Department of Computer Science
University of Copenhagen

Submitted: November 30, 2006
Advisers: Jyrki Katajainen and Per Høgh
Master thesis for the cand. scient. degree in computer science

Jacob de Fine Skibsted
Stephan Lynge Herlev Larsen

Distributing usage of bandwidth for on-demand streaming

Abstract

This thesis seeks to distribute the overall bandwidth consumption in a client-server network providing video on-demand streaming. This goal is reached by designing and implementing a protocol which employs methods resembling those used in peer-to-peer networks.

A protocol specification which enables forwarding of data between clients in order to lower the bandwidth consumption of the server has been developed. The data stream is divided in in order to enable multiple clients to send a part of the data stream to a single receiver and at the same time offer the full functionalities of on-demand streaming such as pause and skip. Finally, the primary functionalities of the protocol specification has been implemented and used in a set of test applications.

We have succeeded in designing a protocol which can distribute the overall bandwidth in a logical network offering video on-demand streaming. Thus, the bandwidth usage of the central server has been lowered. Based on the design a fully operational implementation has been developed. Calculation of the actual savings is still an open question as this will need a large scale real life usage of the protocol in order to obtain empirical measurements.

The following section contains a Danish translation of the abstract.

Resumé

Dette speciale forsøger at distribuere det samlede båndbreddeforbrug i et client-server netværk, som tilbyder video on-demand streaming. Målet opnås ved design og implementering af en protokol, som benytter metoder, der ligner dem som anvendes i peer-to-peer netværk.

Der er udviklet en protokolspecifikation, som muliggør videresendelse af data mellem klienter med det formål at reducere serverens båndbreddeforbrug. Datastrømmen er opdelt for at muliggøre afsendelse af en del af data fra forskellige klienter til den samme modtager og samtidig tilbyde den fulde funktionalitet ved on-demand streaming såsom pause og skip. Slutteligt er protokolspecifikationens primære funktionaliteter implementeret og benyttet i et sæt testapplikationer.

Vi har succesfuldt designet en protokol som kan distribuere det samlede båndbreddeforbrug i et logisk netværk, der tilbyder video on-demand streaming. Således er den centrale servers båndbreddeforbrug blevet reduceret. Baseret på designet er en fuldt funktionsdygtig implementering udviklet. Beregning af den egentlige besparelse er stadig et åbent spørgsmål idet dette vil kræve en egentlig brug af protokollen i stor skala for at opnå empiriske målinger.

Formalities

This report is the master thesis for the cand. scient. degree of Stephan Lynge and Jacob de Fine Skibsted at the Department of Computer Science at the University of Copenhagen. The thesis was written in the period from the 1st of December 2005 to the 30th of November 2006.

Initially, we would like to thank our advisers Jyrki Katajainen and Per Høgh for their thoroughness and commitment.

Contents

1 Introduction

As computers have become increasingly more powerful the possibility of presenting digital media to the end-user has become ever more present. Watching video clips or listening to music through the use of computers is now a part of most end-users everyday usage of computers.

Together with the rapid spreading of the Internet this opens up for opportunities which will redefine how computers are used in our lives. Listening to radio through the Internet is now a common way of distributing media and every self-respecting radio station now distributes content through the Internet and some are even based only on the Internet.

Distributing videos through the Internet has yet to gain the same level of popularity. This is mainly due to the high requirements to the capacity of the available connection of the end-users. During recent years this amount has been growing rapidly and is now reaching a level where watching a video through the Internet is becoming possible. Thus, renting a video is no longer a question of venturing into the cold, spending time in the rental shop, only to realize that your favourite video is rented out. In the future the rental video is only a click away sitting on your sofa and will never be rented out. But it does not end here. TV viewers will no longer be bound to the TV guide but will be able to decide when to watch their favourite soap opera.

Distributing multimedia content through the Internet is normally referred to as streaming. Streaming differs from normal download in the sense that streaming tries to optimize the use of bandwidth in the regard of only using the exact amount of bandwidth needed to watch the video. Hence, data will be received at the same speed as it is watched. This has several advantages; Firstly, it only uses the needed bandwidth at any time freeing the capacity for other use, thus raising the potential amount of connected users. Secondly, it is possible to start watching a video immediately avoiding the delay it would impose if the end user would have to wait for a complete download. Finally, it helps protecting the copyrights of the owner since only the currently displayed fraction of video needs to be present at the client.

This thesis focuses solely upon streaming of video as streaming of sound is not associated with the same difficulty as the bandwidth requirements when streaming sound are small enough not to represent a problem neither to the streaming server nor the client. Thus, streaming of video at high quality represents a problem which will have to be solved before the video rental scenario above becomes reality.

Describing the ultimate goal of this thesis calls for a clarification of the different approaches when streaming video:

Live streaming: This term refers to a method of streaming video ”live” to the end-user. The user receives a stream of video without the ability to fast forward, rewind or pause the stream. This is analogous to a regular TV-signal transmitting a football match or a news broadcast. Thus, only one stream of data is transmitted to all end users.

On-demand streaming: This term is applied to an approach which tries to resemble the functionality of the VCR or DVD player. The user receives a stream of video ”on-demand" meaning that the user controls the content of the stream. Thus the user decides when the stream is dispatched and is able to fast forward, rewind, or pause the stream.

Near on-demand streaming: This term is used for a system using a hybrid of live streaming and on-demand streaming. Using this approach a new copy of the same stream is dispatched with a certain interval. The user signs up for the stream and waits until the stream is dispatched. Thus the user is partly in control of when the stream is dispatched, and can only fast forward, rewind or pause the stream by jumping to a channel dispatched with a different time offset.

The focus of this thesis will be on-demand streaming resting upon the above definition combined with the requirement of distributing bandwidth usage. Distribution is referred to as dispersing the bandwidth usage among several network links, instead of the typical setup where the link to the main server carries the highest load. The total sum of needed bandwidth remains the same, but a more even usage is obtained as the sum is distributed among many links, reducing the required bandwidth capacity of the server.

1.1 Motivation

During earlier work [7] we have been confronted with the fact that the largest part of the costs involved when streaming video over the Internet are bandwidth expenses. This is due to the enormous amounts of data transports incurred when streaming video.

Transporting large amounts of data has always been an issue in computer science. Solving this type of problem sometimes ends up presenting a distributed system using a peer-to-peer model. This can be seen in file sharing environments like BitTorrent [1] or eDonkey [8].

Thus, combining streaming of video with the advantages and ideas taken from peer-to-peer like networks is the primary motivation for this thesis. Using technologies and ideas of this type, the wish is to create a system which offers on-demand video streaming with the ability to reduce the needed bandwidth of the server.

As the required bandwidth for streaming of high quality video is only just starting to become easily accessible to the average consumer, no real *de-facto standard* or off-the-shelf product has yet been developed. Only time will tell which direction this technology will take but by this thesis we hope to make a small contribution to this development.

1.2 Objectives

The main objective is to create a system which can be used to minimize the bandwidth of the streaming server when streaming video on-demand.

This thesis is comprised of multiple parts each defining their own objectives:

Design of a protocol: As the system will constitute server and client entities a network protocol to facilitate communication between these will have to be designed.

Implementation of the protocol: The protocol specification should be implemented with respect to the specification requirements. Furthermore, the implementation should be made modularly.

Implementing of applications: One or more applications should be implemented as a proof of principle possibly a server and a client.

Validation: The design should be validated to create a proof of concept.

The thesis will focus upon the designed and implemented version of the protocol. Thus, the design and implementation will be documented thoroughly with the result as the main objective.

1.3 Report layout

This report has been typeset using LaTeX2e. The report has been written using the Oxford style English language. Thus, synchronize is for instance deliberately spelled using the *-ize* ending.

The structure of this report illustrates the chronological work process used during this project. The report is divided into the following chapters:

I Requirements and solutions: This chapter dives deeper into the subject defining the formal requirements. We describe the requirements from a less technical perspective narrowing down the theoretical areas which will have to be examined by the study. We outline a rough solution to the given problems based on initial ideas and thoughts. These solutions will be used as reference to the set of problems until the final design is laid out.

II Related theory: In this chapter we analyse the found literature. The areas defining this theory are for the main part computer networks, multimedia coding and application development. The selection of these areas takes its starting point in the requirements and solutions found in the latter chapter. The main purpose of this chapter is to examine important parts of the theory, thus giving a foundation for the design of the system.

III Protocol design: This chapter analyses the requirements given in Chapter I and based upon the theory examined in Chapter II the design of the system is accounted for. This chapter will be the final result of several iterative passes aiming at clarifying the flaws and ambiguities of the design.

IV Protocol implementation: This chapter describes the implementation of the protocol and its components developed during the course of this thesis. This will also include a schematic view of the components of the implementation.

V Verification: This chapter seeks to verify the protocol design and the implementation. In the chapter practical tests will be conducted to validate the protocol implementation. Furthermore, the chapter includes a discussion of the achieved bandwidth consumption.

VI Closure: This chapter dives into the perspectives of the thesis and concludes upon the achieved goals.

These chapters illustrate the phases when working with the thesis. All chapters are of course revisited iteratively after the completion of every phase.

This report is written for students at a similar level as the authors or the like.

The thesis aims at realizing a design of a distributed on-demand streaming network and will therefore contain a thorough description of this design. Readers can skip to this in Chapter III if the underlying theory is of little or no relevance.

As the report contains a collection of specific technical terms, a glossary is provided on page 177. Throughout the report these terms, whenever appearing for the first time, will be written in *italics* and a description is given in the glossary.

Chapter I

Requirements and ideas

In this chapter we will present the basic requirements for the system listed by overall areas. For the reader to get a general idea of how these issues can be solved, this chapter furthermore contains a section which sketches a solution to each of the listed requirements. Thus this chapter serves as means of getting the reader acquainted with the fundamental elements of the system before the theoretical foundation can be accounted for.

2 Fundamental requirements

This section describes the main requirements to the system. These requirements are a result of our expectations to a system of this type, our technical knowledge combined with a wish for creating a usable outcome.

2.1 Server bandwidth usage

As described earlier the primary objective of the system is to lower the *bandwidth* usage of the server. Fundamentally, this means that the bandwidth usage at the server must not rise linearly with the number of connecting clients. For instance, if a server has a capacity of 100 Mbit/s and is *streaming* a video requiring 2 Mbit/s this would result in a maximum capacity of 50 streams. This is exactly the problem which we wish to solve by raising the above capacity of 50 streams without increasing the bandwidth capacity of the server of 100 Mbit/s.

2.2 Efficient scalability

The *protocol* should be scalable to connect a large number of clients. Thus the number of connected clients should not be limited by the protocol design. The scalability is defined with respect to the number of connected clients compared to the bandwidth usage of the server.

2.3 Traffic shaping

It should be possible to control and prioritize the individual data streams sent to all parts of the network. We refer to this as *traffic shaping* although this may not be in complete accordance with the traditional definition of the term. For instance, if a service provider wishes to enforce a minimum usage of a specific network link, it should be possible to satisfy this wish.

2.4 Design and implementation

The protocol should be designed using a modular approach offering easy extension of future design. The specification of the protocol should be thoroughly documented.

Furthermore, a *server application* and a *client application* should be designed and implemented, which demonstrate the usage of the protocol implementation. This will be part of our proof of concept verifying that the idea works as intended.

An enclosed implementation of the protocol specification should be created, which offers an easy-to-use interface, enabling developers to implement their own applications using the protocol implementation.

The protocol must be implemented using standard libraries and components and be based on the TCP/IP stack. This ensures that the implementation is usable on the Internet and will be portable as required by the objectives.

For the sake of clarification, we stress that our implementation will be developed for a standard computer with no regard to specialized hardware devices.

2.5 End-user functionality

To fulfill the demands for creating an on-demand video system the end user will need to have a certain degree of control of the video stream. This involves instant play back, pause and go backward/forward in the video, as seen on a common VCR. Media interaction can be broken down into two types [41], continuous and discontinuous functions. The first category comprises fast forward and rewind known from analogous media. The second one, comprises skipping from one part to another as known from digital media like DVDs and CDs.

The system should further support that all end users do not necessarily have the same bandwidth available both regarding upstream and downstream. This will reflect the current situation on the Internet as will be seen in Section 6. As many users of the Internet are using *asynchronous bandwidth* it is not feasible to exclude clients with an upstream bandwidth not sufficient enough to forward the stream — this would simply exclude too large a part of potential users.

2.6 System security

The design of the system should invite implementation of a robust security policy. To obtain this, different aspects of security must be accounted for. Therefore security issues can be divided into three main parts:

Authorization: The system should verify and grant or deny access to newly connecting users.

Data integrity: No end-user should be able to tamper with the data stream and thus influence the content of the stream sent to other users.

Data theft: No end-user should be able to steal data from the system, e.g. receive a video stream without being acknowledged by the system. This does not ensure that clients cannot send the video stream to unacknowledged clients outside the system.

It is unlikely that all of these items will be solved with all security details in mind. The main goal is to design a version of the protocol enabling future development to benefit from the design.

2.7 Quality of stream

To create a usable system, it should deliver a stable stream of data ensuring that the video is shown without disruptions. Also, partial network disruptions should not affect delivery of data. Of course, larger network disruptions such as a link breakdown or a system crash at a central point will have to be accepted as a non-recoverable failure which can potentially affect other users.

3 Fundamental ideas

Based upon the requirements described in the previous section we illustrate the solutions and technical details of the fundamental ideas. For each of the requirements a rough solution will be outlined. These rough solutions will help identify areas of theory which will be studied further to create a theoretical foundation for this thesis. We emphasize that the solutions outlined in this section are only a sketch and will not be finalized until the design phase is completed.

3.1 Server bandwidth usage

To fulfill the requirements concerning minimization of the server's bandwidth, it is obvious that sending a stream of data to every client is not applicable. This approach will result in a bandwidth consumption increasing linearly compared to the number of connected clients as illustrated in Figure 1.

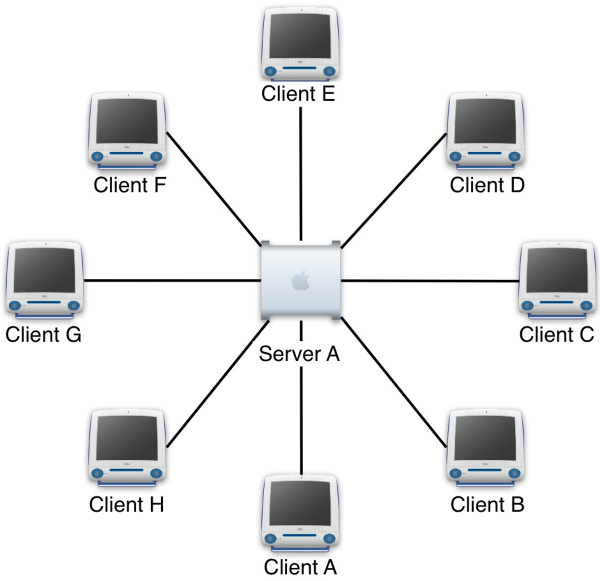


Figure 1: Star topology.

Another approach is to let the server send data to a number smaller than the total number of clients in the network. This combined with the clients forwarding data to each other will create a network with a topology resembling a tree or a mesh as seen in Figure 2. This will reach a bandwidth usage which may be lower than the simple solution using one stream per client. This solution will be referred to as *bandwidth sharing*.

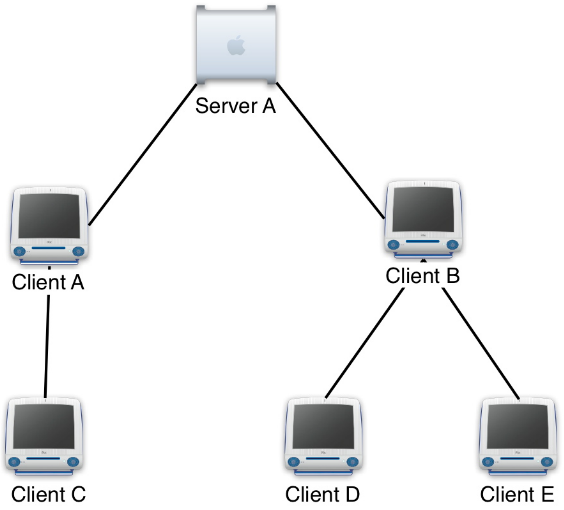


Figure 2: Tree of clients.

However, these two models cannot be directly compared unless we take the following main issue into account: The first model opens up for the ability to stream any data to any client at any given time. The other model is limited by the fact that a client receiving data from another client is restricted to receive data available at the sender. Hence, on-demand streaming is not directly possible, as it will result in all clients watching the same point of time in the video. The solution to this problem is to introduce a buffer at the client containing a certain portion of the video. Thus client A depicted in Figure 2 could watch one portion of the video while client B and attached clients watch another portion. The clients attached to client B do not have to watch the exact same point of time in the video but has to lie within the portion of the video held by the buffer og client B. Client A could even be watching a completely different video than client B and its subtree, hence each different video or different section of a video result in its own subtree.

Connecting clients to the network represents two scenarios. One is the newly connecting client requesting to receive the video from the starting point. The other is the client requesting the stream from a given point of time in the video. The latter scenario could represent a client wishing to resume the stream after pausing it earlier on.

Both of these scenarios are solved by the network selecting the optimal entry point to the connecting client. This ”optimal” choice is taken by the server selecting a client which at the time holds the correct part of the stream requested by the connecting client. If the server cannot find any suitable client, it can either reject the client or stream the video directly to the client. Splitting the video into x parts and enabling the clients to buffer an entire part, will result in the need for, at most, x available streams per movie at the server. All other clients would receive the stream from a client connected to the server. Introducing this functionality we can compare the two models.

Furthermore ensuring basic on-demand streaming functionality such as instant start-up, pausing, fast forward and rewind represent a different and more complicated challenge.

Also client or line break down will have to be handled differently as a client break down in Figure 1 will not effect more than one user.

3.1.1 Multiple servers

Extending the first model in Figure 1 could be done using multiple servers — a scenario which has been practiced with commercial success in other areas of computer networks. In spite of this, the scenario still scales in a linear form based on the number of clients. This solution should by no means be neglected, but this approach has no value to the specific scenarios which are treated in this thesis since it would not result in any further bandwidth reduction in total.

The second model, see Figure 2 could also be replicated to use multiple servers, as shown in Figure 3.

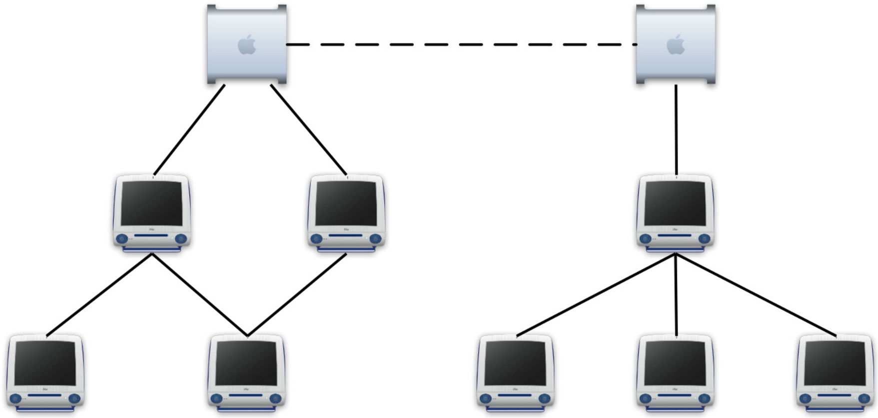


Figure 3: Joined servers.

Adding multiple servers to any of the models does not contribute to a theoretical lowering of bandwidth usage but due to the structure of the Internet, the use of multiple servers could contribute to significant savings regarding bandwidth cost and rapid extension of the server's capacity. In order to connect multiple servers another protocol is needed which will have a main task of distributing data among the servers.

This structure permits that transportation of data in the core of a network can be done at a low cost due to the high capacity connections linking the central network locations. Furthermore, the structure can also have a single string connection to a geographical point from where on data are distributed to a number of nodes. This scenario could also benefit from a multiple server setup avoiding starvation of a single connection.

On the edge of the network, where the end-user is typically situated the connections usually have a lower capacity. Taking a closer look at the market[1] for end-user connections reveals a palette of different options. Most common are the *asynchronous connections* for instance ADSL or Cable. A typical connection has a maximum downstream rate of 2048 Kbit/s and a maximum upstream rate of 512 Kbit/s. These connections are widely spread and faster are often seen, as will be elaborated upon in Section 6. Furthermore, the trend is that connections are getting even faster.

Obviously, we choose a model based upon the second scenario (Figure 2). The model uses a single server communicating with an arbitrary number of connected clients. Combining the two scenarios to yield a multiple server setup could be done by implementing the communication channel between the servers. As the focus of the thesis is upon the client-server communication the server-to-server communication is left as further work.

3.2 Efficient scalability

As previously stated the maximum number of connected clients in the first model scales linearly with respect to the maximum bandwidth capacity of the server. The second model obviously has the ability to scale beyond this point. The theoretical number of connected clients using the second model may seem infinite as the load upon the bandwidth is taken away from the server. Of course, this property introduces complicated issues which will be dealt with in the rest of this thesis.

Thus the demand for scalability is fulfilled solely by choosing the second model.

3.3 Traffic shaping

As described in Section 3.1.1 it can be an advantage if data can be routed by a specific path. It might be preferred to limit the use of specific links based on different causes, e.g. low capacity, high carrier price, long delay, or other factors.

An example could be an island with only one link to the mainland with limited capacity. Should two clients on the island be connected, it would be more efficient to connect these to each other and only one of these to the mainland. This would distribute the bandwidth usage away from the server and furthermore take the load off the link to the mainland resulting in traffic shaping.

Expanding this example, it could be the case that the users on the island are watching a lot of videos. Hence it could be an advantage to place a server on the island as referred to in Section 3.1.1.

[1] As this thesis does not treat the market or economy for end user connections we concentrate upon the market in the western world. This is due to the high availability of Internet connections and the massive presence of the entertainment industry.

3.4 Design and implementation

The design and implementation of the applications using the protocol will be done with regard to the previously mentioned requirements given in Section 2. Except for the requirements regarding the enclosed solution the other requirements do not need further explanation.

The implementation of the protocol presented in this thesis will focus upon simplicity and will be implemented for standard computers.

Enclosing the solution refers to a design which encourages an implementation of a simple interface enclosing the logic in the protocol. This will enable developers using the protocol to implement applications easily. To give a general idea of the system, Figure 4 illustrates the main components of the design. The left side illustrates the server side of the protocol, while the right side illustrates the client side. The middle layer defines the protocol and its general components. The protocol interfaces downwards with *Application Programming Interfaces (API)* provided by the operating system facilitating network communication. Upwards, the protocol provides an API used by applications. These applications will from this point be referred to as server applications and client applications. Accordingly, the server side of the protocol will be referred to as *server protocol* or just *server* while the client side will be referred to as *client protocol* or just *client*. The term 'protocol' will be used when referring to both. The API provided by the operating system is the lower part of the drawing. The protocol layer is coloured green, indicating that this part will be designed and implemented in this thesis. The implementation of the server and client applications will only be done to illustrate the functionality of the protocol, with no consideration to efficiency or scalability.

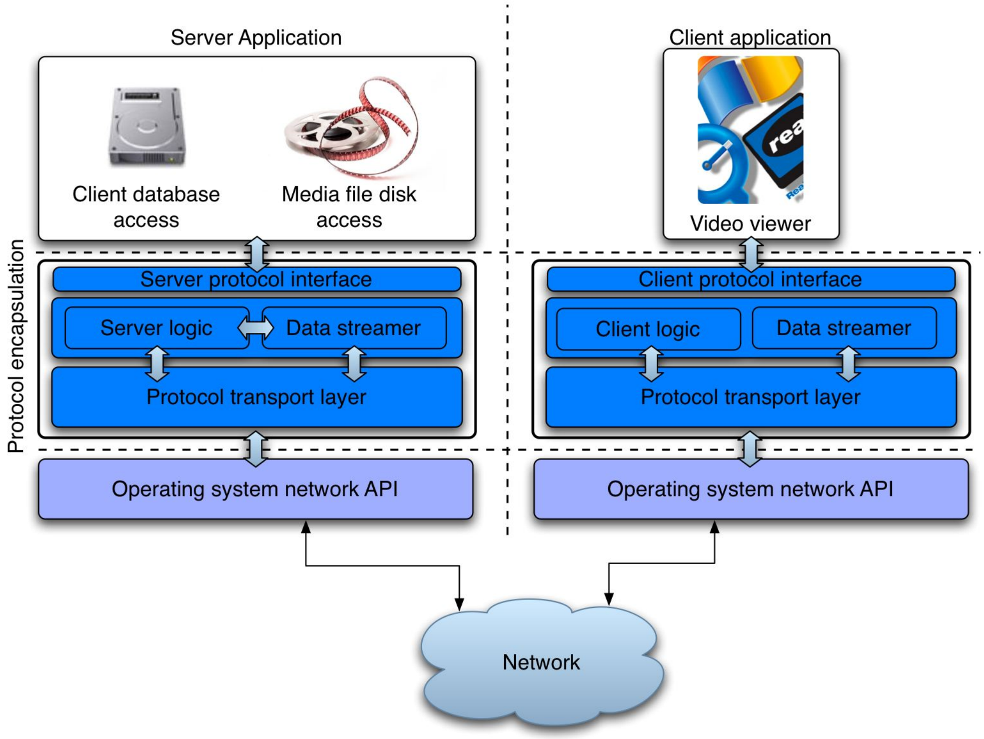


Figure 4: Architectural overview of the system

3.5 End-user functionality

The system should offer the-end user functionalities similar to what is expected of on-demand video regarding display of the video stream. It should be possible to forward/rewind and pause the stream. Apart from this, the user should have the possibility of stopping the video altogether without any effect upon other clients dependent on data sent from the client. Hence, stopping and pausing the stream resembles each other closely, as pausing and stopping the stream has the same effect upon clients receiving data from the interacting client. Finally, instant start-up of the stream is an issue to be examined.

3.5.1 Stop

When a client interacts with the system to stop the stream, the system needs to select another client from which all clients receiving data from the interacting client must now receive data. If the interacting client has no clients connected, it leaves the network without further complications.

The system should be able to detect a link or client failure. This event is opposite to the user stopping the video as there will be no interaction from the client. In the event of interaction the client will notify the server that the user has stopped the video. In the event of a line or client failure this will have to be detected in another way. Clients receiving data from another client which is failing will at some point conclude that the sender has stopped transmitting data.

3.5.2 Pause

Pausing the video could result in a similar behaviour as stopping the movie – attached clients need to be relocated in the network. When the stream is resumed, the system needs to attach the resuming client to another client which holds relevant data. Hence, starting and pausing are closely related, as the only difference is that the resuming client resumes the stream at an arbitrary point in the video stream.

3.5.3 Fast forward/rewinding the stream

The traditional VCR player offers the user possibilities of fast forwarding or rewinding the video. To introduce a true video on-demand system it is important to present functionalities which resemble this.

Advancing the video stream in a speedy manner involves receiving the stream at for example double rate and likewise playing the video at double rate. This injects a heavy load onto both the sender of data and the client advancing the video concerning computing power and bandwidth usage. Alternatively, a scaled down content of the stream could be sent including only a subset of the *frames* contained in the stream. This would remove the bandwidth load from both sender and receiver but the sender would instead have to analyse the stream and extract a subset of the frames and send these accordingly. This would instead raise other complications which would have to be handled.

Instead of trying to resemble the VCR player another approach as argued in [45] which suggests that the perception of interacting with analog media differs from digital. Hence, there is a need for changing the perception of the end-user. If the system had the ability to skip to an arbitrary sequence of the movie, this could constitute and replace the fast forward/rewind functionality of the VCR. Hence, combining the solutions of the ability of the clients to stop and start the video at any point, we see that the problem of skipping to an arbitrary sequence of the video is solved.

Thus, the solution to the issue, will be to implement discontinuous interactive functions which will constitute the functionalities of continuous interaction. The functionality will from here on be referred to as "skip to another sequence".

3.5.4 Asynchronous client bandwidth

As required in Section 2.5, clients equipped with an Internet connection using asynchronous bandwidth should be able to participate in the network. In practice this means that a client can have a downstream capacity sufficient for receiving a video stream but not enough upstream capacity to send data to the next client.

This problem can be solved in different ways. A solution to this problem is to let clients with a low upstream to send only a portion of the data and hence let a client receive data from several sources. This will also imply that a client with a large upload bandwidth could send data to multiple clients.

3.6 System security

Creating a system with a security so robust that no one in any way can bypass the system, is only possible by unplugging the network cable. Realizing that a system can only be secured to a certain degree is important when designing it. The main goal when designing a security system is to make it extremely difficult to bypass even for highly skilled individuals with a firm knowledge of the area.

3.6.1 Authorization

Authorization is a well-known area within computer science. Hence, lots of different approaches can be used to solve this problem. One way is to let connecting clients receive a token from the server and use this to access the network. Another solution is to let the server grant access to new clients and inform all other clients. These are just examples of different approaches useful when designing an authorization mechanism. Common to both of these solutions are that they are based upon a central register maintaining information about users and access rights. Thus, no client can grant others access without going through the server. This basic principle will be the cornerstone of the design of the authorization mechanism in this system.

3.6.2 Data integrity

Constructing a network of this type implies that clients have the opportunity to corrupt data before passing it on to the next client as other clients can be completely dependent upon the behaviour of other clients in the system. For this reason the client must be able to verify that data received from other clients are valid. This can, for instance, be done using a checksum.

3.6.3 Data theft

The issues discussed in this section are closely related to authorization. When the client is authorized, the server cannot know how data are handled by the client. The client could pass data on to other clients without the knowledge of the server and thus compromise security. The design should be extendable with security mechanisms ensuring that data are only passed on to authorized clients. One way of doing this is to encrypt data with certificates sent at intervals by the server. This solution is more safe but will never safeguard the system completely – the malicious client will always be able to decrypt the stream and pass data on in unencrypted form or even exchange certificates. For instance, it will still be possible for the end user to record the stream using an old-fashion VCR or even a hand-held video recording of the screen. All of these issues are out of the scope of this thesis. The system will only be responsible for the security until the stream is delivered to the receiving unit whether being a screen or a streaming application. From here on, other security mechanisms must be brought into use.

Complete securing of data are out of the scope of this thesis. No matter how tight the security is it will always be possible to construct malicious code. This is especially true when working with an open specification available to anyone. Designing the system using an enclosed specification neither contributes as any piece of software can be reverse engineered.

The solution is therefore to make it troublesome for malicious clients to compromise data security. This can be done by letting the clients receive vital information about the video stream from the server at different points in time.

This report will contain theory, which outlines different ways of protecting digital data, but only a very limited fraction of this will actually be implemented.

3.7 Quality of stream

One of the key points when securing a continuous stream of data is to read data into a buffer before play back. The size of the buffer should not be larger than necessary but still large enough to deliver a stable media stream even if the data stream should be delayed in shorter intervals. If the sending client pauses the stream, the buffer on the receiving client should be large enough to leave time for the server to elect a new sending client and initiate the transfer.

Chapter II

Related theory

In this chapter we will cover the basic theory needed for the design. Some of these sections may seem unnecessary, but as the topic of this report covers many domains, we find it only fair to provide a theoretical overview of the areas which are used in this report. Furthermore, the degree of detail provided in the various sections of the chapter will only be taken to a level sufficient enough for the design of the system.

4 Synthesis

In order to map to different kinds of theory needed in this thesis a synthesis will be outlined in this section. This synthesis will serve as a tool to determine the different domains which should be covered in the following sections.

Multimedia encoding: A basic requirement concerning streaming of video is a survey into theory regarding multimedia encoding. This is needed to determine the nature of multimedia standards and formats, how they work, and which of these should be supported by the protocol.

The Internet: As the goal is to send multimedia via the Internet a analysis of the possibilities and weaknesses of the Internet is needed. Streaming the large quantities of data used in multimedia requires a certain degree of Internet connection stability. Thus, we must cover the stability of Internet connections, and how to handle the issues arising from the lack of stability.

Network topology: Building a logical network suitable for streaming multimedia data involves an analysis of different topological models, their advantages, and disadvantages. This analysis will partly be based upon existing theory and partly upon solutions already described by various articles.

Protocol design: Clearly, we must cover theory regarding the principles of designing network protocols. This should cover how protocols are structured and designed, the usage of underlying protocols, and the advantages and disadvantages of these.

Protocol implementation: Having designed a network protocol, theory regarding the implementation of this should be covered. This will include basic software implementation issues which can be specifically applied when implementing protocols.

Data buffering: As described in Section 3.1 the need for injecting a buffer in each client is evident. Therefore we must cover theory regarding buffering of data.

Security: To fulfill the security given in Section 2.6 we need to cover theory and methods related to this.

5 Multimedia coding

Streaming of multimedia normally involves *encoding* and *decoding* the content. This section provides the basic theory regarding this subject. The main focal point of this section is to give an introduction to the area in order for the reader to become familiar with the subject.

5.1 Video codec

The term *video standard* is used in this thesis referring to the structure of how picture and sound[1] is organized. A standard does not explain the process of organizing data or how it is recreated again. These processes are known as encoding and decoding and are done by the codec.

Codec is an abbreviation for ***Co**der*/***Dec**oder* and is basically a piece of software or hardware, which is both responsible for compressing the raw material and decompressing it again. The codec is designed to conform with the video standard which it is coder/decoder for. As previously stated, a standard does not specify how data is converted (encoded and decoded), hence many varieties of codecs implementing the same video standard are available.

5.2 Requirements

This section describes the minimum features which a usable video standard will need to support, should it be able to be transported by our protocol. These requirements are no more than a technical extension of the previous ideas and assumptions found in Section 3.

Streaming: The video standard must be suitable for streaming. A standard which, for instance, requires that the entire file is available locally before play back, cannot be transported by the protocol.

Controllability: Since one of the primary goals is to provide the end user with the ability to interact with the play back of the video, the standard must support pausing and skipping. A standard needs to support a way to implement these functions.

Splitting and joining the stream: Another goal of this project is the ability to distribute the stream among several clients. In order to do this, the video standard must be capable of being transported in smaller entities so that data can be divided and reassembled again. Thus, an arbitrary part of the data file must be transportable by the protocol and hereafter delivered to the multimedia player. How this arbitrary part is handled is completely up to the codec. Thus, the protocol relies upon the codec to be strong enough to handle this.

[1] From this point on referred to as frames and audio, which are the expressions normally used in the literature.

The last requirement is by far the most important. Furthermore, it is obvious that if this requirement can be fulfilled, the remaining ones can be neglected. This becomes evident as skipping can be made by jumping from one part of the video to another. Pausing is just done by halting the display of the video sequence and restarting it again at the same position. Finally, streaming is possible if the codec has the ability to play any part of the video independently of the rest.

Two main issues are important. First, the video standard must support the requirements given above. But apart from this, the codec used when coding/decoding the material must not change the ability of the standard to fulfill these requirements. Thus, if a standard supports controllability, the codec must not be implemented, so that the data stream does not possess this property any longer. Hence, when using the protocol, measures should be taken to verify that the codec does not constrain the functionalities of the standard.

5.3 Standards

Due to the existence of the large variety of different video standards, some of which have become more or less accepted, a brief introduction is in place. The best known group of standards is the MPEG-standards made by Motion Pictures Expert Group [27]. This group was begun in 1988 aiming to develop a set of standards regarding multimedia formats. The rest of this section will mainly involve standards developed by this group, as they have more or less become a de-facto standard.

The first standard released by the group was called MPEG-1. This was mostly based upon well-known techniques used in areas of computer science, for instance compression of still images in the JPEG format. The MPEG-1 standard was designed to use a *bit-rate* of 1.2 Mbit/s. Thus it had become possible to compress a complete video to a relatively small size compared to, for instance, a DVD. This high compression rate of course meant losing quite a bit of quality.

Later came the MPEG-2 standard which is similar to the MPEG-1 standard but designed to deal with higher resolutions. This was originally designed for broadcasting video at a bit-rate of 4 to 6 Mbit/s. The MPEG-2 standard is today known as the standard used by DVDs.

Work was also started on an MPEG-3 standard whose primary target was *High Definition Tele Vision* (HDTV). It was later discovered that the MPEG-2 standard could be changed to cover this and therefore the work was incorporated into the MPEG-2 standard and work on MPEG-3 stopped.

Finally, the MPEG-4 standard was created with Internet multimedia applications as the main target. The MPEG-4 standard is closely related to the well-known DIVX ;-) *[sic]* and DIVX standards made to resolve licensing issues and patent rights. MPEG-4 is designed for heavy compression which it does rather successfully. This has made the standard popular together with the DIVX standards. The technology used to compress data is different from what is used by MPEG-1 and MPEG-2. Quality is said to be not quite as good in DIVX as in MPEG-2 especially when using large resolutions and with a lot of changing frames in for instance action movies. This is due to the method used when compressing frames.

Another standard which deserves mention is Windows Media Video (WMV) — Microsoft's attempt to design a specialized version of the MPEG-4 standard. Unfortunately, the latest versions of this standard are no longer compatible with the original MPEG-4 standard. As a side remark it should be mentioned that the first DIVX standard (the DIVX ;-)) was a hacked version of the WMV standard and by today the WMV is still not an open standard.

5.3.1 Choosing a video standard

At this point it should be emphasized that the goal of this project is not to build a system bound to a specific video standard. The system should rather be used with any standard as long as it fulfills the requirements as stated in section 5.2. Despite this, an introduction to MPEG-2 is in place as this standard was our initial target, since it offers high quality video known from HDTV which we believe will become standard requirements to streaming of video.

5.4 MPEG-2

The MPEG-1 and MPEG-2 standards have a certain amount of shared structure. This section contains a quick overview of the two standards and their differences.

Most literature is focused on either coding of frames or audio. Obviously these two are the main components of multimedia coding but far from the only parts. To give a picture of this, at the time of writing the MPEG-2 standard contains nine different parts, with more under way and only a few of them concerning frames and audio. A third very important part is called system, which acts like the 'glue' between frames and audio.

The following section will contain a quick overview of frames, audio and system in MPEG-1 and MPEG-2.

5.4.1 Frames

Clearly, one of the main elements in a stream of video are the frames of the video. Video can be considered as a continual stream of pictures shown at a fixed interval, for instance 25 frames per second (referred to as the *frame rate*). Hence, another often used name for video is moving pictures.

A very simple approach for implementing moving pictures is to display a stream of JPEG pictures[2]. The only problem is that the compression factor is normally around 10-20 for a JPEG picture. Therefore the size of the video will be substantial. More important, a large

[2] This video format is called MJPEG

amount of the information in the stream would have no relevance as only a small part of the frames in a typical movie changes from one frame to another. This advantage is used in the MPEG-1 and MPEG-2 standards. Instead of sending a complete frame every time, the main idea in MPEG-1 and MPEG-2 is to send a complete frame every now and then and in between send enough information to maintain an appropriate appearance of the picture.

For example: In a movie sequence, a person might only move his or her head, but the surrounding background stays more or less the same during the sequence. Instead of sending a series of full frames, each containing only small changes, the idea is to send the first full frame and then a number of frames correcting the appearance to reflect changes. Every now and then a complete frame is sent, both to ensure that corrections stay synchronized and because eventually, a big part of the picture has changed anyway, that it might as well use a complete picture.

5.4.2 Audio

MPEG-2 audio standard is an extension of the MPEG-1 audio standard which operated only with mono and stereo. MPEG-2 audio supports a total of five channels used to create surround sound. The MPEG-2 audio standard is further extended in some cases to support better sampling rates to improve sound quality. Last but not least the MPEG-2 audio standard is backward compatible with the MPEG-1 audio standard. Hence, an MPEG-2 video stream could use MPEG-1 audio.

The main idea in compression of audio data in MPEG-1, and hence also in MPEG-2, is to cut off all frequencies and parts of the sound which cannot be heard by the human ear. The MPEG-1 audio format is divided into three different layers sharing the same layout consisting of a sequence of audio frames with a header and sound data using constant frame rate. One of the better-known layers of the MPEG-1 standard is MPEG-1 layer 3, also known as MP3.

Finally, a standard called MPEG-2 AAC has been created. This standard is more advanced and builds upon the same principles as those found in MPEG-1 layer 3 standard but with some improvements. It is beyond the scope of this project to look further into this standard and we will for the time being just acknowledge that it exists.

5.4.3 System

This part of the MPEG standards is concerned with combining frames, audio and other data streams into one or more streams. A typical video sequence contains both images and sound and could furthermore contain data like subtitles. The system handles the task of putting these elements together. Synchronizing frames and audio together into one is called *multiplexing*. Doing so, one ends up with one continuous file, where the picture and the sound has been synchronized. The opposite of multiplexing is called *demultiplexing*.

A typical MPEG-1 or MPEG-2 file has been multiplexed (frames and audio is the same file), and when the file is shown using a multimedia player, the data is demultiplexed and handled separately as described above.

5.5 Existing software

To investigate further, we did a set of experiments to examine different implementations of multimedia players and codecs. As a key element of the system is to create a protocol which interfaces with a client embedding a multimedia player, the first focus was set upon different implementations of players. The three main requirements to a player are that the player should be available upon a windows platform. Secondly, the player must be able to be used as a component embedded in a Windows application. Lastly, the player must be able to play videos using the MPEG standards. Quickly this was narrowed down to either Apple's Quicktime or Microsoft's Windows Media Player.

To choose between these, we conducted a test to see how robust the used multimedia players were. A number of media files were played seeking to stress the player. These media files were edited using standard text editors, and random text was inserted at random places. These media files were played accordingly using the same codecs and the behaviour of the player was noted. Depending upon the amount of damage inflicted to the media file both players would in some cases stall for a while, scan forth in the file, until valid data was found. In some cases the player would just jump to the next valid frame. Both players seemed quite robust, with the main difference that Apple has chosen a more open source like approach when designing Quicktime. Contrary to this, Windows Media Player has a more enclosed structure. This ultimately has led to the choice of using Quicktime as the player.

5.6 Video samples

This section tries to show a few examples of typical videos, their sizes and bit-rate providing a foundation for further discussion and give the reader some formal numbers to refer to. All of the shown numbers are based on 90 minutes (5400 seconds) of video.

Type	Quality	Size (MB)	Bit-rate (Kbytes/s)
MPEG1	Low/Medium	200	≈ 38
MPEG4/(DIVX)	Medium/High	800	≈ 152
MPEG2	High (DVD)	4000	≈ 760

Table 1: Typical streaming figures for movie types

Table 1 illustrates typical examples of videos and their respective bit-rates. The examples illustrate only video encoded using *constant bit-rate* (CBR) as the size of a video encoded using *variable bit-rate* (VBR) is dependent on the content of the video stream.

5.7 Conclusions

We acknowledge that the field of multimedia coding is comprehensive. Thus, we have chosen only to elaborate upon the MPEG-2 standards as this was the initial target of the protocol. In addition to this the video stream can be considered as binary data of any type. Because of this, only limited time will be spent in the rest of this project regarding video coding and instead focusing upon the main objectives of the project.

Furthermore, using MPEG forces the need for only using files encoded using CBR, as streaming of data encoded with VBR further complicates the task. Obviously, streaming content encoded with VBR represents a field in itself, as some parts of the movie suddenly become more time critical than others.

6 Structure of the Internet

The Internet as we know it today has been developed over the last couple of decades. Today the Internet consists of millions of *hosts* and an equally large amount of connections between these hosts.

