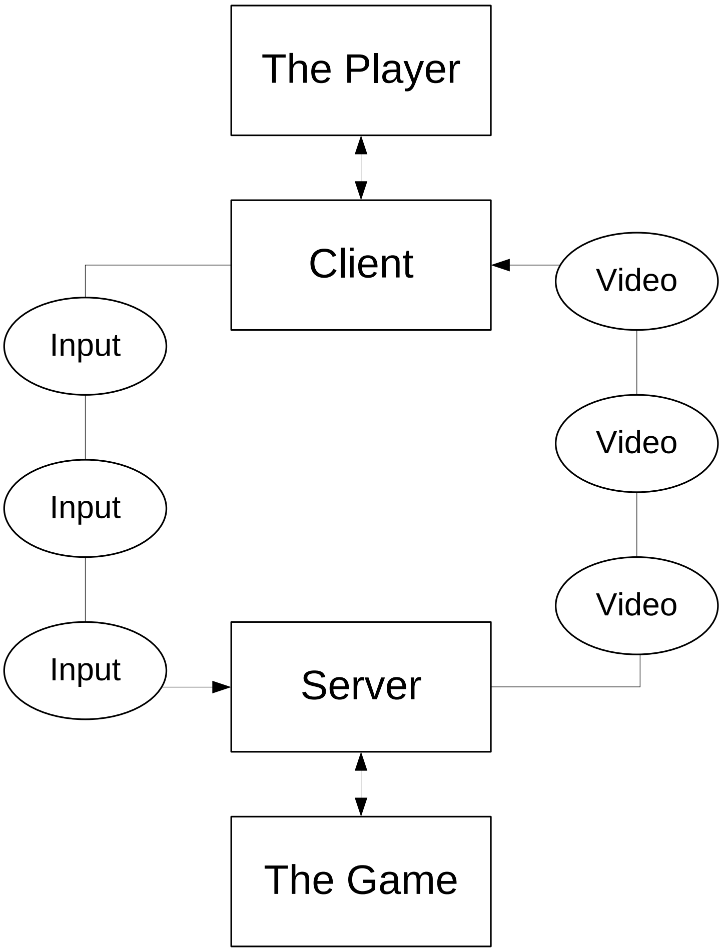
Game On Demand Service



Architecture Proposal & Prototype

DAT5-project by
– Group d501a–
Jais Heslegrave & Thomas Justesen

January 3, 2011 at Aalborg University

Department of Computer Science
Aalborg University
Selma Lagerlöfs Vej 300
Phone +45 9940 9940
Fax +45 9940 9798
`http://www.cs.aau.dk`

Student Report

Titel:
Game On Demand Service
— Architecture Proposal & Prototype...

Theme:
Distributed System, Game on Demand

Project period:
DAT5, fall 2010

Project group:
d501a

Authors:
Jais Heslegrave
Thomas Justesen

Supervisor:
René Rydhof Hansen

Printcount:
4

Nr. of pages:
52

Appendix:
A cd with source and executable and a list of acronyms.

Completed & signed:
January 3, 2011 at Aalborg University

Abstract:

Game On Demand (GoD), a service that enables for streaming of games. The idea has risen from the Video On Demand (VoD)-service, which allows a user to rent, buy or just view a streamed video on any capable receiver. The same can be done for games, given the user has a device capable of accessing an ip-network, viewing streamed video and intercepting and sending enough inputs for the game to function. Such a service could be of interest for both consumers and for the game developers.

This report is minded on the network aspects of a GoD-service. It proposes a client/server architecture for a possible finally implemented solution. The report also evaluates on described prototype implementations of key components of the proposed architecture.

The evaluation has lead to believe, that a GoD service is indeed possible and plausible. For it to function properly however, it demands a lot of optimizing of various functionalities in all parts of the service. Even commercial implementations of GoD services that exist today, which have been under development for several years, suffer from a noticeable delay.

Contents

Preface

This report has been written during the DAT5-project period by group d501a at Aalborg University. The theme is “Distributed System, Game on Demand”.

References to sources are marked by [ABc#], where ABc# refers to the related literature in the bibliography at the end of the report.

The appendix to the report is found as the last chapter of the report and on a compact disc, located on the very last page of the report. Furthermore, the source code created during the project are on the disc.

The entire report is written in English and no translation will be accessible. Abbreviations and acronyms are at first appearance written in parentheses, to avoid breaking the reading stream. They are also found in a list in the appendix. The report is written in LaTeX and is accessible as a PDF-document.

A special thanks for e-mail correspondence with help, knowledge and code examples to Theofilos Karachristos [KAM08] and to Jori Liesenborgs for help on JRTPlib [Lie10], your knowledge on the subject was very helpful.

Signatures:

Jais Heslegrave

Thomas Justesen

Thomas Justesen

Chapter 1

Introduction

This chapter introduces the project itself, the report, the product and it specifies the problem statement.

Consider the concept of Video On Demand (VoD), where the users have the opportunity to rent and stream any available movie or video content directly to the living room. Thus providing a service, enabling users to watch movies without having to buy the movie or store them anywhere. Then imagine the same sort of system for video games. A user would be able to remotely play a video game that is streamed to the local device/computer. Such a solution would, in theory, render investments in newer high-level hardware unnecessary for the consumer. It might also save game developers the trouble of having to develop the games for several hardware-profiles. Such a service could be referred to as a Game On Demand (GoD) service.

A GoD-service will consist of a few key aspects for it to function properly and by the standards of today, it must be capable of streaming the game in high quality and without a noticeable delay. There will need to be a machine that serves the game, it has to be powerful enough to handle one or more of the games wanted in the service but also powerful enough to handle the sending/receiving mechanism. The server-side would also need a certain amount of bandwidth to handle both incoming and outgoing traffic, needed for the GoD-service. There are a number of possible solutions for actually streaming the game, they all have disadvantages and advantages and they will be considered in this report. The client theoretically needs some sort of input-controller (could be keyboard and mouse), it will need enough bandwidth to supply a fluid stream and sending the inputs and enough power to ensure fluid playback of the streamed game.

The main focus of this report is on the networking aspects of the system, so much attention will be paid to the protocols, network problems and network structure of the system. The project is not minded on creating a final solution, but on determining the feasibility of the service as an implemented solution, based on the knowledge accumulated by analyzing and prototyping the smaller aspects of this service. More on the goal of the project is found in the problem statement.

This chapter includes the introduction, as well as the problem statement describing the problems tackled in this project and what approaches are taken. It also describes some limitations as to what will and will not be developed. The second chapter analyses the problems and concerns related to the project, such as how to implement a streaming service, what protocols to use and which encoding schemes to utilize. Several other solutions to the game streaming service are also described. In the third chapter, the design of the architecture of the service is detailed, and each component of the service, as well as how they are supposed to interact is explained. The argument for the chosen protocols can also be found here. Chapter four details the implementation of the service, dealing with the third party libraries used, as well as the code used to mesh them together. The reasoning for choosing certain libraries is mentioned here as well. The final chapter evaluates on the project, the process and the implementation, and describes some of the concerns related to designing and implementing a finished service, this chapter also concludes on the project as a whole.

1.1 Problem Statement

The main focus for this project is a part of the specialization semester, where the goal is to learn more about distributed systems. This particular project is about the challenges encountered while developing a service for streaming games from a server to a client, thus allowing low-end computers to play high-end games. The project is also intended as a stepping stone for a Master's project. The project can be summarized as follows:

- Analyze the components necessary to construct a functional game streaming service.
  - Propose an architecture for such a service.
  - Compare different possible components for each role in the architecture.
- Implement non optimized key parts of the game streaming.
- Evaluate the performance possibilities of such a service.
  - Network characteristics.
  - Gameplay performance.

1.1.1 Limitations

Due to time constraints, and very basic knowledge of streaming implementation of the authors, the entirety of the proposed service in this report will not be implemented, instead there is a focus on getting the streaming part to work to some degree.

Chapter 2

Analysis

This chapter analyzes the various aspects needed for the project. This includes subjects such as coding of video, streaming of such, network distribution protocols, I/O handling and other existing solutions.

2.1 Streaming Methods

For a streaming service dedicated to streaming games and allowing remote clients to play, there exist several different possibilities for providing such a service. The chosen method will dictate what kind of client can use the service, but also effects the quality of the service.

2.1.1 3D Streaming

The 3D streaming approach utilizes the fact that many newer games use some form of graphics abstraction layer, such as Direct3D, OpenGL or Graphics Device Interface (GDI). The Application Programming Interface (API) calls from the games to these graphic libraries can be intercepted from the game process running on the server. Once intercepted, these API calls are transmitted to the client, the client will then pass these API calls to the graphics abstraction layer locally, to generate the graphics of the game on the client side. Audio has to be transmitted seperately and synchronized. This approach requires less processing time from the server, as the graphics API calls are intercepted and transmitted to the remote client. However, the client must be able process the graphics, so a very lightweight client is impossible in this setup. This method also requires that it is possible to get into the inner workings of the games, to intercept the graphics layer calls. An example of how this could work can be seen in Figure 2.1.

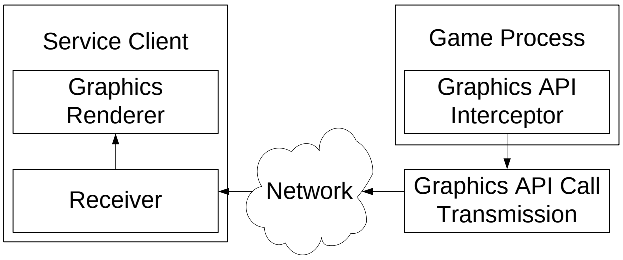


Figure 2.1: 3D Streaming Figure

2.1.2 Audio/Video Streaming

This method involves capturing the video and audio from the running game process on the server side using a capturing tool. When captured, this feed is transmitted to the client as a streaming video. This method requires the server to do more work as it has to encode the video and audio into an appropriate format before transmitting it to the client. The client only has to be able to decode and play the stream in order to use the service, input concerns not withstanding. The client can take the form of a seperate program, mobile application, or browser applet.

The method chosen for the service implemented in this project is the Audio/Video streaming approach, as the 3D Streaming requires intimate knowledge of the games and the graphics abstraction layer API in order to intercept API calls. In addition, tools exist to actually capture video and audio from running processes as detailed in the following section.

