodz and Michael Zink. Dynamic data path reconfiguration. In *M3W: Proceedings of the 2001 international workshop on Multimedia middleware*, pages 72–75, New York, NY, USA, 2001. ACM Press.

[3] Marco Lohse, Michael Repplinger, and Philipp Slusallek. An open middleware architecture for network-integrated multimedia. In *Protocols and Systems for Interactive Distributed Multimedia: Joint International Workshops on Interactive Distributed Multimedia Systems and Protocols for Multimedia Systems, IDMS/PROMS 2002, Coimbra, Portugal, November 26-29, 2002. Proceedings*, pages 327–338. Springer Berlin / Heidelberg, 2002.

[4] `http://www.live555.com/`. Live555 internet streaming media, wireless, and multicast technology, services, & standards.

[5] Daniel P. Bovet and Marco Cesati. *Understanding the LINUX KERNEL.* O'Reilly, November 2005.

[6] http://lwn.net/Articles/132196/. An introduction to kprobes.

[7] Wim Taymans, Steve baker, Andy Wingo, Ronald S. Bultje, and Stefan Kost. Gstreamer application development manual (0.10.12.1).

[8] Marco Lohse and Michael Repplinger and Philipp Slusallek. Dynamic Distributed Multimedia: Seamless Sharing and Reconfiguration of Multimedia Flow Graphs. In *Proceedings of the 2nd International Conference on Mobile and Ubiquitous Multimedia (MUM 2003)*, pages 89–95. ACM Press, 2003.

[9] `http://tools.ietf.org/html/rfc4566`. Rfc4566 - sdp: Session description protocol.

[10] `http://www.mplayerhq.hu/`. Mplayer.

[11] `http://www.ietf.org/rfc/rfc1889.txt`. Rfc1889 - rtp: A transport protocol for real-time applications.

[12] T. A. Dalseng P. Halvorsen and C. Griwodz. Assessment of linux' data path implementations for download and streaming. *International Journal of Software Engineering and Knowledge Engineering*, August 2007.

[13] Juan-Mariano de Goyeneche and Elena Apolinario Fernández de Sousa. Loadable kernel modules. *IEEE Software*, January/February 1999.

[14] Prasanna S Panchamukhi <prasanna@in.ibm.com> Jim Keniston <jkenisto@us.ibm.com>. Kernel probes (kprobes). Linux Documentation.

[15] http://developer.intel.com/drg/pentiumII/appnotes/RDTSCPM1.HTM. Using the rdtsc instruction for performance monitoring.

[16] `http://www.mathworks.com/`. Matlab.

[17] `http://www.r-project.org/`. The r project for statistical computing.

[18] John A. Rice. *Mathematical Statistics and Data Analysis Second Edition*. Duxbury Press, 1995.