All of this has developed along the years and has brought life to new standards and technologies, some of which have become the Internet's main characteristics.

6.1 The elements of the Internet

The Internet consists of a large amount of *nodes*, where the term node covers a large variety of equipment: PCs, servers, routers, switches etc. Equipment like routers and switches are designed for the purpose of connecting network segments together but in overall the term node is used for all the mentioned items. Two logically interconnected PCs on the Internet are denoted hosts while the connection may span across a number of physical *hops* interconnected by nodes.

The nodes transport entities called *packets* which contain the different informations exchanged between the nodes. These packets include a *header* containing various control information.

All of these nodes are interconnected by a wide range of different connections. As the type of connection is of no relevance to this project, no further elaboration will be given on this subject since it makes no difference to this project what type of media is used for data transport. The only thing which is worth notice is that these connections have individual capacity, measured in the amount of data which can be transported per second. Furthermore, sending data across the connection may involve a delay, typically measured in milliseconds.

6.2 Connection capacity

A node located on the Internet is equipped with some kind of connection. This connection has different characteristics such as capacity, measured as the amount of data which can be transmitted per second. This capacity is usually guaranteed by the service provider, but only from the end user to another point in the network of the service provider. The capacity obtained between two users on the Internet using different service providers is usually not guaranteed. Thus, a user cannot always rely upon the capacity advertised by the service provider, when exchanging data with another segment of the Internet. Other properties, such as delay, are associated with the same problems, as the delay between two hosts is a result of the number of hops between the hosts and the bandwidth available at each hop. A more thorough description of the problems encountered when trying to determine the available resources of a link, is found in following section.

Throughout the rest of this thesis, we refer to the capacity of an Internet connection as either capacity or bandwidth.

6.3 Connection stability

As mentioned in Section 1 streaming of video is a bandwidth critical subject. Thus, it is crucial that connection stability is maintained. The fluctuations of the Internet concerning receival of packages out of order or the occasional dropping of packets during transport will be examined in Section 8.3. What remains concerning connection stability is to consider the impact of exchanging time critical data when connection bandwidth is not guaranteed. In other words: How do we know that a connecting client can receive data, possibly from several clients, at a speed high enough to satisfy the bit-rate of the video? And how do we ensure that this client can pass data on to another client, possibly in conjunction with several other clients?

Bandwidth between hosts is defined as the amount of data, which can be transferred between two hosts. Observing a set-up with two hosts logically connected to each other, as shown in Figure 5, indicates that there are three factors which contribute to the determination of the bandwidth between the clients. These factors are client A's ability to send data, client B's capability to receive data and finally the amount of data which can be transferred through the connection. As seen in Figure 5 the physical connections between two nodes form a logical end-to-end connection which determines the available bandwidth between the two clients.

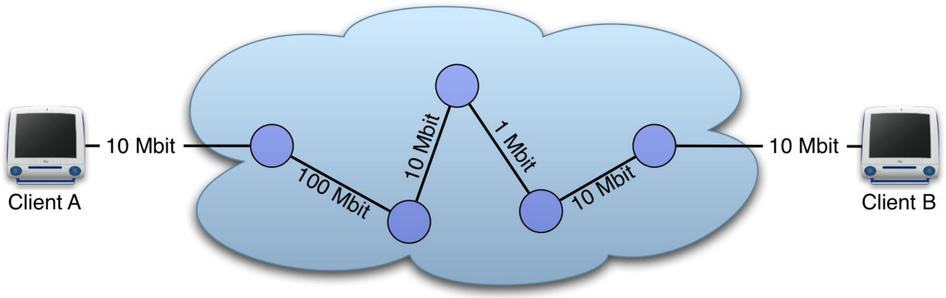


Figure 5: Bandwidth between two hosts.

As the network consists of a large number of interconnected nodes any of these can potentially break down, get congested or in other ways prevent data from passing through. This unfortunate property makes it hard to predict which route a packet will follow, how long time it will take for a packet to reach its destination and hence the calculated bandwidth available between two clients.

The large number of end-users and relatively small number of network providers have shaped the Internet with high bandwidth network connections in the core of the network and thinner connections to the end user. This means that the connection to the end-user (last mile access) is often the bottleneck in the network. This observation tells us that under normal conditions it is safe to use the end user's advertised bandwidth as a forecast to how much data the client may transmit. This observation is of course only valid, if the other end of the connection is placed in the 'thick part of the Internet' like a server at an Internet Provider. If the connection, is placed at another end-user, then the capability of that connection has to be taken into account too.

6.3.1 Guaranteeing connection stability

The Internet's *quality of service* (QoS) is simply 'best effort'. This means that a connection between two hosts tries to send as much as possible as fast as possible. All traffic is treated identically with no regard as to where and from whom the traffic originates and to whom it is destined.

Guaranteeing the bandwidth between two arbitrary nodes located upon the Internet is simply not possible. Hence, an Internet Service Provider cannot sell a specific bandwidth and guarantee that it will be available to an end-to-end connection if one part of the connection lies outside the network of the service provider. This has been one of the Internet's main issues during its lifetime and is apparently hard to solve.

The solution to the problem must account for a range of issues:

- Users need to be guaranteed that the advertised bandwidth is always available.
- Users need to be guaranteed that *latency* times will never rise above a certain point.
- Users need to be guaranteed that packet loss will not occur.
- Service providers need to be able to differentiate traffic.

In a sense the last item must be a consequence of the three first, as the last item will give the means to solve the first three items. Thus, differentiating traffic is the key-point of the solution.

A variety of solutions has been proposed. Some of these have never been used in practice and others are only of limited use. We identify a subset of these to illustrate the variety of solutions that has been presented over time:

Label switching/MPLS: MPLS is not traditionally used to ensure QoS but can be used to enforce the route of packets sent between two hosts. Thus MPLS can be used to enforce the route, but not ensure that packets do not get lost during transport.

Integrated services: Integrated services is a fine grained QoS system aiming at securing QoS at application level. Integrated services uses the Resource ReSerVation Protocol (RSVP) which must be implemented in all nodes along the route between two hosts. All nodes must send a message to its neighbours at a given time interval identifying whether the node is capable of sending QoS data. Thus, the routers in the network will be able to identify the existence of a path between two hosts, which supports integrated services.

Differentiated Services: Differentiated services is a coarse grained QoS system designed to partition the Internet into differentiated service regions grouping differentiated services domains together. Hence, an ISP could implement differentiated services in its own domain and have a Service Level Agreement (SLA) with a number of other ISPs which also implement differentiated services. These ISPs will together form a region which by the use of SLAs can guarantee differentiated quality of service the domains in between

The domains are in turn implemented using edge nodes which classify all incoming and outgoing traffic according to the SLAs. Traffic will be directed through a traffic conditioner which meters and shapes the traffic.

Common to all QoS systems is the need for implementing the systems across the boundaries of Internet service providers. This restriction has the consequence, that all measures taken by the protocol to ensure QoS are useless unless provided by all Internet service providers along the route between two clients. Therefore no further measures will be taken by the protocol to ensure QoS.

6.4 Routing

A packet sent from one host to another normally has to pass through several nodes in order to reach its destination. As the Internet is made up of a large amount of nodes it is evident that more than one path may exist which can be used to exchange data between the two hosts. Finding the right path to pass data between the two hosts is known as routing.

Many aspects can be taken into consideration when choosing the route a specific packet should follow: economy, security and geography are all aspects which can be taken into consideration. But even though a route has been selected it can be altered quickly if a node detects a problem, in which case the end-to-end path from one host to another can change from one moment to another. Routing is performed by the lower network layers as stated in Section 8.

When a packet is sent, it usually reaches its end-point, but in some cases it may be lost underway. If more packets are sent to a node than it can handle, the node starts to drop the packets which it is unable to send. If more packets are sent through a network than its capacity allows, the networks starts to congest. Congestion happens as the network only has a certain capacity. Once this capacity is exhausted the only option is to drop the packets. In this situation, the only solution is to send less data through the network as the situation will only worsen until one or more senders stop sending data for a given period of time.

6.5 The Internet of 2006

Today the Internet has reached a level where a typical end-user seldom observes any breakdowns. Often a breakdown will result in the data being routed another way in the network and therefore the user will not notice anything. The end-user is typically placed in the edge of the network and hence on the lines with the lowest bandwidth. The capacity is much higher near the core of service provider and their network providers, called *backbone providers*. This structure means that the bottleneck from the end-users perspective is 'the last access mile' (the users' 'own' connections) both regarding bandwidth and breakdown. If this part of the network is down, the end-user is cut off from the entire network whereas a breakdown in the more 'central' part of the network often is solved temporarily by routing data through another network connection.

Research [49] indicates that end-to-end connections along an Internet path often remains stable for a period of time, where period is defined as several minutes. But this period can be expected to be several hours or even days. This property ensures that the expected bandwidth of a connection does not vary much over time rendering it possible to test the connection and discover the bandwidth available.

Commonly available technologies for end-users are DSL, Cable and "Community Shared Network" (CSN).[3] These provide a different range of bandwidth as shown in Table 2.

Technology type	Downstream	Upstream
ADSL	256 Kbit/s - 8 Mbit/s	128 Kbit/s - 1 Mbit/s
ADSLv2	256 Kbit/s - 24 Mbit/s	128 Kbit/s - 1.5 Mbit/s
Cable	128 Kbit/s - 4 Mbit/s	64 Kbit/s - 1 Mbit/s
"CSN"	10 Mbit/s - 100 Mbit/s	10 Mbit/s - 100 Mbit/s

Table 2: Typical Internet connections and the associated bandwidth.

As the table indicates, the variation of end-user connection is wide, but a typical connection is ADSL with 2048/512 Kbit/s. As the example shows, the bandwidth is often asynchronous as is the case with ADSL. There are several reasons for this, but one of them is that the typical use of the Internet involves far more download than upload and therefore most Internet Service Providers (ISP) offer a larger range of products which offer a higher downstream than upstream.

6.6 Network byte-order

One of the special properties of the Internet is the way bytes are ordered from host-to-host. Consider a 16 bit integer, which is made up from two bytes. Interpreting the 16 bit number

[3] In some countries it has become increasingly popular to form communities consisting of several end-users, which together have the resources to buy Internet connections with larger bandwidth, which then can be shared among the users.

can either be done by viewing the value starting from the high-order address, known as *big endian* or starting from the low order address, known as *little endian*. This is illustrated in Figure 6. The terms LSB and MSB is an abbreviation of Least Significant Bit and Most Significant Bit, respectively.

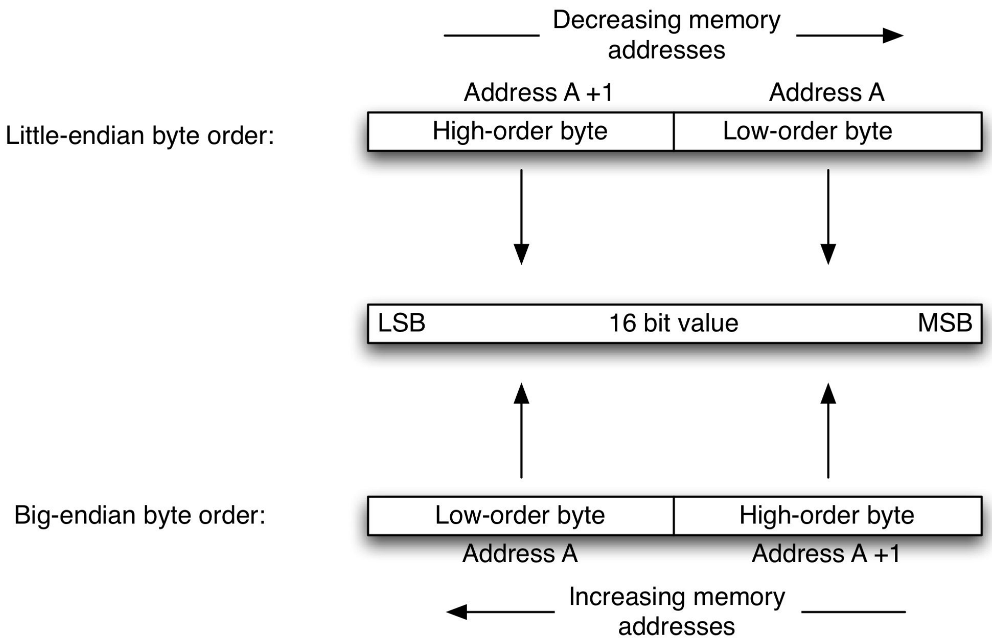


Figure 6: Byte ordering.

This difference in interpreting numbers goes a long way back, and has been a difference in architecture. This becomes a problem when data is transmitted over the Internet between different architectures. Thus, an architecture interpreting numbers using big endian will not have the same understanding of a number as an architecture using little endian. To remedy this, the Internet protocols use big endian format as *network byte-order*. This ensures, that all multi-byte values must be ordered as big endian. This is done using functionality converting multi-byte values to and from network byte-order. Using these functions, the ordering of data is irrelevant, as long as data is converted to network byte-order upon dispatch and to *host byte-order* upon receival.

7 Logical network topology

This section tries to summarize the different approaches which can be applied when designing a network distributing multimedia content. The physical topology defined by the underlying network layer is predetermined by the structure of the Internet as given in Section 6 and is therefore not an issue to examine. On top of the network layer a multimedia network defining its own logical topology can be placed. Before a topology is decided upon a walk through the different approaches is therefore in place.

The topologies outlined in this section are not defined by their traditional sense but rather different approaches used when designing a logical network extending the capabilities of the Internet. These topologies are the focal point of this section. We describe the different topological models, and in what way our design can benefit from them. We describe the relationship between the server and clients in a network and their roles.

7.1 Topological models

Building a network consisting of one or possibly more servers connected to a number of clients involves selecting a reference model depicting the network topology best suited for the purpose. As stated in Section 3.1 two different approaches can be taken when streaming multimedia content to a number of clients. Thus, the server can either send data to every client or to a number less than the total number of clients, referred to as bandwidth sharing as defined in Section 3.1. Separating the two approaches can be difficult as the cut between these two approaches is not necessarily well-defined. Therefore this section describes these approaches and a subset of the variations that have been proposed in the literature. As this literature has become quite comprehensive within recent years the models described in this section are a selection which is of relevance and interest to our design thus defining a theoretical basis upon which the final design of the network topology will rest.

7.1.1 Unicast

Unicasting data to clients is traditionally defined as a simple physical network topology where all connected clients are attached to a single common point. This point is usually a switch or a router, possibly exchanging data with another segment of a larger network. This approach builds a network consisting of one or more interconnected stars. Data exchanged between two clients will pass through the central point, enabling the network to centralize control with the clients.

Applying this topology to a logical network structure suitable for streaming multimedia content could yield a simple set-up where all clients connect to a central server which will distribute content separately to all clients as seen in Figure 1.

The solution is simple as it requires a one-to-one connection between the server and each of the clients. If one would distribute a multimedia stream which requires the clients to have a bandwidth of 4 Mbit/s, the bandwidth requirements for the server would be the number of connected clients multiplied by the required bandwidth. Deriving from this, the bandwidth requirements of the server scale linearly as more clients are connecting.

As described this topology can consist of one or possibly more interconnected stars. This can easily be projected to a network for streaming multimedia content leading to a scenario using multiple servers as described in Section 3.

The advantage of using a star topology is that user interaction capabilities can be implemented easily. When the user interacts with the video, e.g. pauses, the server can be directly notified, and no other clients will be influenced. Opposed to this, the disadvantage is poor scalability due to the high load injected upon the bandwidth capacity of the server.

7.1.2 The proxy approach

Instead of the naive solution described above, other approaches has been developed aiming to stream video content to a larger number of clients. The approach partitions the network into smaller parts, each with a server distributing content to a smaller number of clients. These proxies will each act as server to the clients but as clients to the main server. This way the requirements of the bandwidth of the main server is lowered and the proxies can be set up in strategical places in the network as visualized in Figure 7.

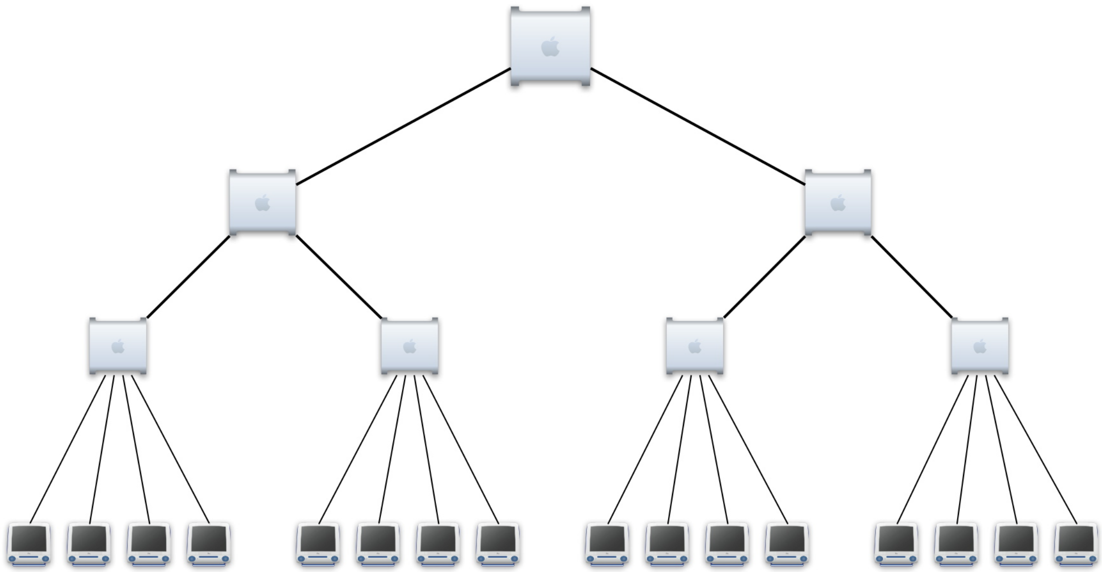


Figure 7: The proxy approach.

Another variant has been proposed in [5], using a hierarchical proxy system aiming to increase the scalability. In this system the main server distributes content to one level of proxies which in turn distributes content to the next level of proxies and so forth. This solution resolves the problem of scaling the system as more levels of proxies can be added as the system grows.

Using proxies has the advantage that the original functionalities of the old VCR player are maintained as every client has a separate connection to the server which distributes the content. Thus fast start-up, pausing and fast forward/rewind is easily implementable.

Unfortunately, using proxies involves greater costs as more clients are connecting. Scaling the network will increase the number of partitions as each proxy is only capable of servicing a certain number of clients. Thus, the scalability of a system of proxies is better than the simple approach but the demand for bandwidth is only distributed to a small number of proxies. The main server's bandwidth is thus lowered which is the goal of this thesis but the sum of bandwidth consumed by the set of servers will still rise linearly.

7.1.3 Multicasting

The multicast approach aims at distributing content to a number of clients at the same time as proposed in [24]. The main server streams content to a router which duplicates packets and forwards them to a number of clients. As a result, clients are collected in batches sharing the same stream of data. Clients are collected until a time-out has been reached and the stream is dispatched from the server as visualized in Figure 8. In its simplest form, this method resembles live streaming and is also referred to as "near video on demand" [40], as described in Section 1.

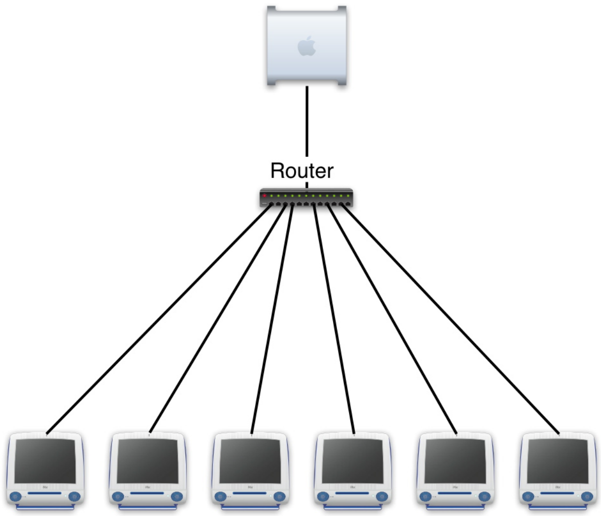


Figure 8: Multicasting.

7.1.4 Patching

Patching is originally a multicasting technique proposed in [11] where the server tries to pack as many clients in one multicast stream as possible. New clients connecting will receive data from the main stream together with data from a "patch" stream. Data received from the main stream will point to some point in the video while data received on the "patching" stream will point to the beginning of the movie. Data from the main stream will be buffered while data from the "patching" stream will be displayed immediately. When the patching stream reaches the point where the main stream started, the contents of the buffer will be displayed. At this point the new client receives data from the multicast stream together with the rest of the clients and the bandwidth requirements for the server are lowered.

Patching has been further developed [11] to a model, called P2Cast, which is based upon bandwidth sharing instead of multicasting. P2Cast utilizes a peer-to-peer model to distribute video content enabling clients to receive content from other clients in conjunction with patching. Thus, a newly connected client will receive a stream of data from one client, and a patch from another. When the patch stream reaches the starting point of the main stream the client switches to the main stream and the patch stream is closed.

Patching is an interesting schema for distributing video content. The model defined by P2Cast is both scalable, and realizes the goal of distributing the total bandwidth of the system away from the main server. The idea is based upon assumptions of the clients being able to receive two concurrent streams of video (the main stream and the patch stream) together with a possible outgoing stream to the next client. These assumptions put high demands on the available bandwidth of the clients. Thus, the problem of lowering the bandwidth costs at the server has been moved to the clients. As this is the main goal of this thesis, P2Cast serves as a good source of inspiration. Unfortunately, P2Cast only implements instant start-up and no pausing or skipping.

7.1.5 Bandwidth sharing

Bandwidth sharing uses methodologies from peer-to-peer systems which rely on the resources of the peers participating in the system. These resources are shared among the peers to the benefit of everyone. In its most pure form, a peer-to-peer network exists without any form of central server. Information vital to the survival of the network, such as login information, is shared among the peers to ensure that no element of the network represents a single point of failure.

Extending these ideas to the area of multimedia content networks could lead to a scenario where the clients receive a stream delivered from another client currently watching the same video instead of receiving the video stream directly from a main server. As a result the overall bandwidth used in the network is distributed among all clients taking the load away from the main server.

As stated in Section 7.1.4, bandwidth sharing has already been adopted in theory. Furthermore, systems resembling pure peer-to-peer systems have also been proposed as in [13] with the construction of PROMISE. PROMISE relies completely upon the resources of the peers in the network operating without a central server. Thus, there is no way to control what data is distributed among the client raising problems concerning copyright to the material.

These ideas have already been used in practice. Thus a system called GridMedia [48] has been developed for live streaming of television. The system builds a network structured as a mesh where all clients have the ability to receive data from each other. On top of this a number of satellite clients is chosen to be connected directly to the server. To show the functionality of the system it has been set up in China where the system reached a peak of 15.239 simultaneous clients using only one central server. The setup had 200 satellite clients receiving data directly from the server. This shows that the potential of streaming multimedia using bandwidth sharing technologies is indeed high.

7.2 Client-server relationship

Designing a network consisting of one or possibly more servers and a number of clients involves defining a precise relationship between client and server entities. This relationship can take two different approaches but is usually a hybrid between the two.

Decentralized control: Decentralized control between server and client is a loose-coupled relationship, in which the server interacts as little as possible. New connecting clients can find the access point in the network by themselves without asking the server. Searching for other clients containing the multimedia content is done by the clients themselves without interaction from the server. Retrieving relevant parts of the video is done by the clients requesting each other for the multimedia content. Thus, the server will only have the job of keeping the multimedia data in its whole. This scenario can be taken even further using an off-coupled approach, using no server as employed in pure peer-to-peer networks. Hence, the server's role of containing the complete video is distributed among the clients making sure that no data is lost when clients leave the network.

Centralized control: Using centralized control, the server is tight-coupled to the clients. The server is in control of where clients are located topologically in the network, and retains information about the point in the video where all clients are viewing. Requesting pieces of the video passes through the server, ensuring that the server maintains information about the network. In the most extreme cases of centralized control, all data passes through the server before reaching its destination. Of course, this cannot be applied to multimedia streaming as the bandwidth requirements for the server would be too high.

The two extremes can be referred to as router-less networking, and centralized routing networks. None of them is applicable in the extreme form, but they can be combined to a routing scheme suitable for streaming networks using bandwidth sharing.

7.3 Sources of inspiration

As already stated, a pure peer-to-peer network does not have a notion of clients or servers. Most incarnations of peer-to-peer networks use some form of hybrid between traditional client-server networks and peer-to-peer networks. This is, for instance, the case with the Napster [28] file sharing system where searching is done by a server while the data transfer is done between clients.

Other networks, such as Gnutella [10] use a more pure model, where no central server exists, but peers can be selected as ”servents” to a specific set of peers as defined in [9]. This set of peers will query the servent for search results which in turn will distribute its information to other peers selected as servents to another groups of peers.

The obvious difference between the two types of systems is the possibility of centralizing control and information. In conjunction with this, identification of peers can be difficult when using a pure peer-to-peer model, as a user's unique information is at risk of disappearing when the user disconnects. On the other hand, this can be an advantage in some environments. The big advantage of the pure peer-to-peer network is the degree of scalability. If the design is robust enough, the scalability is theoretically infinite.

Some peer-to-peer networks use a bit-for-bit model in which users have to contribute a certain amount to the community, before utilizing the resources of the community. Due to the problems incurred by using asynchronous client bandwidth this is not a viable solution, which is also argued in [17].

8 Network protocol design

Before venturing into the principles of network protocol design, we give an unambiguous definition of the subject:

A network protocol is a standardized method for exchanging data between two or more nodes located upon a shared network.

A network protocol is formally described by a document which specifies this standard and tries to uncover the ambiguities arising when the network protocol is attempted to be implemented. In practice, a protocol is a piece of software which has the ability to exchange data with other entities upon a network and possibly pass this data to some other piece of software which employs the functionalities of the protocol. Designing a protocol is therefore a matter which deserves great attention as the strengths and weaknesses are uncovered in the design phase.

Protocols carry packets, which are constructed by a header containing various information (fields) vital to the functionality of the protocol. These fields could be the sender and receiver of the packet alongside with other control fields but vary from protocol to protocol. Some packets have a *payload* containing user data exchanged between some applications relying upon the functionality of the protocol.

8.1 OSI model

To build a network system is a complex task, and is therefore often broken up into layers each with their own well-defined interface to create a modular approach. These layers together define a *protocol stack* in which data is exchanged between the layers through the interfaces. The layers serve as abstractions to the logical entities encountered in the system, hiding the complexity from the programmer. The interfaces between the layers enable designers of the protocol stack to easily change vital parts of the system without affecting the others.

Using this approach, each layer in the protocol stack becomes a separate protocol with its own specifications and standards. The payload of a packet thus becomes user data employed by the layer one level higher up in the protocol stack.

The construction of a protocol stack has been standardized by the Open Systems Interconnection (OSI) model. This model breaks the network down into a set of logical entities to construct a well-defined protocol stack. This model is depicted in Figure 9.

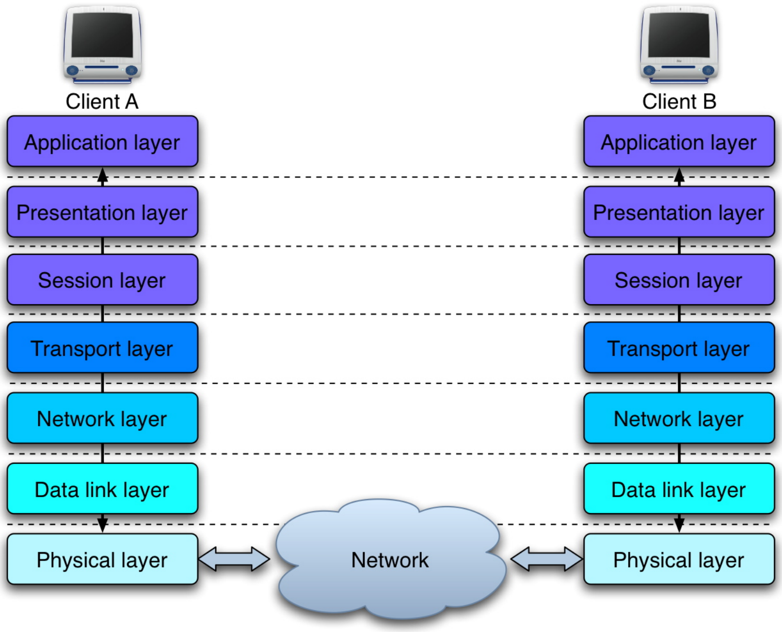


Figure 9: Open Systems Interconnection (OSI) model.

The model defines a common way for computers to communicate through a media link. It partitions the network in seven layers as illustrated in Figure 9.

Physical layer: This layer abstracts the network hardware dealing with electrical signals. The layer transmits bits, the most basic entity manipulated by the network. Electrical signals are converted to bits.

Data link layer: The data link layer collects bits to a stream of data, called frames. The frames are entities of the media type, e.g. Ethernet, ATM etc. This layer exchanges frames between directly connected nodes.

Network layer: The network layer transmits packets between two hosts with possibly many nodes between them. Packets are collected from a number of frames received from the data link layer.

Transport layer: This layer provides a messaging service for the session layer and hides the underlying network from the upper layers. It may need to be very complex in order to deal with a variety of network characteristics and capabilities.

The transport layer can provide a *connection-oriented* or *connectionless* service. In a connection-oriented session, a circuit is established through which packets flow to the destination. In this arrangement, packets arrive in order and do not require a full address or other information because the circuit guarantees their delivery to the proper destination. A connectionless session does not establish circuits or provide reliable data delivery. Packets are fully addressed and sent out over the network. The transport layer protocols at the destination can re-order the packets which arrive out of order and request retransmission of missing or defective packets.

Session layer: This layer manages communication sessions between two processes, namely session creation and termination.

Presentation layer: The layer provides data translation/conversion enabling end-systems from heterogenous environments to exchange information. The aim is to ensure that the messages exchanged between two *application processes* have a common meaning – known as shared semantics – to both processes. The presentation layer is also concerned with data encryption and data security.

Application layer The purpose of this layer is to serve as a window between correspondent application processes so that they may exchange information on the open environment. The programs which use the application layer are known as application processes.

8.2 TCP/IP protocol stack

The OSI model only describes that interfaces should be presented between software implementing one or more layers of the protocol stack. How the software implements the layers internally is not a part of the model. Thus, the OSI model presents a well-defined theoretical approach when implementing a protocol stack. Unfortunately this approach has never been widely adopted, as another standard, the *TCP/IP* model has become the *de-facto* standard upon the Internet. As this standard has come to define the Internet itself, we have no choice but to base our solution upon this as our protocol will be created for the Internet. Therefore, the TCP/IP model deserves a closer look, which will be given in this section and the aspects related to the design of the system at hand will be described.

The basis for the standard is depicted in Figure 10 as it is described in [33], [34] and [32].

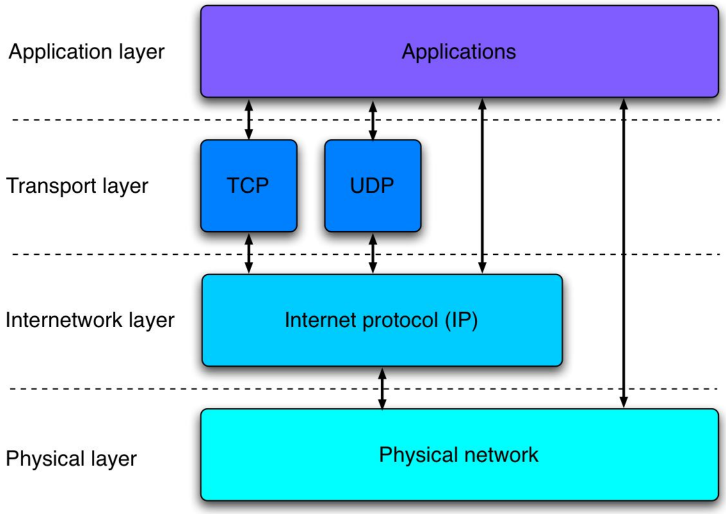


Figure 10: TCP/IP protocol stack.

8.2.1 Physical layer

This layer unifies the lower two layers of the OSI model to only one. Thus the physical properties of the network have been abstracted from. This layer is normally implemented in hardware and is thus hidden by the Networking Interface Card (NIC). Thus, the type of physical network employed between the nodes is hidden by the NIC and can vary from high speed fiber optical interface to a standard Point-to-Point Protocol (PPP) modem connection. The layer operates on these direct physical links connecting the nodes of the network and transports frames.

8.2.2 Internetwork layer

This layer connects the different physical networks constituting the Internet into a single logical internetwork. Thus, the layer abstracts from the physical links between two nodes, regarding these as logical connections between two hosts on a route with one or possibly more hops between them.

At the heart of the internetwork layer sits the Internet Protocol (IP) binding all the hosts together abstracting from the node-to-node connections. This is done using advanced routing algorithms employed in this layer at the routers of the Internet providing the host-to-host packet delivery.

IP binds the different hosts on the Internet together using an unreliable delivery service denoted as *best-effort* meaning that the network does not provide any functionality in regard to lost or corrupt data. Thus IP does not guarantee that data sent from host A to host B is identical upon reaching B. This has the effect that a connection between two hosts cannot be expected to be stabile. Finally, packets are prone to be delivered out of order, as the route between two hosts cannot be expected to be the same all the time. All in all, the internetwork layer reflects the properties of the Internet as discussed in Section 6.

To identify hosts on the Internet, IP uses an *IP-address* to distinguish them.

8.2.3 Transport layer

The transport layer enables applications located upon the different hosts to exchange data. Thus, this layer abstracts from the host-to-host connection to a process-to-process connection.

The transport layer consists of two main protocols, Transport Control Protocol (TCP) and User Datagram Protocol (UDP) which represent two different ways of exchanging data between applications. UDP is a mere extension of the Internetwork layer providing only connectionless best-effort service for the application layer. Thus, UDP provides host-to-host delivery with all the disadvantages that the Internet possesses. TCP is designed as a *reliable byte stream* protocol aiming to abstract the properties of the Internet from the application programmer delivering a reliable, connection oriented byte-stream.

The TCP and UDP protocols use a *port* to identify the process transporting data through the network. Thus, a process located upon a host will be identified by the pair (IP-address, port). The protocol stack demultiplexes incoming data observing the destination port in the header, and delivers the data to the relevant process.

8.2.4 Application layer

The top layer is where the applications are located. Applications located upon different hosts will exchange data through the transport layer using a protocol defined by the applications. Therefore this layer constitutes networked applications created by application programmers.

How the interface between the transport layer and the application layer is implemented varies from one Operating System (OS) to another. Data is typically exchanged using message queues which are filled by the transport layer and emptied by the application layer.

Thus an example with two applications exchanging data is depicted in Figure 11.

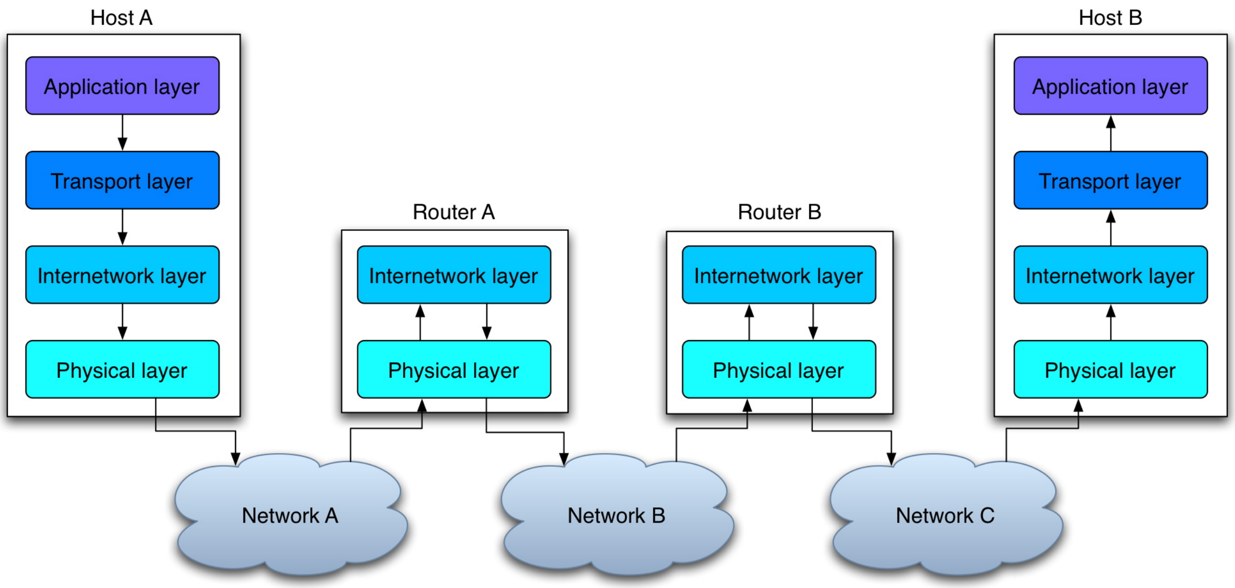


Figure 11: Two applications exchanging data using TCP/IP. The two hosts are located at separate subnetworks with routers routing data between the subnetworks.

8.3 Connection-oriented versus connectionless transport

The type of transport chosen for a given task depends on the task at hand, as the properties of the two transport types are so distinct. Thus, a thorough introduction to these two different ways of transporting data is in place.

8.3.1 User Datagram Protocol (UDP)

As UDP is a mere extension of the host-to-host delivery service of IP, it contains all the disadvantages of the Internetwork layer. UDP is connectionless meaning that data is not sent through a logical connection but rather just sent at request by the application programmer. If no application is listening at the other end of the communication channel, the sender will not be notified.

As one of the disadvantages of the Internet is that data transport is unreliable, UDP due to its simple extension of IP suffers from this[4]. This fact results in a number of drawbacks:

Data corruption: As the communication lines of today do not ensure that data is unchanged at reception, UDP induces the possibility that data can become corrupt during transport. UDP does include a checksum field in the header but using this is not mandatory.

[4]In practice UDP has proven to be more reliable than commonly thought, as the Internet has become more and more reliable during the years.

Therefore the application programmer cannot rely upon the ability of the protocol stack to correct data.

Data displacement: One of the properties of the Internet is its ability to dynamically reroute data when a link fails. As a result, data can be received *out-of-order* meaning that the order of which a series of packets is sent is not guaranteed to be the same in which the series of packets is received. Furthermore, data is not equipped with a sequence number leaving the application programmer to take measures when combining the received packets to the original series of packets.

Data loss: As the routers upon a route between two nodes on the Internet sometimes drop a packet due to congestion etc. UDP induces the risk of data loss. Again, the application programmer must take measures to verify that data is received if this is relevant.

Data duplication: During transport, a node located upon the route between two hosts may erroneously duplicate a packet. Thus, the receiver will encounter two identical packets. UDP does not detect this, and applications using UDP must handle this accordingly.

Taking all these drawbacks into account, a simple question can be raised: Why use UDP? The answer is simply that all these drawbacks become advantages when reliability is not an issue. Furthermore, when exchanging time critical data, speed can be more important than data reliability. As UDP does not implement any logic other than the simple demultiplexing between application processes, it possesses the great advantage of high speed. This is due to the absence of a time consuming connection setup, and the total absence of data reliability and most important that no acknowledgment packets are used.

As streaming of multimedia is a time critical subject, UDP has traditionally been used in this context. In addition, streaming is not subject to high sensitivity as to whether data has become corrupt during data transport. Furthermore, it is important that data is received at some point in time, but not too late to be played. Thus, a streaming protocol must be able to undertake retransmissions when possible but avoid late retransmissions.

In addition, implementations of UDP generally have the advantage that when the application process initiates a data transfer this is done instantly. Thus the application programmer can rely on the fact that data is not queued in the protocol stack before being sent. UDP uses a maximum size of data transmission called MTU of 65 Kbyte. From this must be subtracted the size of the IP header and the size of the UDP header.

8.3.2 Transport Control Protocol (TCP)

TCP represents another approach when transmitting application data on the Internet. Data transport using TCP offers a connection oriented reliable byte stream. In other words, TCP offers a logical extension of the internet protocol.

Where the nodes of the physical layer are directly connected, TCP uses a logical connection between the hosts of the Internet. To enable this, an algorithm called *three-way handshake* is used to establish a shared state among two application processes during connection establishment or termination.

TCP uses a sliding window approach to realize reliable transmission. This type of protocol implements a buffer between the application process and the underlying network which absorbs the irregularities encountered on the Internet. This buffer is used as a "window" sliding along the content of the data stream on both the sending and the receiving side. Data inside the window of the sender is either sent but not yet acknowledged by the receiver or not sent yet (and therefore not yet acknowledged). As data is being acknowledged by the receiver the window slides along the data stream filling up with data which remains to be sent. The window on the receiving side is filled with data which remains to be received, but may possibly be sent. As data is received, acknowledgments are returned to the sender and the window slides along to include new data which can then be transmitted by the sender. If data is lost during transfer the receiver uses timers to ensure that the lost data will be retransmitted. When a time-out occurs, the sender will indicate that the packet was not acknowledged. This time-out will be raised by an adaptive trigger ensuring that the retransmission will occur. For this to work, sender and receiver have to agree on the size of the window. Furthermore they must both know the location of the window in the data stream. In addition the window size can change to reflect the fluctuations of the Internet. An optimal sliding window protocol will keep the connection between the two hosts saturated without overflowing the receiver and avoiding congestion on any links between the two hosts. Furthermore TCP uses credit mechanisms incorporated into the sliding window which modify the sizes of the window of the sender and receiver to account for the fluctuating amount of data which is regularly sent through a TCP connection. These properties induce a large overhead, both in terms of data transmissions, but also in terms of computing power.

Contrary to UDP, TCP does not implement an MTU which has impact on the application process. Whereas an application using UDP cannot send portions of data larger than the MTU, TCP itself fragments data into packets of sufficiently small segments. The receiving host will collect the segments to the originally transmitted data and return it to the receiving application process. As a result TCP may buffer enough bytes from the sending application process to fill a segment and send this to the receiving process on the destination host. In other words, data is split into segments rendering the sending application process incapable of relying upon TCP to send data immediately as an internal parameter called maximum segment size (MSS) is used to decide when data is to be sent. When enough data is collected to fill a segment TCP will send data. In addition some form of time-out is implemented to ensure that data will eventually be sent even though a full segment cannot be filled. The size of MSS and this timer is usually calculated from factors such as the the properties of the underlying network and is therefore unknown to the application process.

Furthermore, TCP uses back-off as part if its congestion control mechanism. Thus, if TCP detects that the connection is congested, it will incorporate a delay before sending any other packets.

Concluding upon these two types of transport, it becomes evident that TCP is unsuitable for transporting multimedia data for streaming. This is furthermore justified in the literature as performance analytic studies have shown "that TCP generally provides good performance when the achievable TCP throughput is roughly twice the media bitrate" [43].

8.4 Network Address Translation (NAT)

Due to the format and use of of the IP-address, the maximum number of hosts located on the Internet is restricted[5]. This has lead to an address shortage resulting in the need for technologies which enable several hosts to be connected to the Internet without having to take up an address each. A newer version of IP, known as IPv6 overcomes this problem, but has yet to gain acceptance.