2.2 Video Capture

Video capturing tools are many and varied, but a more thorough look at some of the available tools is needed to select the right tool for the job. This section describes some of these tools, both proprietary and open source. Also some details relating to hooking into software and capturing from the framebuffer are included.

2.2.1 Capturing

Many different tools exist for capturing audio and video from other programs, for the purposes of making movies or streaming online. Most of them are typically used to upload tutorial videos or game footage to YouTube to show the world. XFire and UStream are services that allow you to record your desktop and stream it to online viewers. They are free to use, but are not open source, so it is difficult to go into further details about their implementation. Fraps is a different approach, it records from a running program and saves it as a video file on your computer, which you can later upload. It is, however, not free to use, and thus proprietary, which makes it impossible to find details about its inner workings. Taksi is similar to Fraps, in that it allows you to record video from a running application and save it as a video file locally. It also allows you to choose a Video For Windows (VFW) codec (which is a codec installed to be available for a multimedia framework developed by Microsoft, used by for instance Taksi) to encode the video, and even allows you to record from a single application at a time, and not the entire desktop. Taksi is also open source, allowing a closer look at the inner workings of the program, and ultimately allowing modification for use in this project.

2.2.2 Taksi & Hooking

Taksi has been chosen as the hooking, and capturing part of our solution, this is done because Taksi is open source and free, allowing examination and modification of the code to suit our needs.

Taksi works by utilizing hooking. This is a process called a hook, which can intercept function calls, messages or events, in the case of Taksi the hook is into a graphical program's DirectX, OpenGL or GDI API calls. It intercepts these calls and reads the frame from them, while also allowing the API calls to proceed as normal afterwards. Taksi now passes the intercepted frame to a chosen VFW encoder, and writes the finished encoded frame to the disk in a target file. Hooking works by Taksi injecting its library into the memory of the process to be hooked, it finds the appropriate places in the memory where the DirectX *Present* and *Reset* methods are called. Afterward the hooking method interjects a jump instruction to the Taksi library's frame capturing code, which is run first, before the DirectX API calls are allowed to proceed. Taksi captures the frame by copying the contents of the backbuffer to be displayed on the screen [KAM08]. The *Present* method of the DirectX API is used to display the contents of the program's backbuffer in the appropriate window or frontbuffer. The *Reset* method is used to clear the presentation parameters of the backbuffer, to alert of fullscreen changes or window resizing. Depending on the quality and speed of the chosen encoding, the frame capturing can actually slow the hooked process down. The captured frame is output in Red, Green & Blue (RGB) *4:4:4* format.

2.3 Video Coding

For the solution covered in this report, video streaming is used. There are several different formats and container types for video and audio, all specialized for their respective purpose. In this case, high resolution such as High Definition (HD) is a must, to keep up with the quality of a consumer game nowadays. It must, however, still be possible to stream through the Internet. Throughout this section the video coding standard and features are described, mostly with respect to the features that increase the coding's efficiency for streaming. Notice that the standards are all on the format of the coding and decoding of video formats, to give way for innovation in the area of encoding.

2.3.1 Choice of Standard

Since there are such a large number of possible video coding formats, the one chosen for the prototyping in this project, is based upon effectiveness, but also availability in open source and popularity. In a search for video streaming codecs, names like MPEG-1, MPEG-2, MPEG-4 and H.261–H.264 emerge, some might also think of the DivX codec. It is important to distinguish between file formats or container formats like AVI (and the widely used FLV, on webpages) and the encoding format of a video. The file format is a wrapper of one or more encoded streams (e.g. video, audio and maybe subtitles), while the encoding format is the format the actual video stream inside the file format, is stored in.

The question of what to choose for this given purpose, needs a little research. For instance, the format of DivX was originally intended as an alternative to DVD, as an attempt to compress an entire movie to the size of a CD-ROM (≈650 MB), instead of ≈4.7 GB on the DVD. Although DivX is possible for streaming purposes as well [Zim03], it is not made for it, and it resembles the MPEG-4 which is minded on streaming.

A more reasonable approach would therefore be to look into the standards made by Moving Pictures Experts Group (MPEG). MPEG is part of the International Standards Organization (ISO/IEC) standardization group, for creating new coding formats for video and audio. Their most popular coding formats are MPEG-1, MPEG-2 and MPEG-4 where MPEG-4 is a preferred coding format by most video streamers at the present time. The International Telecommunication Union (ITU-T) have standardized a number of relatively similar coding standards as MPEG, they are called H.261 to H.264. The wish for a collaborated standard of the MPEG-4/H.264 formed a Joint Video Team (JVT) which consists of ITU-T, MPEG and a third party called Video Coding Experts Group (VCEG).

After having mapped the popular standards in video streaming, it is time to determine which coding standard is best for the prototype. To avoid the need for testing the various different possibilities, we look to existing papers describing recent experimentation on the performance of the popular coding standards. Although most of the recent papers are minded on performance over wireless network, and what IEEE 802.11 technology is best, they still provide some insight into what the best video coding would be. One such paper is [KGW09], that elaborates on some knowledge on popular video coding formats. The two top candidates are the MPEG-4 and the H.264, where the important differences are bandwidth usage, error sensitivity and encoding power needed. The numbers for bandwidth are backed up by performance comparison performed by [OBL+04]. It is found, that the H.264 encoding have better average bit rate savings, than the other candidates. See Table 2.1, for a reference on the characteristics of the popular coding standards.

Codec Name	**Bandwidth needed for PAL**	**Error Sensitivity**	**Encoding Computing Power**
MPEG-1	8 Mbps	Low	Small (x386)
MPEG-2	5 Mbps	Medium	Medium (x486)
MPEG-4	3-6 Mbps (Variable)	Medium	High (x586)
H.264	1 Mbps	High	Very High (Dual Core)

Table 2.1: Characteristics of popular coding standards (Source [KGW09])

As seen in Table 2.1, the H.264 uses less bandwidth, but as a trade off is more sensitive to errors and uses more computing power for the encoding process. The MPEG-4 uses more bandwidth, but is more tolerant of errors and uses a little less computing power. These arguments point in the direction of MPEG-4 as the best proposed video coding format, because we might need the extra computational power for computations on highly demanding computer games this GoD-service might provide. Error tolerance is also of the essence, and a high number of consumers cannot be expected to have a fast Internet connection (mobile broadband for instance, seems to be increasingly popular) and as “Most of the latency is in the last mile” (Steve Perlman, OnLive CEO [Gra10]). The preferred choice for a final implemented solution would be the MPEG-4 coding format. There is no apparent open source available port of an MPEG-4 encoder, so for this small scale prototype, we will settle for H.264, for which the Fast Forward Moving Pictures Experts Group (FFmpeg) has developed an open source implementation, in the form of a VFW codec and a stand-alone encoder. The difference between MPEG-4 and H.264 are rather minute in the FFmpeg implementation, which is a result of the JVT-groups combined work on a new version of MPEG-4/H.264 coding format.

2.3.2 The H.264/AVC Video Coding Standard

To understand why the H.264 video coding standard is recommendable for streaming (as previously noted in Table 2.1), it is necesarry to observe the network-related built-in features of the standard. This section provides a quick overview and explanation of the features of the coding standard, that makes it one of the best video streaming coding standards available at the moment. The information on the subject of the H.264 coding standard is gathered mostly from the standard for a *Real-Time Transport Protocol (RTP) Payload Format for H.264 Video* in RFC 3984 [WHS$^+$05] and from an *Overview of the H.264/AVC Video Coding Standard* [WSBL03]. Since, as mentioned in Section 2.3, the standard is only for the actual format of the coded video and the decoding of the video, there is no information on how the third party encoder has implemented these features.

The standard contains two layers; one to address the representation of the video content (the Video Coding Layer (VCL)), and another to address network abstractions (the Network Abstraction Layer (NAL)). A short description and listing of optimizing features in the two layers are in their respective sections.

VCL-layer & Features

As mentioned, the VCL-layer takes care of the representation of the video as moving images. Each video has a number of images per second (for video normally 24), these images are known as frames. To be able to perform compression, known as encoding for video, on these frames there must be a way to exclude similarities within the frames of the video. This is done by placing a grid over each of the frames and thereby splitting the video into smaller pieces. These pieces are called macroblocks, and for H.264 (H.264) the macroblocks initially consist of 16×16 pixels. The basic encoding of the video is to avoid saving or sending unchanged areas of each frame. If for instance the contents of a macroblock is unchanged since last frame, there is no reason to use resources on this particular macroblock. Hereby in this little example, the video is a macroblock smaller in size. This adds up, if done for each unchanged area of all frames in the entire video.

The splitting of the frames are in H.264 part of a larger hierarchy, as each macroblock is contained within a group of macroblocks and a macroblock. Such a group is referred to as a slice. A slice allows for a faster scan for changes in the frame, the groups can be nested as well. This feature makes it possible to create new frames only containing the changing parts of the video.

Once in a while, especially when streaming a video, a part of the video may be corrupted or lost. This problem will result in holes in the video or areas where some part of the frames are not correct, in relation to the rest of the frame. Therefore, there has to be some kind of fail over. In H.264 there are two kinds of frames being sent or stored, this is the *Intra*-frame and the *predictively coded* frames. Predictively coded means, that in H.264 some of the macroblocks are predicted to avoid more data loss failures. The prediction is based on the macroblocks' contents over the last few frames. Therefore the predictively coded frames are small in size and holds only the new information of the video frame. The Intra-frame is a frame that works a reference for the predictively coded frames, and is a frame that is large in size, because it holds the entire data of the frame.

Another way to compress the video and save space in the stream or file, is to convert the colour space to YCbCr, and also by reducing the resolution of the sampling of the information for Cb and Cr. This conversion makes sense, because the human perception abilities are based on details of brightness and then on colour. YCbCr follows this idea, as the various components are; *luma* (Y) and *chroma* (CbCr). Luma is brightness, Cb is the amount the colour deviates from gray toward blue and Cr is the amount the colour deviates from gray toward red. The human visual system is more sensitive to luma than chroma, therefore H.264 samples the brightness twice as much as it samples the chroma. Such a sampling is called *4:2:0*, and each of these samples has a precision of 8 Bits. This means that each macroblock covers 16 × 16 luma-samples, 8 × 8 Cb-samples and 8 × 8 Cr-samples.