Instead a common solution to this problem is Network Address Translation (NAT) sometimes also called IP masquerading. This technique masquerades a number of hosts behind a NAT-router directly connected to the Internet. This router has an external interface equipped with a global public IP address, and an internal interface equipped with a local private IP address. When a masqueraded host communicates with a host on the Internet, data will go through the NAT-router which will know that the two hosts are connected. Thus data coming from the masqueraded hosts will be forwarded to the end hosts, while data from the end hosts will be forwarded to the masqueraded host. This raises the natural question of how two hosts in the masqueraded network can be connected to the same host. This is done by the NAT router operating upon the transport layer de-multiplexing the packets using the port of the transport protocol.

Using NAT has the disadvantage that hosts located upon the Internet cannot directly open a connection to hosts masqueraded by a NAT router. Thus, a client initiating a stream of data to another client may erroneously conclude that the receiving host is malfunctioning due to the IP masquerading of a router. Most NAT routers provide functionality to permanently forward data incoming on a specific transport port to a given host masqueraded by the router. This can be tedious, and many Internet users are unaware of these capabilities.

8.5 Real-time Transport Protocol (RTP)

Specialized protocols for data streaming do already exists. One of the most well-known is Real-time Transport Protocol (RTP) which was proposed as a standard back in 1995. The protocols' main purpose is to transport real-time data like video and audio.

It is divided into two parts a data and a control part. The data part of the protocol is very thin and provides the possibility to transport continuous data.

The protocol is original targeted to be used with TCP and UDP but efforts have been made to make the protocol independent of the transport protocol. Although the protocol is normally implemented into an application the protocol normally goes under the name as a transport

[5]As the IP address field is 32 bit, the maximum number of hosts is $2^{32} = 4,294,967,296$. However, this number is reduced due to special purposes such as private networks, multicast addresses and early over-allocation of addresses to some institutions as the number of addresses was thought to be 'infinite'.

protocol. It does not support any mechanisms for re-sending or error correction of data.

The main reason not to investigate this protocol any further is the need for streaming data from multiple sources and the use of RTP would not contribute in any way to solve this issue.

9 Protocol implementation

Protocols are often implemented as part of a protocol stack as described in Section 8. Communication between these individual protocols is carried out using interfaces, defining how the protocols interact with each other. These interfaces and how the protocols interact will be the focal point of this section.

9.1 Network layers and interfaces

As described in Section 8.1, networks are implemented as a series of layers on top of each other. The main reason behind this is to hide the complexity of the lower layer together with easy extension. The same technology can be used internally in a protocol or in any software as subdividing software into smaller pieces and defining precise interfaces among these components help keep complexity down and make software more flexible when changing specific elements.

Thus, one of the most important elements of a protocol implementation is to make a simple, unambiguous, solid, and easy-to-use interface which offers application programmers easy implementation of software using the protocol.

9.2 Multithreading

Implementing a protocol stack usually implies some form of communication between the layers. This can be done using multiprogramming where a number of *threads* in the protocol stack handles ingoing and outgoing data.

Roughly the main discussion on multithreaded programming can be divided in two main issues:

Number of threads: What number of threads should be used? Is it best to minimize or maximize the number? Or is there a more balanced approach?

Location of threads: Where should the threads live (in what area of the code/system)? and hence who should create and destroy them?

The issues will be treated in the listed order in the following sections but these issues have one thing in common: there exists plenty of theory about the topics, but none of this has lead to any real conclusion. Hence, a lot of different opinions exists in the area and therefore a large part of this text will also be based on our personal opinion and experience.

9.2.1 Number of threads

Threads used in a multithreaded application are divided into different categories depending on the job they are performing:

- thread(s) used for user interaction,
- thread(s) used for reading/writing data from/to different types of media.
- thread(s) doing background data processing and calculations.

Switching from one thread to another involves an amount of overhead. The more threads an application is using, the more overhead is generated, which could speak against using many threads. On the other hand, using a large number of threads, if used correctly, can lead to better performance, since each individual thread can perform specific tasks.

Regarding protocols, threads are often used to perform background jobs at different intervals. If the job does not need to communicate and need not be synchronized with other threads, then the job is perfectly suited for its own thread. A processor intensive task could also benefit from its own thread. Dividing these tasks among several threads is for example known in a web-server, where a thread receives incoming requests and delegates the work to one or more threads.

When communicating between components in application programming such as layers in protocol stacks, it may be an advantage to use a thread with the sole purpose of retrieving data from another layer, process it, and enqueue it to the next layer. This is especially advantageous when receiving data from a network, as the underlying network layer may discard packets if they are not read fast enough. This may happen due to overflowing of the queues of the network layer which normally have a maximum size limiting how much data it may contain.

A typical role of thumb as described in [4] is that interaction between threads should be minimized, as this involves a lot of synchronization which in turn will result in lots of time spent waiting for other threads. Furthermore, a multithreaded application needs to be analysed thoroughly to identify the critical regions of the code. A critical region is data accessed by two or more threads, possibly at the same time. These critical regions must be protected by semaphores, as an example. Otherwise unpredictable errors may occur as one thread may read the contents of a variable immediately before being switched out by another thread updating the variable. How many threads and how many critical regions arise when multi-threading an application is thus a question to be analysed.

If any conclusion can be drawn from this, it would be that the number of threads should be held as low as possible with respect to the kind of work that the software should perform.

9.2.2 Location of threads

The next question, is to solve where the threads embedded in the software should reside. Furthermore this raises the question of where the threads should be created and destroyed.

Most protocol implementations encapsulate the logic of the protocol and offer only a simple interface. This means that if a program creates an instance of the protocol, it might only make a simple call to a function when receiving data. What happens inside the encapsulated protocol is hidden from the application thread. Thus, if the logic embedded in the protocol decides to wait for a specific event to occur, it might spawn a new thread waiting for a timer to run out.

To conclude this subject, the rule of thumb must be that new threads are created and destroyed by the encapsulated protocol when needed and that the application instantiating the protocol, only creates one thread running the protocol. From here on, it is up to the protocol to start new threads.

10 Buffering of data

As explained in Section 3.1, attaining the goal of lowering the bandwidth usage of the server requires all clients to buffer a certain portion of data. This buffer serves both the purpose of enabling clients to realize the goal of providing on-demand video streaming, and absorb the fluctuations encountered upon the Internet. This section will give an introduction to basic theory on data buffering but with the above specific use in mind.

10.1 Buffering approach

As buffering of data has been used to solve many problems in computer science, the existence of a large amount of theory on the topic is evident. Fortunately, the the largest part of this theory is based on buffering data in specific systems and situations. Because of this we have sought to find literature concerning similar systems and setups but without much success. Two articles did however stand out, both looking at data buffering for redistribution in video streaming systems with similar characteristics as the system we are planning, though both of them only resemble it.

The authors of [35] concludes that a client should buffer at least 25 percent of the video in order to attain a significant bandwidth reduction. Obviously, the larger an amount of data a client buffers, the more it can transmit to other clients. This is evidently a simplification as it raises other issues, such as the need for a large amount of bandwidth, otherwise it would not be able to retransmit the data to other clients.

On the other hand the authors of [19] conclude that, in order to make a good video-on-demand system, the amount of memory should be minimized in order to lower user interaction response time, e.g. fast startup. But this statement is a simplification as well, as it implies a demand for the buffer to be full, or at least partly filled before responding to user interaction. This demand can be relieved so only a fraction of the buffer needs to be filled, before responding to user interaction. This would make sense, as user interaction would imply that all clients receiving data from the interacting client need to be relocated in the network. Thus, the first purpose of the buffer can be overlooked, immediately after a user interaction, only leaving the buffer to resolve the purpose of absorbing network fluctuations. When the buffer is filled, or partly filled, the buffer could again fulfill the purpose of ensuring on-demand streaming to other clients.

The two articles clearly show the difficulties in drawing final conclusions based on work which does not share the exact same prerequisites.

Despite many different types of buffering strategies like; 'FIFO' and 'use-it and toss-it'. these can only be used as a source of inspiration. Instead of discussing a lot of different strategies, we find it more rewarding to define the primary properties needed by the buffer. The main properties of the buffer ordered by importance should be:

1. Ensure that the client always has the needed data available.
2. Keep as much data in the buffer available to transmit to other clients.
3. Lower the size of the buffer to avoid too high memory requirements.

The first item is a direct consequence of the time critical nature of the subject at hand. The two last items seem to be inconsistent but put into practice this only means that these two properties need to be balanced to find a good solution. These properties will be kept in mind when designing the buffer, in Section 14.2.

10.2 Physical memory layout

Data buffering is usually done in physical memory using two different approaches.

Dynamic allocation: This approach allocates memory when needed and deallocates it again when the need ends. This way only a minimum amount of memory will be allocated at any given time. The downside of using dynamic allocation is the overhead spent on allocating and deallocating memory.

Static allocation: This method means that memory is allocated only once and a fixed amount of memory is available at all times.

When implementing software applications used in personal computers, dynamic allocation is often preferred as the software should run alongside a range of other programs. Therefore it is preferred if each individual application uses a minimum of resources.

Static allocation is typically an advantage when an application has to run on hardware specially designed with the application in mind. This could be a protocol build into hardware, manufactured with a limited amount of memory, and the only purpose of streaming a signal to the television. It also has the advantage that no other application can seize some of the memory and hence exhaust important resources.

10.3 Buffering of data

Utilizing a buffer, it becomes obvious that data needs to be read at a continuous rate. If this rate is unable to be satisfied, the buffer will be exhausted resulting in a buffer underflow. This is indeed important in this context, as the video player will halt until data is available resulting in bad user experience. Furthermore, all clients receiving data from a client, the buffer of which underflows, will potentially be affected in the same way. The opposite scenario, when the buffer overflows is furthermore an important scenario, when designing the system. This scenario may seem avoidable, as buffer overflow will intuitively be the result of receiving data at a rate faster than the video player consumes it. Nevertheless, the scenario must not be neglected and caution should be taken to avoid this.

In order to guard the buffer from under- or overflow it must be possible to increase or decrease the speed with which the buffer is filled. In the case of buffering data for on-demand video it is furthermore important that the parts of the movie can be received fast to enable fast startup.

11 Security

As already stated the security of the protocol must cover three different areas, namely authentication, data integrity and data theft.

11.1 Authorization

As described in Section 3.6 validating connection requests will be done by the server performing the authorization procedure. But the procedure of exchanging the information over a shared network may compromise data. Thus, a malicious user may be eavesdropping on the line and read the content of packets containing vital information such as user name and password.

To account for this problem systems does not exchange vital informations in clear text. Instead it is encoded using some encoding scheme. This could, be Public Key Infrastructure (PKI) where the client exchanges a public key with the server before sending user name and password. When the keys have been exchanged the user name and password may be encrypted using the public key of the server before being sent to the server.

In this way all information flowing between the nodes could encrypt a small amount of data which would have to be decrypted and be in accordance with some preset value. If the data differs from this value the sender of data could be regarded as malicious.

11.2 Data integrity

To ensure that data has not been modified by other clients a client needs to have a mechanism to ensure data integrity. This could easily, although not bulletproof, be done using a checksum which has to be calculated from a portion of data. If this checksum differs from a precalculated checksum, data can be regarded as invalid. But the precalculated checksum must further be received from a trusted source. As no client can trust another client, this checksum has to be precalculated and distributed by the server.

11.3 Data theft

Data theft may occur on two different levels. Thus, a malicious client may steal data by eavesdropping on a line or by extracting data from the protocol/client application. In the first case a node located at a central point may read the contents of the packets flowing in the network. The only way to cut off the client from this is to encrypt data in some way. PKI clearly cannot be used in this context as this would involve a client sending data to other clients to encrypt data using different public keys to all connected clients. This would indeed inject a substantial load upon the clients as the process of encrypting and decrypting large amounts of data is processor intensive.

The second case involves a programmer using the protocol to extract data transferred by the protocol. In this way a user might log on to the network and just read data from the protocol and save it to disk.

As a remedy to both of these cases some form of Digital Rights Management (DRM) scheme could be undertaken. DRM is used by media publishers to control access to digital media. The term DRM covers a wide field but the core is to protect software and hardware. But as data need to be delivered to a client application in a form readable to the application, the protocol must not deliver data in a format unreadable to the application. As a result, the server and client need to have an agreement of how data should be read. But as is the case when dealing with security the measures these can always be broken. For example the security of Apple's Quicktime AAC proprietary DRM format for streaming multimedia has been broken making it possible to extract the raw multimedia format from the stream.

Chapter III

Protocol design

This chapter accounts for the design of the protocol as it is illustrated in Figure 4 in Section 3.4. Thus, the design will account for the client-side but also the server-side. The chapter should be regarded as a specification for any implementation of the protocol and will therefore not contain any direct demands as to how the implementation should be made.

12 Fundamentals

This section treats the fundamental design issues. These issues define the underlying platform for the design and will therefore be treated individually in the following sections.

12.1 System control

Following the requirements in Section 2 regarding system security, controlling the network needs to be located centrally. Thus, the main server will control which clients can be authenticated and where the clients are logically located in the network. Thus, the server controls all clients at any time, and decides when and from where the client will receive their data stream. The reason for this choice is to locate all logic centrally opening up for a design allowing different implementations to co-exist independently of each other in the same network. The objective is thus to reduce the activity between the nodes to the essential, namely streaming video data.

This choice does not come without a cost: Centralizing control will inject a high load onto the server, both concerning bandwidth and processing power. If the server needs to be aware of the state of all clients at any given time the amount of control information flowing between the server and all clients will rise. Furthermore, relocating clients logically in the network can require the server to run complex selection procedures. Essentially this choice constrains scalability. Thus, there is a trade-off between centralizing control and the security gain this may result in, and decentralizing control and the possibilities of gaining higher scalability as seen in more pure peer-to-peer networks.

12.2 Data transport

Evidently it is necessary to transport two main types of data, namely video and control. Video data includes the content of the data stream, while control data is used by the server to monitor and control the network, e.g. authenticate clients, relocate clients.

As already stated, control data must be delivered to the main server, to attain the desired centralized control. Thus, control and video data cannot share the same data channel, as video data in most cases will be received from another clients. This forces the design and implementation of two different data transmission channels, one exchanging control data between server and client, and one exchanging video data between clients. These two channels will be implemented as two separate protocols due to the diverse characteristics of the two data channels. However, these two protocols will only be usable together thus in this thesis they will be referred to as only one. The two protocols will from this point on be referred to as Data Communication Protocol (DCP) and Control Communication Protocol (CCP).

It is evident that the two protocols do not share all properties. Clearly, delivery of video data is a time-critical issue, whereas control data does not share this property. Furthermore, the integrity of the CCP is of high importance, as the contents of messages must not be changed during transport. Contrarily, if several bits of a DCP message are changed, this would not have great influence upon the ability of the video player to reproduce the video. Following this, and the conclusions drawn in Section 8, the DCP will be implemented on top of UDP. This gives the protocol the speed of UDP, both concerning data transmission and connection setup/teardown. Conversely, using UDP adds to the complexity as retransmission of lost data packets must be handled by the DCP. But intuitively, data must not arrive too late to be played - hence, late retransmissions must be avoided.

Both protocols need to communicate using separate ports. The server will use a static CCP port which must be known by clients wishing to connect to the server. Which ports the clients will use are decided by the client and advertised to the server. Thus, no static ports will be used by the client protocol. Thus, the CCP port will be static on the server and dynamic on the client, while the DCP will be dynamic on both client and server sides.

12.2.1 Control communication protocol

The CCP will contrary to the DCP be implemented using TCP. Evidently, it is crucial that control data reach its destination, though not crucial as to when this happens. Of course information can reach the destination at a point where correcting an error will be too late, but using UDP would not solve this problem in any case.

It is evident that two communication flows must be established between the server and each client. Thus, information must flow both ways to obtain the required functionality. The transmitted messages will in most cases be request-respond oriented, meaning that either the server or a client will request an action from the other and will accordingly expect a respond. Every client will therefore have a control connection open to the server at all times using this when deemed necessary.

This raises the natural question of how much load the CCP will inject upon the server. The communication with all hosts on the network will require the server and client to set aside a portion of the available bandwidth to the sole purpose of control communication. Therefore it is of utmost importance that the use of CCP is lowered as much as possible. The more bandwidth consumed by the CCP, the higher expenses will be involved at the server and additionally a fewer number of clients will be using the service, as the bandwidth demands will be higher.

It becomes evident that the interval at which control information is exchanged together with the size of synchronization data is the major impact upon the bandwidth of server. Therefore these parameters must be carefully tuned.

TCP in itself often uses keep-alive packets which periodically probe the other end of the connection. This functionality can however be altered to change the interval at which this probing is done. This may offer an advantage to the server which will be notified if TCP discovers that a host is down.

12.2.2 Data communication protocol

As stated in Section 3, clients equipped with asynchronous Internet connections which have the ability to receive, but not send, the full bit-rate of the video should not be excluded from the network. This fact implies that a set of clients can share the task of sending the stream in its entirety to another client. Limiting how many clients can share this task, or rephrased as, how many different clients a client can receive data from, will be accounted for in Section 14.1.

12.3 Logical topology

The physical network topology of the system is defined by the underlying network and the routing algorithms applied in the internetwork layer. Upon this topology, a logical topology defined by the protocol must be constructed.

As already mentioned, the protocol will be able place clients so that they receive data from multiple sources. This means that small fractions of a video can come from virtually all other clients watching the same section of the video. This setup clearly leads to a topology which will take shape as a mesh and not a well-structured shape.

As is the case with routing, selecting a logical topology for the network can either be done using two approaches, namely centrally or de-centrally. Where the first model applies a central unit which decides where a client should be placed logically, the second model leaves the decision to the client itself. Based on information already present in the network the client can find a proper point in the logical network. Due to the requirements given in Section 2 concerning system security the first approach is used by the protocol.

Selecting a logical topology for the network used for on-demand streaming involves a knowledge of the characteristics of the media which is distributed. One topology might be suitable for media with little or no interaction while another topology might not. Evidently the more clients receiving data from an interacting client, the more relocations must be performed. Thus, selecting a logical topology requires knowledge about the behaviour of the users. The only universal property applying to the topology is that it will form a directed graph which in some cases will be disconnected as described in Section 14.2.

This has the consequence that the logical topology of the network is not decided on beforehand by the design. This will be done by the server whenever an event occurs in the network, e.g. when client relocation is necessary due to new clients connecting, clients leaving etc. How this selection procedure is constructed will be described in Section 14.6. However, for this selection mechanism to function the protocol may need various information when determining the point where a client can be located. Traditionally, IP routing applies values like round-trip time and/or numbers of hops between a pair of nodes. Some of these values could be used by the selection algorithm, as a long delay between two clients exchanging data might give rise to complications. Yet another parameter to be considered may be the economical cost incurred when transmitting data across a certain link might be desirable. Lastly, values determined by the functionality of the protocol can be used. These could be values like how many times a user has interacted with the video stream, e.g. with a pause or a skip or the upload capacity of the client.

Traditional graph theory uses node and edge weights. This can be transferred to the protocol in the sense that a given client, its ingoing and outgoing connections can be assigned weights. The construction and assigning of these values can be done using a scheme combining several information. Subsequently, these weights can be used by the selection mechanism.

These possibilities in constructing a complicated selection mechanism gives the protocol great flexibility. Therefore the workings of the selection mechanism is part of an actual implementation of the design, and no part of the design itself.

12.4 Video identification

A method of identifying this video must be provided for the server protocol to identify a request for a given video sent from a client. This identification can either come from the client, or the server can provide the client with a list of videos available at the server.

In the first case, the client might obtain the identification outside the protocol, e.g. from a home-page. Thus, the distributor of the video might have a home-page containing billing information and the identification of the videos. In the second case, the client might request the server of a list of the videos available. In this case, the server protocol would request the server application for this list and send it to the client, which in turn would select a given video and request it. Due to reasons of flexibility and simplicity we choose the first solution. Thus, we assume that the client may identify a video outside the protocol and request this from the server. This identification will simply be a 64 bit long number containing the identification.

12.5 Security

As mentioned in Section 2.6 a requirement to the protocol is the capability to offer different elements of security. These elements can roughly be divided into four parts all of which will be treated in the design of the protocol. A basic design solving the security issues will be made but not implemented in the protocol. Generally, the security enhancements of the design are related to issues arising from the functionality of the protocol. Thus, any digital rights management will be implemented in the applications using the protocol, as described in Figure 12.

12.5.1 Authentication

To verify the authenticity of a sender of a packet a client must be able to be authenticate the client which has sent the packet. This will ensure that no client can impersonate another. This is done using a set of keys. Two clients exchanging data share a set of keys identifying the other. These keys are generated when a client connects. The server will then generate a unique key identifying the client, and a key identifying the server to the client. Only the client knows about the server key and hence no other client is capable of sending packets to the server and pretend it is another client. When two clients need to communicate, the server will distribute their keys to let the clients know the identification of the other.

These key pairs can further be used to identify a client, in regard to IP-address, communication ports etc. Hence, when a client connects, the server will store the information and for the rest of the *session* the client will be identified by the generated keys. These keys will from this point be known as a client id. Thus, two clients may know the keys of each other which will form the key pair described in this section.

This procedure resembles the Kerberos [18] authentication mechanism designed to allow individuals communicating over an insecure network to prove their identity to one another in a secure manner. As the focal point of this initial version of the protocol is not secure authentication of clients this simplified authentication mechanism has been chosen.

12.5.2 Authorization

To authorize itself, a client will need to send a connection request to the server containing a user name and password. This request should furthermore contain identification of the desired video the client would like to watch. The information should not be validated by the protocol, but rather handed to the server application allowing this to authorize the client. Hence, it is not part of the protocol to perform this check. The client and server applications should be implemented using some form of data encryption before handing user name and password to the protocol, as the protocol only transports data in clear text.

12.5.3 Data integrity

To avoid clients tampering with data before forwarding it to another client the functionality of transporting security data is added to the protocol. This is done using CCP by sending security data from the server to all clients containing data related to some predefined security mechanism used by the server application. Thus, the CCP will also handle data related to the security mechanisms of the server application while DCP will handle data related to the video stream. Client-side, the security data will be delivered to the client application which in turn will need to verify if the security scheme defined by the server application is respected. This model is depicted in Figure 12. The reason for this choice of this model is that the security part of the protocol should not be an integrated part of the protocol. As a result, the application protocol needs a way of notifying the client protocol if the security check fails. This notification should be passed on the the server protocol which can take proper actions.

Due to the extra complexity involved in handling and sending security data from the server, only the packet implementing this functionality will be designed. No attempt will be made to implement the actual security mechanisms. Thus, the packets will be designed, but not handled.

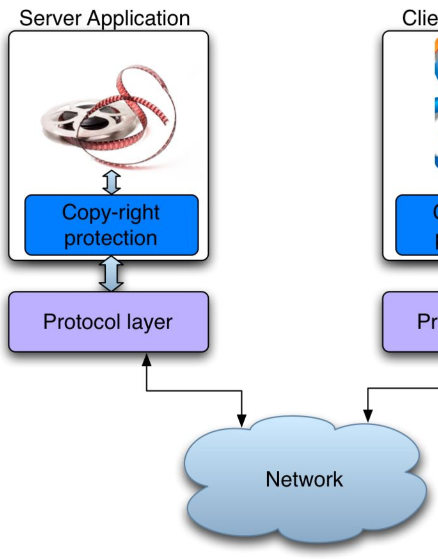


Figure 12: Copyright protection.

12.5.4 Data theft

This issue of ensuring that malicious clients cannot steal data is by far the most important and difficult one. Many solutions have been suggested but none have solved the problem of proper protection against copying of copyright protected material, which has also already been described in Section 11. The solution we have chosen is therefore to make it completely up to the implementer to choose what to do and hence what level of security the protocol implementation should offer. The protocol will facilitate that data related to security can be embedded in the packet facilitating digital rights management. Thus, this packet will contain all information related to security. The protocol design only defines one rule related to the security packet, namely that the client receives security data from the server without requesting it. Thus the server must keep track of what data the client needs and send this accordingly. How much data the server sends at a time is undefined, meaning that the server can choose to send all security data related to the video as soon as the client begins receiving the video.

12.6 Protocol architecture

To give a more precise picture of the design of the protocol a figure extending the contents of Figure 4 is given in Figure 13. The figure shows how a server operates with two clients all using the same uniform interface.

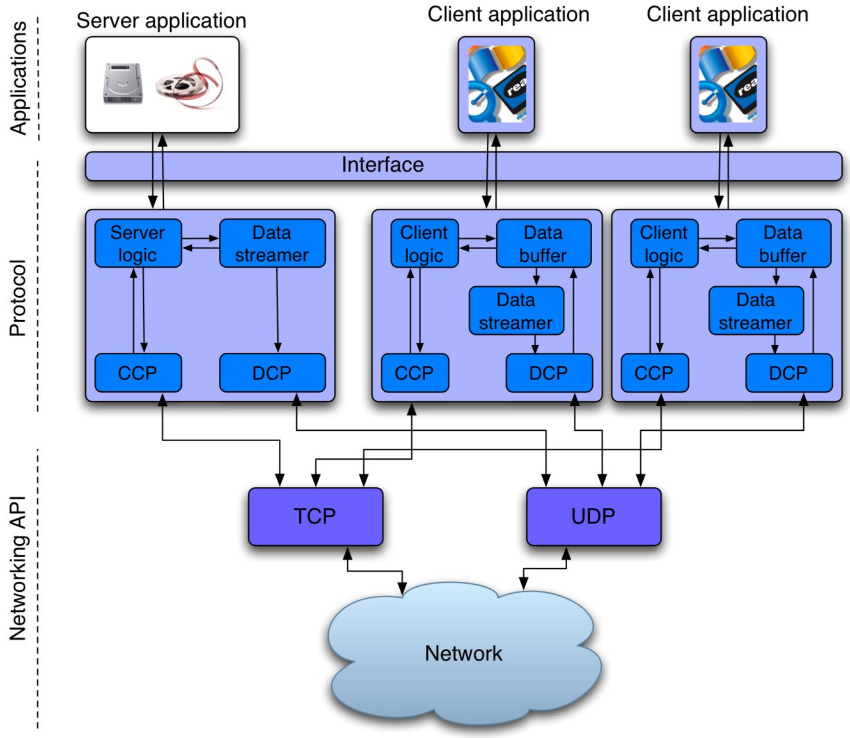


Figure 13: Detailed architecture.

13 Protocol state

In order to determine the validity of incoming packets a perception of the state of the two hosts exchanging data is necessary. An analysis of protocol state reveals that many perceptions of the state can be found:

A client's perception of itself: The client itself does not operate with any state enforced by the protocol. Hence a client is free to define its own state. One could argue that the client can be in a logged on or logged off state, but this is evidently implicit, as it does not make sense to send anything but a log on request if the client is not logged on.

A client's perception of other clients: Again a client has no real interest in which state another client is in and therefore no state is defined for this purpose either. A client should however be able to know when to send data to another client. Again a state which determines whether a client is logged on or not could be used, but communicating with another client is only possible if a unique id is known, as described in Section 12.5.1. This unique id is provided by the server, and acts as a validation of the other clients.

A client's perception of the server: As the server binds the clients together in the logical topology, a client clearly cannot regard the server as malfunctioning. If this was the case the topology would be inconsistent. Furthermore, a client must always abide by the commands of the server. As a result, a client always regards the server as online. Thus, the server does not have any states from the point of view of the clients.

The server's perception of itself: The perception of the server of its own state is only relevant to the server itself. Clearly, the server may either be on-line or not on-line. But apart from this, the server may also be streaming data to one or more clients. However, as the server also functions as a client to other clients it may be argued if this can be considered as the state of the server. As a result we define that the server only has one state which is on-line.

The server's perception of clients: As the server controls the topology, the perception of the server of all clients is clearly the most important. For example, if the server regards a client as not on-line, the client has only one option, which is to request access.

In the following we examine only the perception of the server of both the clients and its own state. As all control communication passes through the server, the most important aspect of the protocol is exactly the perception of the server of the clients and itself.

13.1 Server state

As stated, the perception of the server of its own state is straightforward. Either the server is on-line or not. This may be simple, but needless to say, the server does perform many actions. The flow of these actions defines the procedures which should be applied when a certain action occurs. A diagram depicting the state transitions of the server is found in Figure 14. The diagram, simple as it is, does not say anything of the nature of these actions and how they should be performed.

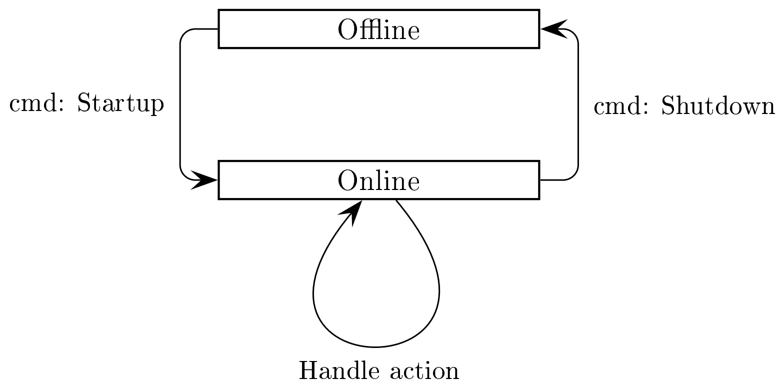


Figure 14: Server state diagram.

The actions performed by the server all correspond to actions taken by some of the clients connected to the network. Thus, some action might occur on the basis of a client interaction which should be handled accordingly. How these actions are performed by the server may be seen as irrelevant to the design of the protocol, but as these actions might spur topology changes, the flow of the actions is needed to to defined following in the next sections.

13.1.1 Client connect

Upon connect, the server must initiate the process of authorizing the new client. This process is depicted in Figure 15. The process initially starts by receiving the connect request. This will be transferred to the server application which must grant or deny access to the client requesting a given video. The response from the server application must provide the server protocol with the measured bandwidth of the client as will be given in Section 14.3. If the server application grants access to the client, the protocol must seek a spot in the topology for the client. If this cannot be found, the protocol may seek to relocate other clients to make room. This relocation may result in disconnecting some other client, if the connecting client can provide a higher bandwidth than already connected clients. Finally, if a spot has been found, the internal representation of the topology must be updated, and the set of clients which are to distribute data to the new client must be informed. The new client must furthermore be informed from which clients the data stream will come. Obviously, if the set of clients initiate data transfer before the receiver has been prepared to receive data, the receiver will begin discarding packets as they will be regarded as invalid. As the receiver is informed by the server that it must prepare for receival, the data received from the set of clients streaming data suddenly becomes valid. But the first part of the packets has been thrown away and the receiver will now issue a number of requests for resending of data.

A solution to this problem might be for the server to wait for an acknowledgment from the receiver indicating that it has now been prepared for receival. Thus, the server will need to retrieve the set of clients, inform the receiver and wait for an acknowledgment. When this has been received, the server may inform the distributors that they should start sending data. However, this solution is not sensible as the nature of streaming, being time critical, may result in the senders not having data available any longer. Hence, the server retrieves a set of clients which has data available and informs the receiver. In the interval from this information is dispatched and to the server receiving the acknowledgment, some or all of the set of clients may not have data available any longer.

A second solution is to let the receiver and the set of senders set up a mutual agreement accordingly before initiating the stream. But once again, the receiver could be compelled to set up an agreement with a potentially high number of clients before the data transfer can be initialized. This can result in some of the clients not having data available any longer due to the long waiting time incurred in setting up an agreement with a high number of senders. Furthermore, the only connection established between the clients a DCP connection transported by UDP. If some packets were lost using the unreliable transmission some logic would be needed to ensure that this was remedied, only to extend the possibility of the clients not having data available any longer.

To simplify matters, the protocol assumes that the receiver of data will always have had enough time to prepare itself to receive data. The simple assumption is thus, that the time spent sending packets through the network will always be lower that the time spent preparing the receiver for data receival.

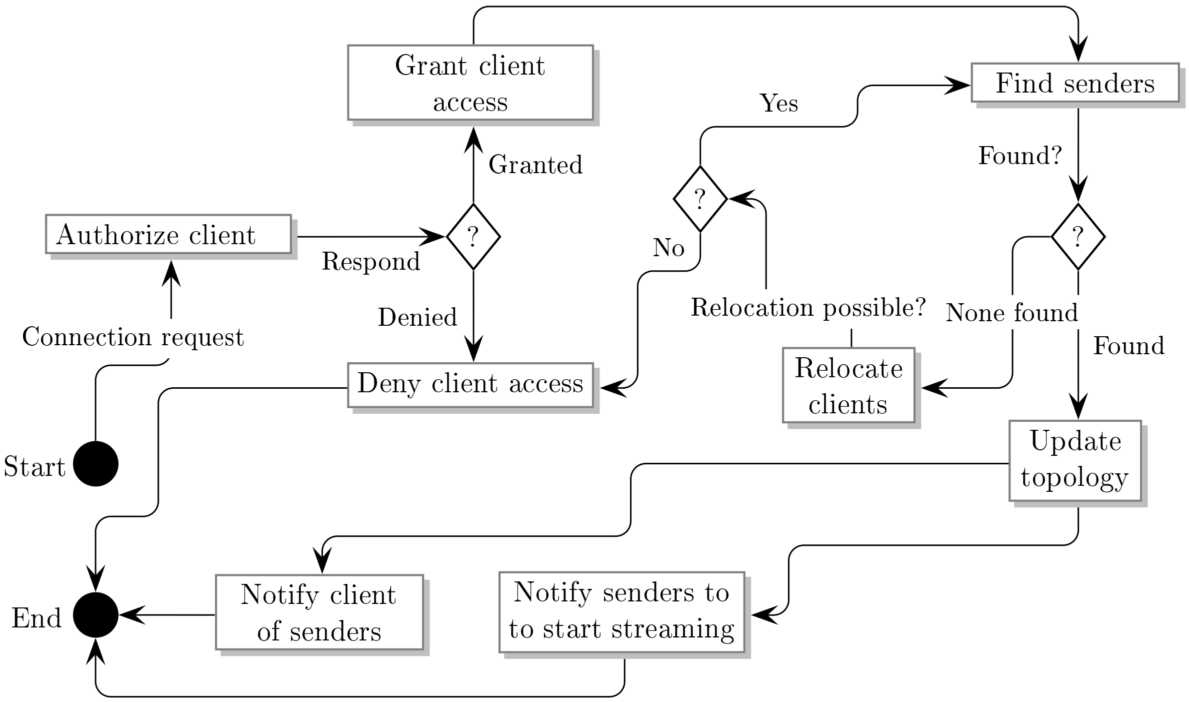


Figure 15: Connect action.

13.1.2 Client Interaction

Upon client interaction requests, the server protocol must first update the internal representation of the topology. If the interaction request is a stop action, this is handled identically to the disconnect request. Thus, the representation of client interaction in Figure 16 only handles pause and skip. If no relocation is needed, e.g. the client wishes to pause and no other clients receive data from the pausing client, no further work is done. If, on the other hand, the client needs to be relocated, the server protocol needs to find a new set of clients which can forward data to the client. If this is not possible, the protocol checks if relocation of other clients is possible, or possibly disconnect other clients to make room for the interacting client. If this is not possible, the client is disconnected. Otherwise, a new set of senders is elected.

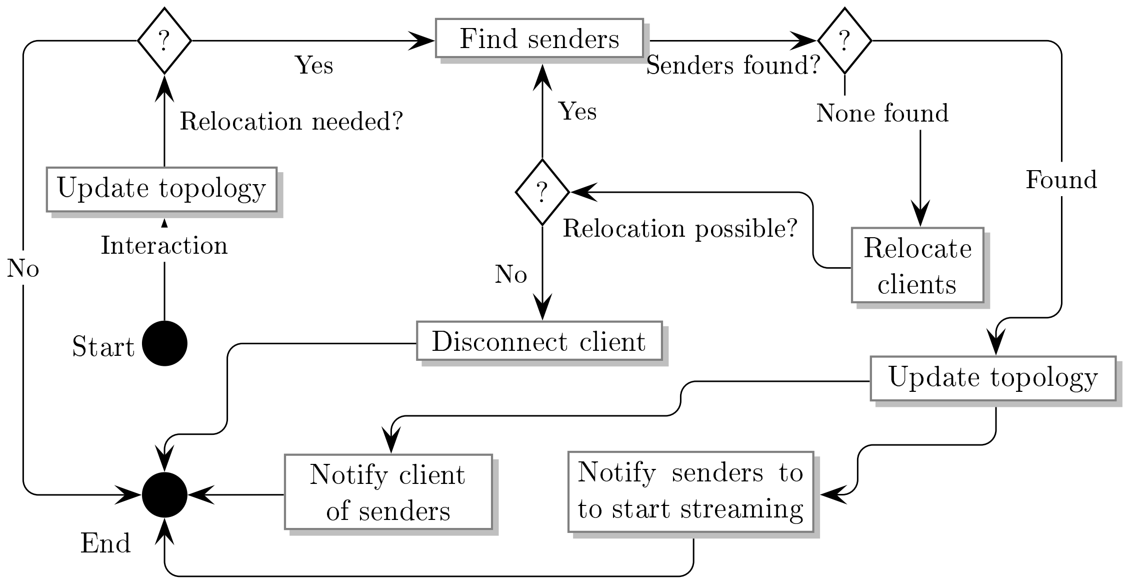


Figure 16: Client interaction action.

13.1.3 Client disconnect

When the server receives a disconnect command, it must perform the action depicted in Figure 17. The server must check if a relocation is needed due to clients receiving data from the disconnecting client. If this relocation is needed, the server may either relocate the affected clients or disconnect some or all of the affected clients if another spot cannot be found.

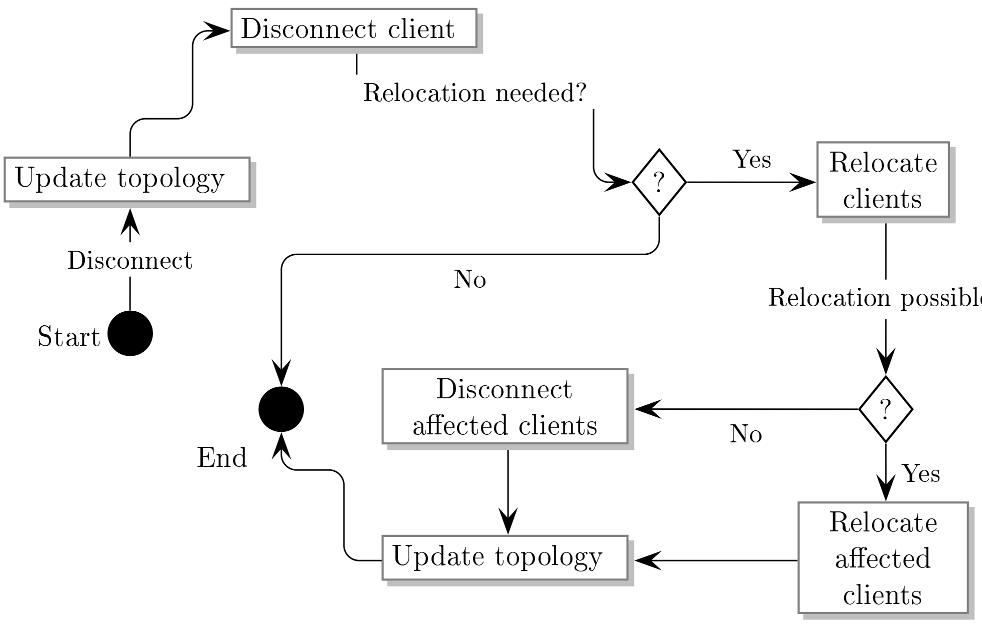


Figure 17: Disconnect action.

13.2 Client states

The state of a client is used by the server to identify if some action requested by the client is allowed. Thus, the action will initiate a transition in the state of the client as it is seen from the server. This state transition diagram is depicted in Figure 18.

The different states the client can be in are:

Not connected: This state is both the entry and exit state of a client. Thus, a client initiates the connect phase as not connected and if the server disconnects the client or the client itself wishes to be disconnected, the state of the client changes to not connected.

Connect pending: When the client has initiated the connect procedure the client must wait until is has been acknowledged or disconnected by the server. In the interval between the connect request and the response of the server, the client is in state 'Connect pending'.

Connected, initializing: When acknowledged, the client must wait until the server has passed vital information about where the video is to be received from before the client may initiate play-back.

Buffering: Before play-back can be initialized, the client must pre-buffer an amount of data. How large this amount is will be elaborated upon in Section 14.2.

Playing: When the client has buffered the required amount of data, its state changes to playing.

Paused: When the client pauses the video, its state changes to paused. From here, the client may only go to playing state.

Skipping: Upon the skip interaction, the client may move to the skipping state. While in this state, the client still performs play-back of the video, while waiting for the server to select a new location in the network. When this location has been found, the client moves to the buffering or playing state, depending on the type of skip performed. This is elaborated upon in Section 14.2.

Stopping: This state is used either when the client wishes to stop the video or the video reaches the end.

The state transition diagram found in Figure 18 depicts how the client changes from one state to another. The diagram is quite straightforward and therefore needs no further explanation. The key point of the figure is simply that the client should obey to the orders sent from the server. However, it should be noticed that many of the state transitions should be equipped with a timeout which should assure that the client cannot be stuck in a state transition from which it cannot continue.

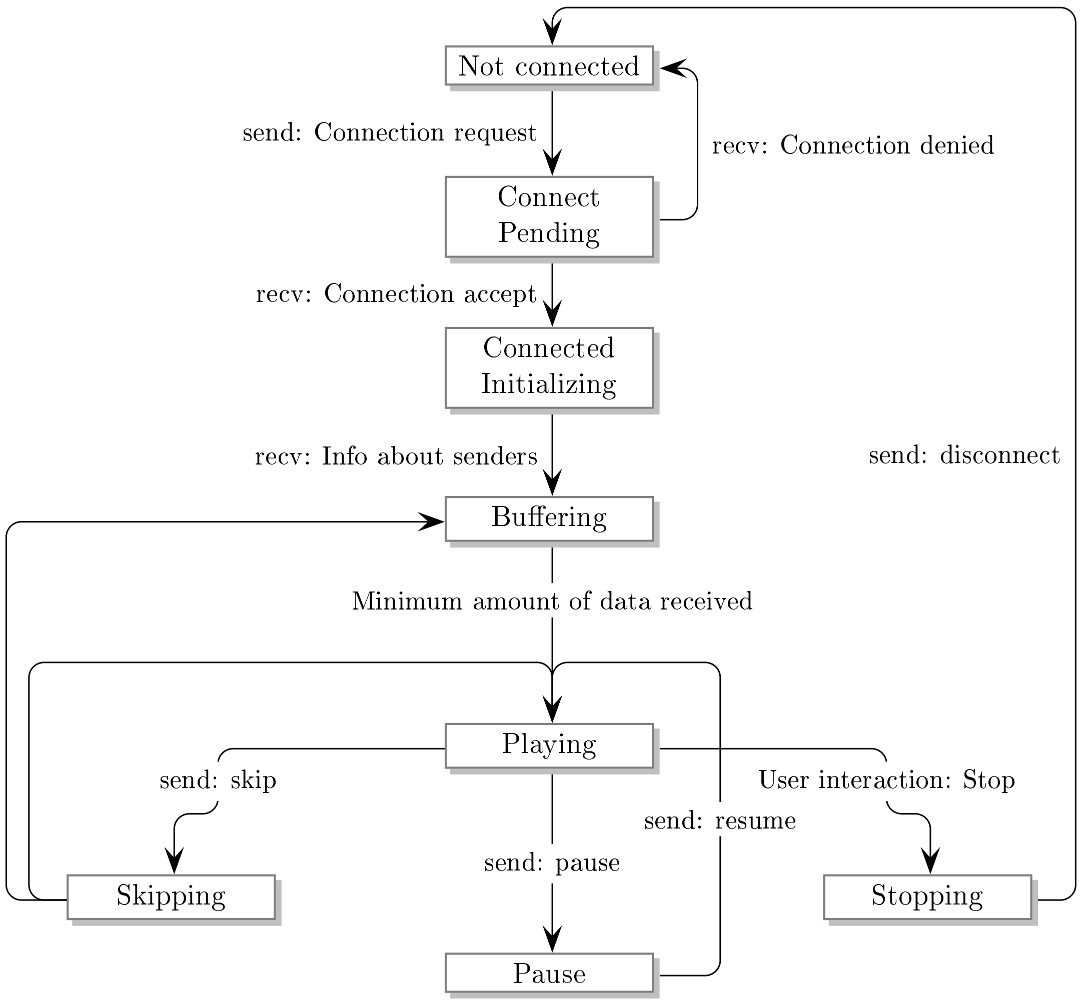


Figure 18: Client state transition diagram.

14 Mechanisms

This section describes the mechanisms which define the internal logic of the protocol. These mechanisms will help identify the different packet types which will be used by the protocol to exchange information the nodes in between. Implementations of the protocol must conform to the workings of these mechanisms in order to enable co-operation between heterogenous implementations. To account for these mechanisms a short description of these is given here.

Fragmenting data: To enable transport of multimedia data, the raw video file must be parted into smaller bits which can be transmitted piecewise. This mechanism describes how data is fragmented and how the bits must be transmitted.

Client buffering of data: The solution to the problems concerning user interaction when using bandwidth sharing is to inject a buffer on each client. How this buffer must work is therefore an important mechanism of the protocol.

Bandwidth: This mechanism refers to a set of methods applied when calculating and regulating the bandwidth of a client. This is needed as the protocol must be able to determine how much bandwidth a client has available and how this is to be utilized.

Round-trip time detection: The protocol must define a mechanism for calculating the round-trip time between two clients. This must be done to set up test scenarios which can help the protocol to identify if errors are occurring.

Error detection: For the protocol to identify if errors are occurring, a set of error detection mechanisms must be present. These mechanisms will be able to detect errors and handle these accordingly.

Selection mechanism: This central mechanism is used to find the logical position in the network of a new or relocated client. This mechanism is given in its initial state the logical appearance of the network and terminates with a possibly updated logical appearance of the network. Thus, it will output a possibly empty list of clients identified to stream data to the new or relocated client. If this list is empty, no suitable entry point could be found. The mechanism is not a direct part of the protocol design, as implementors of the server-side protocol may implement different selection mechanisms. However, the mechanism, regardless of how it is designed is a vital part of the protocol, which is why this has been included in the design. We identify this mechanism as important to the success of the protocol, as it defines a central point where poor design may lead to low performance.

14.1 Fragmentation of data

In order to distribute and stream video as required in Section 2, the data needs to be buffered as described in Section 10. This involves dividing the video into smaller pieces of data. How this is done is the subject of this section.

As described in Section 12.2 video data will be transported using UDP which imposes an upper limit of 65 Kbytes[1] for the size of data transmitted in one packet. Thus, the upper limit on the size of a fragment is limited to the size of one UDP packet. It could be argued that a fragment could be sized to fit more than one UDP packet, as this size could in a high quality video equal a single frame or even less and therefore this is not much data to operate with. But from reasons of simplicity, one UDP packet will contain one fragment.

Furthermore the functionalities offered by on-demand streaming force the need to skip between sequences of the video, as described in Section 3.5. As skipping between fragments sized 65 Kbytes will not be feasible, an entity for grouping fragments is necessary. The size of this entity has to be substantially larger than a fragment, as this will serve the purpose of skipping in the video. Thus, the following entities will be used:

Data Block: The smallest fragment the stream needs to be divided into will be referred to as a Data Block (DB). These blocks will be transmitted piecewise using DCP.

Buffer Block: Grouping of DBs will be done using a Buffer Block (BB). BBs constitute the pieces of the video between which a skip can occur. Following this, a BB will contain a number of DBs.

Referring to a specific point in a video is done using the tuple $\langle x, y \rangle$ indicating DB y inside BB x. The further use and size of these blocks will be discussed and determined in the following sections.

14.1.1 Buffer Block

The main purpose of dividing the video into BBs is to define precise points in the video to skip between. There are several things to consider when deciding how many, and thus how large blocks, a video should be split into. This decision is taken in the light of the following properties:

Distance between skips: As BBs enable skips from one block to another, the size of a BB will determine the distance in time between skips.

[1]The maximum size of an UDP packet is 65 Kbytes, but this includes the IP header and the UDP header. Furthermore, some architectures do not support the full size of the UDP packet, which may lower the size further.

Number of streams dispatched from server: Evidently, the more BBs the video is split into, the more consecutive streams the server will be able to dispatch. Thus, under the assumption that a client holds at least one BB at a given time it will be able to forward this BB to another client. Therefore, in theory, if the video has been parted into 100 BBs, then the maximum number of clients the server needs to stream the video to is 100.

14.1.2 Data Block

The DB is the smallest fraction a video will be divided into. As stated above, a DB is meant to be contained in one packet dispatched by the DCP. The property of dividing each BB into a number of DBs is a result of the requirements in Section 2, introducing the need for a client to receive data from several sources. Furthermore the time critical property of multimedia streaming forces a need for a certain amount of data to be available at the client at all times. As a result of these two properties, several clients transmitting data to another client will transmit one BB at a time, as depicted in Figure 19. The figure depicts that a number of DBs constitute the contents of a single BB. This BB is received from multiple sources, where several clients each send a subset of the set of DBs contained in the BB. Following this, each client will transmit a number of DBs of each BB causing a full BB to be received at a time. This property is of course only approximate, as some clients will transmit data faster than others, and the receiver may in practice end up receiving data from the next BB before having received the full contents of the first. But absorbing these fluctuations is exactly one of the purposes of the buffer.

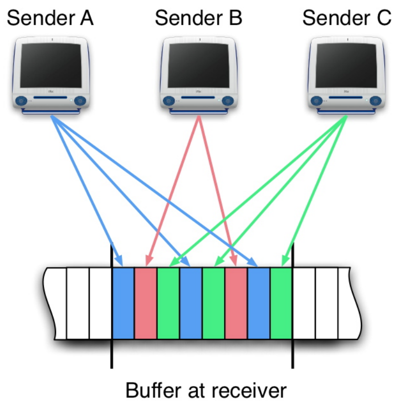


Figure 19: Receiving data from multiple sources.

Determining how many DBs should be contained in a single BB requires an analysis of two factors, namely:

Physical limitations: As already mentioned the DCP will be implemented using UDP, which induces a limit of 65 Kbytes upon the size of the DB, assuming that a DB should fit into a DCP packet. This causes a 65 Kbytes DB of a high quality video to contain a much smaller part (in terms of playing time) than a low quality video.

Number of concurrent streams: The number of DBs in a BB must be at least the number of clients from which a client can receive data. This property is a direct result of enabling a set of clients to send a collected BB together. Clearly, the number of data senders cannot be higher than the number of DBs contained in a single BB as this would unnecessarily complicate the organization of which clients should send which DBs. As a result, if the protocol is setup allowing a client to receive data from 10 other clients, a BB should contain at least 10 DBs. The reason for this demand is merely a design decision made with the purpose of ensuring that all the clients are sending data to the same BB, hence one DB is filled one at a time as described above.

Briefly, it becomes clear to the observer that the collected number of DBs contained in a video file is related to the size of the video. A file occupying 4 Gbytes of space will thus have a minimum number of $4\,Gbyte/65\,Kbyte \approx 65.527$ DBs. How many DBs contained in a BB thus becomes a question of how many BBs a video contains as given by the server application. The relationship between BBs and DBs and how these entities are used in the underlying network is depicted in Figure 20.

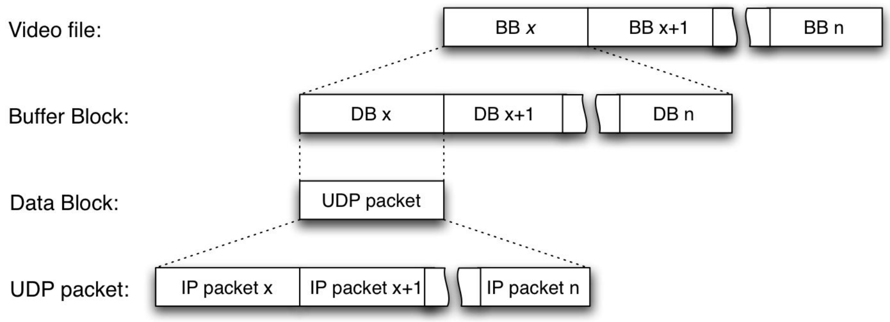


Figure 20: Relationship between BBs, DBs and the underlying network.

Based on these assumptions and facts it would be impossible to set accurate numbers on these parameters and therefore these will be made as variables which can be individually set.

14.1.3 Blocks and buffer size dependency

Logically the buffer will be divided into BBs. As the BB is only a logical entity, it could be argued that the buffer be divided into DBs, but as the BBs are made up of DBs this is merely a matter of definition. Furthermore, this graduation enables us to store and delete BBs one at a time, which will give better control of what data is available and when, as a small block will be written out of memory much faster then a larger one. Thus, determining which parts of the video a client holds in its buffer is done based on BBs, hence its name buffer block.

It should now be clear that the sizes of the BBs and the DBs are closely related. Hence, changing one parameter regarding the size of a DB will also affect the size of a BB. With respect to these dependencies we try to construct a set of equations which in a more formal way display these dependencies.

$$\texttt{sizeof}(\mathrm{BB}) = \texttt{sizeof}(\mathrm{DB}) * \texttt{numberof}(\mathrm{DB\ in\ BB}) \tag{1}$$

Equation 1 expresses the relation between the sizes of BBs and DBs. Furthermore, the number of BBs contained in the video is determined by the server application as a relationship between the skip distance in the video and the maximum number of streams dispatched by the server. Thus, the demands for the buffer becomes a function of the BB size and the capacity of the clients as depicted in Equation 2.

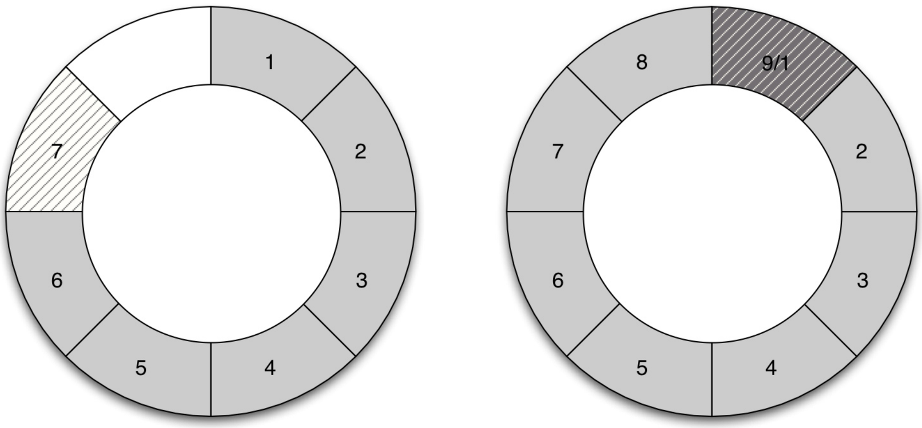
$$\text{buffer demands} = f(\texttt{sizeof}(\mathrm{BB}), \mathit{capacity}) \tag{2}$$

14.2 Client buffering of data

In order to enable redistribution of video content to other clients, a client must buffer a certain amount of data. This amount of data is limited by the amount of data available at the client. As mentioned, an earlier study [35] has shown that to render peer-to-peer streaming efficiently, over 50 percent of the video should be buffered. But there are several reasons why this is not a preferable solution. Some of the most important reasons are security and the costs incurred when producing clients implementing the protocol in hardware.

14.2.1 Buffer contents

The buffer is conceived logically as a circular buffer, but in practice it will be implemented as a static continuous piece of memory. This is implemented by moving a pointer back to the beginning of the buffer once it reaches the end, as depicted in Figure 21. This scenario shows how the buffer at time t' wraps around and overwrites the contents of the first buffer block. In order to illustrate the functionality of the buffer a number of expressions needs to be defined.



(a) Buffer at time t (b) Buffer at time t'

Figure 21: A circular buffer.

First several pointers need to be defined which determine points inside the buffer where a certain activity takes place:

Play Pointer (PP): This pointer indicates the present position of the client in the video pointing to the data being displayed on screen. The buffer block containing the PP will be referred to as the *play block*.

Receive Pointer (RP): This indicates the point where data is being received from other client(s). The pointer thus indicates where the next DB received from the sender will be written. As data can be received from multiple sources, one receive pointer will be used per sender. The buffer block currently being filled will be referred to as the *receive block*.

Stream Pointer (SP): This indicates which DB is currently being sent to another client. One pointer will be used for each client data is forwarded to.

Apart from these pointers, the buffer is divided into different areas, each with their specific purpose. As the buffer is visualized as a circular memory segment these pointers and areas are moved around the buffer all the time. Thus, it makes no sense to depict the position of these pointers and areas unless it is clear that this can only be viewed at a certain point in time. As the PP moves into the next block, the areas slide further along the buffer.

Absorption Area (AA): This area lies ahead of the play block and contains data which must absorb network fluctuations. The size of this area is defined individually by each client, as network stability varies from client to client. Under optimal conditions, the buffer block lying immediately after the AA will be the receive block. The buffer is said to be in a *critical state* if the amount of valid data in the AA is below the required amount of BBs (AA size). Put in another way there has to be a certain amount of data to be played in front of the Play Pointer. This amount of data is what under normal circumstances gives the client enough time to react to a line breakdown etc. and request the server to solve the problem.

Reserved Area (RA): This area contains data which is not allowed to connect new clients into. The RA lies immediately ahead of the AA and will under normal circumstances begin with the receive block. A skip outside the buffer will move the PP into the first block of the RA, hence the receive block becomes the play block and the RA becomes the AA. Skipping will therefore result in filling the RA with data from the new point in the movie where the PP has moved to.

Connect Area (CA): Connecting other clients can only be done in a certain portion of the buffer. This data has already been played by the media viewer, or is under immediate play back. Clients can only be connected to contents of the *connect area* if data is sequential. This entails that if a skip has occurred earlier on and some data in the CA is not sequential in relation to the PP, then clients can only be connected to the part which is sequential in relation to the PP.

As the CA of the clients' is used for attaching new or relocated clients into the sizes of the clients CA constitute the total amount of memory available to the protocol. Evidently, the smaller the amount of aggregated memory provided by the clients the higher an amount of clients must be attached to the server. Therefore, the sizes of the clients CA must be as high as possible. To assure this, the server must have a way of informing the clients of the desired size of the CA. But as the memory available to the clients may differ this desire may only be perceived as a guideline by the client. Therefore, clients which use a CA smaller than what is advertised by the server are regarded as *anti-social*. The reason is that the memory of the clients is regarded as a shared resource used for buffering data and passing this on to other clients. If a client does not contribute with a portion of memory large enough to satisfy this demand, the client will potentially starve the data in the system. Being anti-social does not mean that the client cannot be connected, but it means that it has a higher risk of being disconnected and may experience poor service.

Send Area (SA): The send area contains the area in which sending data to other clients is permitted. The area spans the entire buffer. If data contained in the buffer does not lie sequential with respect to the receive pointer of an attached client, the client will stop sending data once the send pointer reaches the invalid data. The attached client will hereafter need to contact the server to be relocated.

To further account for these terms, Figure 22 displays a simple buffer sized 8 BBs. The size of the AA is set to two which is also the size of the RA. The size of the connect area is thus 4, as no clients can be connected to data placed in either the AA or the RA. As the SA spans the entire buffer it has size 8.

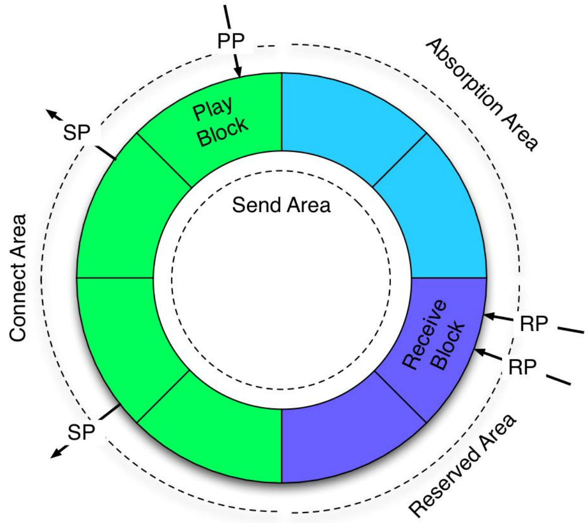


Figure 22: Visualizing the contents of the buffer.

As will be described in Section 14.3 when clients are relocated to receive data from another client due to a skip, this will result in speeding up the rate of which the video is sent to fill the AA of the receiver. Therefore, connecting clients must receive data at a higher speed than the sending client is viewing the video, possibly resulting in the SP passing the PP. This is no immediate problem as long as the SP does pass the RP. This can only happen if a client needs to receive more BBs at extra rate than the sender has in its AA. As the size of the AA varies from client to client, this could indeed happen if the receiver has an AA of higher size than the sender. Thus a constraint must be introduced:

A connecting client cannot be attached to a BB
closer to the end of the sender's AA than the size of its own AA.

This situation is visualized in Figure 23. A client wishing to connect (not displayed in the figure) to the client depicted in the figure has an AA of only five BBs. From the above constraint, the connecting client can only connect into buffer blocks 28-35 as connecting outside this window would result in the need for receiving data at extra speed for a full five blocks, possibly resulting in the send pointer crossing the receive pointer.

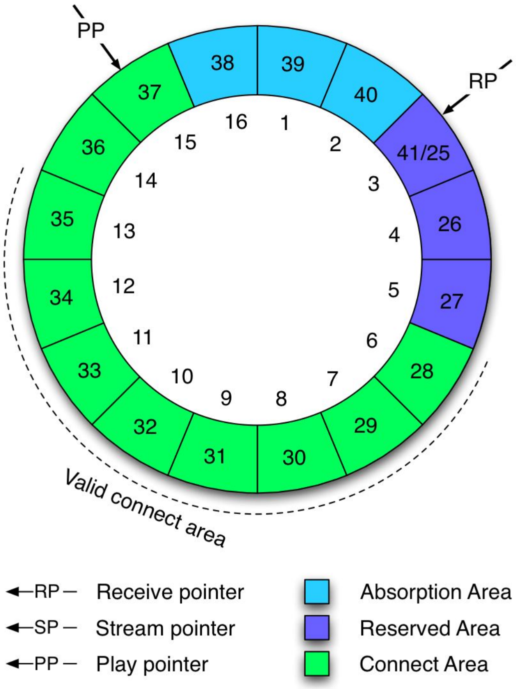


Figure 23: Connecting clients inside the Connect Area (CA).

14.2.2 Buffer size

As described in Section 10 the buffer serves not only the purpose of absorbing network fluctuation but also forwarding data to other clients. Thus, there is a need for an amount of data, both ahead of the client's playing point (to absorb network fluctuations), and behind the playing point (to forward data to other clients).

The size of the buffer is thus formed by the amount of BBs required, the sizes of the AA, RA, and the CA. How large these portions of data need to be is inherently hard to determine, as they are both dependent upon the stability of the network and the amount of memory available to the client. These three two sizes will therefore be variables at each client. The collected buffer size will thus be made up by:

$$\texttt{sizeof}(\mathit{Buffer}) = \texttt{sizeof}(AA) + \texttt{sizeof}(RA) + \texttt{sizeof}(CA) \tag{3}$$

Deriving from this equation the minimum buffer size must thus be 3 BBs. In this case, the play block will follow immediately after the receive block, and the play block will be the only block allowed to attach new clients into. In this scenario the block which is currently being filled may wrap into the next block, which is in the process of being played possibly overwriting data which has not been played yet. Thus using a buffer of size 3 BBs is obviously not preferable, yet possible. Using 4 or more blocks would ensure higher stability and the possibility of connecting clients into another block.

14.2.3 Initiating play back

Upon initiation of data receival, the AA will be completely empty. Therefore, to ensure smooth play-back, the client could be forced to wait for the area to be filled. Depending on the size of the AA and the size of a single BB this may sum to a substantial waiting time for the end-user and may not be considered instant play-back. Thus, there is a need for initiating play-back before the AA is filled. But as streaming in its nature only sends data with the same speed as the play-back rate of the video, a client needs to be able to receive data faster than the play back rate. This will be elaborated upon in Section 14.3.

This involves playing the video before a full BB is present. As shown on Figure 24 a client has skipped to BB number 51 in a movie, and has received only the first 4 DBs of this BB. Considering this scenario the client could potentially start displaying the movie starting from DB 1 in BB 51. But as different clients may posses different properties as to the quality of the Internet connection the client will have to be in control of this. Thus, controlling how many DBs the client will have to buffer before play-back can be initiated is up to the client. Furthermore, there is a need to receive a certain portion of the video, as some video standards use a header in the beginning of a multimedia file which identifies media bit-rate, frames per second etc. Thus, the protocol must support a way of identifying how many DBs should be received before the initial play-back can be done.

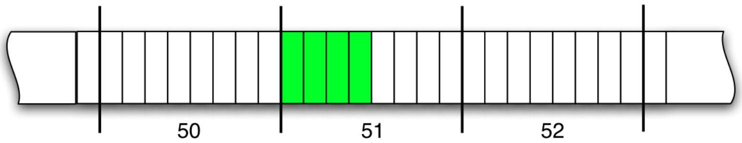


Figure 24: Video playback immediately after receival of data.

A visualization of how the buffer is used during initiation of play-back is found in Figure 25. The scenario shows how the buffer is used during startup with an empty buffer. When the minimum needed DBs are received, the client starts displaying the video. The data in the play block lies within the AA until enough data has been received to fill the entire AA. As data is being filled into the CA other clients can be attached, provided they fulfill the constraint given in Section 14.2.1 regarding the valid connect area of a client.

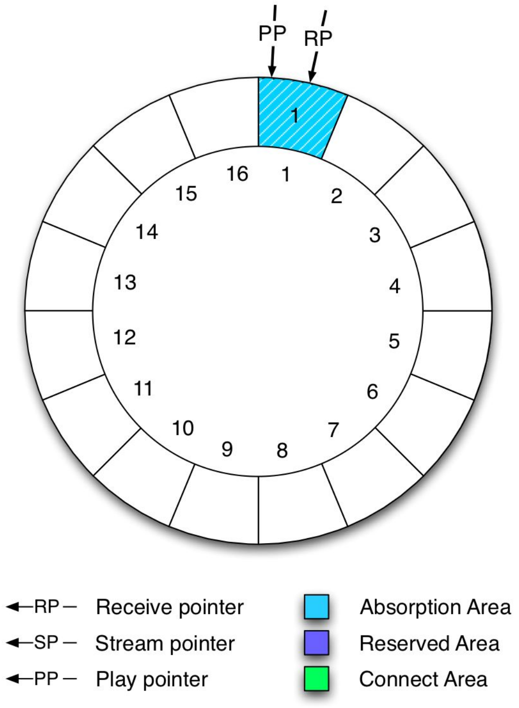


Figure 25: Contents of buffer at startup.

14.2.4 Normal play-back

Normal play-back is the typical state of the buffer during a streaming session. The PP has made zero or more trips around the buffer and all the data behind the PP is in sequential order meaning that data contained in the CA is ordered sequentially. The client receives data from two other clients and forwards data to three other clients. The scenario is visualized in Figure 26.

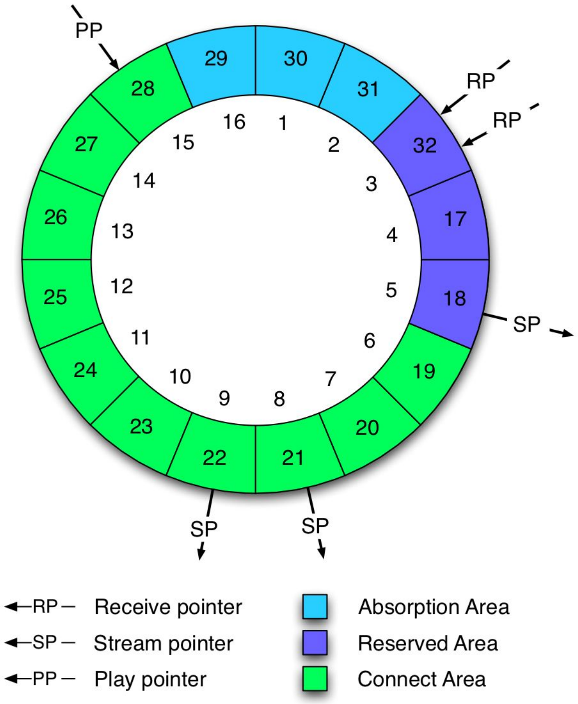


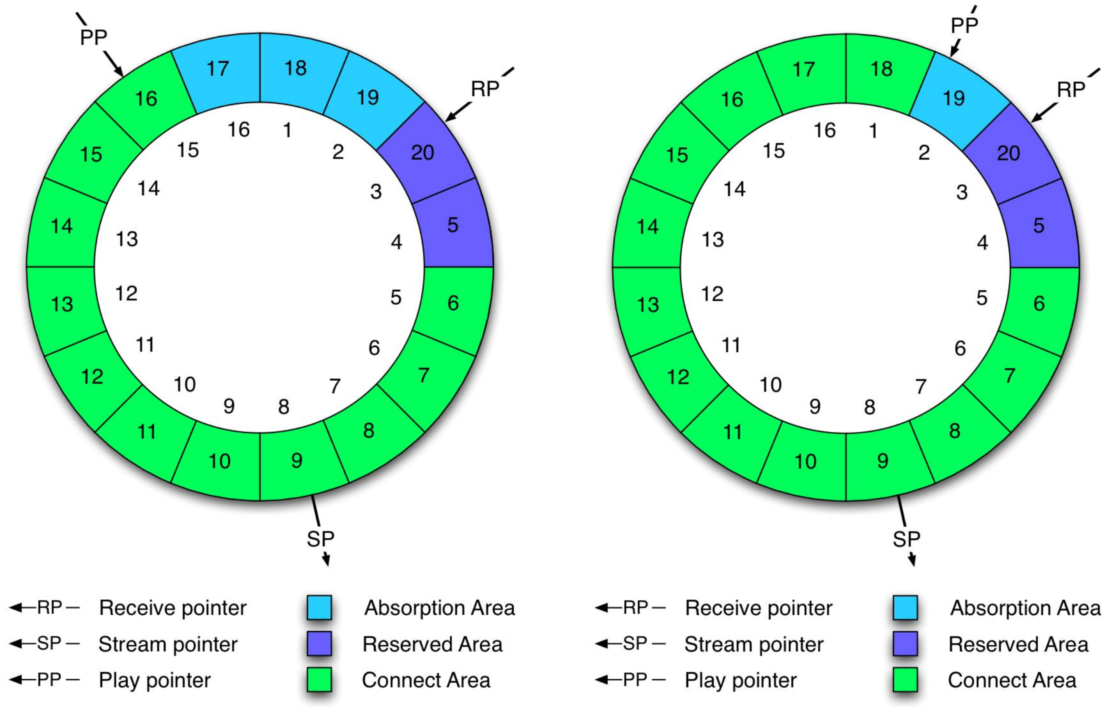
Figure 26: Contents of buffer at normal playback.

14.2.5 Skipping

As described in Section 3.5 skipping is handled in identical to initiation of play-back. Thus, when skipping to another sequence, displaying the video instantly should be possible. A deeper analysis reveals that three types of skips can occur:

Skip forward inside buffer: As the name implies, this is a forward skip to a block which is already contained in the buffer. Thus the PP will be moved forth to the buffer containing valid data. To avoid buffer starvation the AA needs to be filled quickly resulting in the need for receiving data at a higher speed. This will result in the RP moving faster than the PP, possibly overwriting data which is needed by attached clients. In this event the client must inform the attached clients that data is unavailable. If dispatching data needed by attached clients cannot be satisfied, these will have to be relocated to other clients by the server.

The scenario is depicted in Figure 27, where the client has skipped to block 19 at time t'. The result is, that the AA has been reduced, but at the same time the CA has been enlarged. This calls for the need to receive data at a higher rate to enlarge the AA.



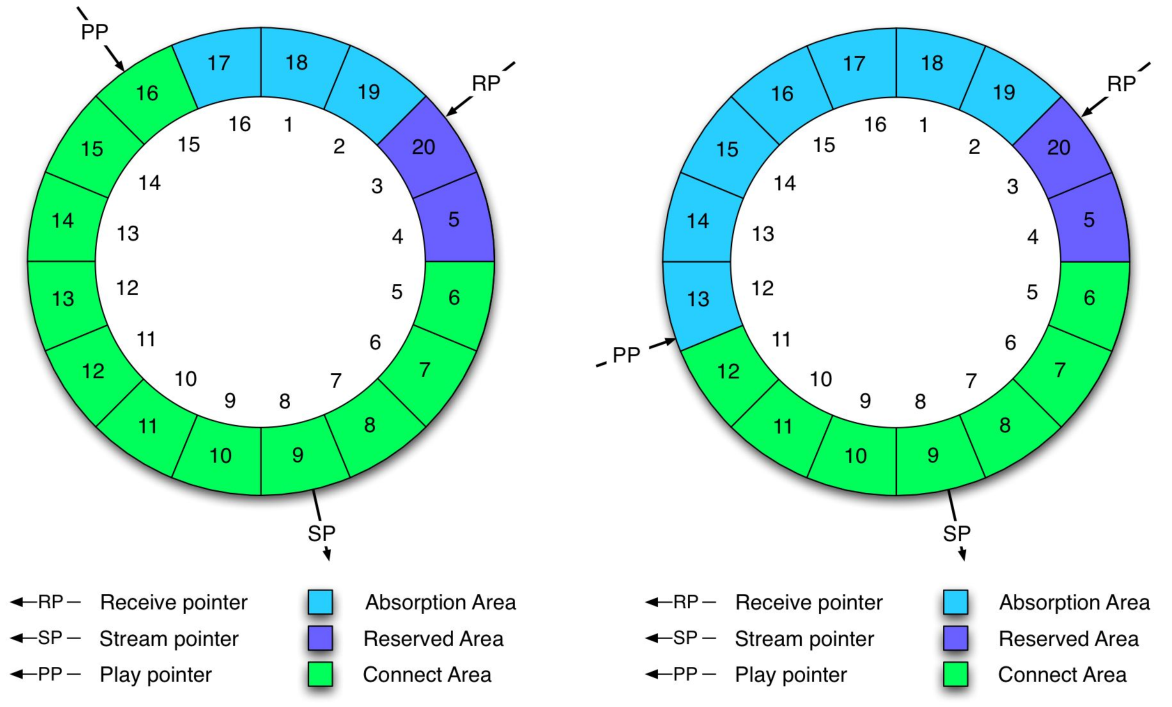
(a) Buffer at time t

(b) Buffer at time t'

Figure 27: Skipping forwards inside the contents of the buffer.

Skip backward inside buffer: Skipping backwards inside the buffer can be done until the receive block. In other works a skip can occur to all data lying between the RP and the PP. This will result in the PP moving back to the intended block. As the PP moves closer to the RA, the CA is reduced to fill as little as only the play block. Thus, the play block will follow immediately after the receive block minimizing the CA. To enlarge the CA once again receival of data must therefore be throttled down, until the size of the CA is once again satisfied. When throttling down speed, there is a risk that attached clients cannot receive data fast enough, possibly resulting in starvation of the buffer of attached clients. Again this will result in the need for relocating these clients to other locations in the network by the server.

Skipping backwards inside the buffer is illustrated in Figure 28. The scenario shows that the client has skipped backwards to block number 13, thus enlarging the AA and shrinking the CA. To accommodate for the highly enlarged AA, the client will need to receive data at low rate, to widen the gap between the RA and the PP and thus enlarge the CA.

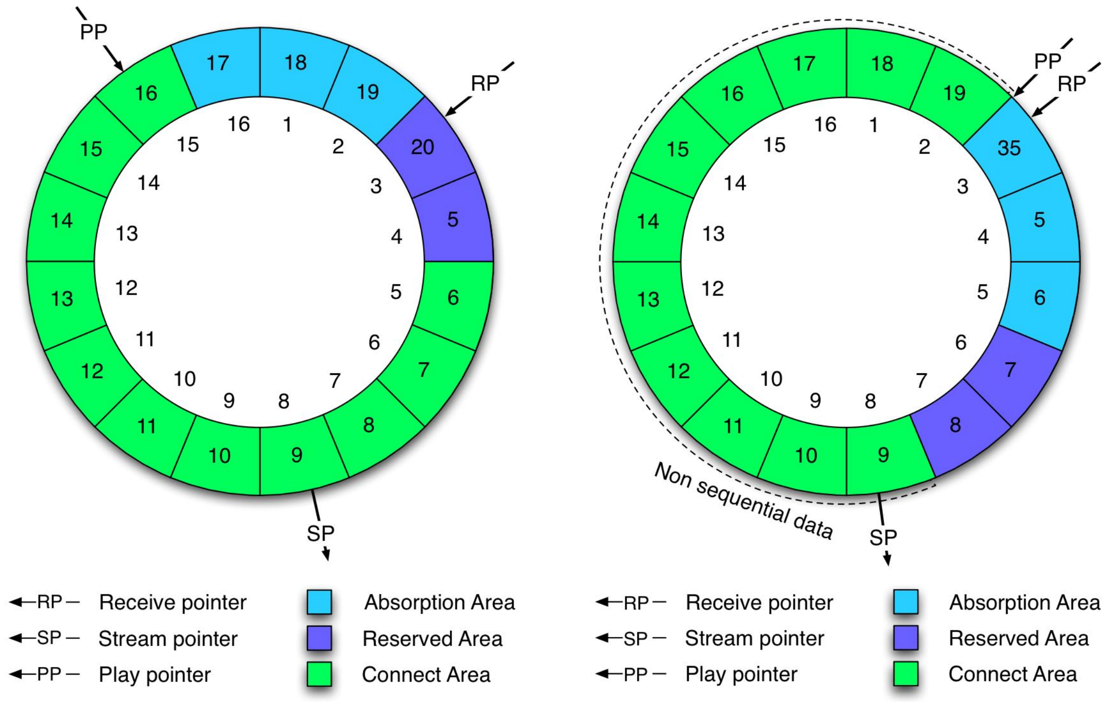


(a) Buffer at time t

(b) Buffer at time t'

Figure 28: Skipping backwards inside the contents of the buffer.

Skip outside buffer: If the PP is moved further back, i.e to a point where it crosses the RP, this would no longer be a skip backward inside the buffer as data would not be available. The same applies to a forward skip where the PP crosses the RP, as this is skipping forward outside the buffer. These scenarios are handled identically by overwriting the contents of the receive block with data reflecting the new point in the video form where content is to be shown. At the same time, the PP is moved to the receive block, thus the block becomes the play block and the receive block at the same time. This scenario closely resembles initiation of play-back as no data in the buffer is valid. Therefore these two scenarios are handled identically, using the same variable which controls how many DBs should be available before play-back is initiated. Skipping outside the buffer renders all data in the buffer *non-sequential*, meaning that data lying in the SA is not sequential with respect to the data contained in the receive block. This will result in the need for relocating attached clients elsewhere by the sever. The scenario is depicted in Figure 29. After the skip, the CA is marked as non-sequential thus resulting in the server being unable to attach clients into the area until the RP has moved away from the PP.



(a) Buffer at time t

(b) Buffer at time t'

Figure 29: Skipping outside the contents of the buffer.

14.2.6 Pausing

As described in Section 3.5 one of the goals is is to provide a pause function similar as to what is seen on a VCR. This should result in the user experiencing a halt in the video stream until he or she wishes to continue playing. How this is visualized by the client application is of no concern to the protocol, but the protocol needs to determine what to do.

One of the characteristics of a media pause is that the length of the pause is not determined on beforehand. One could give an estimate from statistics on the average length of a media pause, but the type of media distributed by the server might have an impact upon this average. Thus, one type of video could generally result in shorter pauses than another.

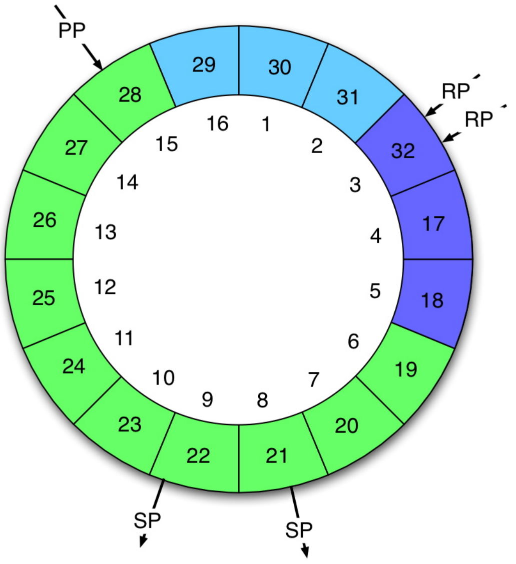
What is interesting about the length of a pause, is that a client attached to a pausing client might not need to be relocated as data transfer can proceed until sending side does not possess relevant data any longer. As long as this does not happen, the pausing client could receive data without displaying it and forward it on to other clients as long as the sender does not overwrite data not displayed by the video player. On the other hand, it might be desirable to anticipate events and act as soon as a client pauses by relocating all attached clients.

To sum it up, the protocol can either choose to relocate a client as soon as the sender pauses the stream, or wait until data is unavailable at the sender. Using the first approach, the protocol might have greater possibility of avoiding stalls in video play-back at the receiver. On the other hand the second approach exploits the possibility of the sender only pausing the stream for a short time. After the sender resumes play-back everything will proceed as normal. How the protocol handles this situation is not part of the protocol design but in the implementation given in this thesis the second approach is chosen.

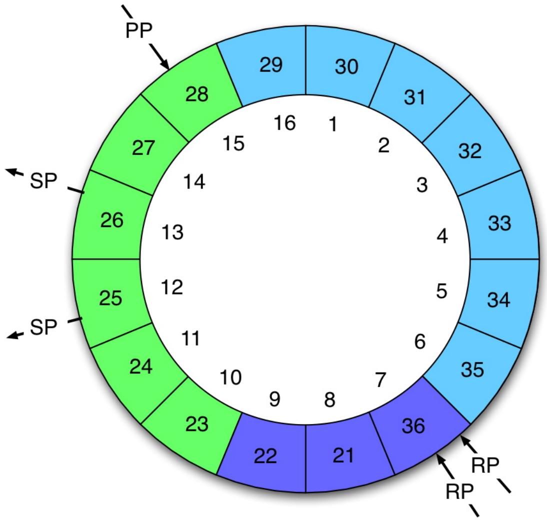
Evidently, attaching clients to a client which is paused is not desired. Attaching a new client will result in the need for receiving data at high rate as playback will need to begin immediately. Thus, if the paused client suddenly runs out of buffer space as it may overwrite data not displayed by the player yet, data receival must be brought to a standstill. In this event the newly attached client might not yet have its AA completely filled but needs to be relocated. This relocation might take up too much time, resulting in the player stalling. Hence, there is a need to first inform the server of a pause in order to let the server know that no clients should be attached before a resume has been performed. Secondly, the client must be able to inform the server that it has run out of buffer space and cannot hold any more data. In this event, the server needs to relocate all connected clients and stop transmission to the paused client from other clients. In a sense, this will result in the client still being a part of the network, but without any connections. Thus, the network will form a disconnected graph.

Pausing the video is visualized in Figure 30. At time t, the client pauses the video and the server is notified. At time t' the video has been paused for an amount of time, resulting in the enlargement of the AA. Notice the similarities between skipping backwards inside the buffer and pausing. From these similarities it can be concluded that pausing and skipping backwards are closely related as the play pointer moves backwards in time in both cases. This is in fact consistent with [41] which notes that *'the pause action belongs to the backwards interactions'*.

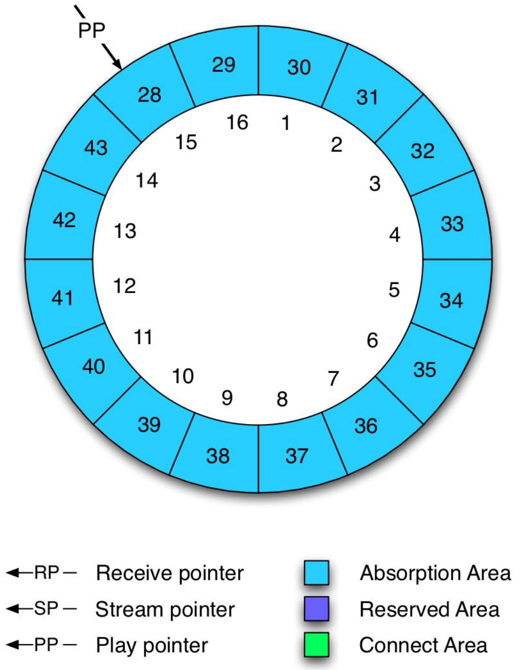
At time t'' the buffer has wrapped around and the receive block has reached the play block. At this time all attached clients has been relocated and receival of data has been brought to a standstill.



(a) Buffer at time t



(b) Buffer at time t'



(c) Buffer at time t''

Figure 30: Pausing the video stream.

14.2.7 Resume

Resuming can evidently only be done after a pause has been performed. If the client still receives data from other clients, the client can play-back the video without further ado. If the client on the other hand has informed the server that it has run out of buffer space, the client must be relocated to some point in the network before data transmission can be initiated.

14.3 Bandwidth

As already stated bandwidth is referred to as capacity. Obviously, all clients must be able to receive data at a rate equal to the bit-rate of the video. Put in another way, a client must be able to receive data at the same speed as is used when playing back the video plus the amount bandwidth of needed by the CCP.

Unfortunately the ability to receive data at this rate is not enough. The reason for this is that, in order to buffer data, a client must be able to fill up a certain amount of its buffer ahead of the PP in concurrence with displaying the video. Thus, the sender of data will need to raise the speed at which data is sent. Another approach to this is to let a third client send the extra amount of data in situations where this may be needed. Hence, a client not using its upstream bandwidth could be found to deliver the extra amount of data. But this would not lower the total amount of bandwidth needed, and would greatly add to the complexity. Therefore the simple approach of demanding that clients can provide an extra amount of bandwidth is chosen. Furthermore, the client must be able to throttle down the rate in order to widen the gap between the PP and the RP in the event of a backward skip inside the buffer as described in Section 14.2.5.

To connect a new client to the network therefore requires a way of figuring out how much bandwidth the client can provide.