NAL-layer & Features

The purpose of the NAL-layer, is to provide an abstraction to available network protocols. Hereby, the layer maps from the H.264 VCL-data to the transportation layers, this could be file formats, but also RTP for instance.

The data has been coded and therefore compressed at this point, and the next step is to either send it or store it in a file. To enable this data is organized into something called NAL units, which first holds a byte with header information and the remaining is the video data. This means that the NAL unit is already a small packet. Instead of packet sized NAL units, it is also possible to get the data in a Byte-stream format, which is needed in some systems. In the case of a Byte-stream there are identifiable boundaries of the NAL unit's beginning and end.

There is some data information that seldom change for a video, this information can be contained in *Parameter Sets*. There are two types of the parameter sets, one for the sequence that hold information for a sequence of video frames, and one for the single frame. Both the sequence and the picture parameter sets can be sent ahead of the NAL unit, thus making it more robust towards errors.

There are many features in the H.264 video coding standard and further details can be found in for instance [WSBL03] along with [KGW09].

2.4 Routing Schemes

Since this project is about utilizing a network connection, it makes sense to consider what kind of routing schemes are plausible, for the proposed system. There are a number of different routing schemes and they all have their own features, making them effective for various situations. A short description of the different possibilities is listed below. The different schemes are depicted in Figure 2.2.

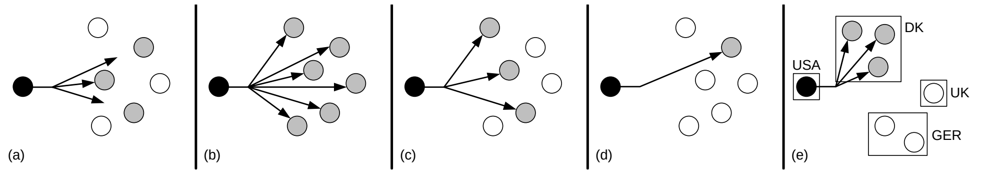


Figure 2.2: Example of the different Routing Schemes. *(a)* is Anycast, *(b)* is Broadcast, *(c)* is Multicast, *(d)* is Unicast and *(e)* is Geocast.

Anycast *(a)* is a routing scheme used, if any one receiver is needed. A datagram package is sent from a single sender, to a single receiver from a group of possible receivers. Often the receiver is the closest to the sender in the group. One to one.

Broadcast *(b)* is a routing scheme used, when all nodes in a group needs to receive a datagram simultaneously. One sender tries to reach all possible receivers with this scheme. One to all.

Multicast *(c)* is used for sending to many nodes in a group, but not all. Only the interested receivers get the datagram that is sent using multicast. One to many.

Unicast *(d)* is used to deliver a datagram to a single specific receiver. If the datagram is for one receiver by one sender, unicast is the routing scheme to do this. One to one.

Geocast *(e)* is used, if a datagram are for a specific region. It is a special multicast delimited by geographic region. One to many.

To use in a solution proposed by this project, the probable routing scheme would be unicast, or maybe even multicast in some situations. Unicast is a natural selection, because there is one user connected to one game server, therefore the client sends to one server, and the server sends to one specific client. Multicast will make sense, if it should be possible to follow another player's progress in a game. Multicast will provide the possibility to stream ones gameplay to many clients, who might be interested in following a particular client's progress.

2.5 Input Handling

In order for the streaming service to make any sense, the user input from the client side needs to be collected, sent across the network and then fed into the game that is run on the server. This is a necessity in order for the client to be able to play and interact with the game remotely.

There are two different ways of capturing the input on the client side, either the client receives all the input from mouse and keyboard and parses it, which in turn will require the client to know every possible input combination. Or, the client simply intercepts the raw message input data containing the keyboard and mouse input and sends it to the server to be fed into the running game directly.

The server needs to receive these raw input messages from the client, and insert it into the event queues for the game process. The game will then react to the input and the visual feedback will be streamed to the client.

2.6 Network Transmission Protocols

For a final implementation, three different types of data needs to be sent through network interfaces. These types are split into a bootstrapping of the service, the input from the client and the audio/video stream from the server. This section provides argumentation for what transmission protocols are recommended for the different purposes.

Each data type to transmit, are described under its own section below.

2.6.1 Bootstrap Connection

The bootstrapping connection is used to initialize the session of connections between the client and server. This is therefore the first connection between the client and server, and it should be handshaking data for the service. Handshaking involves, for a final implemented solution, some authentication data and it involves information about game to play and the IP/port for the server entitled to run the game.

The data for this connection is important, and it has to be reliable that all the information reaches both ends. Speed of the transferring data is not important at this stage.

A very good solution for this connection is to create a Transmission Control Protocol (TCP) socket connection between the requesting client and the (authentication) server. The standard of TCP version 4 is described in RFC 793 [Pos81], and belongs to the Transport Layer. The purpose of TCP is to create and sustain a stable connection between two end points in a network, allowing for data exchange. TCP is often used in connection with the Internet Protocol (IP) in the Internet-/Network- Layer, to route messages across the network participants. This correlation is referred to as the Internet Protocol Suite (TCP/IP). The ability for TCP to create a direct connection between endpoints makes it ideal for the purpose of the Bootstrap connection. See Figure 2.3 for a small example of the bootstrap connection, using TCP.

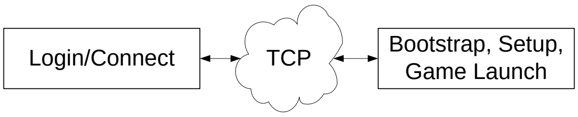


Figure 2.3: Example of the Bootstrap Connection. Client is on the left side and Server is on the right.

2.6.2 Input Connection

The purpose of the input connection is to send the inputs from the client to the server. This is inputs from controllers, as for instance keyboard keys and mouse positions. For this data connection, there is expected to be many inputs and they may be very rapid (this may depend on the type of game – if it is a fast paced game like an First Person Shooter (FPS) e.g.). Therefore the speed of the data connection is of the essence in this connection. For most games, key presses and mouse positions are often repetitive, which means that it might not be a problem if some of the data are lost. Although it is tolerable to lose a few of the inputs, it cannot happen too often, as it will be a nuisance for the player.

To handle this connection a User Datagram Protocol (UDP) socket will be a possibility. As opposed to TCP, the UDP protocol is more minded on speed than reliability. The standard of UDP is formed in the RFC 768 [Pos80] and is, as TCP, part of the Transport Layer. UDP is also part of the Internet Protocol Suite, but differs with TCP by not demanding anything of the transmission paths through the network. The IP is forced to find a route through the network participants for each datagram sent by the UDP. This behavior does not ensure datagram delivery, but it allows for a timely delivery, which makes UDP a reasonable choice for the Input connection. In Figure 2.4 there is a small example of the input connection.

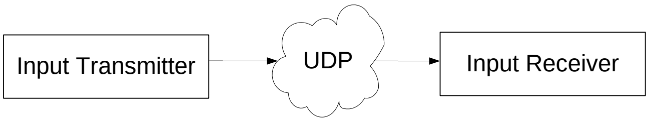


Figure 2.4: Example of the Input Connection. Client is on the left side and Server is on the right.

2.6.3 Stream Connection

As with the input connection, it is more important for the audio/video streaming to be fast than reliable. If a screen update does not arrive at the client's side, another will arrive shortly.

A possibility is to use the UDP connection here as well, but there are protocols already designed and optimized for streaming of video and audio. One of such is the RTP, as stated in the book RTP [Per03], RTP is standardized for streaming audio and video in RFC 3550 [SCFJ03] by the Internet Engineering Task Force (IETF) and later adopted by ITU-T. This standardization means it is easier to find support for RTP in both players and programming libraries. RTP is part of the Application Layer and uses the UDP protocol in the Transport Layer, and therefore utilizes the timely delivery of the UDP socket.

RTP works by packing the data into packets with added header information to inform the receiver of how to handle the containing data. The RTP header has a minimum size of 96 bit or 12 byte and can be larger if using optional extension header information. The header of the RTP packet is constructed as seen in Table 2.2, and is explained in the following description.

Bit	0–1	2	3	4–7	8	9–15	16–31
0	Ver.	P	X	CC	M	PT	Seq. Nr.
32	Timestamp						
64	SSRC Identifier						
96	CSRC Identifiers ...						
96+32×CC	Profile-specific Ext. Header ID						Ext. Header Length
128+32×CC	Extension Header ...						

Table 2.2: The header format of a RTP packet. From bit 96 the header information is optional. Source RFC 3550 [SCFJ03].

Ver. (Version) indicating the version of the protocol. Default is version 2. (2 Bits)

P (Padding) indicating if there are extra bytes in the end of the packet (if it needs a certain size). (1 Bit)

X (Extension) indicating if there are Extension headers before payload data in the packet. (1 Bit)

CC (CSRS Count) indicating the number of CSRC Identifiers in the header. (4 Bits)

M (Marker) indicating (at application level) if the packet is of special relevance. (1 Bit)

PT (Payload Type) indicating the type of the data in the payload. (7 Bits)

Seq. Nr. (Sequence Number) indicating the order of the packets, is incremented by one for each packet. (16 Bits)

Timestamp indicating time for synchronizing of the feed. (32 Bits)

SSRC indicating the synchronization source of the stream, a unique identifier for the source in the RTP session. (32 Bits)

CSRC indicating source IDs for all the sources that are contributing to the stream, if more than one exist. (32 Bits per CSRS, hence 32×CC)

Profile-specific Ext. Header ID indicating a profile-specific identifier. (16 Bits)

Ext. Header Length indicates the length of the extension header. (16 Bits)

Extension Header indicating the actual extension header. (32 Bits per header unit)

Figure 2.5 is an example of the stream connection.

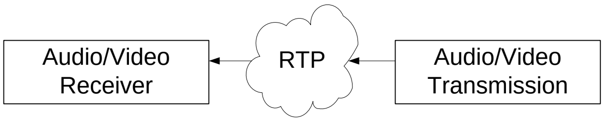


Figure 2.5: Example of the Stream Connection. Client is on the left side and Server is on the right.