14.3.1 Bandwidth detection

Upon connect the available bandwidth of the client needs to be determined. If the system is controlled by an Internet Service Provider, the bandwidth of its customers may easily be available. But if the system is used by a third part with no knowledge of the bandwidth of the client, other methods must be applied. The simple way is to let the clients advertise their bandwidth, but as described in Section 6.3 the structure of the Internet causes the advertised capacity to be very unreliable. Measuring the available bandwidth is a difficult subject. A lot of research has been done in this area resulting in a large variety of methods and theories as seen for instance in [22]. This area is by far large enough to be a topic for a master's thesis on its own and since, it does not play an important role to this project, it will be regarded as out of scope.

The assumption is that some other software located outside the protocol handles this measurement, but even this assumption has its limits. If the bandwidth can be measured one further assumption needs to be made: The capacity from point A to point B is equal to the capacity from A to another point C provided that the advertised capacity of C is not lower than the advertised capacity of B. This enables us to trust the measured capacity of a connecting client and rely upon the fact that wherever the client is logically placed in the network, its speed will never be lower than the advertised speed. Note that the assumption is not related to the rule of mathematical transitivity.

This advertised speed has to be provided to the protocol and will include only the upstream bandwidth. If the downstream bandwidth is too low to receive the bit-rate of the video, the client should be rejected right away. The provided upstream bandwidth should be the real upstream bandwidth, meaning that, the clients ability to receive data is not affected by the amount of data being sent or vice versa. If this is the case, the downstream bandwidth used to receive the video should be subtracted. A client which cannot provide an upstream bandwidth high enough to send the complete video will be regarded as anti-social. Thus being anti-social is not only defined as providing less memory than what is advertised by the server, as stated in Section 14.2, but also if clients contribute less bandwidth than they consume and hence add to a potential starvation of the available bandwidth of the network.

14.3.2 Bandwidth calculation

For the system to determine how and when data is to be streamed, the server needs to know the available bandwidth of all clients. As this information will be supplied from outside of the protocol, the server application instantiating the protocol must pass these informations to the protocol. Apart from this, information about the video which is streamed must be available to the server. This includes, media bit-rate, the duration in milliseconds and size in bytes, BB size in bytes, and the number of DBs contained in a single BB. These numbers are used to calculate how much data each client should and can stream.

The main idea in these calculations is to translate the available bandwidth of the clients into the maximum number of DBs the client is able to send per second. To calculate this number, the protocol needs to calculate the following units:

First the total number of DBs in the video is calculated:

$$\texttt{numberof}(\text{DBsInVideo}) = \frac{\texttt{sizeof}(\text{video})}{\texttt{sizeof}(\text{DB})} \tag{4}$$

This number is used to find the duration of each DB based on the total duration of the video. This number is measured in milliseconds, as the rate of which DBs are sent is expected to be finely grained.

$$\texttt{duration}(\text{DB}) = \frac{\texttt{duration}(\text{video})}{\texttt{numberof}(\text{DBsInVideo})} \tag{5}$$

This number is the time span in seconds in which the client consumes a DB or in other words a new DB should be sent to the client at this interval. Hence, it is possible to find the number of DBs which should be sent per second by:

$$\text{DB/sec} = \frac{1}{\texttt{duration}(\text{DB})} \tag{6}$$

When this number is found we need to look at the maximum upstream bandwidth of the client. This bandwidth needs to carry more than just the amount of data needed to stream the video. As already described the client needs to have the ability to transfer extra data in order to fill the buffer of the connected client. On top of this, the client needs some of the bandwidth for control data sent using CCP. Hence the upstream bandwidth available to send the video from a given client is found by solving the following equation:

$$Band_{total} = Band_{available} - Band_{extra} - Band_{control} \tag{7}$$

where $Band_{total}$ denotes the available bandwidth of the client when data is streamed at normal speed and $Band_{extra}$ is the extra amount of bandwidth needed for throttling up the speed. This value of this variable is a portion of the bit-rate. If $Band_{total}$ is equal to or higher than the bit-rate of the video, the client will not be regarded as anti-social. If the number is lower, it means that the client is not capable of streaming the entire video on to another client and hence is regarded as anti-social.

Based on the above formula the protocol can determine how much bandwidth a client may contribute. This is calculated as the maximum amount of DBs the client may send per second. The calculation is found in Equation 8, where $Band_{total}$ denotes the value computed in Equation 7.

$$\text{DB/sec} = \frac{Band_{total}}{\texttt{sizeof}(\text{DB})} \tag{8}$$

This variable is stored by the server protocol for each of the connected clients. The numbers calculated in equations 4, 5 and 6 are precalculated and stored together with other video information and therefore the only calculations needed during a new client connect is those of Equations 7 and 8.

14.3.3 An example

To visualize the dependency between the above calculations a short example will be given in this section. The following data is given on beforehand:

- The video is divided into 1000 BBs each containing 100 DBs.

- The total playing time of the video is 5,400,000 msec (90 minutes).
- The size of a DB is 40 Kbytes.

The duration of a BB is therefore:

$$\frac{5,400,000\,msec}{1000} = 5,400\,msec = 5.4\,sec \tag{9}$$

Thus, the duration of a DB is:

$$\frac{5,400\,msec}{100} = 54\,msec \tag{10}$$

The number of DBs a client needs per second is:

$$\frac{1000\,msec}{54\,msec/\text{DB}} \approx 18.52\,\text{DB}/sec \tag{11}$$

As given above the size of a DB is 40 Kbytes. Thus the bandwidth used for streaming video data per second is:

$$18.52\,\text{DB}/sec * 40\,Kbyte \approx 740.7\,Kbyte/sec \approx 6.068\,Mbit/sec \tag{12}$$

All of the above are calculated once based on the number of DBs the video is divided into. This is done when information about a video is received from the server application.

The following is calculated each time a client is connected. The server is provided the client's available upstream bandwidth, and on top of this the protocol uses the following variables:

- control data takes 10 Kbyte/s,
- bandwidth maximum should be 20 % extra,
- bandwidth minimum should be 20 % less.

As described above the minimum bandwidth which should be available to a client, if it should be able to forward the entire video to another client must be: $(741\,Kbyte/sec * 1.2) + 10\,Kbyte/sec = 899.2\,Kbyte/sec$. If a client can only provide for example a maximum of 300 Kbyte/s, we can calculate the bandwidth which is available to the protocol:

$$(300\,Kbyte/sec - 10\,Kbyte/sec)/1.2 = 241.7\,Kbyte/sec \tag{13}$$

This means that the client can send only 241.7 Kbyte/sec of the collected bit rate of the video. Since this is below the 741 Kbyte/s needed for the entire video, the client is marked as anti-social with the possibility of experiencing poor service.

Now it is interesting to find the number of DBs the client can deliver out of each BB. The amount of DBs the client can deliver per second is:

$$\frac{241.7\,Kbyte/sec}{40\,Kbyte/\mathrm{DB}} = 6.042\,\mathrm{DB}/sec. \tag{14}$$

And finally the number of DBs the client can deliver in each BB is found:

$$6.042\,\mathrm{DB}/sec * 5.4\,sec = 32.63\,\mathrm{DB}. \tag{15}$$

Meaning that 32.63 DBs out of each BB can be delivered from the client per second. The number is always rounded down, e.g. 32 DBs per BB containing 100 DBs.

Now the server knows that this client is calculated to be able to assist with 32 DBs in each BB in normal mode, but it can deliver $32\,\mathrm{DB}/sec * 0.2 \approx 6\,\mathrm{DB}/sec$ extra, should the attached client(s) need it.

Furthermore, if the client is requested to send e.g. 17 DBs out of each BB, then the client can calculate how long the interval between sending each packet should be:

$$\left(\frac{100}{17}\right) * 54\,msec \approx 317.64\,msec. \tag{16}$$

which is be rounded down to an interval of 317 msec. From this we can calculate that if the client is required to send at maximum speed it would be: $317\,msec/1.2 = 264.17\,msec$ and with minimum speed it would be: $317\,msec * 1.2 \approx 380\,msec$.

14.3.4 Adjustment of client bandwidth

As stated in Section 14.3.1 the protocol will not implement a method for determining the bandwidth of a client. But another obstacle remains: Deciding how to adjust the used bandwidth of a client when streaming still needs to be taken care of. The question is simply, how much bandwidth should be consumed by the client? And what if the client fails to contribute the measured bandwidth? Thus, adjusting the bandwidth will be described in this section.

Several variables are used to store information about the bandwidth of the clients. These variables will be used to select the right client or clients to whom a new client should be connected.

MEASURED_BANDWIDTH: This parameter is set upon startup when a client's bandwidth is measured and handed to the protocol. The value is given as in the number of DBs the client is able to send per second. The value could be higher than what is required to send the entire video indicating that the client is capable streaming more than one full video. Hence, the client cannot be marked as anti-social as the client will at least contribute the same amount of bandwidth as it consumes. This value is not changed during the session of a client and is only calculated upon connect.

EXPECTED_BANDWIDTH: This parameter denotes the expected bandwidth of a client and is adjusted during the client's session. If a client is calculated to contribute with more bandwidth than required by the video, this parameter is set upon startup to only what is required to stream the video. As this parameter is also given in DBs, it would for instance be set to 100 if the rate of the video required is 100 DBs per second, even if MEASURED_BANDWIDTH is higher than 100. Afterwards the value may be adjusted up if necessary.

Adjusting the value down is done when a receiver of data notifies the server that receival of data is not done at the advertised speed. The server will then assume that the expected bandwidth of the client has been set too high and accordingly adjust it down.

Adjusting the value up is done when bandwidth equilibrium cannot be obtained with the amount of bandwidth available to the network. Thus, if the expected bandwidth of a client is below the measured bandwidth, the expected bandwidth of the clients is adjusted up.

The only rule which applies when adjusting the parameter is that it should never be set higher than the measured bandwidth. If the expected bandwidth falls below the rate of the video, the client may be marked anti-social.

USED_BANDWIDTH: This parameter indicates how much bandwidth is currently being consumed. The value is adjusted during the client's session and cannot be raised higher than expected bandwidth, though it may be lower.

EXPECTED_UP: This value is a simple counter used to track how many times the EXPECTED_BANDWIDTH has been raised. This is done in order for the server to keep track if it continues to raise and lower the bandwidth of a client.

EXPECTED_DOWN: This is the counterpart to EXPECTED_UP.

14.4 Calculation of round-trip time

As the DCP will use the UDP protocol, there is a chance the packets may disappear during transport. This unfortunate property raises the simple question of what to do, if a packet gets lost. But answering this question cannot be done, before another question is answered, namely, how do we detect if a packet is lost? As already accounted for in Section 8.3, UDP also has the unfortunate property that packets may be received out-of-order. Thus if packet seven is received before packet six, we cannot rely upon the assumption that packet six has been lost – it might simply be delayed underway.

The protocol could wait a given time to see if packet six would arrive, and when this time runs out take some action indicating the assumption that packet six was lost. But how long time should the protocol wait? As a minimum, it should wait the time it takes for a packet to be dispatched from the sender and until it reaches the receiver. This is known as the round-trip time (RTT), or actually half the round-trip time, as the round-trip time is the amount of time it takes for a packet to reach its destination and come back.

Calculating the round-trip time can only be done using some form of synchronization between the two clients. Thus, a client can send a request, get a response back, and calculate the round-trip time simply by looking at the time between sending the request and receiving the response as depicted in Figure 31.

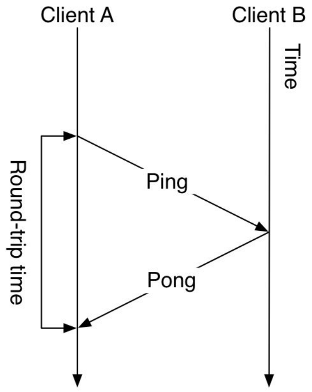
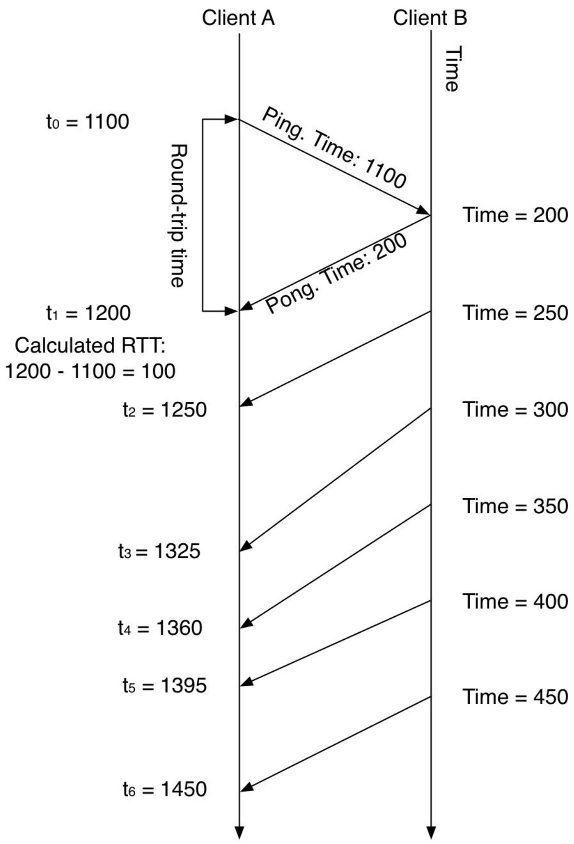


Figure 31: Round-trip time.

This value needs to be updated as it may change over time. This may either be done by initiating a new synchronization procedure or updating the value every time a packet is received. Calculating the round-trip time is therefore straightforward:

1. Upon initialization of data transfer, a ping packet is sent from the data receiver to the data sender.
2. The ping packet is responded by a pong packet containing a timestamp indicating the time the packet was sent.
3. The data receiver will then calculate the difference between sending the ping and receiving the pong.
4. Furthermore, the data receiver will calculate the time difference between the clients by subtracting the timestamp given in the pong packet and the time of arrival. This value will correspond to the time difference between the clients and the time spent by the packet before it has arrived. Note that this variable may be below zero. This is due to the difference in time between the clients, as their clocks will not be synchronized. Subtracting the time stamps may therefore return a value below zero.
5. Consecutively, every time a data packet is received, the receiver can calculate the time difference by looking at the sender's timestamp given in each data packet. If this difference has changed from what was originally calculated, the difference will be added to

a running round-trip time variable. Thus, the protocol can maintain a round-trip time for all clients from which it receives data. The procedure is visualized in Figure 32.



Time	Time difference	Estimated RTT
t_0	-	-
t_1	*Round-trip time calculated. Time difference = 1000*	100
t_2	1250 - 250 = 1000	100
t_3	1325 - 300 = 1025	125
t_4	1360 - 350 = 1010	110
t_5	1395 - 400 = 995	95
t_6	1450 - 450 = 1000	100

Figure 32: Calculating the round-trip time.

Finally, the simple question of what to do if packets arrive out-of-order can be answered. We now have a knowledge indicating how long it should take for the packet to reach its destination under normal circumstances. If packets do not arrive timely, we may now be able to wait an amount of time corresponding to a factor of the round-trip time between the clients before requesting a resend of data. The observant reader will now ask: 'what if the ping, or the pong packet is lost?'. The simple answer to this question is to let the data receiver start out by sending a range of ping packets. This will serve not only the purpose of assuring that eventually one of the ping packets will be responded but also as a remedy to calculate a more precise round-trip time as an average can be calculated from the range of pong packets.

14.5 Error detection

Protocols communicating over a network will eventually encounter situations where an error has occurred. Protocols located at different clients must therefore have a mutual understanding of how errors should be handled. As a result, error detection and handling is a central part of the design of a protocol.

We identify a number of situations which we define as erroneous and how these must be handled.

Data not available: The sender of data may be asked by the server to distribute data from a certain point in the video. In the event that the buffer of the client has overwritten the data which should have been sent, the client cannot send the relevant data. Handling this, can either be done by notifying the server that data was not available or simply doing nothing.

Notifying the server would result in a new selection process which would select a new subset of clients. But if the client suffered from a line failure it would never be able to notify that data was not available. Furthermore, the request to distribute data might never have been received. Therefore, the solution to handling the error is simply by doing nothing. Thus, the receiver would have to notify the server that no data was received. Additionally, the solution handles the situation identically when the sender of data experiences a line failure. Thus, if data reception stops, regardless of whether the receiver has been newly relocated to the spot or whether data distribution has been up and running the client will send the same notification to the server.

Data loss: As data transport using UDP may result in data loss, resending packets must be possible. But due to the time critical nature of multimedia streaming late retransmissions must be avoided. A late retransmission occurs when the data consumer receives retransmitted data at a point in time where the PP has moved to a point beyond the retransmitted data. Hence, the data receiver must avoid requesting data if there is chance that data may be received too late. This can be done using the calculated round-trip time which determines how long time the procedure of dispatching and receiving the fulfilling of a resend request takes. But a resend request may also be received too late. This happens when the buffer of the data sender has overwritten the data requested by the receiver. In this event the data sender must notify the receiver that requested data is no longer available. The receiver may in this event either ignore the lack of data with the problems this may impose on the video player or request data from somewhere else. This may be done by notifying the server that data receival has been brought to a standstill, thus requesting a relocation of the client. This may not be preferable if the needed data may only affect a few data blocks. Therefore, a second solution if offered to the client, which is to send an emergency request of data to the server. The server may fulfill this request if possible or ignore it. Thus, the client cannot rely upon the server to fulfill the request. Hence, the protocol must facilitate this functionality.

Data corruption: If data gets corrupted during transport the data receiver has no means of identifying this. Therefore the receiver simply delivers the corrupted data to the client application. This works in accordance with other implementations of streaming applications, as data corruption is seldomly seen even though UDP is an unreliable form of transport. Furthermore, the amount of corruption which might be seen is normally so small that the impact on the display of the video is bearable.

Data duplication: Data duplication is simply handled by discarding the duplicate data. Thus, the packets must have an identification of the contents which will enable the receiver to decide if the packet has already been received.

Security error: If the data received fails to comply with the security mechanisms used by the protocol, the client must inform the server that the error has occurred. The server may hereafter decide how the error should be handled.

Connection error: In the event of a line breakdown, the rate at which packets are received clearly drops to zero. Furthermore, if the connection suffers from temporary fluctuations the rate may also drop. Handling these must be done by notifying the server of the error and what rate the speed has fallen to.

Buffer overflow/underflow In the event of a buffer underflow or overflow the client will need to request the sender of data to modify the sending speed. Therefore, if the AA of the client is not completely filled with valid data, the client will need to receive data at maximum speed to avoid buffer underflow. Furthermore, the client will need to request the sender of data to lower the speed, if the receive block comes too close to the play block as can happen in the event of a skip.

14.6 Selection mechanism

This section describes the mechanism which is used to find the set of clients selected to send data to a newly connected or relocated client. The mechanism returns the set of clients and the speed used for each of the clients.

As stated in Section 12.3 the construction of the selection mechanism will define the logical appearance of the network. Intuitively, choosing the entry point for a newly connected client will shape the logical topology. Thus, the construction of this mechanism is crucial to the performance of the topology. Selecting the right topology therefore becomes the design of this mechanism. As written in Section 12.3, selecting the right topology of the network involves knowledge about the characteristics of the video among others. Thus, selecting a topology on beforehand might result in poor performance when distributing one kind of video, while another video might obtain good performance. Hence, the selection mechanism becomes an issue which must be solved by the implementation of the protocol.

It should be stressed that the mechanisms can be designed in many ways - hence, designing an optimal solution to these algorithms is not the goal.

15 Underlying protocol usage

This section describes the usage of the underlying transport protocols. Thus, some of the uncertainties of the underlying protocols will be cleared to avoid ambiguities in the protocol design.

As the protocol relies upon TCP the question of how and when the underlying TCP connection should be closed must be answered. Thus, two different events may trigger the down-tearing of the connection, namely:

1. When the client sends a disconnect request to the server, the client must wait until the server closes the connection. This will ensure that the server will not end up maintaining several unused connections due to malfunctioning clients.
2. When the server sends a disconnect request to a client, the server also closes the TCP connection. The server must ensure that all data pending on the connection is purged to ensure that the disconnect eventually reaches the client.

As given in Section 8.4 several routers located on the Internet utilizes NAT to masquerade hosts. Designing a protocol that can handle clients located behind masqueraded routers involve a lot of quirks as other hosts cannot send data to the client if the client has not connected out through the router in advance. For these reasons the protocol cannot handle masqueraded clients. However, the ports used by the client protocol can be selected by the client application which may, in turn, leave the user to select which ports the protocol should use. Thus, the user may select the ports, and configure the router to forward these accordingly.

The UDP header includes a checksum field which can be used by the sender of the packet to specify a checksum for the data contained in the packet. This field could be used in our protocol to allow for the receiver of the packet to check if data contained in the packet has been corrupted during transport. However, using this field is by no means mandatory in the TCP/IP v.4 UDP protocol specification [32]. Therefore the protocol does not rely upon other senders to specify the checksum.

16 Protocol phases

The different phases of the protocol constitute the stages which are used when communicating between either the server and a client or two clients in between. Thus, the phases describe the logical communication flows between the network entities which must be completed in predefined order.

Connection phase The procedure of connecting clients is done when a new client wishes to connect to the network. The client must first issue a connection request to the server and wait for an appropriate answer. This either comes as a connection granted respond, or a connection denied respond. The connection request needs to contain an identification of the video which is requested by the client. This is needed by the server application to determine if a client is to be granted access. Furthermore, the client needs to handle a timeout if the server does not respond to the request. How long this timeout should be is not part of the protocol specification as this would require knowledge about the properties of the physical connection between the server and the client. Furthermore, the server may be busy doing other tasks, which may result in a long response time.

The connection procedure has the responsibility of allocating a key pair as described in Section 12.5.1. This key pair is generated and distributed by the server and is contained in the connection granted packet. Thus, the server only allocates a key pair for the client if the server authorizes the client.

Configuration phase To maintain control of the connected clients the server needs tools to configure all clients. This is needed whenever a change in the topology occurs. When a client has been connected to the network, it must be located at some point resulting in the need for providing some information to the clients which will need to deliver data to the new client. Furthermore, the newly connected client will need to be informed of what data to expect and from which clients.

Streaming phase Once the connect procedure has been successfully completed and the client has been configured the video data transfer will begin. This is done using a data packet sent to the stream receiver from the data sender.

Interaction phase Performing interaction in the video is done by sending a request to either pause, resume or skip in the video. As stated in Section 14.2 pausing the video is done using two different methods. When the player pauses the video, a pause video player packet is sent. Next, when the buffer runs full, the client protocol dispatches a pause stream packet. When the player resumes the video a resume packet, containing the BB which the client targets for the resume, is dispatched from the client.

Skipping between BBs in the video is done using a skip packet also containing the target of the client.

Status phase For the server to maintain a routing table describing the topology of the network all clients must pass their status at a desired interval. This information is needed when the server has to determine the contents of the buffer of a client in the event of a relocation. Evidently, clients may have to send their status at different intervals. If a client is well functioning, and the status of the client does not change much over time, the need for a short interval between status updates disappears. Contrarily a client may need to send updates often if it has proven unstable.

To minimize the amount of data sent through the CCP, requesting the clients for status at this interval is not feasible. Instead the client will send a status packet at an interval given by the server. Still, the server must be able to request a client for status. This request will contain the interval at which the client must send following status updates. As a result, the server will need to monitor all clients to determine if they are providing status at the given interval. If they fail to do this, they will ultimately be dropped by the server protocol. This will furthermore serve to investigate if a client has broken down – if the client does not respond, it will eventually be certified as dead.

Round-trip calculation phase Calculating the network round-trip time between two clients is done using two different packets using the method described in Section 14.4.

For a client to successfully display a video, it must first complete the connection phase. This must be followed by the configuration phase which must be succeeded by the streaming phase. At any given time during the streaming phase the server can force the client into the configuration phase again if necessary. If the client performs an interaction during the streaming phase, the interaction phase will have to be completed, possibly succeeded by the configuration phase. Finally the connection phase will be executed to disconnect the client.

In between these, the status and round-trip calculation phases will be conducted.

17 Packet description

Based upon the purpose of the two protocols it is possible to point out the packet types needed by the protocols. These types are again divided into different domains identified by the type of communication and the direction they handle:

Server-to-client request packets: These packet types are sent from the server to a client indicating that the client needs to take some kind of action.

Server-to-client respond packets: These packet types act as a respond to a request sent earlier on by a client. The server needs to respond with a meaningful answer to the request.

Client-to-server request packet: This packet is sent from a client to the server as a request to perform some kind of action by the server.

Client-to-server respond packets: A client will reply with this type of packet to satisfy a request sent from the server to the client.

The different packet types are partitioned into domains representing the different phases of the protocol as given in Section 16.

17.1 Packet types and flows

The protocol will use a uniform packet header for all packet types. Thus, the header will only contain data which is shared by all packet types containing a payload which constitutes data private to the different packet types. Furthermore, a packet may contain a payload data which can contain a variable length field private to the packet types.

The uniform packet header contains the following fields in the stated order:

Name	Size	Description
VERSION	1 byte	This field contains a one byte version field, indicating the version of the protocol used by the sender. The contents of this field should correspond to the design specification of the relevant protocol version.
OPTIONS	1 byte	This field is reserved for future use – is always set to zero in this version of the protocol.
PACKET_TYPE	1 byte	This field contains the type of the packet. Thus, the field groups the packets enabling the receiver to quickly determine if the packet should be accepted or discarded.
ACTION_TYPE	1 byte	Like the PACKET_TYPE this field is one byte and contains the action type of the packet. By looking at this and the previous field the receiver can determine whether the packet is usable.
PAYLOAD_LENGTH	4 byte	This field contains the length of the trailing payload. The size of the header is not included in the field.
PAYLOAD_DATA_LENGTH	4 byte	This field contains the length of the trailing payload data. The size of the header and payload is not included in the field.
RESV_ID	8 byte	This field contains the unique ID of the packet receiver as described in Section 12.5.1.
SND_ID	8 byte	Contains the unique ID of the sender which is matched by the receiver to validate if the packet is sent from a legal host.

This brings the total size of the header to 28 bytes meaning that the total size of a packet is the value of the PAYLOAD_LENGTH field plus the value of the PAYLOAD_DATA_LENGTH field plus 28 bytes.

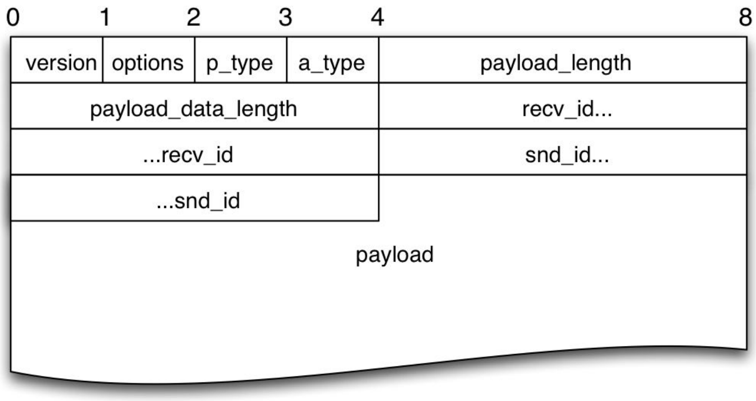


Figure 33: Protocol header.

The following sections will describe each of the packets used in both the DCP and CCP protocols. All descriptions will contain the following elements:

Protocol type: This part contains the description of which protocol is used to transport the packet. This may either be CCP or DCP.

Short name: This part contain the short-name of the packet.

Packet type, action type: This part gives the packet type and the action type of the packet. Both values are present in the header. The packet type value refers to the type of the packet, e.g. server request, client respond etc. whereas the action type refers to the action which should be triggered by the packet. An easy overview of the packet and action types is given in Appendix A.

Usage: This describes the purpose of the packet.

Payload: This part describes the payload of the packet in a table.

Payload data: This part describes the payload data of the packet.

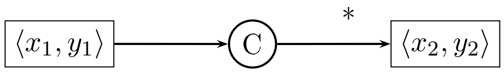
Logic: This part describes what happens once the packet is received and how the logic embedded in the protocol must react. This part further describes the flow of the packet which defines which packets this should result in being sent from the receiver. This is described in words and by a diagram. A specification of the notation used in the diagrams is given in Section 17.1.1.

17.1.1 Diagram notation

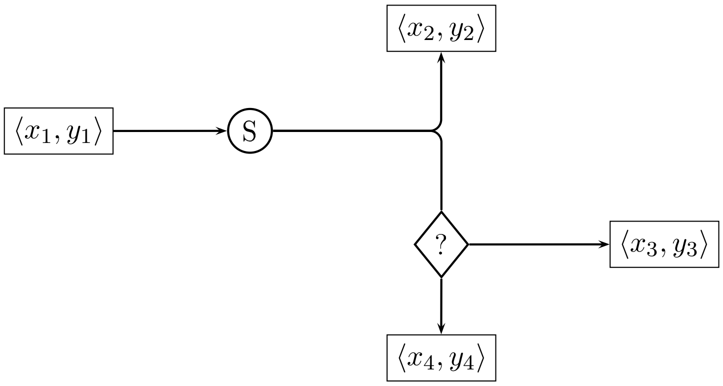
This section contains a description of the notation used by the protocols packet descriptions. These sections account for the different packet types and contain a number of diagrams illustrating the specification of the packet flow which should be initiated by the server and client upon receiving a specific packet.

All of the diagrams follow the same structure, illustrating the reception of a packet and how this should be responded with a range of different packets, if any. The receiver can be either a client or the server denoted by an upper case S or C. In the case of DCP packets the notation SR or SS is used, which is short hand for stream sender and stream receiver.

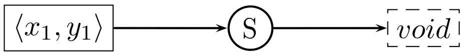
The diagram below illustrates a scenario where the server receives a packet requesting some action is performed. Subsequently the client sends a packet to the server responding to the incoming request. The diagram illustrates that the packet received by the server has packet type x_1 and action type y_1. The packet is received by a client (upper case C) and is responded with a packet of packet type x_2 and action type y_2. The star denotes that zero or more packets of this type can be sent. If no star is present, exactly one packet must be send.



The next scenario shows the server receiving a packet of packet type x_1 and action type y_1. The server responds with two packets, a packet of packet type x_2 and action type y_2. Aside from this, the server must send either a packet of packet type x_3 or x_4. The question mark indicates that based on different circumstances, only one of the two branches will be followed.



Finally, a diagram can illustrate that the reception of a packet does not result in any further data flow.



17.2 Connection

The packets given in this section correspond to the connection phase, and include a disconnect packet, which is sent from the client upon disconnect.

17.2.1 Packet: Request connection

Protocol type: CCP

Short name: CONN_REQ

Packet type, action type: $\langle 40{,}10 \rangle$

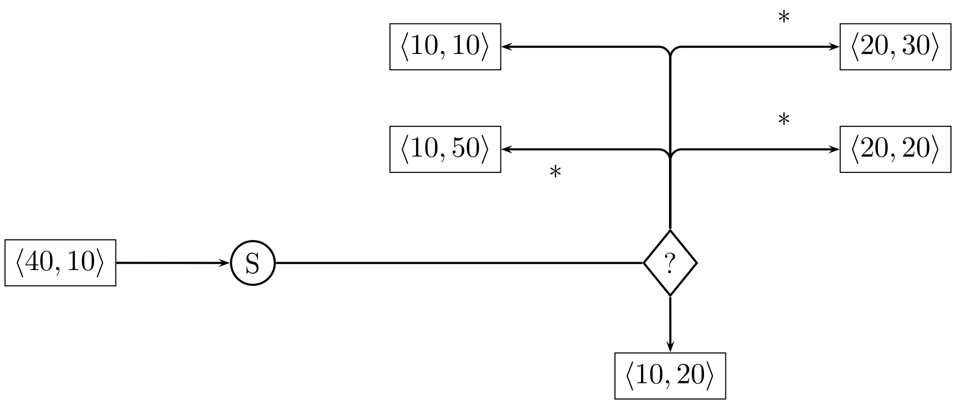
Usage: This packet is dispatched from a client to the server with the purpose of connecting the client to the server. The packet notifies the server that the sender wishes to initiate receiving a specific video given in the payload.

Payload: A total of 140 bytes:

Name	Size	Description
CCP_PORT	2 bytes	This field indicates the TCP port opened by the client to be used for the CCP.
DCP_PORT	2 bytes	This field indicates the UDP port opened by the client to be used for the DCP.
VIDEO_ID	8 bytes	This field indicates the video which the user is requesting access to.
USERNAME	64 bytes	A field indicating the user name provided by the client to be authorized by.
PASSWORD	64 bytes	A field indicating the password provided by the client to be authorized by.

Payload data: The payload data field of this packet is empty.

Logic: Once this packet is received, the server must determine if the client can be authorized. Furthermore, the server must decide whether the client can be given access to watch the requested video. The actual process of authorizing a client is not done by the protocol but is passed to the server application which in turn decides whether the client can be granted access. Passing the result of this to the client must be done by sending either a connection granted packet $\langle 10{,}10 \rangle$ or a connection denied packet $\langle 10{,}20 \rangle$. Following this, the server must find the set of clients that will be deployed to stream data to the client. Selecting these clients will be separated in an encapsulated mechanism as given in Section 14.6. This mechanism should find and return the clients which should provide data. This information must both be sent to the client initiating video play-back and to the clients dispatching the data stream.



17.2.2 Packet: Connection granted

Protocol type: CCP

Short name: CONN_GRANTED

Packet type, action type: ⟨10,10⟩

Usage: This packet is used to respond to a connection request from a client. The server determines whether the client is allowed connection and responds with this packet if connection is granted. The packet furthermore contains information specific to the video requested by the client.

Payload: The payload of the packet contains a total of 310 bytes:

Name	Size	Description
CLIENT_ID	8 bytes	This 8-byte field contains the unique client ID generated by the server. The client ID must be stored and will serve as the identification of the client until disconnected.
SERVER_ID	8 bytes	This 8-byte field contains the unique server ID corresponding to the id of the client. This server ID must be stored and will serve as the identification of the server to the client until it is disconnected.
VIDEO_SIZE	8 bytes	The total size of the video in bytes.
VIDEO_LEN	4 bytes	The duration of the video in milliseconds.
BB_TOTAL	2 bytes	Number of BBs in the video.
DB_TOTAL	2 bytes	Number of DBs contained in each BB.
DB_SIZE	2 bytes	Size of DBs in the video in bytes.
VIDEO_HEADER_SIZE	4 bytes	Minimum number of required DBs of the video before playback can start initially or after a skip has occurred.
RECOM_CA_SIZE	2 bytes	Number of recommended BBs in the CA, as given in Section 14.2. If this number is not satisfied, the client may be tagged as anti-social.
SPEED_LOW	2 bytes	This field indicates the rate data should be sent with, when throttling down the speed. The number is measured in milliseconds denoting the maximum interval at which a data block can be sent. The higher this number is, the lower the minimum speed is, since the number illustrates how long time should pass between each packet transmission.
SPEED_NORMAL	2 bytes	Denotes the speed data blocks must be sent with at normal speed.
SPEED_HIGH	2 bytes	Denotes the speed data blocks must be sent with at high speed.
SEC_TYPE	2 bytes	Determines the type of security used.
SEC_SIZE	2 bytes	Determines the size of a security data per data block.
SKIP_DISTANCE	4 bytes	The amount of milliseconds in a skip. Equals the duration of a BB.
VIDEO_TITLE	256 bytes	The title of the video.

Payload data: The payload data field of this packet is empty.

Logic: When a client receives a connection granted packet it may regard itself as logged on successfully to the server. The client must store the contents of the payload in an appropriate way. When the client starts receiving video data the protocol must not provide the client application with data until the number of data blocks given in

VIDEO_HEADER_SIZE is available. This restriction is enforced since many multimedia formats demand that the start of the video is available as it contains vital information about the format which must be present before play-back can be initiated. The contents of RECOM_CA_SIZE should be taken into consideration when determining the size of the buffer. If this recommended size is not available, the client will be tagged as anti-social and therefore has a higher risk of being disconnected. The pair of identification keys which are contained in the payload must be available whenever messages are exchanged with the server.

$$\boxed{\langle 10, 10\rangle} \longrightarrow \text{C} \longrightarrow \textit{void}$$

17.2.3 Packet: Connection denied

Protocol type: CCP

Short name: CONN_DENIED

Packet type, action type: ⟨10,20⟩

Usage: This packet is used as a counterpart to the connection granted packet ⟨10,10⟩. The packet indicates that the client could not be connected by the server.

Payload: The payload contains a single field.

Name	Size	Description
DESCRIPTION	256 bytes	Description with an indication of why the client was not allowed to connect.

Payload data: The payload data field of this packet is empty.

Logic: The server must respond with the connection denied if a client is to be denied access. The server may include a textual response indicating why this has happened. The only option the client has in this situation is to retry connect, which will be regarded as another session.

$$\boxed{\langle 10, 20\rangle} \longrightarrow \text{C} \longrightarrow \textit{void}$$

17.2.4 Packet: Connection closed

Protocol type: CCP

Short name: CONN_CLOSED

Packet type, action type: ⟨10,30⟩

Usage: This packet is sent from the server to a client, if the server decides to disconnect the client. The client is not offered any possibility to respond, but is immediately disconnected from the network.

Payload: The payload contains a single field.

Name	Size	Description
DESCRIPTION	256 bytes	Description with an indication of why the client has been disconnected.

Payload data: The payload data field of this packet is empty.

Logic: Once this packet is received from the server, the client has definitively been disconnected from the network. The client should update its own state to disconnected. The packet can be sent at any given time and is therefore not necessarily a respond to a request packet. Any respond from the client must be ignored by the server as the client is no longer considered connected. The only option for the client is to reconnect, but this will be handled as a new session.

$$\boxed{\langle 10, 30\rangle} \longrightarrow \text{C} \longrightarrow \mathit{void}$$

17.2.5 Packet: Disconnect

Protocol type: CCP

Short name: DISCONN

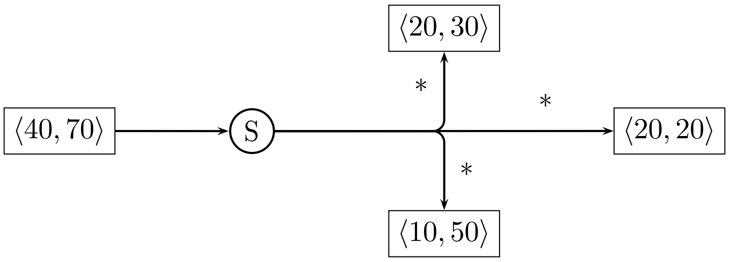
Packet type, action type: ⟨40,70⟩

Usage: This packet is used when the client wishes to stop viewing the movie. Once a stop has been issued the client gets disconnected from the network and hence if it wants to start playing the movie again, it has to login again.

Payload: The payload of this packet is empty.

Payload data: The payload data field of this packet is empty.

Logic: Upon reception of this packet the server needs to relocate all clients receiving data from the sender of the packet. Furthermore, all clients sending data to the client must be informed to stop sending data. The server must hereafter regard the client as disconnected and purge all information concerning the client.



17.3 Configuration

The configuration phase configures the client in regard to where it will receive data from and to whom it must send data to. Some of the packet types used by the configuration phase will in their payload data transport a number of elements called STREAM_INFO. These elements are used as information sent from the server to notify clients about the properties of a data stream. Thus, a client will be notified by the server using a STREAM_INFO entity that the client should send data to another client. The STREAM_INFO contains the nature of that data stream, such as which DBs. The client which will receive the data stream will be sent the same information using the STREAM_INFO entity. Clearly, the length of this entity is variable as some clients will send/receive more DBs than others. This leads to the following fields:

Name	Size	Description
CLIENT_ID	8 bytes	The client ID field is used differently, depending on the type of packet the STREAM_INFO is embedded in. If the packet is used to inform a client that it must send a data stream to another client, the client ID identifies the receiver of the stream. If the packet is used to inform a client of the nature of the data stream it should receive, the field indicates the sender of the stream.
CLIENT_IP	16 bytes	This field contains the IP-address. As with the CLIENT_ID the field either contains the IP-address from where it receives data or to where it must send data, depending on which packet the STREAM_INFO is used in. The field has room for further extension to enable use of IPv6.
CLIENT_PORT	2 bytes	This field contains the port number to which data must be sent to the receiver. The field is only used, when the STREAM_INFO is used in packet 17.3.2.
START_BB	2 bytes	This field contains the BB starting point in the video from where the data stream should begin.
START_DB	2 bytes	This 2-byte field contains the DB starting point in the video from where the data stream should begin.
DB_NO	2 bytes	This field indicates the number of DBs from each BB, the data stream must contain.
DB_RANGE	x bytes	This variable length field specifies each DB number the stream will contain. Thus, this field is repeated for each of the number of DBs given in the field DB_NO. This collection of fields takes up 2 bytes for each of the specified DBs. The DBs must be transmitted in the order listed in the entity, starting with the first DB given in the START_DB field.

Visually, a STREAM_INFO is presented in Figure 34.

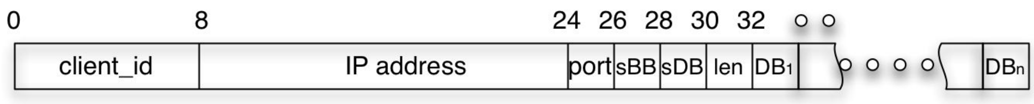


Figure 34: The STREAM_INFO entity.

17.3.1 Packet: Receive data stream

Protocol type: CCP

Short name: RECV_DATA_STREAM

Packet type, action type: ⟨10,50⟩

Usage: This packet contains information about the stream of data a client is to receive. Whether this stream will come from the server, a client, or a number of clients in conjunction with each other will be indicated by this packet. Thus, this packet informs a client about what data it can expect and from whom it can expect it. This packet is counterpart to the distribute data packet in the sense that this is sent to the client providing the data stream.

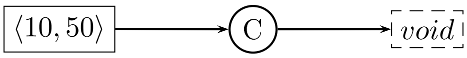
Payload: The length of the payload of this packet is variable, as the number of clients streaming data is variable.

Name	Size	Description
CLIENT_NO	2 bytes	Indicating the length of trailing data containing information about each of the clients which will send data.
CLIENT_RANGE	x bytes	This variable length field contains a STREAM_INFO entity for each of the clients given in the field CLIENT_NO in the payload.

Payload data: The payload data of this packet is empty.

Logic: When a client receives a RECV_DATA_STREAM, it must prepare itself to receive data from the clients indicated in the packet. The received client IDs should be stored in a proper way with that in mind that the IDs must be accessed every time a packet is received in order to validate the sender. The client does not respond to this kind of packet.

The sending speed of each sending client is implicitly received in this packet as the number of DBs received from a client indicates together with the full length of the video and the total number of BBs, the rate at which DBs should be received. Upon reception, a mechanism monitoring the speed at which it receives data from the clients needs to be set up.



17.3.2 Packet: Distribute data

Protocol type: CCP

Short name: SND_DATA_STREAM

Packet type, action type: ⟨20,20⟩

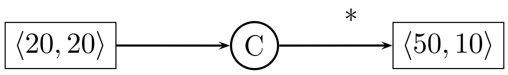
Usage: This packet is sent to a client in order for the client to initiate the video stream to another client.

Payload: The payload contains information concerning what to stream, with what speed and to whom. More formally this is:

Name	Size	Description
STREAM_INFO	x bytes	This field contains a STREAM_INFO entity indicating what and to whom data must be sent.