When using RTP, there is also the possibility of extending the use, with optional additions in Session Description Protocol (SDP), Real-Time Transport Control Protocol (RTCP) or Real-Time Streaming Protocol (RTSP). More details on the SDP, RTCP and RTSP protocols follow.

Session Description Protocol

SDP is an Application Layer protocol, usable as a format for describing a media stream during its initialization. It is designed to supply parameters, options and descriptions needed for the stream to be understood correctly by the receiver. The standard is described in RFC 2327 [HJ98].

SDP is originally made for the Session Announcement Protocol (SAP), but has proved itself useful as a supplement for other media protocols, such as RTP. It is not designed to stream any media, so it is supposed to be a supplement to the existing stream solution.

The SDP format is a newline-separated series of fields containing a single character as identifier, followed by an equal-sign (=) and then the value of the field ($= *$ indicates that the field is optional). There are three main sections in the SDP format to group the fields. The sections are Session, Time and Media that contains fields and values to either describe the current session, the timing or the media in the stream. Some relevant fields are displayed in Table 2.3.

Real-Time Transport Control Protocol

The purpose of the RTCP is to supply statistics on the RTP media stream. The statistics are transmitted both to the sender and the receiver of the media stream and the purpose of the statistics, is to detect transmission faults and can be used for adaptive media encoding on the server side. RTCP runs side by side with a RTP stream and can manage timing between a stream of video and a stream of audio. The RTCP standard is described in RFC3550 [SCFJ03].

Session Description	Time Description	Media Description
$v =$ (protocol version) $o =$ (originator and session identifier) $s =$ (session name)		
	$t =$ (time the session is active)	
$a = *$ (zero or more session attribute lines)		
		$m =$ (media name and transport address) $a = *$ (zero or more media attribute lines)

Table 2.3: Some relevant fields in SDP. Source RFC 2327 [HJ98].

Since RTP can multicast, RTCP also has to provide this feature. In multicasting, the traffic increases proportionally with the number of participants. To avoid network congestion, RTCP has built-in bandwidth management by dynamically controlling the report frequency.

RTCP has the following message types:

Sender Report (SR) is sent by the server in controlled frequencies. It holds information such as an absolute timestamp, allowing RTP streams with video and audio to be synchronized.

Receiver Report (RR) is sent by the client. It holds information on the quality of service.

Source Destination (SDES) is source identifier information, as a canonical end-point identifier or CNAME to all the participants. It may also be information such as stream-owners e-mail or stream description.

End of Participation (BYE) is sent to signify a closing stream, or a participant leaving the session.

Application-specific Message (APP) is used for application specific additions to the stream. Can be used, if extra (custom) information is needed, for the application.

Real-Time Streaming Protocol

To give the client the ability to control a stream. It is designed to send simple commands between client and server, such as PLAY or PAUSE, to control the media stream.

RTSP is another independent Application layer protocol that are often used together with RTP, but not only minded on RTP. RTSP normally use TCP as Transport layer. The standard of RTSP is defined in RFC 2326 [SRL98].

A short description of the possible commands are listed below.

OPTIONS will return the enabled request types the server has available for the current session.

DESCRIBE is to provide a description of the stream available. This is descriptions formed in SDP format (seen in Section 2.6.3), and holds the same kind of information.

SETUP is performed to start up the stream session and must be done before a PLAY request. The request specifies how the stream is transported. Contained in the request are the Uniform Resource Locator (URL) and port for the RTP and/or RTCP stream. Server response is the needed information for the streams to run.

PLAY is to start the stream. The PLAY request can be combined with a range parameter, to play a certain part of the stream.

PAUSE it to simply pause the stream. The stream will resume when using the PLAY request again.

RECORD is to a stream from the client to the server, for storage.

TEARDOWN is to stop and terminate the session and media streams.

2.7 Other Solutions

A number of other solutions exist currently to stream games and play them remotely, which have different uses and applications. Such solutions include OnLive and StreamMyGame, as well as solutions currently under development, such as Games@Large and Gaikai. These applications share the ability to run games on one machine, while streaming the visual and audial output to a client on a different machine, allowing this client to be the player of the game. They, however, have different motivations, implementations and uses, of this functionality, and they are described below.

2.7.1 OnLive

The OnLive [Tea10b] service provides a game streaming service, as well as servers to stream the game from. A user of OnLive would sign up for the service, download a small client application. After having purchased access to a given game, the user will be able to run the game on OnLive's data centres, and stream it to their end user client. This allows a user to play high-end games on low-end machines. The service also provides for the usage of a MicroConsole, which is similar to a set top box, connected to a compatible HDTV, which allows a user to play OnLive games directly from their TV.

The service also provides a variety of community related features, such as spectating or cross-game chat. The software and hardware used for OnLive are proprietary, and as such it is difficult to descern how OnLive implements streaming, input handling, and other features. OnLive is available in the United States and in Europe, and was generally received well [BA10], some concerns do remain about how this service will perform under massive pressure, such as the millions of users on Steam [Ste10].

2.7.2 StreamMyGame

StreamMyGame [Tea10c] is a software solution to stream games from one computer to another, the difference between StreamMyGame and OnLive, is that while OnLive requires you to use their servers and pay to stream and play the games, while StreamMyGame has you set up your own server with your own games that you or others can connect to using the StreamMyGame application and play remotely. It can be used both over Local Area Network (LAN), but also over the Internet if the users and server have sufficient bandwidth for the service. Other features of StreamMyGame includes allowing the user to record their gameplay for upload to media sites such as YouTube.

The service is free to use, however, if a user wishes to provide streaming in high quality, then a subscription fee is required. The software for StreamMyGame is thus also proprietary, and unavailable for examination, so it is impossible to know the implementation details of the service and components.

2.7.3 Under Development

Several other solutions dealing with streaming games remotely in some form are under development, some of them as a response to OnLive. One of these is Gaikai [Tea10a], which uses streaming of games to allow users to play games to demo the experience right in their browser using Flash, before purchasing the game. Gaikai is currently in closed beta.

Another system under development is called Games@Large, which is detailed in a series of research articles [JBDG$^+$09], [JBDG$^+$10], [LLJ$^+$10] and [JFE$^+$09]. This service aims to allow streaming of games to some extent, the status of the development of this service is as of this time unknown, however.

Chapter 3

Design

This chapter provides an overview of the design proposed for a final implementation of the project. It describes the architecture of the service, the various components, details about the service as a whole and how the service should be incorporated as a Client/Server network architecture. The architecture also details which components make up the client and the server, and which of the many protocols are used to connect the functions of the client and server together.

3.1 Choice of Protocols

Prior to defining an architecture for the proposed GoD-service, this section considers choices for the different protocols needed in the various functions in the service. The different functions of the service require different services from the chosen protocol, as argued in Section 2.6. Thus it makes sense to choose a seperate one for each functionality, based on its use.

For the Authentication part of the service, the TCP has been chosen. This is based on the need for a reliable, two-way connection between the client and the server. The TCP is designed for creating a reliable connection between two end parties, which makes TCP a reasonable choice.

UDP is an ideal candidate for the Input Handling functionality. Lost packets can be tolerated since input is often repetitive. Speed is also an important factor for the choice of protocol, for the input messages, as low latency and response is essential in many games. UDP is designed for timely deliveries, which makes it useful in this functionality.

To implement the streaming service, RTP has been chosen, which works on top of UDP. This protocol has been developed specifically for streaming video and audio, and is therefore optimized for the purpose. RTP is also standardized and is supported by many existing implementations.

All the background analysis for these choices can be found in Section 2.6.

3.2 Service Architecture

The architecture proposed for the final service is detailed in this section, it is the ideal architecture for a completed service. Not all of it will be implemented. It details both the client and the server part of the service, as well as each of their components. The architecture schematic also includes abstractions over the various protocols used to communicate between the components of the client and the server based on the analysis in Section 2.6. The proposed architecture for the service is depicted in Figure 3.1. In Figure 3.1, service components are depicted using boxes, while client-/server- functionality is delimited by dashed lines.

As shown in Figure 3.1, the architecture consists of a client and a server. These components come together to provide different functions to the entire service, and it gives a satisfactory overview of how the complete service is envisioned. These components are also intended to use many of the approaches and techniques described in Chapter 2. The server components and the client components are explained in the following sections, and the functionality provided by collaboration of components, is described in Section 3.3.

3.2.1 Server Components

As seen in Figure 3.1, the server consists of a number of different components. To provide a better understanding of the server architecture, this section explains the components one by one. A more complete description of the server components' collaboration and use is in Section 3.3.1. Notice the dashed lines in the server side of Figure 3.1 represents an opportunity to split the service between application implementations or, for a more final implementation, it can signify splitting of the components over a number of servers in a cluster.

The server components are as follows:

Game Process The actual game, played by the user on the client. For a non demanding game the server might be able to run more than one game.

Video/Audio Capture The capture of video and audio from the game process, done by hooking into the process.

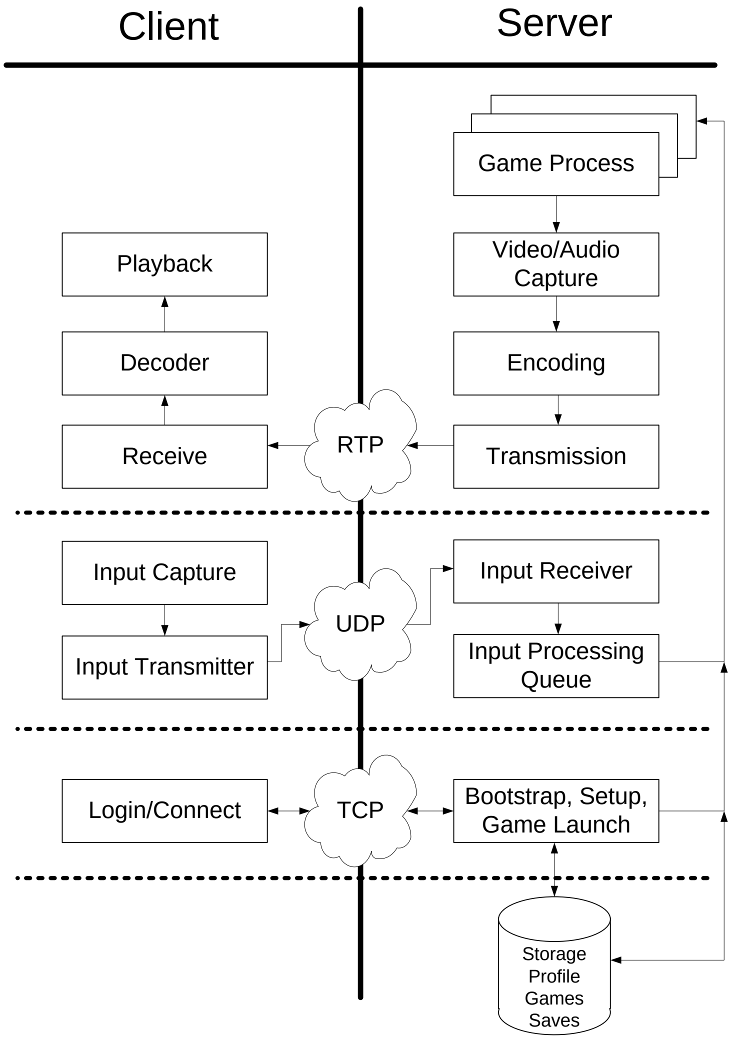


Figure 3.1: Streaming Service Architecture

Encoding The codec chosen to encode the captured video and audio, so it is suitable for transmission over the network.

Transmission Transmitting the encoded frames over the network using the RTP protocol.

Input Receiver UDP socket receiving the sent input from the client.

Input Processing Queue Queue of received input from the client that is to be fed to the game process.

Bootstrap, Setup, Game Launch Initial connection handshakes with the client, setup of connections and launching of the chosen game, as well as loading the profiles and saves associated, using TCP. For a final implemented solution, this component would be a reasonable choice for a resource manager in the cloud of game servers. It could be responsible for choosing a server to run the game for the client.

Storage Storage of the game software, user profiles and game saves.

3.2.2 Client Components

As the server part in Figure 3.1, the client also consists of a number of components. This section explains the components of the client side. Contrary to the server, the dashed lines only signify the possibilities of dividing the client-part into different executables. It will make more sense to have all the components implemented in a single client application though. The Video/Audio component however, could be made as a standalone part, if the service is set to multicast the video feed to other interested viewers. More details on the correspondence between the various components, will be found in Section 3.3.2.

The client consists of the following components:

Video/Audio Receiver Receiver of the RTP packets, containing the encoded frames from the server.

Decoder Decoder of the encoded frames, so they can be displayed on the client-side.

Playback Display the decoded frames as video with sound on the client.

Input Capture Capture the raw input from the user who is playing the game remotely.

Input Transmitter Transmit the raw input to the server using UDP.

Login, Connect Connect, login and authenticate with the server to load the appropriate profiles, saves and games on the server using TCP.

3.3 Functionality of Server & Client

The different components work together in a multitude of ways to create different parts of the service, for example, the video capture, encoding and transmission part of the server work together to create the game streaming functionality. Some of these functions could possibly be divided to run on multiple machines if relevant, such as the login and connection bootstrapping functionality. This could profitably be run on a dedicated authentication server, tuned for handling these types of requests. The other parts of the server service, devoted to running and streaming the games, could be handled on machines optimized for games.

3.3.1 Server Functionality

This section follows up on the architecture in Figure 3.1, with more details about the collaboration between the components of the server part. The server is divided into three different major functions; Game Streaming, Input Handling and Authentication, which are elaborated in the following sections.

Game Streaming

The Game Streaming part of the server encompasses hooking into the actual game process, intercepting the frames and streaming the game as a video feed. The intercepted frames are passed to the encoding component, which encodes the frame using the chosen H.264 encoder. Once properly encoded, the frames are passed to the transmission component, which encapsulates the frame in an RTP packet. The packet is transmitted, over the IP-network, to the client. The game process that is hooked, is fed input from the Input Handling functionality.

Input Handling

The components that make up the Input Handling functionality on the server are tasked with receiving the input from the client, which are sent using UDP datagrams. Once received and unwrapped, it will be enqueued in the input queue for the appropriate game process. The queued input is fed into the game process, which handles it as if the user had been playing locally. The new game state is then displayed for the client, via the Game Streaming functionality.

Authentication

The Authentication functionality of the server is for the clients of the GoD-service, to authenticate and login to the service. If the service is run on a larger scale, in a server cloud, the Authentication component could run on a different server, dedicated to authentication requests. The idea is that clients wishing to use the streaming service, establish a TCP connection to the authentication server, login using their username and password, and then pick whichever of their games they want to play via the service. The games, profiles, saves along with other persistent data, are stored on a different dataserver and loaded to a server suited for running the game. The RTP and UDP services of the input and streaming parts can begin communicating with the client after it has authenticated.

3.3.2 Client Functionality

The functions of the client, depicted in the architecture 3.1, is detailed in this section. This is a description of how the different components in the client communicate and create a complete functionality for the client. The client is divided into three different major functions; Game Streaming, Input Handling and Authentication, which are elaborated upon in the following sections.

Game Streaming

The client houses a Game Streaming part. This involves receiving the incoming RTP packets from the server, and unwrapping these packages, to extract the encoded video and audio data inside. These encoded frames are sent to the decoder, which decodes them using the chosen codec, so they can be displayed on screen for the user. Ideally this process can happen without significant delay, so the user can play the game properly.

Input Handling

Input Handling is done by the client, so the user's input, meant for the game, can be captured and sent to the server and thereby delivered to the actual game process. The Input Capture component captures the raw input directly from input devices, so the client is not concerned with determining which buttons were pressed. This input is sent to the server via UDP datagrams to be received by the game process in the end.

Authentication

Concerning login and connecting to the service, the client has a component, that will establish a TCP connection to the server. This allows the client to enter needed login information, thus assuring security for the games and the users persistent data. Once logged in, the client can select which game to play and load profiles, to use the service properly.

Chapter **4**

Prototype

In the Prototype chapter, it is detailed how the prototype parts of the streaming service are actually implemented. This includes information on development tools used for the project, arguments for choice of the language C++, details on how the various third party libraries contribute to the implementation, as well as the C++ code that tie these third party libraries together. This chapter also explains several significant problems encountered during the implementation process.

It has been chosen to only implement part of the proposed service from the architecture in Chapter 3.2. Only the streaming part of the service is actually implemented and functional, using various third party libraries.

4.1 Development Tools & Language

For the development of this prototype, the choice of programming language is C++. A primary reason for this choice is that Taksi is written in C++ and that the implemented features (such as hooking and capturing, which are outside the scope of this project) in Taksi forms an important base for the service.

Taksi was initially created as a Microsoft Visual Studio project, so it was chosen as development environment for continued work on this project. The prototype consists of a Taksi project and the needed libraries from Jori's Real-Time Transport Protocol Library (JRTPlib), an implemented RTP server to stream the data and debugging utilities implemented to locate problems in the implemented prototype and execution flow.

4.2 Components

This section details the implementation of the various components that make up the streaming part of the service. This includes many third party libraries, as well as code that make these third party libraries work together.

4.2.1 Video Capture

The video capturing components used in the service are implemented using an open source program and library called Taksi, described in Section 2.2.2. The Taksi program allows for hooking directly into a running game, and captures frames directly from the backbuffer of the game.

Taksi is used nearly as is, with a few modifications. Instead of writing the captured frames to a video file on the disk, Taksi has been modified to send these encoded frames to the RTP server to be further processed. The Taksi default Graphical User Interface (GUI) is still used as GUI for the streaming service, as it starts sending as soon as Taksi hooks into a process and starts recording.

The main change done to Taksi is in the CAVIFile.cpp file, where the frames are sent to the RTP server, instead of being written to the file. This is shown in Code Example 4.1.

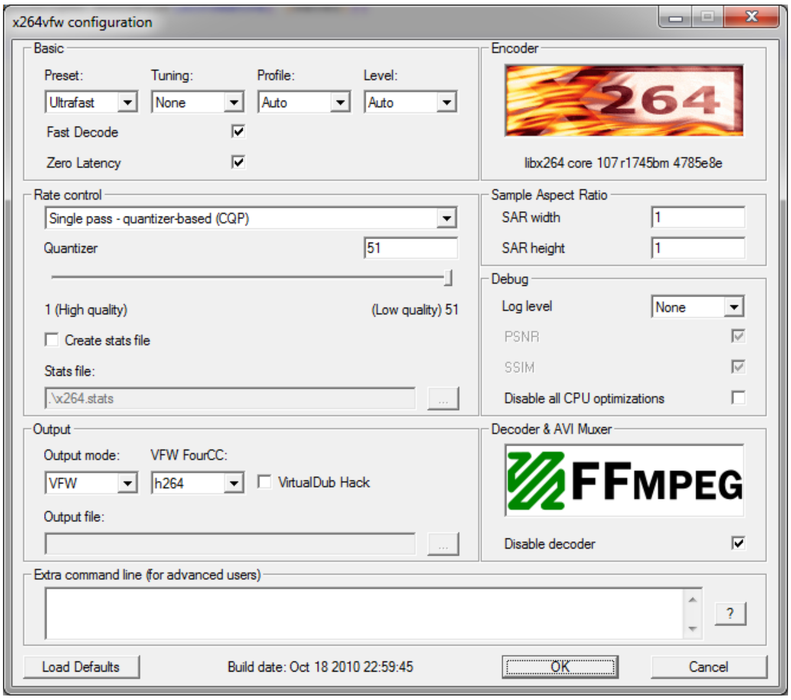
```
    SendPackage(pCompBuf, (size_t) nSizeComp);
    dwBytesWrittenTotal += (DWORD) nSizeComp;
```

Code Example 4.1: The code inserted into the Taksi program, that sends frames to the RTP server.

In this code, the *SendPackage* method is used, while the original Taksi code that writes to a file on the disk has been out commented. The *SendPackage* method takes a pointer to whatever data is to be sent, and the length of this data as arguments. The second line increments the internal Taksi counter of how many bytes have been written so Taksi knows that it is making progress with its frames.