Payload data: The payload data of this packet is empty.

Logic: Upon receiving a distribute data packet the client must immediately begin distributing data to the client at the speed specified in the payload. This is done using the data packet transported by the DCP.



17.3.3 Packet: Stop data distribution

Protocol type: CCP

Short name: STOP_STREAM

Packet type, action type: ⟨20,30⟩

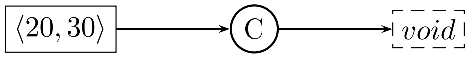
Usage: This packet is sent from the server to a client with information signifying that the client must stop streaming data to a specific client. Information about why the stream should be stopped has no relevance and is hence not given in the packet.

Payload: The payload contains a total of 12 bytes.

Name	Size	Description
CLIENT_ID	8 bytes	The field specifies the client ID of which the receiver must stop streaming data to.
STOP_POINT_BB	2 bytes	This field indicates the stop point in form of a BB. The client must stop sending data once it reaches this point. The field may be empty which entails that the client must stop streaming at once.
STOP_POINT_DB	2 bytes	A 2 byte field indicating the stop point in form of a DB, the field is used together with the field above.

Payload data: The payload data of this packet is empty.

Logic: Once this packet is received by a client it must stop streaming data to the specified client. If no stop point is defined, the client must stop streaming data immediately. Otherwise, the client must send data until the given stop point is reached. The reception of this packet does not trigger any respond, but it may stop transmission of data packets.



17.4 Streaming

17.4.1 Packet: Data

Protocol type: DCP

Short name: DATA

Packet type, action type: ⟨50,10⟩

Usage: This packet is used to send a DB to a client.

Payload: The payload of this packet contains a total of 5 bytes:

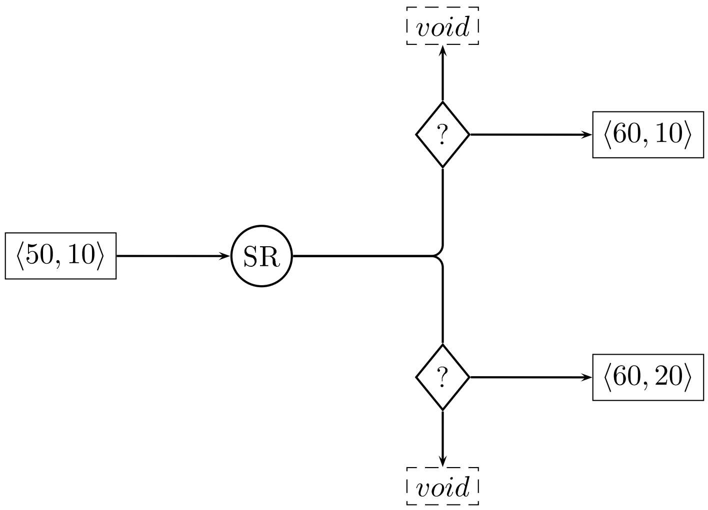
Name	Size	Description
BLOCK_BB	2 byte	The field indicates the BB from which the current DB is contained.
BLOCK_DB	2 byte	The field indicates the DB of the BB given in the field BLOCK_BB contained in the packet.
SPEED_LEVEL	1 byte	The 1 byte field indicates the speed at which the data stream is sent from the client. The value 10 denotes low, 20 denotes normal and 30 denotes high.

Payload data: The payload data of this packet contains the video data embedded in the packet:

Name	Size	Description
DATA	x byte	This field denotes the data contained in the fragment of the video given in the fields BLOCK_BB and BLOCK_DB in the payload.

Logic: Upon receival of this packet, the receiver must check if it needs the data contained in the packet. If the packet is received too late, e.g. the play pointer has moved beyond the point in the video where the data belongs it should be discarded. Likewise if the client is already in possession of the data or it does not need it, the data should be discarded. Otherwise the data should be stored in the buffer.

Upon receival, the receiver may send a packet indicating that data was received out-of-order. Thus, the sender assumes that a packet was lost underway and dispatches a request for resending the lost packet.



17.4.2 Packet: Data not available

Protocol type: DCP

Short name: NO_DATA

Packet type, action type: ⟨50,20⟩

Usage: This packet is sent by the stream sender to inform the stream receiver that the stream sender does not possess a certain DB requested by the stream receiver.

Payload: A total of 8 bytes.

Name	Size	Description
REQ_BLOCK_BB	2 bytes	The field indicates the BB which cannot be found at the sender.
REQ_BLOCK_DB	2 bytes	The field indicates the DB which cannot be found at the sender.
NEXT_BLOCK_BB	2 bytes	This field indicates the next BB available following the block given in the field REQ_BLOCK_BB.
NEXT_BLOCK_DB	2 bytes	This field indicates the next BB available following the block given in the field REQ_BLOCK_DB.

Payload data: The payload data of this packet is empty.

Logic: If a client receives this packet it must decide if data contained between the requested block and the next block can be dispensed with. If this is not the case, data must either be retrieved from the server by requesting emergency resending of the data by sending a request backup data packet.

$$\boxed{\langle 50, 20\rangle} \longrightarrow \text{SR} \longrightarrow \boxed{\langle 40, 100\rangle}$$

17.4.3 Packet: End of data

Protocol type: DCP

Short name: NO_MORE_DATA

Packet type, action type: ⟨50,30⟩

Usage: This packet is used by the stream sender to inform the stream receiver that it does not have any more of the data which it have been asked to send.

Payload: The payload of this packet is empty.

Payload data: The payload data of this packet is empty.

Logic: When this packet is received the client knows that it no longer can receive the data that it needs from the data sender. The receiver hereafter informs the server by sending an error packet indicating that data receival has been brough to a standstill.

$$\boxed{\langle 50, 30\rangle} \longrightarrow \text{SR} \longrightarrow \boxed{\langle 40, 80\rangle}$$

17.4.4 Packet: Request data resend

Protocol type: DCP

Short name: RESEND_DATA

Packet type, action type: ⟨60,10⟩

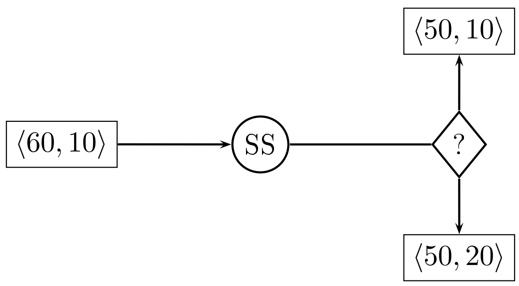
Usage: This packet is used by the receiver of video data to ask the sender to resend a packet which was lost during transport.

Payload: A total of 4 bytes.

Name	Size	Description
BLOCK_BB	2 byte	This field indicates the BB of the block being requested for retransmission.
BLOCK_DB	2 byte	This field indicates the DB of the block being requested for retransmission.

Payload data: The payload data of this packet is empty.

Logic: Upon reception of this packet the client must send the contents of the data block contained in the packet. The client should send this packet as the next in line, before resuming the data stream. Furthermore, the sending speed should be increased to maximum rate to ensure that the buffer of the receiver is not starved.



17.4.5 Packet: Adjust speed

Protocol type: DCP

Short name: ADJUST_SPEED

Packet type, action type: ⟨60,20⟩

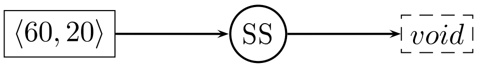
Usage: This packet is used by the receiver of video data to request the sender to adjust the stream according to the request.

Payload: A total of 1 byte.

Name	Size	Description
SPEED_LEVEL	1 byte	The field contains the value of 10, 20 or 30 which indicates 10 for low, 20 for normal and 30 for high speed.

Payload data: The payload data of this packet is empty.

Logic: Upon reception of this packet the sending speed of consecutive data packets should be increased/decreased to the value contained in the SPEED_LEVEL field.



17.4.6 Packet: Error receiving data

Protocol type: CCP

Short name: STREAM_ERROR

Packet type, action type: $\langle$40,80$\rangle$

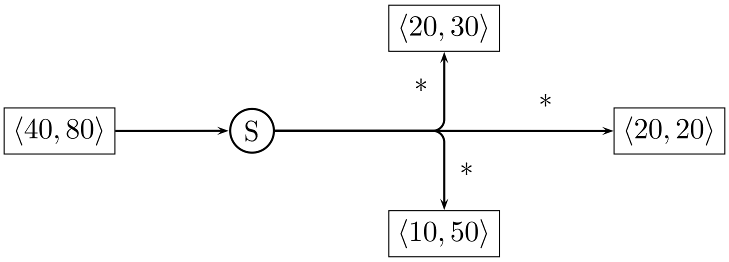
Usage: This packet is used to notify the server of a speed failure when data is received at a different speed than what is advertised. Thus, if a client is not receiving data from another client at the advertised speed, the client must dispatch this packet to the server.

Payload: A total of 12 bytes:

Name	Size	Description
CLIENT_ID	8 bytes	This field contains the client ID of the failing client.
RATE	2 bytes	A rate calculated by the client indicating the speed of which data is being received from the failing client. This is calculated as the average speed between receival of each data block. 0 indicates that no data is being received as the sender of data could not satisfy the request. Values above are treated as the number of milliseconds between consecutive data receival.
BLOCK_NO	2 bytes	This field contains information about the BB in the video where the error has occurred.

Payload data: The payload data of this packet is empty.

Logic: Upon reception of this packet the server needs to determine whether the client sending the packet or the client reported in the packet is causing the error. Accordingly the server may choose to relocate or drop any or all of the affected clients.



17.4.7 Packet: Request backup data

Protocol type: CCP

Short name: REQ_BACKUP_DATA

Packet type, action type: ⟨40,100⟩

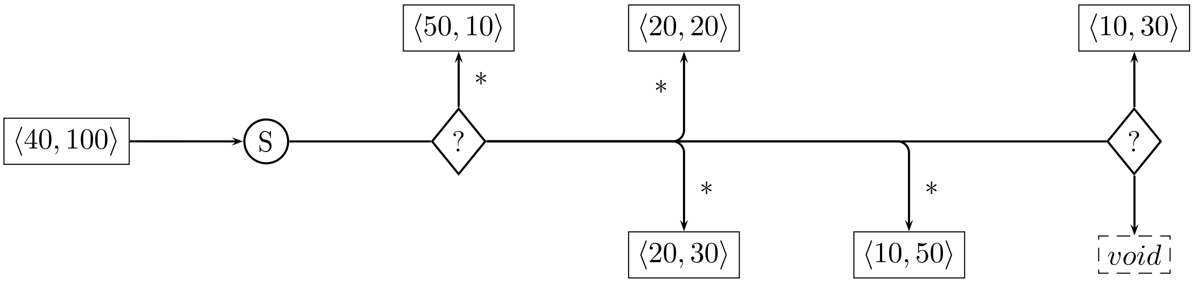
Usage: This packet is used by a client to inform the server that it needs a specific part of the video stream.

Payload: The number of bytes contained in the payload of this packet is variable:

Name	Size	Description
BLOCK_BB	2 bytes	This field indicates the BB from which the client is requesting data.
NUMBER_DB	2 bytes	This field contains the number of DBs requested from the given BB.
DBS	$n * 2$ bytes	The field contains the DBs which the client wishes to receive.

Payload data: The payload data of this packet is empty.

Logic: Upon reception it is purely up to the server how the request is handled. Firstly, the server may ignore the request. This may not be a good solution but it can be necessary if the server is very busy. Secondly, the server may send one or more of the requested DBs to the client. A third solution is to find a new client which should send data to the client. This option could be used if a client continues to ask for data which should have been received from another client. If the client keeps asking for data, the server may in the end disconnect the client.



17.5 Interaction

17.5.1 Packet: Pause video player

Protocol type: CCP

Short name: INTERACT_PAUSE

Packet type, action type: ⟨40,30⟩

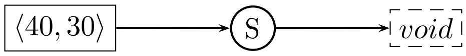
Usage: This packet is sent when the client wishes to pause the video. Data distribution from/to other clients continues except no data is returned to the client application. Once the buffer is full, the client notifies the server that packets will now be dropped and sends a pause stream packet.

Payload: The payload contains 1 field.

Name	Size	Description
PP	2 bytes	Indicating the current play pointer of the client.

Payload data: The payload data of this packet is empty.

Logic: Upon receiving this packet, the server must flag the client as paused. Furthermore, the server stops connecting more clients to the pausing client. Thus, the clients connected to the pausing client will continue to receive data until the client's buffer runs full. Sending this packet must only be done if the server regards the client as in running state. Otherwise the packet will be discarded by the server.



17.5.2 Packet: Pause stream

Protocol type: CCP

Short name: INTERACT_PAUSE_STREAM

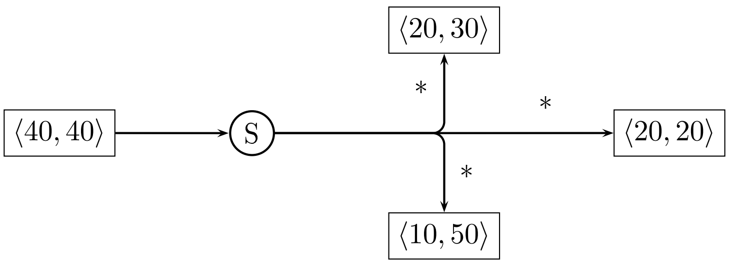
Packet type, action type: ⟨40,40⟩

Usage: This packet indicates to the server that the client has run out of buffer space while collecting data in paused state. If this occurs, the client must send this packet to stop other clients from sending data which will otherwise be discarded.

Payload: The payload of this packet is empty.

Payload data: The payload data of this packet is empty.

Logic: Upon reception of this packet the server needs to relocate all clients receiving data from the sender of the packet. Furthermore, all clients sending data to the client must be informed to stop sending data. Sending this packet must not be done unless the client has previously sent a pause video player packet to set the client in paused state. If this has not been done, the server discards the packet as invalid. This packet then informs the server that the client has been paused long enough to fill its buffer and all clients should stop sending data to the client. Apart from this, the client may soon run out of data which should be sent to other clients, and these clients (if any) may be relocated.



17.5.3 Packet: Resume stream

Protocol type: CCP

Short name: INTERACT_RESUME

Packet type, action type: ⟨40,50⟩

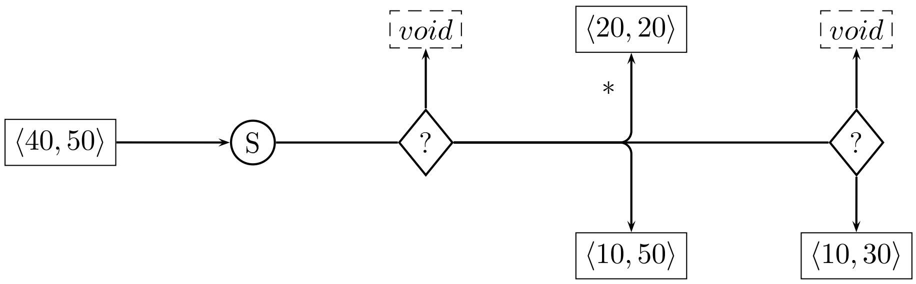
Usage: This packet identifies that a client wishes to resume the video after a pause has been carried out.

Payload: A total of 2 bytes:

Name	Size	Description
TARGET_BLOCK	2 bytes	Indicating from which BB the client wishes to receive data.

Payload data: The payload data of this packet is empty.

Logic: Upon receiving this packet, the server must determine whether the client needs to be relocated. This will only be necessary if the buffer of the client has run full and a following INTERACT_PAUSE_STREAM packet has been transmitted. In this event, the client will have stopped receiving data and needs to be relocated to a new spot in the network from where it can receive data. In this event the client will provide the next block which needs to be filled in the payload of the packet to identify what data the client needs to receive. If no INTERACT_PAUSE_STREAM has been transmitted this field must be empty as the client still receives data and has no need for relocation. Thus, the server only needs to update its internal representation of the client and mark it as playing.



17.5.4 Packet: Skip

Protocol type: CCP

Short name: INTERACT_SKIP

Packet type, action type: ⟨40,60⟩

Usage: This packet is used when the client performs a skip.

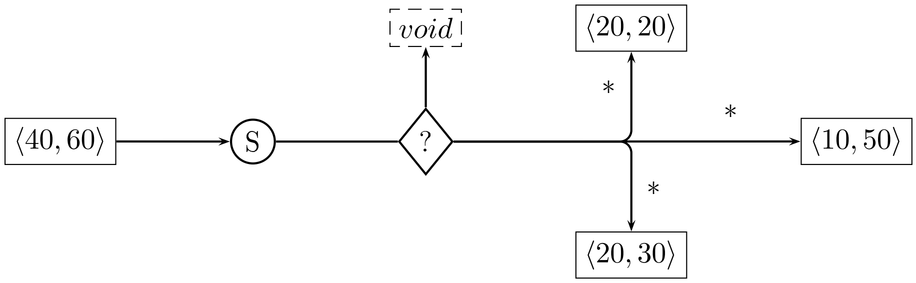
Payload: A total of 2 bytes:

Name	Size	Description
TARGET_BLOCK	2 bytes	A field containing the BB to which the client wishes to skip.
SKIP_TYPE	1 byte	This field indicates whether the skip moves the PP outside the buffer. Thus, zero indicates a skip inside the buffer. All other values are treated as a skip outside the buffer.

Payload data: The payload data of this packet is empty.

Logic: As described in Section 14.2 three different kinds of skips can be performed. This can either be a skip forward inside the buffer, a skip backwards inside the buffer, or a skip outside the buffer.

If an INTERACT_SKIP packet is received from a client that is not in playing state, the packet must be discarded. Sending this packet will result in the server either doing nothing or forcing the client to be relocated.



17.6 Status

17.6.1 Packet: Request status

Protocol type: CCP

Short name: STATUS_REQ

Packet type, action type: ⟨20,10⟩

Usage: This packet is sent from the server to request a client of its current status. The packet is furthermore used as an 'i am alive' request to which the client must respond.

Payload: The payload contains a single field:

Name	Size	Description
STATUS_RESP_INTERVAL	4 bytes	Indicating the interval at which the client must transmit the STATUS_RESP packet. The field denotes this interval in milliseconds.

Payload data: The payload data of this packet is empty.

Logic: A status request packet can be sent from the server at any time. It is completely up to the server to probe the client for its status. Upon receival of this packet, the client must update its relevant timers to reflect the interval given in the packet. If a status response packet is not transmitted according to this interval, the client may eventually be disconnected by the server.

There is no limit in terms of how many or with what interval this packet can be received. Neither are there any specific number of attempts which the server should ask the client for an answer before it is disconnected. Because of this it is very important for the client to respond rapidly in order to ensure its connection.

$$\boxed{\langle 20, 10\rangle} \longrightarrow \text{C} \longrightarrow \boxed{\langle 30, 10\rangle}$$

17.6.2 Packet: Status respond

Protocol type: CCP

Short name: STATUS_RESP

Packet type, action type: ⟨30,10⟩

Usage: This packet is used to notify the server of the present status of a client. Thus, it is both used as a respond to a status request packet, dispatched by the server, but also as a packet sent from the client at any time.

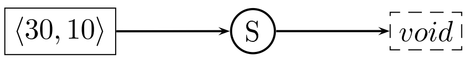
Payload: The payload of this packet contains a number of fields of a variable length:

Name	Size	Description
PLAY_BLOCK	2 bytes	This field contains the BB holding the PP of the client at the time of transmission.
CA_SIZE	2 bytes	A field containing the size of the client's present Connect Area measured in BBs as described in Section 14.2.
AA_SIZE	2 bytes	A field containing the size of the client's present Absorption Area measured in BBs as described in Section 14.2.
RA_SIZE	2 bytes	A field containing the size of the client's present Reserved Area measured in BBs as described in Section 14.2.
RECV_BLOCK_BB	2 bytes	This field contains the last received BB which lies farthest off the play block.
RECV_BLOCK_DB	2 bytes	This field contains the last received DB which lies farthest off the play block.
BUFFER_CONTENT	x bytes	This variable length field contains the buffer numbers contained in the buffer of the clients starting at the play block. Each BB is represented by a 2-byte field indicating the video BB number contained in the field. Thus, the total size of the field is 2 times the total buffer size in BBs.

Payload data: The payload data of this packet is empty.

Logic: Once the server receives this packet it must store these data. The contents of the packet enables the client to alter the size and distribution of its buffer during play-back if necessary. In this initial version of the protocol the server will however assume that the client does not alter this.

Upon receival of a status response packet the server must update its internal data-structures to represent the current state of the client. The server can calculate the total buffer size of the client by adding the fields CA_SIZE, AA_SIZE and RA_SIZE. From these values the server must calculate the portion of the client's CA which may be used for attaching clients to, as described in Section 14.2. This will be calculated to the lowest and the highest BB number contained in the buffer which are valid for connection.



17.7 Round-trip time calculation

17.7.1 Packet: Ping

Protocol type: DCP

Short name: PING

Packet type, action type: ⟨60,30⟩

Usage: This packet is used to initiate the round-trip time calculation procedure.

Payload: The payload of this packet contains 8 byte.

Name	Size	Description
SEND_TIME	8 bytes	Indicates the time at which the packet has been sent. This number must be given in milliseconds since some predefined starting point. Whether this is given in *unix time* or the beginning of the session has no relevance. However, it is important that this same starting point is used whenever packets are timestamped.

Payload data: The payload data of this packet is empty.

Logic: Upon receival of this packet the receiver must respond with a pong packet.

$$\boxed{\langle 60, 30\rangle} \longrightarrow \text{SS} \longrightarrow \boxed{\langle 50, 40\rangle}$$

17.7.2 Packet: Pong

Protocol type: DCP

Short name: PONG

Packet type, action type: ⟨50,40⟩

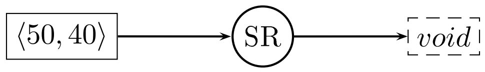
Usage: This packet is used as respond in the round-trip time calculation procedure.

Payload: A total of 16 bytes:

Name	Size	Description
SEND_TIME	8 bytes	This field contains the time which was given in the ping packet for which the pong packet is a respond to.
RECEIVE_TIME	8 bytes	This field contains a timestamp given in milliseconds indicating the time of receival of the corresponding ping packet. The starting point of this timestamp does not need to be in concordance with the starting point used by the sender of the ping packet which the pong packet at hand is a respond to. However, care should be taken to ensure that the same starting point is used to timestamp all consecutive data packets.

Payload data: The payload data of this packet is empty.

Logic: Upon receival of this packet, the round-trip time between the two clients should be calculated. Hereafter a round trip time should be maintained using the method described in Section 14.4.



17.8 Security

17.8.1 Packet: Security data

Protocol type: CCP

Short name: SEC_DATA

Packet type, action type: ⟨10,60⟩

Usage: This packet is used by the server to send security data to a client.

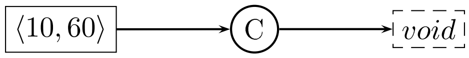
Payload: The payload contains one field:

Name	Size	Description
BB	2 bytes	A number indicating which BB the contents of the packet is related to.

Payload data: The payload data contains the data related to the security mechanisms used by the server and client applications.

Name	Size	Description
DATA	x bytes	Contains the security data.

Logic: Upon receival of this packet the client must store the info given in the packet. The data contained in the payload data must be stored until the client application requests these.



18 Timers

An analysis of the protocol reveals that several situations may occur where the use of timers are important. These timers are used to enable a form of synchronization the server and clients in between.

- The functionality of the server clearly relies upon the ability of the client to send status updates at a regular interval. As already described in Section 16 the server decides an interval at which the clients must send a new status update. Whether this interval is maintained needs to be monitored by the server. This may be done using a timer indicating that when the timer runs out, a status update should have been received from some client.

- As the time critical nature of video streaming requires data to be received at a continuous rate, there is an evident need to monitor whether data has been received at the required speed. This will need to be done using a timer monitoring all incoming DCP connections. This timer will be used in different situations:

  Request speed change: If the client requests the sender to dispatch data at another rate than the one packets are received at, the client must wait for the changes to take place. Thus, the line of packets being delivered must be emptied for the speed change to become effective. The client must therefore utilize the knowledge about the round-trip time between the clients before issuing a new speed change request.

  Request resending of data: Performing request for missing data there are two timing issues to consider. First it should be detected when data is missing and secondly when the request for resend has gone lost. The first value is a parameter which should be adjustable from client to client and hence may be set through the client interface. Secondly the client should utilize knowledge about round-trip time before issuing a new resend request.

- As opposed to the item above, a client must also use a timer to dispatch packets at a correct rate, when streaming video data to other clients.

19 Interface

To enable applications to make use of the protocol, an interface must be provided. This interface must carry commands dispatched from the applications and be accordingly handled by the protocol.

The server interface should provide the following mechanisms:

- Provide information regarding videos to the protocol.
- Authorize new connecting clients and provide the bandwidth available to the client.
- The server protocol must be able to request video data of the server application which, in turn, must provide this to the protocol.
- The server protocol must be able to request security data of the server application which, in turn must provide this to the protocol.
- The server application must be able to request status from the server protocol.

Client-side, the interface must provide the following mechanisms:

- The client protocol must provide the client application with video data when available.
- The client protocol must handle video interaction requests from the client application.
- Security data must be provided to the client application possibly in the same manner as in the case of video data.

The interface is depicted in Figure 35.

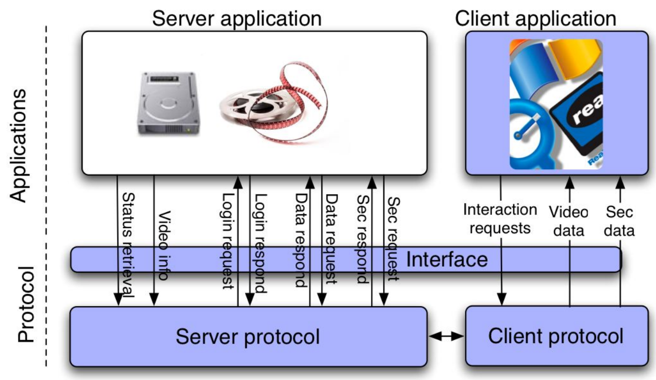


Figure 35: Interface between protocol and applications.

Chapter IV

Protocol implementation

This chapter contains a description of the implementation of the protocol. This will be used when implementing the protocol. Thus, the implementation sketches an implementation of the main functionality of the protocol.

20 Fundamentals

Evidently, the protocol needs to be implemented as a multithreaded library. The protocol could be implemented as a single passive library using only the thread of the application instantiating the protocol. This would however, lead to many inconveniences, as this one thread would need to carry out many separate tasks. As stated in Section 9.2 the number and location of these threads is the main issue when implementing multithreaded applications.

To simplify the implementation, the solution is divided into a number of main components, each providing an interface accessible by the other. Some components need to interact with each other and hence when created a reference to the other components which it must know about must be passed to it. The threads will of course act as the driving force among these components and they will be implemented with a minimum amount of code to obtain centralizing all logic in the components. This way a component can be easily changed as long as it retains the interface.

To simplify the construction of these components, C++ used as programming language with an objective approach is chosen. This may come with a small price regarding performance but enables us to structure the implementation better dividing the different tasks of the implementation process. Communication between the protocol instances running on different hosts will use the socket API.

The protocol design can in some cases be found ambiguous as it does not always define how the implementation of the server side protocol should act. Thus, the protocol design does not specify if the server should remain passive until an error occurs, or anticipate events and act before the error occurs. The implementation could monitor all clients and from statistical material act when some action occurs, as this normally would trigger an error later on. To keep the implementation simple, we choose only to act only when an error or event occurs.

20.1 Main components

As both a server-side protocol and a client-side protocol will be implemented, identifying the main components must be done with care to enable reuse of the components on both sides. Thus, tasks which are shared by both sides of the protocol can benefit from this.

Packets: To facilitate communication between the components, a standard for exchanging data must be defined. As the interfaces of the components may be complex, due to the large amount of data which may flow between the components, this standard must simplify the interfaces. As data is sent and received as packet entities the nodes in between, data flowing from the network layer is represented in a packet. Thus, transporting packets through the interfaces of the components is simple. The protocol will therefore embed packets in a packet component which can easily be transported between the components of the protocol.

Stream Engine: Evidently, streaming of data, either from a client or from the server will be done similarly. The difference between the server and a client is simply that client can stream only one video to a number of clients while the server can stream a number of different videos to a number of clients. The stream engine will therefore be based upon streaming of one video, encouraging the need for one streaming entity per video dispatched from the server. This does in fact make sense as streaming of different videos can be seen as having well defined boundaries. Hence, a client will use a single instance of the stream engine entity, while the server will use one per video distributed by the server.

Data container: Along with a number of streaming engines, the protocol will require some form of buffering mechanism. This data storage will work differently depending upon the type of protocol instance. This does in fact describe the main difference between the server and client. Thus, a client stores data which has been received from another client, while the server has all data available through the interface, see Section 19, to the server application. As the data container used by the client side, called the buffer, is a central part of the protocol design it has been thoroughly described in Section 14.2. Opposed to this, the server side protocol also needs some form of data storage. This may come as a surprise, as data will always be available to the server. But as the server protocol cannot rely upon caching all data transferred through the interface some form of data cache must be implemented. Contrary to the client-side buffer, the server-side data container, called data cache, has not been touched yet, as the design of the protocol does not state anything concerning this.

As the server will serve many different kinds of videos, the properties of the content stored in the data cache will accordingly be different. Thus, storing data cached for different videos will add to the complexity of the data cache. As already stated, streaming of different videos are different tasks therefore one data cache will be used per stream engine on the server.

Application Task Queue: As the server-side protocol needs to be serviced by the server application accessing data, validating login requests etc. the protocol needs to be able to issue tasks for the application to perform. Since the interaction between the protocol and the application involves a number of threads accessing the same data, a queue of some kind is needed server-side enqueueing these tasks for the application.

Incoming Packet Queue: The protocol needs the ability to receive a high number of packets continuously. This must be done at a high rate to ensure that all data received on the socket is read and nothing is lost due to small operating system buffers etc. Therefore the packets must be read and pushed into a packet queue.

Transport handler: To hide the complexity of the underlying operating system dependent network functionality, a transport handler will be implemented. This transport handler will wrap the socket layer providing only a simple interface to send or receive data through. Thus, the component will be exchangeable, should the implementation be used in an environment not supporting the socket API. The component will be used on the server-side and on the client-side, as sending and receiving data will be handled identically.

Data bank: The protocol needs a data structure to hold various information regarding connected clients, receival speed and video characteristics. For this purpose, the protocol will use a data bank component used both on the server and protocol side. On the server side this data bank will serve to contain the logical appearance of the network and contain all crucial information about each client. On the client side, the data bank will only contain information about clients sending and receiving data to/from the client. As the server can function as a client the server will of course contain information about clients receiving data from the server.

Logic client/server: Both the client and the server side of the protocol need to embed special logic which is focused on either the server or the client. The server logic needs a method to decide which client (or server) should stream data to a new or relocated client. This method will be described separately in Section 22.

Gluing the components together is best described visually. How the server is organized is therefore depicted in Figure 36.

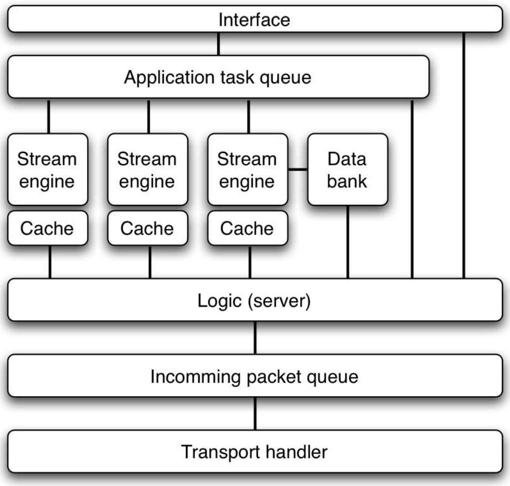


Figure 36: Main components in the server.

Packets are received and sent using the transport handler component, which handles the representation of data as it is understood by the socket layer. Incoming packets are enqueued in the packet queue component and accordingly distributed to the logic which controls the stream engines. The application task queue is filled by the stream engines and the logic, and is emptied by the server application through the interface defined in Section 19.

The client is presented in Figure 37.

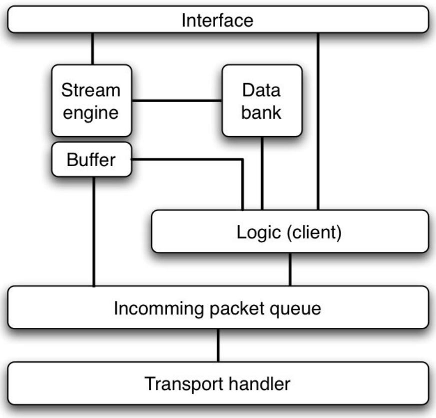


Figure 37: Main components in the client.

The main differences between the server and the client are:

- The number of stream engines differs from the client to the server. Where the server uses multiple (one for each unique video) the client always only uses one.
- The server uses a cache where the client uses a buffer.
- The client does not need to have an application queue, since it does not need to dispatch requests to the application. The client application only needs to receive data and does not need to deliver anything back to the protocol except for user interactions which are handled by the logic.
- The server needs to hold more information in the data bank regarding all clients which are connected where the client only needs information about the clients to whom it must send video data.

20.2 Memory management

Handling the amount of data incurred when streaming video data, calls for avoiding data copying as much as possible. Needless to say, this restriction only applies to video data, as control data only constitute a small amount of the total data. Thus, minimizing the amount of video data being copied is a must. This is especially true for the server which may distribute a vast amount of data.

As a basic rule the server protocol must not copy video data at all. As the server application has the role of reading data from disc etc. and supplying it to the protocol the memory must be allocated by the server application and passed through the interface as a pointer. This means that all data which is sent by the protocol will be held by the server application and is only represented by a pointer to the protocol. Thus, the protocol needs to be able to notify the application when it is done using the data.

Passing video data to the client application from the protocol works in a different way. Here, copying the memory between the protocol and the application is possible, as the amount of memory transferred to the client application does not have the same size as on the server. Thus, the client application allocates a segment of memory and accesses the protocol, which in turn copies data from the buffer into the memory segment.

At the bottom of the protocol, the socket layer will be supplied a pointer to an amount of data allocated by the protocol, and accordingly filled by the socket layer. This piece of memory will be represented by a pointer which will be transported between the various components. Once the protocol is done using the data it must deallocate it. Deallocating the memory is thus not up to the transport handler, as the data is passed to another component, even though the transport handler has the job of allocating it. This is the case both for the server side protocol and the client side protocol.

20.3 Thread design

As already stated, the threads will work as glue between the components. Thus, a thread will for example extract data from the transport handler and push it to the packet queue with as little logic as possible. Only the code which handles the flow of data is placed in the threads meaning that only the code which connects the different components is placed here. This design has the great advantage that each of the components can be tested and debugged separately without having to work with multiple threads and only the final assembling will involve many threads.

21 Class design

This section contains a detailed description of the components described in Section 20. Each component will be presented using a UML class diagram. These diagrams will only display member variables and member procedures relevant to the understanding of the implementation. Private members are also presented where it leads to easier understanding.

As previously stated the components will be reused by both client and server-side to the extent that it is possible. This section will walk through the details of the components, and when necessary describe the differences between the server and the client. Furthermore a description of the memory management of each component will be presented.

21.1 Packets

All the packet types described in Section 17 are wrapped in a `packet` class. This packet class is inherited to a number of different classes, each representing one of the packets. Data which is shared between all packet types are defined in the base class, limited to only the contents of the packet header. Data specific to each of the packet types is private to each of the specific inherited packet classes.

Internally, data is represented in network byte order, as described in Section 6.6. Thus, all information must be converted to host byte order when accessed by the protocol. This gives the advantage, that when the packet is dispatched from the transport handler, no conversion and no memory copy is needed. Likewise, when data is received from the network, data is simply embedded in the packet without any network-to-host data conversion.

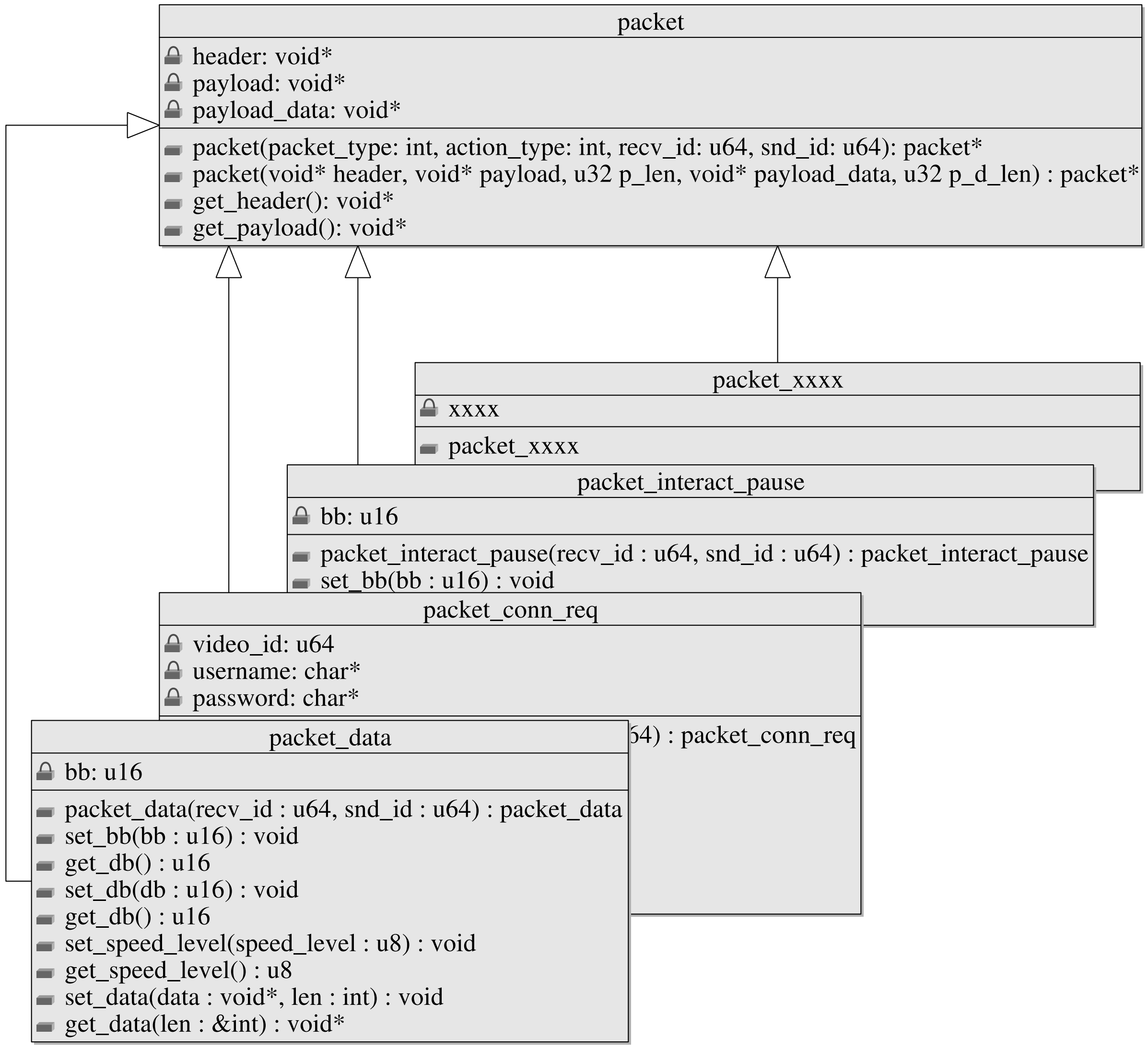


Figure 38: Class design of the packets.

21.1.1 Memory management

As indicated by Figure 38 the packet contains three types of data: A header and a payload which are present in all packets and a payload data which is only present in a subset of the packets. The two first entities are allocated by the packet once it is instantiated while the third field is set by a pointer to an already allocated area. The advantage of this is that each of the entities can be deallocated without impact on the others.

21.2 Transport handler

This component is embedded in a single `transport_handler` class utilizing a library wrapping the socket API. This library is a simple extension of the socket API which raises an exception when an error occurs.

The `transport_handler` class has member functions used for sending data through either CCP or DCP. These functions take as argument a `packet` and send the data embedded in the packet via TCP or UDP. Receiving data is done via a single member function returning the first packet available on all channels regardless of the underlying transport protocol. Data is returned as a `packet`.

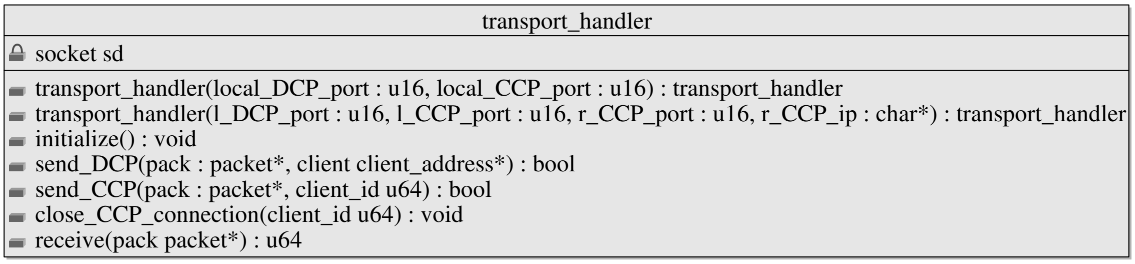


Figure 39: Class design of the transport handler component.

The difference between the client-side and the server-side `transport_handler`, is only the constructor of the class which takes different arguments. Special to the `transport_handler` is, that the class generates the client id of new connecting clients on the server side. This is due to the fact that the client must be assigned a socket descriptor before the server application has actually granted a connection. If the server application rejects the client, the protocol needs to be able to close the TCP connection by using the client id.

21.2.1 Memory management

Sending data through the `transport_handler` is done without the class interfering with data contained in the packet. Thus, the caller of the send procedure must deallocate the `packet` as according to Section 21.1.1.

Receiving packets is different. Upon reception of a packet the class will allocate the needed amount of memory in three pieces, one for each of the three fields in the `packet`.

21.3 Incoming packet queue

In order to control the amount of memory used by the client protocol, the packet queue will be implemented using a limit which controls how much data can be stored in the queue. Access to the queue will be controlled by a mutex which ensure mutual exclusion and by a counting semaphore which will force the thread retrieving packets from the queue to wait if the queue is empty.

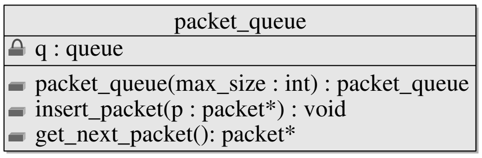


Figure 40: Class design of the packet queue.

21.3.1 Memory management

Data inserted into the queue is simply a pointer to the received `packet` allocated by the `transport_handler`. Hence, no memory is copied during insert and retrieval.

21.4 Data bank

The data bank component contains all information needed in relation to clients, their capacity, and the properties of the stream they are receiving. Hence, the data bank will constitute a central point of the implementation, as many components will need to access the data contained herein.

The data bank differs from the client version of the protocol to the server version on two distinct points.

1. The client needs information about all other clients currently attached to the client in order to stream data correctly.
2. The server data bank needs information regarding to all connected clients.