4.2.2 Encoding

The Taksi program, that is used as a base for the capturing, furthermore allows for a choice of a VFW encoder to easily be used to encode the frames into an appropriate format. For this the x264vfw encoder [xT10], has been chosen. This encoder allows several different customization options, related to speed and quality of encoding, which can be seen on Figure 4.1.



x264vfw configuration

Basic

Preset: Ultrafast

Tuning: None

Profile: Auto

Level: Auto

Fast Decode ☑

Zero Latency ☑

Encoder

libx264 core 107 r1745bm 4785e8e

Rate control

Single pass - quantizer-based (CQP)

Quantizer 51

1 (High quality) (Low quality) 51

□ Create stats file

Stats file: .\x264.stats

Sample Aspect Ratio

SAR width 1

SAR height 1

Debug

Log level None

PSNR ☑

SSIM ☑

Disable all CPU optimizations □

Output

Output mode: VFW

VFW FourCC: h264

□ VirtualDub Hack

Output file:

Decoder & AVI Muxer

Disable decoder ☑

Extra command line (for advanced users)

Build date: Oct 18 2010 22:59:45

Figure 4.1: A screenshot showing how the H.264 encoder has been configured.

In order for the basic streaming service to work, the settings for the encoder have been set to fastest encoding speed and lowest possible quality, this can be seen in Preset being set to “Ultrafast”, and Quantizer being set to “Low Quality”. This is done so the streaming service actually works, due to RTP library used for transmitting packages not supporting splitting of the packets for bigger payloads. If packet splitting was implemented, the quality of the encoding could be increased, but the fast speed of the encoding is still a factor.

4.2.3 Transmission

The implementation of the RTP server is created as a Microsoft Visual Studio project called RTPService. This is the setup of the RTP server with all the necessary initial configurations and it also holds the *SendPackage* method. The RTPService is the connection between the encoded video data from Taksi and the JRTPlib RTP library.

In Code Example 4.2, the includes, defines and global accessible variables for the RTPService project are shown.

```
#include "RTPService.h"
#include "PracticalUtils.h"
```

```
 3 #include <crtdbg.h> //for _ASSERT
 4 //JRTP includes
 5 #include "RTPSession.h"
 6 #include "RTPIPv4Address.h"
 7 #include "RTPSessionParams.h"
 8 #include "RTPUDPv4Transmitter.h"

10 #define MCAST_IP       "239.216.30.54"
11 #define MCAST_PORT      4000
12 #define SERVER_PORT     5000

14 #define MAX_PACKET_SIZE ((1024 * 64) - 1)

16 PracticalUtils utils("RTPServiceLog.txt");
17 RTPSession rtpSession;
18 int counter = 0;
19 bool sessionCreated = false;
```

Code Example 4.2: The RTP server component. Includes, defines and variable initialization.

More detail of Code Example 4.2 is in the following line-by-line description:

Lines 5–8 are the references to include the needed methods of the JRTPlib.

Lines 10–14 are defines of the constants for the server setup. Notice that the IP of the server in RTPService, is set for multicasting. This is to enable testing of the multicasting possibilities of the service. Otherwise the ports for the server and service, and a delimiter for a maximum package size are defined here.

Lines 16–19 holds initialization of some globally accessible variables needed primarily for debugging, but also to store the created RTP session in a location reachable by all the methods that need it.

The first step towards creating the RTP server, is to initialize a Windows Socket, to use in the RTP server. Code Example 4.3 indicates how this is done using the WSAStartup method.

```
int InitRTPStream()
{
  utils.logThis("InitRTPStream executed...");

  //WSACleanup called in StopRTPServer
  WSADATA wsaData;
  WORD wVersionRequested = MAKEWORD( 2, 2 );
  WSAStartup(wVersionRequested, &wsaData);

  return RunRtpServer();
}
```

Code Example 4.3: The RTP server component. InitRTPStream method.

As seen in Code Example 4.3, the *InitRTPStream* executes the *RunRtpServer*, which is responsible for starting the RTP server using the JRTPlib. The *RunRtpServer* can be seen in Code Example 4.4.

```
int RunRtpServer()
{
  int res = 0;
  utils.logThis("RunRtpServer executed...");

  //setup session parameters
  RTPSessionParams sessParams;
  sessParams.SetSessionBandwidth(MAX_PACKET_SIZE);
  sessParams.SetOwnTimestampUnit(1.0 / 30.0); //30 video frames per second
  sessParams.SetUsePollThread(1); // Background thread to call virtual callbacks
  sessParams.SetMaximumPacketSize(MAX_PACKET_SIZE);
  //setup transmission parameters
  RTPUDPv4TransmissionParams transParams;
  transParams.SetPortbase(SERVER_PORT);

  //CREATE THE SESSION
  utils.logThis("Creating RtpSession...");
  int status = rtpSession.Create(sessParams, &transParams);
  if (ReportError(status))
    return -1; //unable to create the session
  utils.logThis("RtpSession created with portbase " + utils.itos(SERVER_PORT) + "\n");
  sessionCreated = true;

  //SET TRANSMISSION DEFAULTS
  rtpSession.SetDefaultPayloadType(96);
  rtpSession.SetDefaultMark(false);
  rtpSession.SetDefaultTimestampIncrement(160);

  //ADD THE MULTICAST to our destination
  unsigned long intIP = inet_addr(MCAST_IP);
  _ASSERT(intIP != INADDR_NONE);
  intIP = ntohl(intIP); //put in host byte order
  RTPIPv4Address rtpAddr(intIP, MCAST_PORT);
  status = rtpSession.AddDestination(rtpAddr);

  // Only if there are an error, this will have an impact.
  ReportError(status);

  return status;
}
```

Code Example 4.4: The RTP server component. RunRtpServer method.

Further details on what *RunRtpServer* in Code Example 4.4 does, is in the following line description.

Lines 86–93 creates and defines session parameters. These parameters define the maximum packet size allowed to be sent, a timestamp unit based on having 30 frames per second and the server port for the transmission.

Lines 96–101 creates a session based on the session parameters defined in lines Lines 86–93.

Lines 104–106 defines default transmission parameters. One of these is payload type, which is chosen to be 96 as this is one of the non-reserved payload types for dynamic types. Another is whether the single packet should have the marker bit set, by default. The third default parameter is the timestamp increment between packages.

Lines 109–113 sets the defined multicast ip address to the session.

Lines 116–118 checks if errors occurred when creating the session and returns the result.

At this point, a socket is opened, the server is started and a session is created. Code Example 4.5 shows how the prototype sends the data from the Taksi library to the RTP session. Notice that lines 33–52 are all for debugging and uses more computational resources.

```
21 int SendPackage(const void *data, size_t len)
22 {
23   int res = 0;
24   utils.logThis("SendPackage executed...");
25   utils.logThis("Sending Package of size " + utils.itos(len));

27   if (!sessionCreated)
28   {
29     utils.logThis("Ooops, session not created..");
30     InitRTPStream();
31   }

33 #ifdef _DEBUG
34   void* localData = malloc(len);
35   size_t localLen = len;
36   memcpy(localData, data, localLen);

38   if (counter < 60)
39   {
40     string filename = "PointerDumps - SendPackage - Server " +
           utils.itos(counter) + ".txt";
41     char* charfilename = new char[filename.size() + 1];
42     strcpy(charfilename, filename.c_str());
44      utils.DumpPointer(charfilename, localData, localLen);

46      ++counter;
47      delete charfilename;
48    }

50    res = rtpSession.SendPacket(localData, localLen);
51    utils.logThis("Status of the send package: " + utils.itos(
          res) );
52 #else
53    res = rtpSession.SendPacket(data, len);
54 #endif
55    return res;
56 }
```

Code Example 4.5: The RTP server component. SendPackage method.

Further details on the lines of Code Example 4.5 are in the description below. The lines marked with “Debugging” are lines only run in a debug-build.

Lines 27–31 makes sure that a session has been created. It is important that the session is properly created before the the RTP-server tries to send anything, else it will fail. The *SendPackage* method is also responsible for starting the RTP server at first run, it will also fail however, if it tries to create a second session.

Lines 36–48 (Debugging) These lines are explained in detail in Section 4.4 about debugging tools used.

Lines 53–55 simply sends the data to the RTP session, which builds a packet, transmits it over the network and finally returns the status of the transmission.

During the prototyping, a problem with the video stream not displaying correctly on the client side, caused many different experimentations with the prototype. Most of this experimentation is mentioned in Section 4.4 about the debugging tools created and used in the prototype, while Section 4.3 about the difficulties, describes the most important problems that were faced during the prototyping. The experimentations also lead to a threaded version of the RTP service, which is also mentioned in further detail in Section 4.4. Both the server prototype source codes are located on the CD-ROM attached as appendix.

4.2.4 Receiving, Decoding, Playback

To avoid having to implement a complete prototype client, video decoder and player, it was chosen to use VideoLAN Client (VLC) to view the RTP stream. VLC is freely available as open source and feature, amongst other things, the ability to receive RTP streams, decode H.264 video and display the video frames. This is an easy way to test the streaming of video.

To ease the process of testing the stream and opening the video feed in VLC, an SDP-file was created. This file was created with the knowledge of Section 2.6.3 about SDP. The contents of the file is seen in Code Example 4.6.

```
o=VideoServer 305419896 9876543210 IN IP4 239.216.30.54
s=VideoStreamTesting
t=0 0
c=IN IP4 239.216.30.54
m=video 4000 RTP/AVP 96
a=rtpmap:96 H264/90000
a=fmtp:96 packetization-mode=0; profile-level-id=4D4033; sprop-parameter-sets=Z01AM5ZkBQHtCAAAAwAIAAADAYR4wZU=,a048gJ==
```

Code Example 4.6: The SDP file used to receive from the streaming service.

The lines of Code Example 4.6 are explained in the following description.

o contains an identifier for the session and the originating ip of the session. This is defined as the multicast ip the server was set to use.

s contains a simple name for the session.

t specifies the time in which the session is active, this is zeros because there are no limits on this.

c is another line indicating the origin of the session, still refers to the multicast ip set for the server.

m is the media name and a transport for the address. It specifies that video is sent on port 4000, using RTP and that this video feed uses the dynamic payload type of 96.

1. a is an attribute that informs that the stream identified by payload type 96 is an H.264 stream with a clock rate of 90000. This is a demand proposed for the standard in RFC 3948 [HSV+05].

2. a holds sequence parameter and picture parameter sets (sprop-parameter-sets). Without these, the decoder would not be able to properly decode the image. This is a base-64 encoded representation of the bits that indicate profile id, level id flags and chroma and luma information and

> scaling for the encoded video. The comma seperates the sequence parameter set from the picture parameter set. This is all specified to further detail in the recommendation for H.264 by ITU-T in [itu10].