Because of these differences the data bank is divided in two different components. As the server must function as a client in regard to video data dispatching etc. all sub-components included in the client data bank, will also be part of the server data bank.

21.4.1 Data bank - Client

The `data_bank_client` needs to contain information about all clients to whom it should stream data. The `data_bank` must have a way of identifying to whom and when the next data packet should be dispatched. Thus, the `stream_engine` will use this facility to decide what data should be sent next.

The `data_bank_client` should also keep track of where data should be received from and with what rate data must be received. Hence the `data_bank_client` must continuously keep track of the time interval in which data packets are received.

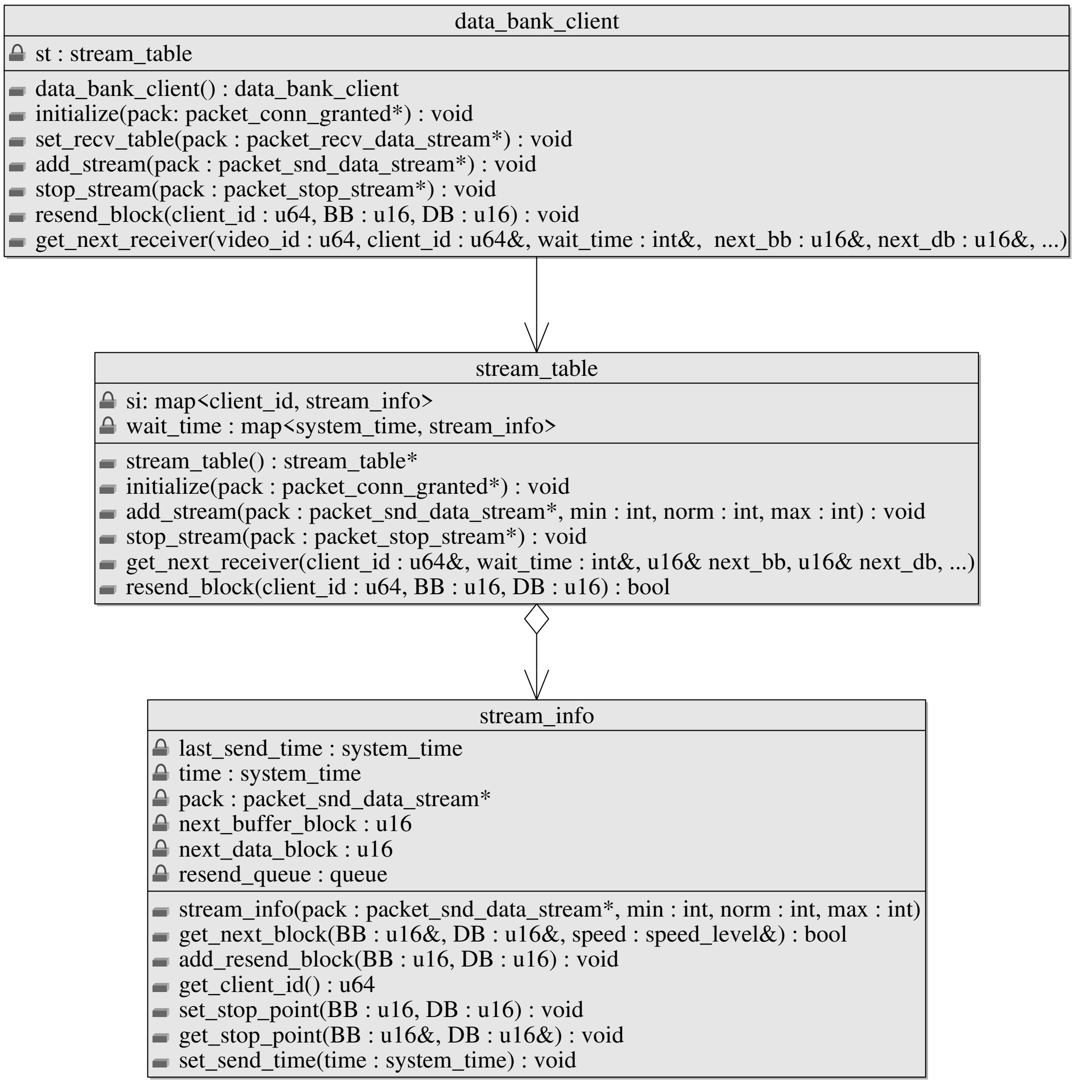


Figure 41: Class design of the client data bank.

Internally, the `data_bank_client` is structured using an instance of the `stream_table` class utilizing a number of `stream_info` instances each representing the stream of data which must be sent to a client given in the `stream_info` instance. If the nature of this stream changes, e.g.. the speed is changed, or the distribution of data blocks changes, the `stream_info` instance is simply changed by setting the packet sent from the server.

21.4.2 Data bank - Server

The server-side data bank will be structured using a table containing a number of `client_info` instances, each containing information related to any connected client. Updating this information is simply done by handling the `client_table` a status packet when this is received by the server. As with the client-side data bank the class `stream_table` functions similarly by holding a number of `stream_info` entities. When the server changes the characteristics of the data stream dispatched to the client identified by the `stream_info`, this is simply done by handling the SND_DATA_STREAM packet, as described in Section 17.3.2, to the `stream_info` class which will update itself accordingly.

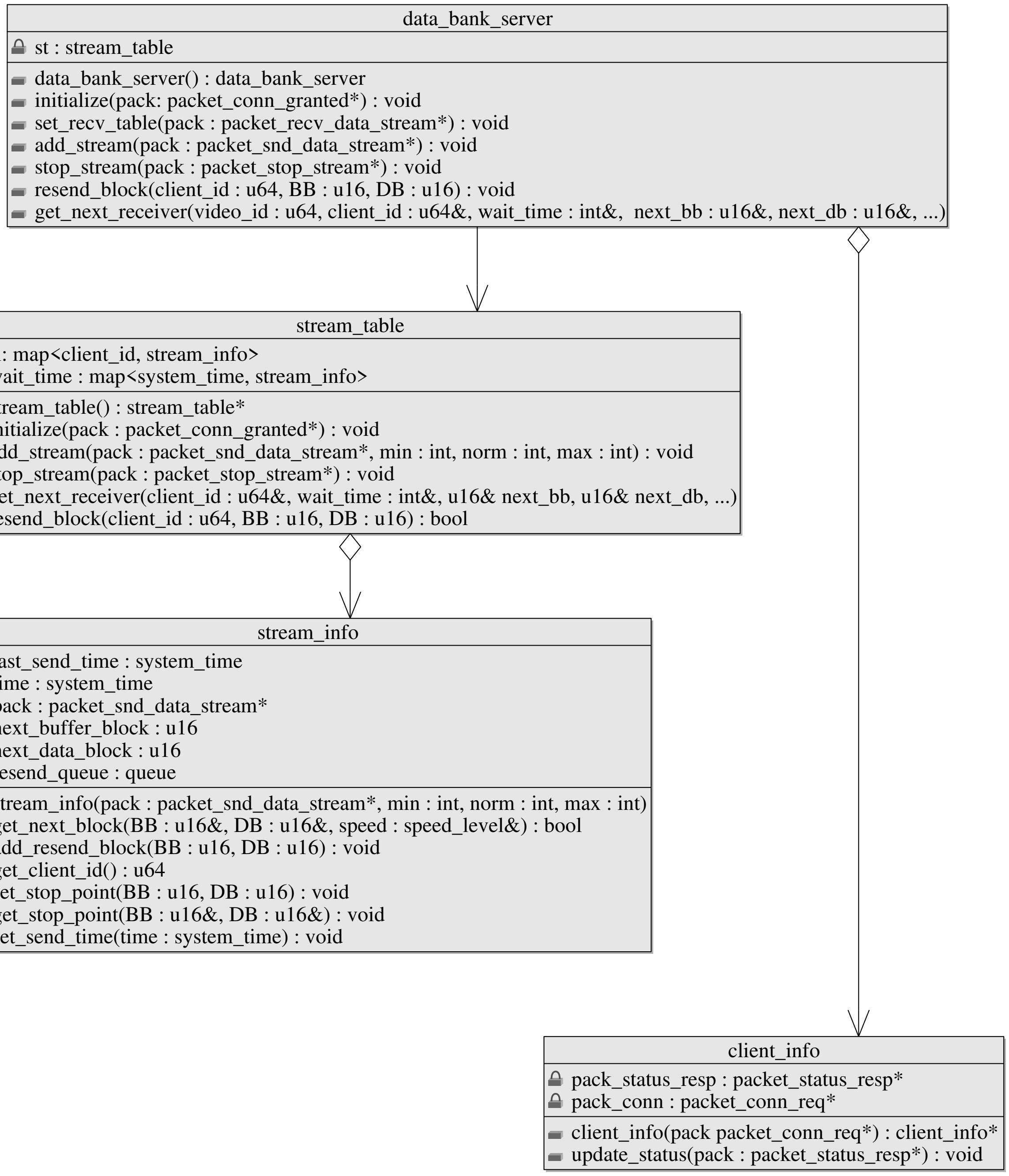


Figure 42: Class design of the server data bank.

21.5 Stream engine

The stream engine component is tied to the functionality of streaming the content of the video to a number of clients. This functionality will be embedded in a `stream_engine` class, which in itself distributes the data contained in a video. The server-side protocol needs to hold several instances, as one instance only distributes a single video while the client needs only one.

As the data bank component contains the information related to each of the data streams dispatched from the stream engine component, the two components must interact directly. Thus, the stream engine must hold the data related to the video which it distributes, and the information related to the data streams must be accessed through the data bank.

To distribute data, the stream engine accesses data held in the data container. This component will be tightly integrated with the stream engine as the stream engine must access video data directly. Deciding what data must be sent will be done by continuously calling the `data_bank` to extract the next client which should receive data, which data should be sent, and the delay in milliseconds until data should be sent. The calling thread must wait the given amount of time before dispatching data to maintain the interval at which packets are be sent.

The only difference between the server-side `stream_engine` and the client-side `stream_engine` is the way that data is stored. Data kept on the client-side protocol is received from other clients and only a part of the video is available. The server also keeps part of the video but data can always be retrieved from the server application. Unifying the functionality of the `stream_engine` requires that the data containers must provide a uniform interface which will hide the differences from the `stream_engine` class. A diagram depicting the design of the `stream_engine` class is provided in Figure 43.

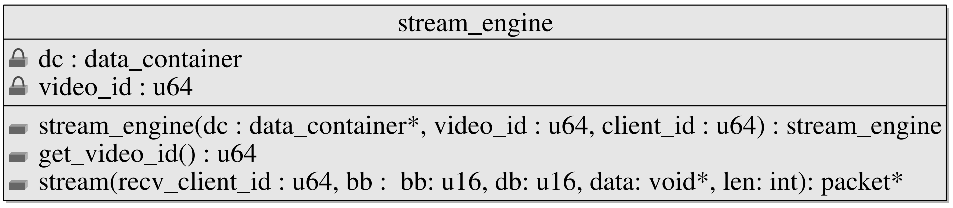


Figure 43: Class design of the stream engine.

21.6 Data container

The data container will be constructed as an abstract class providing the uniform interface which all derived classes must implement. The classes `data_buffer` and `data_cache` will be constructed to be used client and server-side.

21.6.1 Data buffer

To conform to the protocol design given in Section 14.2 regarding buffering of data, the `data_buffer` class will be implemented as a circular buffer. The `data_buffer` will be filled with data received by the logic, while the stream_engine will retrieve data from the buffer and send this to the receivers given in the data bank.

21.6.2 Data cache

The server-side data container will be implemented in the class data_cache. This class differs from the client-side data_buffer in the respect that it must fetch data by itself. The resemblance to paging in modern operating systems is thus immediate. One could argue that video, being a continuous media, prefetching of data can never be random, as the protocol will always know what block a client must be provided with next. But the added complexity incurred when opening up for resending of data, and the role of the server when a client requests emergency data results in the server having to access DBs randomly. Under optimal circumstances, the data_cache will simply prefetch the next BB if this is not present. This implies that the data_cache must have a knowledge of what data the receivers expect. Thus, when data is dispatched, the cache must check if the next block is present and retrieve this from the server application if necessary. This raises the simple question of how the protocol handles a cache miss. As a starting point, the cache should not encounter cache misses, but as already described requesting emergency or resending of data may occur. Furthermore, data must be requested from the server application which entails that some waiting time may be expected. In some cases data may even not be available when needed as the server application may be heavily loaded. Thus, the stream_engine cannot rely upon data being contained in the cache when it is needed. Conversely, data must sooner or later reach the data cache. Thus, if the server application fails to provide the protocol with data either the application is constructed erroneously, or the hardware utilized by the host is too constrained. Therefore, the protocol assumes that data will reach the data cache before it is too late.

As the name implies, a cache only contains part of the data needed by the protocol. Thus, as the cache is being filled, it will at some point be necessary to overwrite some of the blocks contained in the cache. This also resembles the workings of a paging system, as some pages will have to be swapped out when new ones are needed. Paging systems therefore uses different algorithms to determine which page should be swapped out before a new one can be read. In the scenario at hand, a walk through of the workings of these algorithms is far from relevant as this will be out of the scope of this thesis. To simplify the procedure, we note the existence of two simple ways of determining which block should be deleted. The first and simplest is to delete the block which was last accessed. Thus, when the blocks are used, the protocol needs to time stamp the blocks to determine which block has been used last. The second model exploits, that the server is updated regularly with information from each client indicating where the play pointer is situated. Thus, the server knows that when the play pointer of a client has passed a given block, it will not be requested by the client except if a skip should occur. This would however still require some algorithm which can decide which block should be deleted, as this may be relevant in some cases. Instead a more simple approach is chosen. When the server application fulfills a data request the call through the interface returns a pointer to the data which has been possibly released by the protocol. Thus, the control of the data has been transferred to the server application.

21.6.3 Class design of the data container

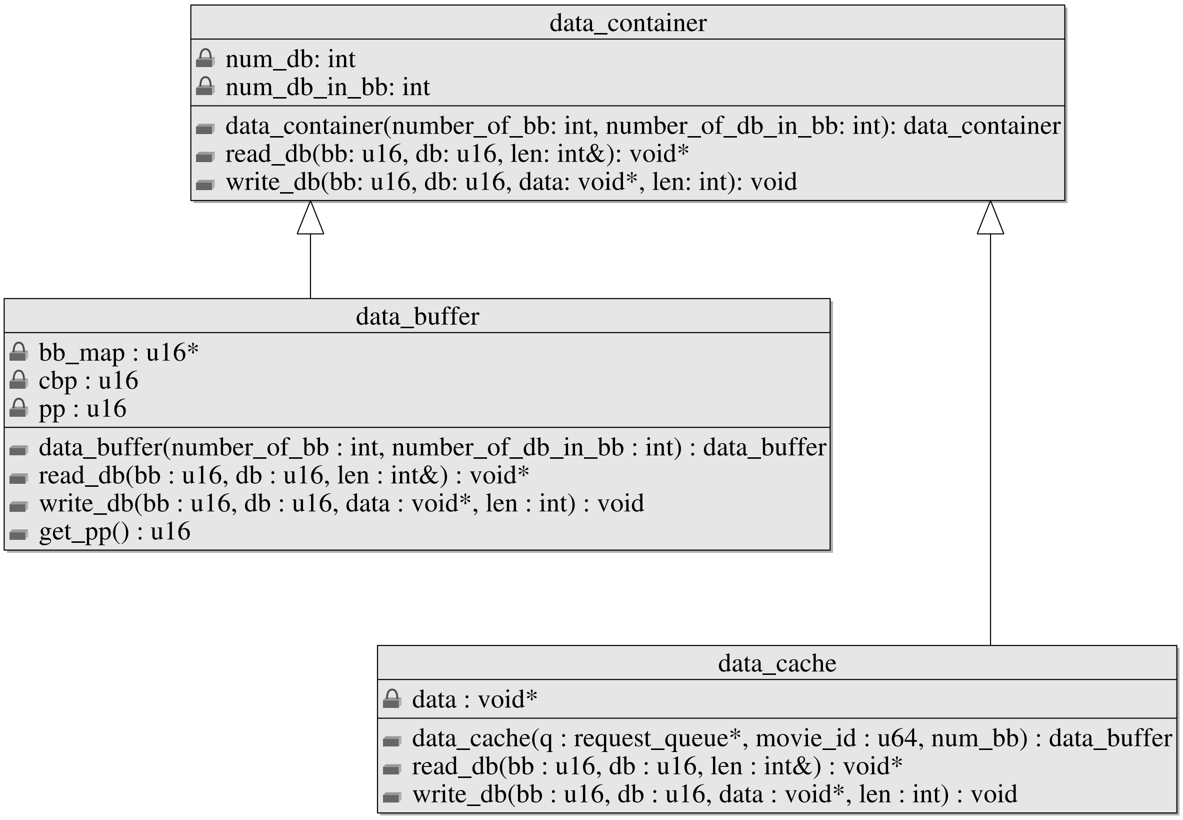


Figure 44: Class design of the data container.

21.7 Application task queue

This component is used as the link between the protocol and the server application. Since all data and login validation etc. should be handled by the server application a communication queue is needed. In this way the application can be implemented as a multithreaded application listening for the different tasks enqueued by the protocol. Thus, when a new task is enqueued by the protocol it must signal to the server application that a new task is ready.

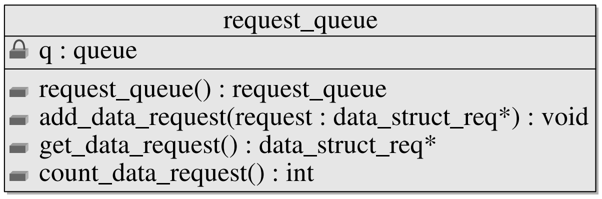


Figure 45: Class design of the application task queue.

21.8 Client-side logic

The client-side logic handles the processing of incoming packets. These are handed to the logic which are handled according to the logic described in each of the packet definitions given in Section 17. Evidently, packets of type $\langle 10, * \rangle$, $\langle 30, * \rangle$, $\langle 50, * \rangle$, and $\langle 60, * \rangle$ are to be handled by the client-side logic. These four packet types define packets sent from server to client and packets sent from a video data receiver or sender to a client. All other packets must be neglected.

Furthermore, the client-side logic must handle requests from the client application. These will be passed from the interface to the client-side logic component responsible for performing the requested task.

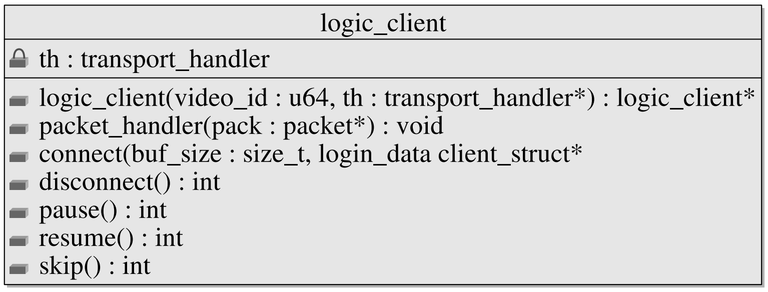


Figure 46: Class design of the client logic.

21.9 Server-side logic

As is the case with the client, the main task of the server logic is to handle incoming packets. The packets received by the server is the remaining part of the packets which is not treated by the client logic. This fundamental difference is the main reason why the logic is divided in two. Several packets result in clients being relocated which is done by the logic using the selection algorithm as described in Section 22.

Actions on the server are triggered by the reception of packets with one exception. The server needs to keep track of whether connected clients send status packet with the correct interval. Otherwise, a status request must be sent.

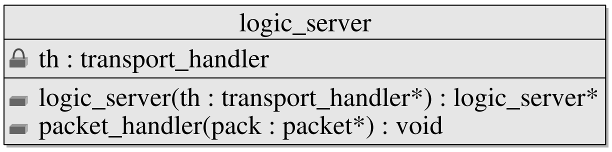


Figure 47: Class design of the server logic.

22 Selection algorithm

As stated in Section 14.6 the selection mechanism seeks to build up the logical appearance of the network.

For the selection mechanism to function a sub-mechanism will be used to serve the purpose of identifying the most anti-social client. This mechanism will be used in the event of a topology change which will result in the need for disconnecting a client. This may happen when the protocol cannot find a suitable point for a new or relocated client which provides an amount of bandwidth higher than an already connected client. Thus, if equilibrium cannot be obtained when attaching the new client, another client must be disconnected to make room for the new client.

Furthermore, the selection mechanism will make use of another sub-mechanism which is used to identify which parts of the video stream a set of clients should send to another client. Thus, the set of clients will each send a part of the data stream to constitute the total stream. The sub-mechanism distributes the amount of DBs of each BB between the set of clients while accounting for the bandwidth available to each client. Hence, the sub-mechanism will output a list identifying which DBs each client must transmit to the receiver of the data stream.

For the mechanisms to function, a number of working parameters can be present. These parameters can be used as a foundation for the mechanisms to make proper decisions. The parameters may be given and maintained for each client and can be separated into three categories:

Static data: Variables related to static information collected upon connection of a client

Dynamic data: Variables related to information collected during the course of a client's session. The information can be based upon status retrieved from each client regarding the current position, buffer contents, etc.

Statistics: Variables related to statistics gathered by the server during the session of each client. The statistics can be gathered every time some event occurs, e.g. when the client pauses, skips, or a client notifies the server of some error, etc. Furthermore, statistics can be gathered from the dynamic data, enabling the server to keep a history of the client's performance.

Intuitively, much information can be gathered to create a more solid ground for taking decisions. The more variables used by the algorithm the more complex and time consuming it will be although it may also build a more efficient topology. We note the almost endless possibilities in this subject but choose only to implement a simple selection mechanism which illustrate how the topology may be built. How this is performed is described in Figure 48.

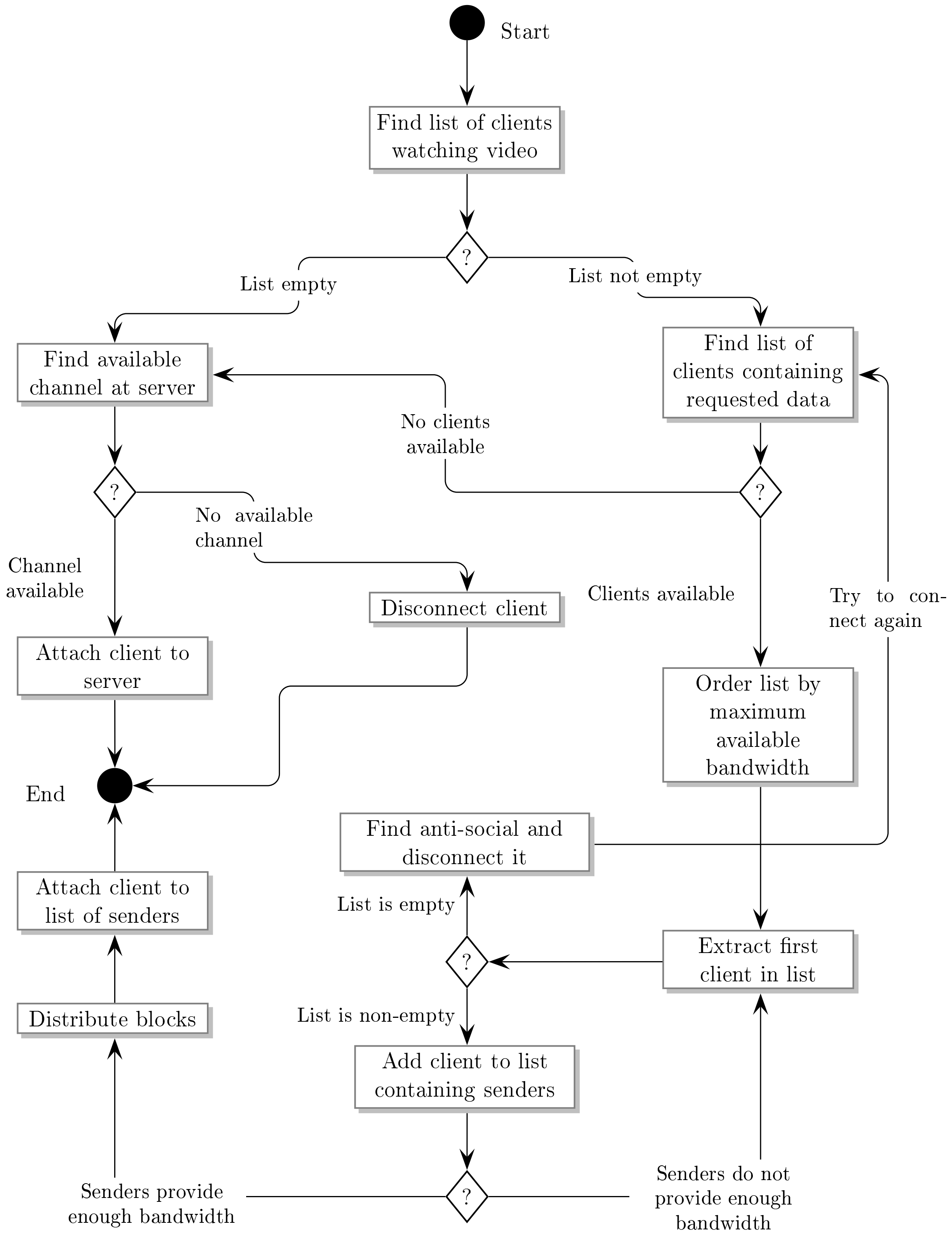


Figure 48: The selection algorithm.

22.1 Finding the most anti-social client

The purpose of this mechanism is to find the most anti-social client. This is only needed in situations where new connecting or relocated clients cannot be attached to any clients because of bandwidth or buffer starvation. If the new or relocated client provides a higher bandwidth than the most anti-social client which can be disconnected in favour of the new client this is done. How the most anti-social client is selected is therefore the purpose of this mechanism. We note that there are many possibilities in implementing this mechanism, as the variables given above may be used in finding the most anti-social client. Therefore an example is given below:

Step 1: The first step of the mechanism is to determine if the new or relocated client is anti-social in itself. If this is the case the client it self should immediately be chosen and disconnected and the algorithm finished.

Step 2: Find all candidates watching the same video as the new or relocated client.

Step 3: From the list of clients found in step 2 all clients that do not contain the requested BB in their CA must be removed.

Step 4: Remove all candidates which are not marked anti-social. These are clients which provide at least the same amount of upload capacity which is required to forward the video unassisted.

Step 5: Sort the remaining clients by available bandwidth. The first client in this list is best candidate. If only one client is found providing the smallest available bandwidth, this client should be disconnected in favor of the new or relocated client.

Step 6: If more than one client has been found in the previous step, the client which forwards data to the lowest number of clients is disconnected in favor of the new or relocated client.

Step 7: If the previous step yields to more than one client the client which has performed the highest number of user interactions is disconnected in favor of the new or relocated client.

The last three steps define the central workings of the mechanism. Therefore, these can be varied in a infinite number of ways. We note these interesting possibilities but choose only to implement a simple mechanism which will disconnect the first client which can be substituted with the new or relocated client. Furthermore, clients that provide a buffer size lower than what is advertised by the server is not considered by the mechanism. As being anti-social is defined as not providing sufficient bandwidth or buffer this should also be part of the mechanism.

22.2 Block distribution mechanism

This mechanism seeks to distribute the DBs of each BB to multiple clients in order to enable multiple clients sending the total data stream to another client.

As an illustration, consider two clients selected to send the total stream to another client. Each BB consists of 100 DB. The first client can only provide 2 percent of the required bandwidth while the second one 98 percent. Thus, the first client can only send two DB of each BB while the second will send the remaining 98 DBs.

Choosing which DBs each of the clients should send may seem trivial. But as DBs should be received in an order resembling the order of which they are distributed throughout the BBs this may not be trivial. Thus, letting the first client send the two first DBs of each BB may result in the capacity of the sending client being over used with the impact that data is received long after it should have been used. In this example, a better solution would be to only let the first client send DB 1 and DB 50 of each BB. Clearly, as the number of clients sending data concurrently to another client rises, and the more their bandwidth varies, the more complex the best solution will be. The solution to this, is to construct an algorithm which traverses all the DBs in a BB and for each DB decides, which client is best suited to send the current DB. This may be done in numerous ways, as the mechanism can, like the rest of the selection mechanism make use of variables collected by the server. However, we choose to implement this simple by performing the following steps:

Step 1: For each client the average distance between consecutive data dispatched is calculated. This distance changes as the number of DBs gets fewer. To illustrate this, if a BB contains 100 DBs and a client needs to send 10 of these, the average distance is 100/10. As the DBs are traversed and distributed among the clients this average distance changes. Thus, when the first 5 DB has been traversed and a sender has been chosen for these, the average distance for this particular client becomes 95/10. As the client is allocated DBs the denominator of the fraction falls.

Step 2: Based on the above calculated distances the best suited client is found for the current DB. This is done by looking at the distance of all the clients and selecting the one which comes closest. This client is then allocated the block and the next block is traversed.

The above algorithm will yield a list of clients and the DB numbers of each BB they must send to the client which is to receive the video data stream.

23 Threading

This section analyses and describes how the protocol will make use of multi-threaded functionalities. The section will be based upon the components of the implementation accounted for in Section 21.

23.1 Receiving packages

Both the client and the server share the same structure in reference to reception of packets. All data is received by the transport handler and based on the received data a packet is instantiated. The packet is then pushed onto the incoming packet queue.

To empty the buffer of the socket quickly the implementation will make use of a thread which will have the sole purpose of extracting packets from the transport handler and placing these on the packet queue. As a large amount of data may be received the thread should be occupied as little as possible with other tasks. Therefore no logic will be applied to the packets before they are pushed onto the queue.

23.2 Incoming packet processing

All packets queued in the incoming packet queue, must eventually be processed. A single thread is sufficient to handle this task. Furthermore this thread will need access to most components and may therefore require a lot of synchronizing with other threads to avoid accessing critical regions concurrently. This synchronization may generate a large amount of overhead, therefore care must be taken to only access regions which are needed by the thread.

A single thread is therefore employed which must pop packets from the incoming packet queue and process these using the logic component. The processing of the packets can lead to a number of events as given in Section 13.

23.3 Buffer and cache

Data is filled into the buffer of the client by the thread which processes all incoming packets. Furthermore, the client needs to keep track of the speed at which data is received. At first glance one would argue that this could be done by the thread filling the buffer, but clearly if data receival is brought to a standstill this would never be detected. Therefore, a separate thread is employed for this task.

23.4 Stream engine

For the stream engine to work, a separate thread with the sole purpose of sending video data to all attached clients will be employed. If data cannot be retrieved from the buffer, either because of buffer underflow or overflow this thread will also be responsible for sending packets identifying that the data request could not be fulfilled. Thus, each instantiated stream engine component will employ a thread for this purpose.

23.5 Status thread

Furthermore, the logic needs to send status packets to the server with a specified interval. This will be done using a separate thread which will be awakened at an interval specified by the server.

23.6 Application threads

Instantiating the protocol either as server or client will result in the constructor of the protocol spawning a number of the above threads. When this has been performed, the class has been constructed, and control will be returned to the calling process. Hereafter, the application may use the interface as given in Section 24. Some calls dispatched to the protocol may result in the protocol blocking the calling thread until some event has occurred. Thus, the server application needs to call the protocol to block until the protocol requests data. Contrarily, the client application needs to call the protocol for incoming data which the client needs to display by the video player. Beside this role the client thread will be responsible for dispatching user interactions to the protocol. This means that when the user interacts with the video the application thread must call the interface. This call will be handled by the logic which in turn may end up transmitting a packet.

Whether the applications handle these tasks using one or more threads is completely up to the implementor and can be done in numerous ways.

23.7 Summarizing

To briefly summerize, the threads present in the protocol are:

Transport handler thread: This thread receives packets and inserts them onto the incoming packet queue. The thread will be present both in the server and the client instance of the protocol.

Logic thread: This thread takes one packet at a time from the incoming packet queue and process them using the logic component. The thread is also present in the server and the client.

Stream engine thread: This thread has the job of sending video data packets to attached clients. The thread is not active if the client does not stream data to another client. As the server also functions as a client, the thread is present on both sides. Furthermore, one thread is used per instance of the `stream_engine` class.

Status thread: This thread is used differently on the server and client. On the client side, the thread is awakened at a specified interval and sends status packets to the server. On the server side the thread has the job of ensuring that all clients are sending status respond packets at the correct interval.

Reception speed: This thread is only present on the client and has the job of ensuring that video data packets are received at the correct speed. This thread furthermore ensures that data reception does not come to a standstill.

24 Interface

This section focuses upon the interface between applications and the protocol. The interface is presented through the classes `vod_server` and `vod_client`. The interface is comprised of the publicly accessible procedures implemented in the classes. The classes conform to the interface specifications as described in Section 19.

The interface will transport classes and structures, thus precluding applications implemented in a programming language which does not understand the C++ class construct.

As the protocol provides both functionality for clients and servers, this section is divided in two, one describing the server side of the implementation and one describing the client side.

24.1 Server interface

The section describes the calls which can be made to the server instance of the protocol.

- `vod_server* vod_server(int data_port, int control_port);`
  The constructor of the class `vod_server` returns an instance of the class. The `data_port` specifies the UDP port used by the DCP for transferring video data to connected clients. The `control_port` specifies the TCP port used by the CCP for communicating with connected clients.

- `int open();`
  The `open` call binds the two ports which have been set in the constructor. The `open` must be performed consecutively after the class has been constructed. Otherwise the `open` call will return an error as indicated by a negative return value.

- `int close();`
  The `close` call disconnects all connected clients and closes all open file handles and sockets. May be called at any time after the `open` call has been performed.

- ```
  void poll(int& video_data,
            int& video_data_hp,
            int& sec_data,
            int& sec_data_hp,
            int& login_req,
            int& events);
  ```
  After the open call has been performed, the server application may start listening for incoming tasks by using the `poll` call. Regardless of how many requests ready to be served the call will return control to the calling thread immediately. The number and nature of the requests which are ready to be served is given as references used as arguments in the call:
  ```

1. `int& video_data`
   Indicates that a number of data request are ready to be served by the server application. The number given in the reference corresponds to the number of enqueued requests.
2. `int& video_data_hp`
   Indicates that a number of data request assigned high priority are ready to be served by the server application. The number given in the reference corresponds to the number of enqueued requests. This may be used in the event of a emergency resend of data not currently held by the data cache.
3. `int& sec_data`
   Indicates that a number of security data request are ready to be served by the server application. The number given in the reference corresponds to the number of enqueued requests.
4. `int& sec_data_hp`
   Indicates that a number of security data requests assigned high priority are ready to be served by the server application. The number given in the reference corresponds to the number of enqueued requests.
5. `int& login_req`
   Indicates that a number of login requests are ready to be served by the server application. The number given in the reference corresponds to the number of enqueued requests.
6. `int& events`
   Indicates that a number of events are ready to be read by the server application. The number given in the reference corresponds to the number of enqueued requests. This queue can be used by the protocol to inform the server application that some event has occurred. This should simply be regarded as a log functionality.

After retrieving these values the application may serve these requests at will. Requests placed in the high priority queues should be served first, but this is not a demand.

- `int get_data_req(struct data_struct_req& data, DATA_TYPE type);`

  The `get_data_req` procedure returns the content of the next data request of the type given in `type`. The call blocks the calling thread if no request is available. The content returned in the structure `data` specifies the offset and length of the data from a given video needed by the server. In this structure a field `transaction_id` specifies a unique id which must be supplied to the protocol when the request is served. This enables the protocol to monitor if a request has been serviced.

  The argument `data` is defined as the struct:

```
struct data_struct_req
{
  unsigned long video_id;
  unsigned long byte_offset;
  size_t len;
  long transaction_id;
};
```

The field `type` is defined by:

```
enum DATA_TYPE
{
  VIDEO_DATA,
  VIDEO_DATA_HP,
  SEC_DATA,
  SEC_DATA_HP
};
```

- `int get_login_req(struct login_struct_req& user);`

  The `get_login_req` procedure returns the content of the next connection request held by the protocol. The call blocks the calling thread if no request is available. The content returned in the structure `user` specifies the user and password of the user trying to connect. In this structure a field `transaction_id` specifies a unique id which must be supplied to the protocol when the request is served. This enables the protocol to monitor if a request has been serviced.

  The argument `user` is defined by the structure:

```
struct login_struct_req
{
  unsigned long video_id;
  user_name;
  password;
  long transaction_id;
};
```

- `int get_event(struct event_struct& event);`

  The `get_event` procedure returns the content of the next event held by the protocol. The call blocks the calling thread if no request is available. The content returned in the argument `event` specifies the event encountered by the protocol.

  The event returned by the protocol in the argument `event` is defined as:

```
struct event_struct
{
  EVENT_TYPE event;
  unsigned long timestamp;
  char* msg;
  int msg_len;
};
```

  With `event` defined as:

```
enum EVENT_TYPE
{
  // Currently unused
};
```

- `int get_video_info_req(unsigned long& video_id);`

  The `get_video_info` procedure returns the content of the next video info request held by the protocol. The call blocks the calling thread if no request is available. The content returned in the reference `video_id` specifies the video which the protocol wishes to retrieve data for.

- `data_class* deliver_data(data_class* data, DATA_TYPE type);`

  The `deliver_data` procedure delivers a type of data requested earlier by the protocol. The procedure is used by the server application, maintaining the three types of data, to deliver whatever data has been requested by the protocol. The `data_class` construct is allocated by the server application, and passed to the protocol which then uses it.

  The procedure returns a pointer to a `data_class` instance which has been released by the server protocol. Thus, the protocol uses this to signal the application that it no longer needs the `data_class` instance in the cache. If the return value is `NULL` no instance has been released by the protocol. This may at first seem as a cumbersome solution, but in fact it enables advanced memory management on the application side. The solution enables the protocol to avoid moving data across the interface which can be very expensive.

  Data is thus delivered as a class defined as:

```
class data_class
{
  private:
    void* data;
    size_t len;
    unsigned long video_id;
    long transaction_id;
  public:
    data_class(unsigned long video_id,
               long transaction_id,
               void* data,
               size_t len);
    void* get_data();
    size_t get_data_length();
    unsigned long get_video_id();
    long get_transaction_id();
};
```

- `int deliver_login(struct login_struct_resp login);`

  The `deliver_login` procedure authorizes clients which try to connect to the server. The protocol will match a previous connection request by the transaction id embedded in the argument `login`. This will furthermore contain the measured upstream capacity of the client measured in available byte/sec.

  `login` is defined as:

```
struct login_struct_resp
{
  long transaction_id;
  boolean login_allow;
  int upload_bandwidth;
};
```

- `int deliver_video_info(struct video_struct_resp video);`

  The `deliver_video_info` procedure provides the protocol with information related to a video. Thus, this call is used to provide information about new videos to the protocol. The argument `video` is defined by the structure `video_struct_resp` given by:

```
struct video_struct_resp
{
  unsigned long video_id;
  int bit_rate;
  char[256] title;
  unsigned long video_len;
  unsigned long video_duration;
  unsigned int skip_distance;
};
```

24.2 Client interface

The client interface defines the applications programmers interface used when programming a client application.

- ```
  vod_client* vod_client(size_t buf_size,
                         struct client_address* client);
  ```

  The constructor is given a number of arguments used for initializing the protocol. The `size_t buf_size` argument specifies the maximum amount of memory which may be used by the internal buffer of the protocol instance. This limit concerns only the buffer as this will be the the main memory consumer. The overall memory used by the protocol will therefore exceed this limit. The argument is measured in bytes.

  The argument `struct client_address* client` specifies the address of the local client as found in the structure:

```
struct client_address
{
  ccp_port;
  dcp_port;
};
```

`ccp_port` defines the local TCP port used by the CCP while `dcp_port` defines the local UDP port opened by the DCP.
```

- ```
  int connect(unsigned long video_id,
              struct server_addr* address,
              struct login_data* client);
  ```

  This call is performed when a client wishes to connect the protocol to a given server. This call may be followed by a `recv_data` call. The arguments are defined as followed:

  1. `unsigned long video_id` defines the video which the client wishes to stream. This id is of no relevance to the protocol and is defined by the server application. The id is only used by the protocol to identify a video, but its contents is defined by the server application.
  2. The `struct server_addr* address` argument defines the address of the server offering the video material. The call will connect the client to the server contained in `address` as defined by:

     ```
     struct server_addr
     {
       char* ip_address;
       int port;
     };
     ```

  3. `struct login_data* client` identifies the local client wishing to connect to the server. The argument contains among others a user name and password which must be authorized by the server application before the client can be granted access. The argument is defined as:

     ```
     struct login_data
     {
       unsigned long video_id;
       char[64] user_name;
       char[64] password;
       unsigned int video_len;
       unsigned int video_duration;
       unsigned int skip_distance;
       char[256] title;
     };
     ```

- ```
  size_t recv_data(void* buf, size_t max_len,
   unsigned long& offset);
  ```

  from the protocol layer. Data returned will be the next data available. Data will be copied into the memory area given in `buf` at a maximum length of `max_len` bytes. The call blocks until data is available and returns the amount of bytes written to `buf`. The argument `offset` identifies the byte offset in the video.

- `int disconnect();`
  This call disconnects the client from the server. The call may be regarded as stopping the video stream.

- `int pause();`
  The call pauses the data stream until it is resumed. Subsequent calls to `recv_data` until the stream has been resumed will fail.

- `int resume();`
  This call resumes the data stream and enables the client application to retrieve data. This call can only be performed after a pause.

- `int skip(unsigned long position);`
  The `skip` call will skip to a given place on the video. The argument `position` specifies the point in the video where the skip must occur to given in milliseconds. This call can only be performed in playing mode, e.g. the video must not be paused.

25 Our implementation

This section describes our implementation. The section is therefore meant as documentation for the problems incurred in the process of implementing the protocol. Furthermore, the section clarifies the shortcomings of the implementation and where we have chosen to restrain the implementation related to the protocol design.

Initially the protocol was implemented using standard C++ on a Linux platform using the BSD POSIX socket API and the BSD pthread implementation. The idea was simply to implement the protocol using Linux and port the code to the windows .NET platform, thus compiling a dynamic linked library. This library was simply to be included when developing the applications making use of the protocol. This would give us the strength of Linux, when implementing the protocol and the flexibility of the .NET platform concerning graphical user interface. Unfortunately, porting the protocol to windows was more problematic than imagined. Thus, neither the socket API and the pthread library works seamlessly under windows. Therefore, we decided to move the implementation of the protocol to the .NET platform at a late state.

The socket implementation used under windows, called Winsock unfortunately does not provide all of the functions described in the POSIX socket specification. Thus, neither the `sendmsg` call or the `recvmsg` call are available using Winsock. This forced us to implement receiving and sending of packets using expensive copying of memory. Furthermore the implementation of the MSG_PEEK option when reading from connectionless sockets is unstable which is acknowledged by Microsoft.

As stated, the pthread library is not directly accessible using the .NET platform. However, implementation of the library does exist but these are made by third part providers and not Microsoft self. Therefore we chose to use the standard threading libraries provided by the .NET platform. This, however, caused a lot of frustrations as mixing unprotected and protected C++ code when using the Microsoft .NET compiler can be quite tricky.