This information can also be sent inline, for each packet, but this is data that does not change during the video stream, so it was chosen to save the bandwidth for the video stream instead.

4.3 Implementation Difficulties

During the development of this project, several difficulties arose, some of which perhaps were underestimated. The development and implementation dealt with the key features of encoding and streaming video, which are barely scratched at the surface in the Analysis Chapter. As such, working with technically heavy topics such as these demands significant expertise, which cannot realistically be acquired during a small development period as was the case for this project. This expertise also had to be built from the ground up, as we were novices in these fields. Thusly, the development of the streaming service in this project, is based on a very basic level of skills in the area of video encoding and streaming. Trying to grasp these topics at a novice level also meant that debugging errors was made more difficult.

Furthermore, to facilitate development of this service, several third party libraries were chosen for specific tasks in the service implementation. Many of these libraries were the only choice for certain aspects of the service, such as Taksi, and some of them suffered from being infrequently maintained and sparsely documented. Other choices were restricted by propriety. Working with these third party libraries and making them work together in many cases proved to be a much greater challenge than anticipated, and thus turned out to be a lot more time consuming and problematic part of the development process than was originally planned.

4.4 Debugging Tools

As mentioned in Section 4.3, there were a number of problems during the prototyping. Mostly because it was first time experience with video capturing, encoding and streaming, but also because different third party libraries are used. The result was, that we could not be sure where the problems occurred, so we needed to create our own debugging tools to find the culprit of the problems. Since many hours were spent on resolving these problems, this section about the tools used documents the time spent.

4.4.1 Practical Utilities

Practical utilities were created as a class of utilities that were useful for debugging the code. These utilities could be used where needed, and proved very useful in locating the problematic areas of the execution flow.

Code Example 4.7 shows one of a number of simple conversion methods. This particular one converts from an integer to a string, this is useful to be able to print the integers or if we need to concatenate it with a string. All the conversion methods follow the same basic principle as in Code Example 4.7.

```
string PracticalUtils::itos(int i)
{
    stringstream s;
    s << i;
    return s.str();
}
```

Code Example 4.7: The Practical Utilities. ItoS method.

The *LogThis* method in Code Example 4.8, is used to log any desired string to a file with a timestamp. This is a useful feature when we need to create a log of what has happened at what time. This code has been widely used to check if the code is executed in a proper manner. Notice that *LogThis* does nothing if the build is not a debug build, to save system resources in a release build.

```
int PracticalUtils::logThis(std::string logthis)
{
#ifdef _DEBUG
  ofstream out(fileName, ios::app);
  if(!out) {
    cout << "Cannot open file.\n";
    return 1;
  }

  time_t rawtime;
  time ( &rawtime );
  out << ctime(&rawtime) << " :: " << logthis << "\n" << endl;
  out.close();
#endif
  return 0;
}
```

Code Example 4.8: The Practical Utilities. LogThis method.

The *DumpPointer* method in Code Example 4.9 is used to get the raw data from any pointer and dump it directly to a file. This method is highly useful for instance to see whether the data is the same on the client as it was on the server.

```
int PracticalUtils::DumpPointer(char* FileName, const void* ptr, size_t size)
{
#ifdef _DEBUG
  FILE *fp;
    size_t count;

    fp = fopen(FileName, "w");
    if(fp == NULL)
  {
        logThis("failed to open PointerDumps.txt");
        return 1;
    }
  count = fwrite(ptr, 1, size, fp);

  string success;
  if (fclose(fp) == 0)
    success = "succeeded";
  else
    success = "failed";

    logThis("Wrote " + itos(count) + " bytes to PointerDumps.txt. fclose(fp) " + success + ".\n");
#endif
    return 0;
}
```

Code Example 4.9: The Practical Utilities. DumpPointer method.

4.4.2 Debugging in RTPService

In Section 4.2.3, the Code Example 4.5 has some debugging lines. These need some further explaining and are elaborated upon in the following description.

Lines 34-36 (Debugging) creates a local buffer of the video data, and information of the size of the data. This is done by creating a new pointer and copying the data to the new pointer, called *localData*. This is to test if it was necessary to create a local buffer for the data, this could have been a reason for some problems that occurred with the data sending. It turned out that the buffer is not strictly necessary, but an implemented First In First Out (FIFO) queue, working as a buffer, might be a good idea to avoid sending bursts of packets by adding a small delay. This is backed up in [HP08] which holds simulations of packet loss performance under varying network conditions with path diversity.

Lines 38–48 (Debugging) dumps the contents of the first 60 pointers with data, to their own file. This provides the ability to find errors in the

> application flow. It is possible to see what data the pointer holds at the moment, and this can be compared with later dumps of the data, either on the client side or other places in the application flow, to see if the data is still as expected, and assure that it has not been corrupted or changed unexpectedly.

Furthermore there is a small method to translate and print error codes including descriptions (where available), provided by the JRTPlib as return values. This method is listed in Code Example 4.10.

```
int ReportError(int errCode)
{
  int isErr = (errCode < 0);
  if (isErr) {
    std::string stdErrStr = RTPGetErrorString(errCode);
    utils.logThis("Error " + utils.itos(errCode) + ": " +
        stdErrStr.c_str() + "\n");
  }
  return isErr;
}
```

Code Example 4.10: The RTP server component. ReportError method.

The *ReportError* method in Code Example 4.10 is simple. It checks if the *errCode* parameter is below zero, if so, it is an error and it gets an error description string from the JRTPlib and prints it using the practical utilities.

4.4.3 A Dummy Client

To test the data on the client's side of the video stream, a small and simple dummy client was created. This client simply joins the multicast ip group, and receives any packages that are sent by the server. This dummy client runs as a console application and also uses the JRTPlib to handle the RTP data. When a packet is received by the client, it prints the length of both the packet and the payload to the screen, and dumps the data to a file using the *DumpPointer* method from Code Example 4.9. This it to ensure that all packages arrive and that the data has not been corrupted on its way.

4.4.4 The Threaded Version of The RTPService

As a solution to the network congestion problem, related to sending bursts of packages, a threaded version of the RTPService was attempted. The differences between the threaded version and the other version, is that the threaded server only sends packages within fixed intervals, and it uses a local data buffer. This means that instead of sending the packet as soon as it is encoded, it is stored in a local buffer in the threaded RTPService and when the fixed interval has passed, the server sends the contents of the buffer.

This prototype is a bad solution for the problems, because many frameupdates might be lost, since the local buffer is overwritten for each frameupdate. This might also mean, that the big reference frames are not sent, which will be bad for the stream. The buffering for this solution could be improved by creating a FIFO queue buffer instead of overwriting.

The threaded solution was not inteded as a final solution, it was rather a way to control when and how often packages are sent during the program execution. It was an experiment to see how this affected the stability of the stream. The source for the threaded solution can also be found on the CD-ROM in the appendix.

Chapter 5

Evaluation

This chapter holds the evaluation of the implemented prototype of the service. It holds small testing experiments, results and evaluation of these and conclusion on the project itself.

5.1 Prototype Performance

This section discusses the implemented prototype. Aspects such as testing, the performance and what must be done to improve the prototype, are described in the section.

5.1.1 Testing

Due to the many problems encountered during the implementation of the service prototype, it was relevant to perform a series of tests and use various testing tools to try to debug the complicated mesh of third party libraries working together.

One of the tests performed was using the tool Wireshark [Tea10d], this tool allows interception of all kinds of packets sent and received on a computer, and examining the contents of these packets. This tool facilitated interception of the packets sent by the stream service server, to see if it was actually RTP packets being sent, and what these packets looked like. This gave some assurance that the service was actually transmitting packets properly, and not just failing to transmit.

Another type of debugging performed was to implement several package dumps at various points in the service, as described in Section 4.4.1 and seen in Code Example 4.9. This meant that the intercepted frame was written to a binary file on the disk before being packetized by the RTP library. Another dump was implemented after the RTP packet had been created, and the entire packet was written to a binary file on the server as well. In addition to this, two dumps were also implemented on the simple dummy client that could receive the RTP packets, mentioned in Section 4.4.3. This client wrote the received packet, and the payload of the packets, to seperate files. The package dumps helped with debugging the service. Even though the binary files were largely unreadable to a human, they still revealed fatal flaws in the implementation due to discrepancies in package sizes between what was supposedly sent from the server, and what was received by the client.

Furthermore, various settings for the encoder were tried. The source of the problem was uncertain, and it could very well be the settings in the encoder configuration that produced problems. After getting the video stream to work, these settings were alsu used to tweak the setup. The configuration options are easily managable in a GUI for the x264 VFW encoder, as seen in Section 4.2.2.

The SDP settings were also subject to much trial and error. As seen in Section 4.2.4 and Code Example 4.6, most of the settings are simple and hard to set wrong. The $sprop - parameter - sets$ however are in base-64, and originates from hexadecimal values indicating various flags, luma/chroma settings, ids and scalings of the video encoding, this therefore was subject to much experimentation.

5.1.2 Gameplay Performance

Games usually depend on fast responses and update times, so the user can react to events that happen in the game appropriately, and have a satisfying gameplay experience. This means a game with a significant delay between sending input to the game and the reaction in the game, will decrease enjoyment, and perhaps replace any enjoyment with frustration.

The implemented game streaming prototype service, suffers from a significant delay between the game updating server side, to the client witnessing these updates. This means that currently the service is not suited for the intended purpose of game streaming.

5.1.3 Improvements

In order to make the service work properly and to a satisfactory level, a number of improvements could be made, that could potentially make the service live up the demands and the gameplay performance requirements.

One of these improvements is to implement packet splitting for the RTP server, such that it is possible to send bigger payloads than the limit of 65Kb, and have a higher quality image for the client. This can be done by splitting the payloads up in smaller chunks of less than 65Kb in size, and transmitting these using the RTP server. The RTP packets's payloads can be reassembled by the client, provided each packet that makes up a bigger payload has the same timestamp, and only increments the sequence number to allow the client to know how to reassemble the larger payload. The final chunk of the bigger payload then needs the marker set to signify this. Implementing this type of packet splitting would allow the service to transmit larger frames, and thus allow for an increase in image quality.

Another improvement possible, is to throttle the rate at which packets are transmitted, as discussed in Section 4.4.4 about the threaded version of the RTPService, this theory is backed up by [HP08] on the packet loss performance under different network conditions. Too many packets at once, can choke the network and the client, resulting in higher packet loss and thus lowering the quality and increase the latency of the service. In order to alleviate this problem, packet throttling could be implemented in the server, such that only X packets are transmitted every Y seconds. This could be implemented by keeping track of the time since the last packet was sent, and then transmitting another if enough time has elapsed.

These improvements will help better the service, both in quality and latency.

5.2 Performance of Other Solutions

It was possible to test two of the existing solutions available mentioned in Section 2.7. The two tested solutions are StreamMyGame and OnLive, which are both proprietary. The fact that none of them are openly available, makes it difficult to perform precise testing and measurements of delays between input and reaction. This section provides a subjective evaluation of the solutions. Both solutions cost a subscription fee, but trial versions are available, upon which these tests are based.

OnLive offers the ability to play some games on a trial basis. It is simple to install and it takes short time to get into a game. The look and feel of the client is good, but playing a game can be frustrating, because there is a delay long enough to be a nuisance between input and reaction. The games can be played, but for a fast paced game, it is a delay that will keep many people from using the service. It is important to point out, however, that the OnLive client informed it had detected a high latency for the connection, and thus warned about lowered quality of gameplay experience. This latency issue might be a result of not having any OnLive servers in Europe yet. Opening the service for

Europe is also a recent development, which might be a reason that no servers are available locally.

StreamMyGame provides the possibility to setup a server on the user's computer, and play games remotely. The trial version of this gives full access to the service, but with reduced resolution. The look and feel of StreamMyGame is not as professional as OnLive, but it has lower delay. The lower delay in this case, is due to the server (in the setup we tested it) running on the same wired LAN as the client, and it may also be because the resolution is small.

5.3 Conclusion

A GoD service is possible and plausible, at the moment two implemented proprietary solutions of two different types of GoD systems exist. OnLive stores the games on remote servers, while StreamMyGame allows the user to setup a server locally. This also means that the experience was very different between the two services, where the US-based OnLive service suffers from too much delay when used from Europe, while the StreamMyGame trial version has too low resolution to be a reasonable comparison. Testing of our own solution, has shown that when using the service on a wired LAN connection, a significant delay exists, which meant that it was unsuited for any real gameplay. This delay can come from encoding speed, package loss, as well as other factors. Issues such as these can be solved by tweaking encoder settings, choosing a different encoder, and implementing some degree of package throttling to not drown clients or stress the network.

A number of different network protocol possibilities for the service have been analyzed and from this, it was argued which suited the different connection needs for a finally implemented solution. The result was a proposed complete architecture of a final GoD service, described and explained in Section 3.2.

Finally a prototype was implemented, to explore what kind of problems existed and had to be addressed, if such a proposed GoD service was to be finally implemented. This proved to be no easy task, and the fact that we had novice experience with many of the parts of such a system, resulted in a very steep learning curve for solving many of the problems.

It is clear why it has taken a company like OnLive eight years [Tea10b] to perfect such a system. There are certainly many aspects that need to be fully optimized, for such a system to properly function and even be playable, but it is certainly possible.

Bibliography

[BA10] Rich Brown and Dan Ackerman. Onlive cnet review, 2010. http://news.cnet.com/8301-17938_105-20009033-1.html.

[Gra10] Kris Graft. Onlive turned on (interview), 2010. http://www.gamasutra.com/view/feature/5861/onlive_turned_on.php.

[HJ98] M. Handley and V. Jacobson. SDP: Session Description Protocol. RFC 2327 (Proposed Standard), April 1998. Obsoleted by RFC 4566, updated by RFC 3266.

[HP08] Richard Haywood and Xiao-Hong Peng. On packet loss performance under varying network conditions with path diversity. In *Proceedings of the 2008 International Conference on Advanced Infocomm Technology*, ICAIT '08, pages 106:1–106:7, New York, NY, USA, 2008. ACM.

[HSV$^+$05] A. Huttunen, B. Swander, V. Volpe, L. DiBurro, and M. Stenberg. UDP Encapsulation of IPsec ESP Packets. RFC 3948 (Proposed Standard), January 2005.

[itu10] ITU-T Recommendation H.264 : Advanced video coding for generic audiovisual services, March 2010. http://www.itu.int/rec/T-REC-H.264-201003-I/en.

[JBDG$^+$09] A. Jurgelionis, F. Bellotti, A. De Gloria, P. Eisert, J.P. Laulajainen, and A. Shani. Distributed video game streaming system for pervasive gaming. *STreaming Day*, 2009.

[JBDG$^+$10] A. Jurgelionis, F. Bellotti, A. De Gloria, J.P. Laulajainen, P. Fechteler, P. Eisert, and H. David. Testing cross-platform streaming of video games over wired and wireless LANs. In *2010 IEEE 24th International Conference on Advanced Information Networking and Applications Workshops*, pages 1053–1058. IEEE, 2010.

[JFE$^+$09] A. Jurgelionis, P. Fechteler, P. Eisert, F. Bellotti, H. David, J.P. Laulajainen, R. Carmichael, V. Poulopoulos, A. Laikari, et al. Platform for distributed 3D gaming. *International Journal of Computer Games Technology*, 2009:1–15, 2009.

[KAM08] T. Karachristos, D. Apostolatos, and D. Metafas. A real-time streaming games-on-demand system. In *Proceedings of the 3rd international conference on Digital Interactive Media in Entertainment and Arts*, pages 51–56. ACM, 2008.

[KGW09] Aleksander Kostuch, Krzysztof Gierlowski, and Jozef Wozniak. Performance analysis of multicast video streaming in ieee 802.11 b/g/n testbed environment. In Jozef Wozniak, Jerzy Konorski, Ryszard Katulski, and Andrzej Pach, editors, *Wireless and Mobile Networking*, volume 308 of *IFIP Advances in Information and Communication Technology*, pages 92–105. Springer Boston, 2009. 10.1007/978-3-642-03841-9_9.

[Lie10] Jori Liesenborgs. Jrtplib website, 2010. `http://research.edm.uhasselt.be/jori/`.

[LLJ$^+$10] A. Laikari, J.P. Laulajainen, A. Jurgelionis, P. Fechteler, and F. Bellotti. Gaming Platform for Running Games on Low-End Devices. *User Centric Media*, pages 259–262, 2010.

[OBL$^+$04] J. Ostermann, J. Bormans, P. List, D. Marpe, M. Narroschke, F. Pereira, T. Stockhammer, and T. Wedi. Video coding with h.264/avc: tools, performance, and complexity. *Circuits and Systems Magazine, IEEE*, 4(1):7–28, 2004.

[Per03] Colin Perkins. *Rtp: audio and video for the internet.* Addison-Wesley Professional, first edition, 2003.

[Pos80] J. Postel. User Datagram Protocol. RFC 768 (Standard), August 1980.

[Pos81] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFCs 1122, 3168.

[SCFJ03] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC 3550 (Standard), July 2003. Updated by RFCs 5506, 5761, 6051.

[SRL98] H. Schulzrinne, A. Rao, and R. Lanphier. Real Time Streaming Protocol (RTSP). RFC 2326 (Proposed Standard), April 1998.

[Ste10] Steam. Steam user stats, 2010. `http://store.steampowered.com/stats/`.

[Tea10a] Gaikai Team. Gaikai website, 2010. http://www.gaikai.com.

[Tea10b] OnLive Team. Onlive website, 2010. http://www.onlive.com.

[Tea10c] StreamMyGame Team. Streammygame website, 2010. http://www.streammygame.com.

[Tea10d] Wireshark Team. Wireshark website, 2010. http://www.wireshark.org.

[WHS+05] S. Wenger, M.M. Hannuksela, T. Stockhammer, M. Westerlund, and D. Singer. RTP Payload Format for H.264 Video. RFC 3984 (Proposed Standard), February 2005.

[WSBL03] T. Wiegand, G.J. Sullivan, G. Bjontegaard, and A. Luthra. Overview of the h.264/avc video coding standard. *Circuits and Systems for Video Technology, IEEE Transactions on*, 13(7):560–576, 2003.

[xT10] x264 Team. x264 sourceforge website, 2010. http://x264vfw.sourceforge.net/.

[Zim03] Roger Zimmermann. Streaming of divx avi movies. In *Proceedings of the 2003 ACM symposium on Applied computing*, SAC '03, pages 979–982, New York, NY, USA, 2003. ACM.

Appendix A

List of Abbreviations & Acronyms

VoD	Video On Demand	1
GoD	Game On Demand	1
API	Application Programming Interface	5
GDI	Graphics Device Interface	5
VFW	Video For Windows	7
HD	High Definition	8
MPEG	Moving Pictures Experts Group	8
ISO/IEC	International Standards Organization	8
RGB	Red, Green & Blue	8
ITU-T	International Telecommunication Union	8
IEEE	Institute of Electrical and Electronics Engineers	
JVT	Joint Video Team	9
VCEG	Video Coding Experts Group	9
FFmpeg	Fast Forward Moving Pictures Experts Group	9
MPEG-1	MPEG-1	
MPEG-2	MPEG-2	
MPEG-4	MPEG-4	
H.261	H.261	
H.264	H.264	10
AVC	Advanced Video Coding	

RTP	Real-Time Transport Protocol	10
VCL	Video Coding Layer	10
NAL	Network Abstraction Layer	10
TCP	Transmission Control Protocol	14
IP	Internet Protocol	14
TCP/IP	Internet Protocol Suite	14
FPS	First Person Shooter	14
UDP	User Datagram Protocol	15
IETF	Internet Engineering Task Force	15
SDP	Session Description Protocol	17
RTCP	Real-Time Transport Control Protocol	17
RTSP	Real-Time Streaming Protocol	17
SAP	Session Announcement Protocol	17
SR	Sender Report	18
RR	Receiver Report	18
SDES	Source Destination	18
BYE	End of Participation	18
APP	Application-specific Message	18
URL	Uniform Resource Locator	19
LAN	Local Area Network	20
JRTPlib	Jori's Real-Time Transport Protocol Library	29
GUI	Graphical User Interface	30
VLC	VideoLAN Client	36
FIFO	First In First Out	39