25.1 Limitations

The implementation of the protocol focuses upon the important parts of the protocol design. Thus, the implementation of streaming a single video from the server, and on to other clients has been the primary goal of the implementation. Hence the following issues remains unimplemented in this prototype of the protocol:

- No security mechanisms has been implemented.
- The server protocol interface has been partially implemented.
- No error handling concerning buffer overflow or underflow has been implemented.
- The selection mechanism of the protocol has been implemented simply.

- The protocol does not considerate the bandwidth of connected clients.

The above issues has been omitted to focus upon the core workings of the protocol which has been implemented to work as intended.

Chapter V

Verification

This chapter contains tests and validations of the protocol. It is divided in two parts. The first part serves as a simple end-to-end test while the second part contains a discussion of the performance obtained by protocol.

26 Verification of the implementation

The end-to-end test considers a setup using a server connected to a number of clients. Based on this we perform a series of tests which should determine whether the implementation fulfills the basic requirements. In order to do this a client and server applications have been developed. Both of these applications have been developed with this specific test in mind. Because of this they are implemented in the simplest manner with no regard to performance. Both applications are implemented using the Visual Studio 2005 .NET framework. The client makes use of the Apple Quicktime component in order to enable easy display of a video. Therefore, the video provided by the server application must be compatible with a decoder which can be used with Apple Quicktime.

In order to minimize the time spent developing the client a simplification has been made. As the quicktime component does not support reading directly from a stream data are written to and streamed from a file concurrently. Thus, the client application both reads data from the protocol and writes it to a file on disk, which is displayed concurrently in the player.

The server application is simple. Once started, a new instance of the server protocol is made and video data is read directly from the disk. Hence, no attempt has been made to optimize the performance of the server application.

26.1 Test scenarios

In this section a set of test scenarios will be designed. These will form the base of the test, which we will perform. The main focus in testing will be one the following cases:

Scenario 1 - Simple streaming of data: Verify that a client can connect to the server and that the client receives data which can be displayed.

Scenario 2 - Data consumption: Verify that the video stream is sent at the correct speed – only the needed amount of data is received.

Scenario 3 - User interaction: Verify that user interaction requests can be send from the client and that the server fulfills the requests correctly.

Scenario 4 - Connection of multiple clients: Verify that a client can send data to a second client.

Scenario 5 - Receiving streams from multiple sources: Verify that a client can receive data from more than one client hence that a client is capable of streaming only a part of the video.

Scenario 6 - Streaming to multiple sources: Verify that a client or the server can stream to more than one other client.

Scenario 7 - Reallocate client: Verify that if a sender of data stops sending, the receiving client will be attached to another client or the server.

The following section will describe how these scenarios are tested. All of the scenarios are performed using both a single computer running the server and one or more clients and on multiple computers running only one application each.

The tests were performed on 4 laptop computers each with a CPU in the range of 2-3 GHz running Microsoft Windows XP. These were connected using a 100 Mbit Ethernet. The video employed in the tests had the following characteristics:

Size: 19.64 MB.

Bit-rate: 1172 Kbit/sec.

Duration: 137050 milliseconds.

Number of BBs: 45.

Number of DBs per BB: 100.

Size of client buffer: 15 BBs.

Media type: MPEG-4.

Scenario 1 - Simple streaming of data

This scenario verifies that a single client can connect to the server and receive the video stream. Furthermore, the scenario verifies that the client is capable of displaying the video.

The expected result is the clients ability to display the video stream continuously without interruptions. Furthermore, after play-back the client should be logged off from the server, and the client should no longer be considered connected.

Scenario 2 - Data consumption

This simple scenario can be verified by testing that during the run of scenario 2 data must not be received slower and not substantially faster than it is consumed. This is easily determined by checking that the consumed ingoing bandwidth used for transmitting video data does not rise much above the bit-rate of the video.

The expected result is that data is only sent at a rate slightly faster than the rate at which it is consumed. This is done using a packet sniffer application which can monitor all incoming and outgoing packets.

Scenario 3 - User interaction

This scenario verifies whether the client can perform a user interaction. The test is conducted by performing a sequence of skips while displaying the video.

The expected result is that a skip in the video stream will display the correct part of the video. Furthermore, skipping outside the buffer will result in small interruptions as the data delivery to the client application is done one BB at a time to simplify data deliverance. Hence the multimedia player must wait until a full BB has arrived before viewing can be resumed.

Scenario 4 - Connection of multiple clients

This test is performed by connecting a second client meanwhile streaming data to the first client. The second client will be attached to the first client as the data needed by the second client is held in the buffer of the first client.

It is expected that both clients are able to display the video from start to end smoothly.

Scenario 5 - Receiving video data from multiple sources

Scenario 4 is run again. A third client is now connected while data is still contained in the buffers of the two first clients. Thus, this third client will be attached to the two first clients. These will each send only half the data stream to the third client. This is done due to the workings of the selection algorithm.

The expected result is that all three clients will display the video from beginning to end without interruptions.

Scenario 6 - Streaming to multiple clients

Scenario 5 verifies that one client can send data to two other clients as the first client sends the full data stream to the second client and half a data stream to the third client. Continuing this scenario by stopping the second client, the server will take over the data stream which was dispatched from the second client. Thus, the server will still provide the full data stream to the first client, and half the data stream to the third client.

It is expected that the first and third clients display the video in its entirety regardless of where the data stream is received from while the second client only displays the video until it is stopped.

Scenario 7 - Reallocate client

Reallocation will happen in scenario 6. Once the second client stops the third client will be reallocated to the server.

It is expected that when a reallocation occurs, no interruptions in the video will be seen.

26.2 The test result

Generally all test scenarios were run successfully and with the expected result. A simple screendump of the test is presented in Appendix E. Thus, we can conclude that the implementation and the design works as intended although this first and simple implementation does not comprise all functionalities of the design. As a result, the implementation is not always completely stable although in this limited setup this does not affect the overall result of verifying that the implementation works as intended. However, some elements discovered during the test deserve to be emphasized.

As skipping between portions of the video is done using BBs we can conclude that sometimes a skip to a given BB will result in interruptions. This is because two adjacent BBs may part the contents of the elements which the video standard is divided into.

During the test we experienced that the protocol did indeed hide all the functionality for the applications which was precisely the goal. However, this did not help to make the test easier as it became difficult to determine how the protocol reacted to different events without digging into the code. In addition to this, some of the functionalities which were deliberately out out also contributed to making the test difficult. This indicates that the functionalities of the protocol is tightly coupled together.

Finally we can conclude, that the intention of creating reusable components which could be used both server-side and client-side has been successful. Thus, the stream engine, the main parts of the data bank, and all the packets were directly reusable.

27 Discussion of the performance

In this section we will discuss the obtained performance of the protocol and whether the design actually reaches the overall goal of lowering the bandwidth usage of the server. As part of this we try to determine the parameters which influence the performance of the protocol leading to a discussion on the worst case and best case scenarios.

As already described in Section 14.1 it is at least in theory possible to connect an infinite number of clients. This can be illustrated by dividing into x BBs. Since all clients can be attached to another clients BB, the total needed number of channels provided by the server will be x. Clearly, this is a simplification as this obviously requires that all clients are capable of streaming at least one whole video stream. But if a client does not have the needed amount of bandwidth to forward the data stream the client is regarded anti-social possibly resulting in the need for a higher number of channels available from the server. Thus, the number of clients which can be connected becomes dependent upon the sum of bandwidth available in the network. The extreme situation is if all connecting clients cannot provide any upstream bandwidth resulting in a network resembling the unicast model presented in Figure 1.

Based on this we can conclude that the bandwidth consumption of the server is strongly related to the upload capacity of the clients and is linearly dependent on the number of BBs the movie is divided into.

27.1 Parameters

Arising from this, a number of parameters determine the ability of the protocol to lower the number of video streams dispatched from the server. Throughout the design and implementation of the protocol we have experienced that these parameters are closely linked together. This became even more true as testing the protocol revealed the relationship between these. Thus, it has become clear that these parameters can be set to an infinite number of combinations which makes it extremely difficult to determine a precise performance gain or loss. Each of the parameters not only influence upon the performance of protocol but they also have a huge impact upon each other.

Client buffer size: Obviously, the size of the clients buffer plays an important role to the performance of the protocol. If a client is able to hold the entire video in its buffer it clearly has a much higher chance of providing the needed data of another client. But the size of the buffer is tightly linked to the amount of bandwidth the client provides as there is nothing gained if the client does not have the bandwidth capacity to distribute data to other clients.

Client upload capacity: The upload capacity of the client has a direct impact on the number of clients which the client can stream data to. But if all clients provide enough bandwidth to at least stream data to one other client there would always be enough bandwidth to connect one more client. But again, if the clients does not provide a sufficiently large buffer, chances are that all new clients will be attached to the server as this will be the only one which can provide the needed data.

Video subdivision: The number of BBs the video is divided into has an effect on how many streams are needed by the server to distribute the video. This parameter arises from the design of the protocol, which determines that each client must have one of the BBs as a starting point. Again if a client has the specific BB in its buffer the client can stream to a new client even if its own PP is not at the exact same BB hence this is closely linked to the clients' buffer size.

CCP consumption: The amount of overhead used to send control packets between the server and each client makes it impossible to add an unlimited number of clients since all clients will need to communicate with the server. Thus, the bandwidth of the server will at some point be exhausted effectively blocking the connection of any new clients. The question is therefore how much bandwidth the CCP consumes. Measuring the individual sizes of the packets is of course possible but determining how many and how often these packets are sent is simply not possible. The reason is simply that the interval between consecutive sending of these packet is dependent upon the users behaviour.

User/network behaviour: This type of parameters is clearly impossible to estimate. One could argue that the users' behaviour changes depending on the type of video – indeed this is supported by [20] which notes that videos can be separated into different categories according to the behaviour of the user. Therefore estimating this would require a survey collecting and analysing data in order to determine this. The result of this survey would be based on averages and would therefore only be useful in large setups. Furthermore as the protocol is vulnerable to the amount of user interactions one could argue that based upon the result of this survey the used selection algorithm should be modified.

The network behaviour clearly have the same impact. Simulating or collecting statistics determining the behaviour of the Internet is a large and complicated subject which is a field of computer science in itself. Again the statistics and knowledge gathered in this area could be used to modify the selection algorithm to employ mechanisms to handle network problems.

The above descriptions have revealed the different parameters are tightly linked and none of them can be assigned a specific value which is more correct than any other.

Based upon the analysis of the behaviour of these parameters we can conclude that to set up formal calculations of the performance of the protocol is extremely difficult. Furthermore, if a formal calculation was made, this would be associated with much uncertainty as the behaviour of each user and their connection is almost unpredictable. However, the worst case and best case scenarios for the performance can be identified.

27.2 Boundaries of the performance

In this section we will based on the above discussed parameters try to determine the limits of the performance of the protocol by assigning values to these parameters.

27.2.1 Worst case scenario

Clearly the worst case scenario involves that all connecting clients do not have the ability to forward the video to other clients. Furthermore, if the clients do not have a sufficiently large buffer to hold at least a significant part of the video chances are that the clients will not even be able to utilize the bandwidth which is available. Thus, all channels dispatched by the server will only have one client attached each, resulting in the simple unicast model. Indeed this scenario represents worse performance than traditional unicasting as the inclusion of data transported by CCP will raise the bandwidth requirements of the server.

27.2.2 Best case scenario

The best case scenario involves connecting a client which provides a very high capacity and a buffer sufficient enough to hold the entire video. This client would then forward data to a number of clients which had the capacity of forwarding data on to the next client. If the clients furthermore have a buffer capacity of the whole video and none is performing any user interaction, the server would need to send data to only a single client. This scenario is of course unrealistic in practice but never the less it becomes the best case scenario.

27.2.3 Conclusion

Based on the above assertions we can conclude that the bandwidth usage can be lowered using our protocol as long as certain conditions are fulfilled. Most important the clients need to be in possession of a significant amount of bandwidth.

An important remark is that each time a client is connected a new CCP connection is needed. This connection is not taken into account in the above scenarios. This means that even though the best case scenario suggests that if all the connecting clients have at least capacity to forward the video an infinite number of clients could be connected. The only problem is that each client takes up a constant amount of the bandwidth of the server to maintain a CCP connection and hence the infinite number of clients does not hold in practice.

Chapter VI

Closure

This chapter contains our final remarks on the thesis. This includes the perspectives and the final conclusion.

28 Perspectives

To widen the aspects of the thesis, this section presents some of the perspectives of the designed protocol, its usage, and how this initial implementation can be extended.

It should be evident, that designing and implementing a protocol facilitating video on-demand using bandwidth sharing is a large project. Therefore the protocol design has gone through many ideas and extensions. Some of these ideas, although favourable, have been kept out of the thesis in order to impose a limit on the amount of work.

Server-to-server protocol: As described in Section 3 implementing a streaming network for on-demand video may easily require more than one centralized server. Thus, extending the protocol to also implement a server side protocol used between a number of servers which may balance the load equally and ensure that a video is available on the right server when requested by a client may prove to be of great value. This would furthermore benefit from scenarios where servers are geographically dispersed. Thus, a protocol enabling multiple servers to exchange data would bring higher efficiency. Furthermore a setup resembling the proxy system described in Section 7.1.2 may be deployed using a centralized server controlling a number of ”super clients” located at different places. These ”super clients” would only have to be slightly modified clients and should provide high bandwidth.

Security: One of the main obstacles for using a protocol which distributes video by the use of bandwidth sharing is securing copyright to the distributed material. But due to the problems incurred this field has been partially left out of the protocol. Therefore, one of the main perspectives of the protocol is to ensure digital rights management. As given in section 11 this field is a large subject, which is why this has been limited in this thesis. Implementing a proper security scheme in the server and client applications would surely remedy this and widen the perspectives of the protocol. Fortunately, this is already part of the protocol design.

Simulator: Even though the test and validation of the protocol have given good hints about the usefulness of the protocol a variety of other tests may be performed. In order to test the protocol with a large number of clients a simulator could be developed. The simulator could be built with the purpose of testing the main issues:

- The protocol's ability to connect and distribute video data to a large number of clients.
- The protocol's ability to handle network interruptions and other failures.
- The protocol's ability to handle user interactions like skip and pause.
- Test of the different selection algorithms to find the best suitable solution.

The protocol could be simulated without any regard to the network layer of the operating system. This would simulate a 'perfect' network, where no network interruptions or connection fluctuations would occur.

As the protocol does not define which ports are used by the clients, it is possible to run a large number of instances of the protocol on the same computer. Hence a simulator could use the implemented protocol by instantiating a server and a number of clients locally. Next, the instances of the client protocols could connect locally to the server and each perform various actions to test the performance of the protocol.

Complete implementation: A key point of the future work of the protocol obviously is to implement all the features of the design. This will serve to rectify all functionalities. Furthermore, the protocol could be implemented using other programming languages such as Java opening up for the implementation of different client and server applications.

Extending the selection algorithm: The selection algorithm used to select the set of clients which are to stream data to a new or relocated client could be implemented in a number of ways. Varying this implementation may result in many performance gains as described in Section 22. Therefore the implementation of the algorithm can be made very complex and may depend on a wide range of parameters. One could for instance add weights to the edges of the network and use these to determine how the topology should be built. These weights could be based on the clients' upload capacity or their physical location. Another parameter could be a credit system which assigned each client a number of points which would then be used by the algorithm. This could be used commercially by assigning points to customers according to the amount of money they had paid to see the video. Furthermore, the more bandwidth the customer provided, the more points would be given. As an illustration of the almost endless possibilities in constructing an efficient selection algorithm we present three approaches which present different paradigms when constructing the topology which we have worked with throughout this thesis:

Dense network: This algorithm will seek to construct the network as densely as possible. This means that when a client logs on, the server will attach the client to a client which is already streaming to other clients. Thus, the algorithm tries to attach as many clients as possible to clients already streaming data. The main idea when using this approach is to utilize all of the bandwidth present at some client, before extending to another client. Thus, if a client providing high bandwidth connects, the protocol will seek to use all the bandwidth of the client before another client is used.

Sparse network: This algorithm works opposite a dense network, aiming to construct a sparse topology where all clients, regardless of their provided bandwidth use a part of their bandwidth. Thus, the network will be shaped as a sparse graph aiming to distribute resources as evenly as possible.

Reliability: This algorithm will work based on the statistics gathered by the server. The server will try to determine which client is the most stable, in terms of error occurrence rate, user interaction, etc.

Applications: Even though the applications have not been the focus of this thesis some issues need to be addressed if one should wish to implement new applications. The client application could of course be built in a variety of ways and will not be subject to any further discussion. The server application on the other hand has one key element which should be addressed if one wishes to implement a strong application. This is the way data is fetched from disc. The protocol facilitates that the server application can be implemented with an advanced algorithm when retrieving data. Streamed data is obviously sequential and this property could be used in implementing algorithms which for instance prefetch data from disc and keep it ready in a buffer pool ready to hand it to the protocol upon request.

The fetching of data from disc could become a bottle neck and therefore considerations have been put into the protocol design in order to make it possible to build more advanced server applications.

29 Conclusion

The main goal of the thesis was to lower the amount of consumed bandwidth of a central on-demand video streaming server by designing and implementing a network protocol for this purpose. The result is a comprehensive design specification which enables the development of heterogenous implementations operating alongside each other in a network. The specification should further be considered as an evidence of the large amount of considerations and discussions which have gone into the final design of the protocol.

The protocol has been designed to enable delivery of a stable stream of data guaranteeing that a video stream is delivered without disruptions. Furthermore, the protocol has the ability to handle partial network disruptions ensuring that the delivery of data is not affected. In addition, the protocol enables user interactions such as skipping and pausing which is exactly the characteristics of video on-demand. As these properties were among the primary goals of the thesis we can conclude that we have designed a protocol which fulfills the expectations.

The amount of security mechanisms built into the protocol has been limited to only granting or denying clients access. The protocol has been designed to enable applications using the protocol to employ security measures as needed. Thus, the applications may ensure that clients cannot change the content of the data stream before it is forwarded to another client and ensure that clients cannot receive video data without being acknowledged by the server.

The implementation of the protocol has been carried out with the primary focus of illustrating the core functionality of the protocol design. Thus, the implementation is meant as a proof of concept which verifies that the protocol design works as intended. The implementation defines an interface which enables applications to utilize the embedded functionality of the protocol. The protocol implementation may be ported to other platforms though not as straightforward as intended because of the diversities of the implementations of C++ in different operating systems.

An analysis of the designed protocol has shown that distribution of the overall bandwidth consumption is possible. Thus, we have successfully minimized the bandwidth consumption of the server hence making it possible to maximize the number of clients which can concurrently receive the video stream. The thesis has shown that the ability to undertake a calculation of the conserved bandwidth is extremely difficult. Estimates can be performed, but these will be based upon several unknown variables associated with high inaccuracy caused by network reliability and human behaviour. Therefore no exact calculation proving the efficiency of the protocol has been made. However, a firm understanding of the relationship between these unknowns has been accounted for. The conclusion on this subject is that to get a more precise indication of the efficiency of the protocol a simulation and/or empirical results obtained through practical usage must be performed.

In spite of the challenges experienced throughout the implementation of a multithreaded and memory intensive protocol we have constructed an operational implementation which demonstrates all the key aspects of the design. Through a series of test scenarios, we have proven that this implementation works as intended.

On a more personal level we have learned that the completion of this thesis has covered many subjects within computer science. Thus, one of the most difficult subjects has been to study and combine these. The development of the thesis has clearly shown that a substantial amount of further development could be put into our work. However, we have succeeded in producing a solid foundation even though it has been difficult to set the right limitations and at the same time produce a complete thesis. Furthermore, our initial beliefs concerning clients' ability and willingness to share bandwidth may be doubtful. Hence a stronger use of servers located at advanced points in the network may be feasible. However, the developed protocol can easily be modified to support this method and the knowledge acquired through this thesis still applies.

At last we can conclude that we have succeeded in designing and implementing a network protocol which largely fulfills the main goals of this thesis as was initially set in the synopsis. Thus, we have succeeded in constructing an alternative to the simple unicast model offering all the functionalities expected from a video on-demand system.

Glossary

A

Application Programming Interface (API): An API defines the interface that software provides in order to allow requests for functionality implemented in the software to be made by other computer programs.

Anti-social: A client is regarded anti-social if at least one of two things apply. First, if a client does not have the bandwidth capacity to send at least one whole video. Secondly, if a client has a connect area (CA) which is less then the size advertised by the server.

Application processes: A process which resides in the application holding an instance of either the server or client protocols.

Asynchronous bandwidth: Is defined as bandwidth where the maximum upstream differs from maximum downstream. Thus, a client could have a maximum downstream bandwidth of 2048 Kbit/sec but only 512 Kbit/sec upstream – a typical specification of todays ADSL lines. As downstream speed is often assigned higher priority than upstream speed Internet users will often have higher downstream speed at their disposal.

Asynchronous connection: An Internet connection equipped with asynchronous bandwidth.

B

Backbone providers: Companies which deliver networks connections used by Internet Service Providers (ISP).

Bandwidth: There is no single universal definition of bandwidth but normally the term is used as a measurement of a frequency range. More recently the word has been used when describing the capacity in networks (and the Internet) where it is used to denote the amount of data which can be transferred through a connection.

Bandwidth sharing: This term refers to a type of network where the bandwidth of the users is a shared resource.

Big endian: Refers to a way of storing multi-byte values in memory where the most significant bit is located in the high order byte in terms of memory address space.

Bit-rate: Is the number of bits which is consumed during a given interval. In the case of digital video it defines the amount of data consumed per second while displaying the video.

Best-effort: This term refers to the quality of service of an Internet connection. Basically this means that the connection will try to send as much data as possible as fast as possible.

C

Client application: Refers to an application which holds an instance of the client protocol.

Client protocol: The client side of the protocol or a specific instance of the protocol in client mode.

Connect area (CA): The area of the clients buffer where new clients can be attached, also know as CA.

Connectionless: The term is used about protocols which do not require a connection to be established before data can be transmitted. A classic example is UDP.

Connection-oriented: Opposite of connection-less. A classic example is TCP.

Constant bit-rate: In accordance to video the term means that the bit-rate of the video is constant. Abbreviation: CBR.

Critical state: The client enters a critical state if the Absorption Area (AA) of the buffer is not completely filled with relevant data.

D

Decoder: Referring to video, decoding is the process of converting data to its original format.

De-facto standard De facto is Latin for "in fact" or "in practice". A de facto standard is a technical standard that is so dominant that everybody follows it as an authorized standard.

E

Encoder: Opposite of decoder.

F

Frame rate: The rate at which a video is displayed e.g. the number of frames per second.

Frames: A video is build up of a series of still images called frames.

H

HDTV: Is short for High-Definition TeleVision. HDTV offers significantly higher resolution than traditional formats like NTSC and PAL.

Header: In computer networks the header of a packet is a sort of control data which is placed in the beginning of the packet. The header may contain information about how the packet should be handled.

Hop: One step, from one node to the next, on the path of a packet on an Internet Protocol (IP) network.

Hosts: A network attached device which communicates over the network. In this thesis it refers to both a client or a server.

Host byte-order: Refers to the byte-order (big endian or little endian) of a given host. Whether this is big endian or little endian is decided by the architecture of the host.

I

IP-address: An IP address (Internet Protocol address) is a unique number that is used by hosts on networks using the Internet Protocol (IP).

L

Latency: The term refers to the delay experienced in network communication.

Little endian: Refers to a way of storing multi-byte values in memory where the most significant bit is located in the low order byte in terms of memory address space.

N

Network byte-order: Refers to the byte-order (big endian or little endian) used in a network. Upon the Internet the used byte-order is big endian.

Non-sequential: The term is used in this thesis regarding the circular buffer of the client. Data is said to be sequential if data contained in the buffer is sequential in terms of the order of which the BBs are placed in the buffer from the Play Pointer (PP) and back to a specified point. Non-sequential is the opposite and will occur when then client has performed a skip outside it buffer area.

Node: Refers to an entity on the Internet. This may be either a computer, router, switch, or other equipment along an Internet path.

O

Out-of-order: In relation to network communications this term is used when data is received in a different order than the one in which it was sent.

P

Packet: In computer networks a packet is a formatted block of data which is transferred over the network.

Payload: The payload is the data part of a packet which is being transported by the layer resting on top of the protocol.

Peer-to-peer: A type of network in which each host has equivalent capabilities and responsibilities. This differs from the client/server architectures, where some computers are dedicated to serving others. Abbreviation: P2P.

Play block: This term is used to describe the Buffer Block in the clients buffer from where the client is currently viewing the video.

Port: In the TCP and UDP transport protocols a port number is used to multiplex between different applications. Thus, the pair (ip-address, port) is used to identify an application upon the Internet.

Protocol: In computers, a protocol is a convention on how data is transferred between two hosts on a shared network. This convention can besides the data transfer, include how the connection is established, synchronization of the connection etc.

Protocol stack: The term protocol stack refers to a collection of protocols 'packaged' in a stack to offer easy interchangeability of the individual protocols of the stack.

Q

Quality of Service: In the field of computer networking, the term Quality of Service (QoS) refers to the probability of the network meeting a given contract. In many cases it is used to refer to the probability of a packet succeeding in passing through the network within a desired latency period.

R

Receive block: The receive block is the Buffer Block (BB) in the data buffer of the client which holds the current buffer block being received.

Reliable byte stream: The term is used in connection with network protocols if the protocol implements mechanisms which ensures that the bytes which emerge from the communication channel at the recipient are exactly the same, and in the exact same order, as the sender inserted them into the channel.

S

Server application: Refers to an application which holds an instance of the server side protocol.

Server protocol: The server side of the protocol or a specific instance of the protocol in server mode.

Session: The term session is used throughout this thesis regarding the time from where a client connects until it disconnects. This interval is referred to as the clients session.

Streaming: In computer networking, a stream is succession of data elements made available over time. The term video streaming is referred to a method of transferring data at the speed at which it is consumed.

T

Three-way handshake: This term is used for the way some protocols create a connection. First the connecting host sends a connect request and the counterpart sends a connect respond back and finally the connection client sends an acknowledgment back.

Traffic shaping: This term refers to a way of attempting to control computer network traffic in order to guarantee performance, latency, and bandwidth.

Threads: Threads are a method of dividing software into several simultaneously running tasks sharing the same resources. Although threads are said to run simultaneously, they are run in a round-robin like manner assigned only a portion of execution time using time slices.

U

Unix time: Unix time, or POSIX time is the number of seconds elapsed since midnight UTC on the morning of January 1, 1970.

V

Variable bit-rate: In accordance to digital video the term means that the bit-rate of the video is variable from frame to frame. Abbreviation: VBR.

Video standard: A standard which describes how frames and audio are structured and combined into a video.

Bibliography

[1] BitTorrent, Inc., Bittorrent, Website accessible at `http://www.bittorrent.com` (2006).

[2] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss, *An Architecture for Differentiated Service*, RFC 2475 (Informational) (1998). Updated by RFC 3260.

[3] Y. Cai and K. A. Hua, An efficient bandwidth-sharing technique for true video on demand systems, *Proceedings of the Seventh ACM International Conference on Multimedia (Part 1)*, ACM Press, New York, USA (1999), 211–214.

[4] P. C. Chapin, pthread tutorial (2005).

[5] F. Cores, A. Ripoll, and E. Luque, A fully scalable and distributed architecture for video-on-demand, *PROMS 2001: Proceedings of the 6th International Conference on Protocols for Multimedia Systems*, Springer-Verlag, London, UK (2001), 59–73.

[6] Y. Cui and K. Nahrstedt, Layered peer-to-peer streaming, *NOSSDAV '03: Proceedings of the 13th international workshop on Network and operating systems support for digital audio and video*, ACM Press, New York, USA (2003), 162–171.

[7] J. de Fine Skibsted, S. Lynge, and et al., *eXstream – High performance multimedia streaming server* (2005).

[8] eDonkey, edonkey2000 – overnet, Website accessible at `http://www.edonkey.com` (2006).

[9] Gnutella file sharing and distribution network, Gnutella protocol development, Worldwide Web Document (2006). Available at `http://rfc-gnutella.sourceforge.net/developer/stable/index.html`.

[10] Gnutella, Gnutella.org, Website accessible at `http://www.gnutella.org` (2006).

[11] Y. Guo, K. Suh, J. Kurose, and D. Towsley, P2cast: peer-to-peer patching scheme for vod service, *WWW '03: Proceedings of the 12th international conference on World Wide Web*, ACM Press, New York, USA (2003), 301–309.

[12] F. Halsall, *Multimedia Communication*, Addison-Wesley (2001).

[13] M. Hefeeda, A. Habib, B. Botev, D. Xu, and B. Bhargava, Promise: peer-to-peer media streaming using collectcast, *MULTIMEDIA '03: Proceedings of the eleventh ACM international conference on Multimedia*, ACM Press, New York, USA (2003), 45–54.

[14] M. Hefeeda, A. Habib, B. Botev, D. Xu, and B. Bhargava, Promise: peer-to-peer media streaming using collectcast, *Proceedings of the eleventh ACM international conference on Multimedia*, ACM Press, New York, USA (2003), 45–54.

[15] M. Hofmann and L. R. Beaumont, *Content Networking: Architecture, Protocols, and Practice*, First Edition, Morgan Kaumann (2005).

[16] K. A. Hua, Y. Cai, and S. Sheu, Patching: a multicast technique for true video-on-demand services, *Proceedings of the sixth ACM international conference on Multimedia*, ACM Press, New York, USA (1998), 191–200.

[17] Y. hua Chu, J. Chuang, and H. Zhang, A case for taxation in peer-to-peer streaming broadcast, *PINS '04: Proceedings of the ACM SIGCOMM workshop on Practice and theory of incentives in networked systems*, ACM Press, New York, USA (2004), 205–212.

[18] MIT Kerberos, Kerberos: The network authentication protocol, Website accessible at `http://web.mit.edu/kerberos/` (2006).

[19] S. H. Lee, K. Y. Whang, Y. S. Moon, and I. Y. Song, Dynamic buffer allocation in video-on-demand systems, *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, ACM Press, New York (2001), 343–354.

[20] F. C. Li, A. Gupta, E. Sanocki, L. wei He, and Y. Rui, Browsing digital video, *CHI '00: Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM Press, New York, USA (2000), 169–176.

[21] T. Liu and S. Nelakuditi, Disruption-tolerant content-aware video streaming, *Proceedings of the 12th Annual ACM International Conference on Multimedia*, ACM Press, New York (2004), 420–423.

[22] C. Logg, L. Cottrell, and J. Navratil, Experiences in traceroute and available bandwidth change analysis, *NetT '04: Proceedings of the ACM SIGCOMM workshop on Network troubleshooting*, ACM Press, New York, USA (2004), 247–252.

[23] D. Loguinov, A. Kumar, V. Rai, and S. Ganesh, Graph-theoretic analysis of structured peer-to-peer systems: routing distances and fault resilience, *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ACM Press, New York, USA (2003), 395–406.

[24] H. Ma and K. G. Shin, Multicast video-on-demand services, *SIGCOMM Comput. Commun. Rev.* **32**,1 (2002), 31–43.

[25] H. Ma and K. G. Shin, Multicast video-on-demand services, *SIGCOMM Comput. Commun. Rev.* **32**,1 (2002), 31–43.

[26] F. Mittelbach and M. Goossens, *The LaTeXCompanion*, Second Edition, Addison-Wesley (2004).

[27] Moving Picture Experts Group (MPEG), MPEG home page, Website accessible at `http://www.chiariglione.org/mpeg/` (2006).

[28] Napster, Napster.com, Website accessible at `http://www.napster.com` (2006).

[29] Y. Okada, M. Oguro, J. Katto, and S. Okubo, A new approach for the construction of alm trees using layered video coding, *Proceedings of the ACM workshop on Advances in Peer-to-Peer Multimedia Streaming*, ACM Press, New York, USA (2005), 59–68.

[30] W. B. Pennebaker and J. L. Mitchell, *JPEG Still image data compression standard*, Van Nostrand Reinhold (1993).

[31] L. L. Peterson and B. S. Davie, *Computer Networks*, Second Edition, Morgan Kaufmann (2000).

[32] J. Postel, *RFC 768: User Datagram Protocol* (1980).

[33] J. Postel, *RFC 791: Internet Protocol* (1981).

[34] J. Postel, *RFC 793: Transmission Control Protocol* (1981).

[35] M. Rocha, M. Maia, I. Cunha, J. Almeida, and S. Campos, Scalable media streaming to interactive users, *Proceedings of the 13th annual ACM international conference on Multimedia*, ACM Press, New York, USA (2005), 966–975.

[36] W. Shi and S. Ghandeharizadeh, Buffer sharing in video-on-demand servers, *SIGMETRICS Perform. Eval. Rev.* **25**,2 (1997), 13–20.

[37] R. Sinha and C. Papadopoulos, An adaptive multiple retransmission technique for continuous media streams, *NOSSDAV '04: Proceedings of the 14th international workshop on Network and operating systems support for digital audio and video*, ACM Press, New York, USA (2004), 16–21.

[38] W. R. Stevens, B. Fenner, and A. M. Rudoff, *UNIX Network Programming: The Sockets Networking API*, Third Edition, Addison-Wesley (2004).

[39] W. R. Stevens, B. Fenner, and A. M. Rudoff, *UNIX Network Programming: The Sockets Networking API*, Third Edition, Addison-Wesley (2004).

[40] A. S. Tanenbaum, *Computer Networks*, Third Edition, Addison-Wesley (1996).

[41] M. A. Tantaoui, K. A. Hua, and S. Sheu, Interaction with broadcast video, *Proceedings of the tenth ACM international conference on Multimedia*, ACM Press, New York, USA (2002), 29–38.

[42] D. A. Tran, K. A. Hua, and T. T. Do, Scalable media streaming in large peer-to-peer networks, *Proceedings of the tenth ACM international conference on Multimedia*, ACM Press, New York, USA (2002), 247–250.

[43] B. Wang, J. Kurose, P. Shenoy, and D. Towsley, Multimedia streaming via tcp: an analytic performance study, *Proceedings of the 12th annual ACM international conference on Multimedia*, ACM Press, New York, USA (2004), 908–915.

[44] B. Wang, J. Kurose, P. Shenoy, and D. Towsley, Multimedia streaming via tcp: an analytic performance study, *Proceedings of the 12th annual ACM international conference on Multimedia*, ACM Press, New York, USA (2004), 908–915.

[45] K. Wittenburg, C. Forlines, T. Lanning, A. Esenther, S. Harada, and T. Miyachi, Rapid serial visual presentation techniques for consumer digital video devices, *UIST '03: Proceedings of the 16th annual ACM symposium on User interface software and technology*, ACM Press, New York, USA (2003), 115–124.

[46] C. Wu and B. Li, Optimal peer selection for minimum-delay peer-to-peer streaming with rateless codes, *Proceedings of the ACM Workshop on Advances in Peer-to-Peer Multimedia Streaming*, ACM Press, New York (2005), 69–78.

[47] C. Wu and B. Li, Optimal peer selection for minimum-delay peer-to-peer streaming with rateless codes, *P2PMMS'05: Proceedings of the ACM workshop on Advances in peer-to-peer multimedia streaming*, ACM Press, New York, USA (2005), 69–78.

[48] M. Zhang, L. Zhao, Y. Tang, J. G. Luo, and S. Q. Yang, Large-scale live media streaming over peer-to-peer networks through global internet, *Proceedings of the ACM Workshop on Advances in Peer-to-Peer Multimedia Streaming*, ACM Press, New York (2005), 21–28.

[49] Y. Zhang and N. Duffield, On the constancy of internet path properties, *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, ACM Press, New York, USA (2001), 197–211.

Appendix A

Packet table

Packet name	Protocol	Short name	⟨P type,A type⟩	Presented in
Connection granted	CCP	CONN_GRANTED	⟨10,10⟩	17.2.2 (p. 104)
Connection denied	CCP	CONN_DENIED	⟨10,20⟩	17.2.3 (p. 106)
Connection closed	CCP	CONN_CLOSED	⟨10,30⟩	17.2.4 (p. 107)
Receive data stream	CCP	RECV_DATA_STREAM	⟨10,50⟩	17.3.1 (p. 109)
Security data	CCP	SEC_DATA	⟨10,60⟩	17.8.1 (p. 125)
Request status	CCP	STATUS_REQ	⟨20,10⟩	17.6.1 (p. 121)
Distribute data	CCP	SND_DATA_STREAM	⟨20,20⟩	17.3.2 (p. 110)
Stop data distribution	CCP	STOP_STREAM	⟨20,30⟩	17.3.3 (p. 111)
Status respond	CCP	STATUS_RESP	⟨30,10⟩	17.6.2 (p. 122)
Request connect	CCP	CONN_REQ	⟨40,10⟩	17.2.1 (p. 103)
Pause video player	CCP	INTERACT_PAUSE	⟨40,30⟩	17.5.1 (p. 118)
Pause stream	CCP	INTERACT_PAUSE_STREAM	⟨40,40⟩	17.5.2 (p. 119)
Resume stream	CCP	INTERACT_RESUME	⟨40,50⟩	17.5.3 (p. 119)
Skip	CCP	INTERACT_SKIP	⟨40,60⟩	17.5.4 (p. 120)
Disconnect	CCP	DISCONN	⟨40,70⟩	17.2.5 (p. 107)
Error receiving data	CCP	STREAM_ERROR	⟨40,80⟩	17.4.6 (p. 116)
Request backup data	CCP	REQ_BACKUP_DATA	⟨40,100⟩	17.4.7 (p. 117)
Data	DCP	DATA	⟨50,10⟩	17.4.1 (p. 112)
Data not available	DCP	NO_DATA	⟨50,20⟩	17.4.2 (p. 113)
End of data	DCP	NO_MORE_DATA	⟨50,30⟩	17.4.3 (p. 114)
Pong	DCP	PONG	⟨50,40⟩	17.7.2 (p. 124)
Request data resend	DCP	RESEND_DATA	⟨60,10⟩	17.4.4 (p. 115)
Adjust speed	DCP	ADJUST_SPEED	⟨60,20⟩	17.4.5 (p. 115)
Ping	DCP	PING	⟨60,30⟩	17.7.1 (p. 124)

Appendix B

Protocol source files

data_bank_client.h

```
#ifndef DATA_BANK_CLIENT
#define DATA_BANK_CLIENT

#include "../include/packet.h"
#include "../include/packet_recv_data_stream.h"
#include "../include/packet_snd_data_stream.h"
#include "../include/packet_stop_stream.h"
#include "../include/stream_table.h"
#include "../include/packet_conn_granted.h"

#include "../include/types.h"
#include "../include/enum.h"

class data_bank_client {
  bool v_initialized;

  // Packet containing video information regarding bitrate, bb size, etc.
  packet_conn_granted* v_video_info_pack;

  // Packet containing info about the nature of from who, where and when
  // video data is received.
  packet_recv_data_stream* v_recv_table;

  // Table containing info about to who, and when we must send data
  stream_table* v_streams;

 public:
  data_bank_client();
  ~data_bank_client();

  // Initialize the class
  void initialize(packet_conn_granted* pack);

  // Set receive table. Sets the packet containint
  // information regarding the contents of the ingoing
  // data stream.
  void set_recv_table(packet_recv_data_stream* pack);

  // Add a stream to the data bank
  void add_stream(packet_snd_data_stream* pack);
```

```
  // Stop a stream
  void stop_stream(packet_stop_stream* pack);

  // Resend a block to a client
  void resend_block(u64 client_id, u16 BB, u16 DB);

  // Adjust speed
  void adjust_speed(u64 client_id, speed_level level);

  // Get the next receiver in line
  bool get_next_receiver(u64 video_id,
             u64& client_id,
                         int& wait_time,
             u16& next_bb,
             u16& next_db,
               speed_level& level,
              u64& ip_addr,
              u16& port);

  // Get the length of the video in milliseconds
  u64 get_video_duration();

  // Get the length of the video in bytes
  u64 get_video_length();

  // Get the skip distance of the video
  u64 get_skip_distance();

  // Get the total number of db in the video
  u32 get_num_of_bb();
};

#endif
```

data_bank_client.cpp

```
#include "stdafx.h"
#include "../include/data_bank_client.h"

//**********************************************
// Constructor
//**********************************************
data_bank_client::data_bank_client()
{
  v_initialized = false;
  v_streams = new stream_table();
  v_video_info_pack = NULL;
  v_recv_table = NULL;
}


//**********************************************
// destructor
//**********************************************
```

```
data_bank_client::~data_bank_client()
{
  delete v_video_info_pack;
  delete v_recv_table;
  delete v_streams;
}

//**********************************************
// Initialize
//**********************************************
void data_bank_client::initialize(packet_conn_granted* pack)
{
  if (!v_initialized) {
    // Initialize the stream_table. Do we need the pack information in the
    // stream_table element?
    v_streams->initialize(pack);
    v_video_info_pack = pack;
    v_initialized = true;
  }
  else throw new data_bank_exception("Error: databank already initialized!");

}

//**********************************************
// Set receive table
//**********************************************
void data_bank_client::set_recv_table(packet_recv_data_stream* pack)
{
  if (v_initialized) {
    v_recv_table = pack;
  }
  else throw new data_bank_exception("Error: databank accessed before it was initialized!");

}

//**********************************************
// Add a stream to the data bank
//**********************************************
void data_bank_client::add_stream(packet_snd_data_stream* pack)
{
  if (v_initialized) {
    try {
      int fac = (int) ((v_video_info_pack->get_db_total() /
                        v_video_info_pack->get_bb_total()) / pack->get_number_of_db());
      v_streams->add_stream(pack, v_video_info_pack->get_speed_low() * fac,
                            v_video_info_pack->get_speed_normal() * fac,
                            v_video_info_pack->get_speed_high() * fac);

    }
    catch (stream_table_exception* ex) {
      throw ex;
    }

  }
  else
    throw new data_bank_exception("Error: databank accessed before it was initialized!");
 }
```

```
//*********************************************
// Stop stream
//*********************************************
void data_bank_client::stop_stream(packet_stop_stream* pack)
{
  if (v_initialized) {
    v_streams->stop_stream(pack);
  }
  else
    throw new data_bank_exception("Error: databank accessed before it was initialized!");
}


//*********************************************
// Resend block to client
//*********************************************
void data_bank_client::resend_block(u64 client_id, u16 BB, u16 DB)
{
  if (v_initialized) {
    v_streams->resend_block(client_id, BB, DB);
  }
  else
    throw new data_bank_exception("Error: databank accessed before it was initialized!");
}


//*********************************************
// Adjust speed of a client
//*********************************************
void data_bank_client::adjust_speed(u64 client_id, speed_level level)
{
  if (v_initialized) {
    v_streams->adjust_speed(client_id, level);
  }
  else
    throw new data_bank_exception("Error: databank accessed before it was initialized!");
}


//*********************************************
// Get the next receiver
//*********************************************
bool data_bank_client::get_next_receiver(u64 video_id,
 u64& client_id,
 int& wait_time,
 u16& next_bb,
 u16& next_db,
 speed_level& level,
 u64& ip_addr,
 u16& port)
{

  if (v_initialized) {
    return v_streams->get_next_receiver(client_id,
                                        wait_time,
                                        next_bb,
```

```
                                        next_db,
                                        level,
                                        ip_addr,
                                        port);
  }
  else
    throw new data_bank_exception("Error: databank accessed before it was initialized!");
}


//**********************************************
// Return the duration of the video
//**********************************************
u64 data_bank_client::get_video_duration()
{
  return v_video_info_pack->get_video_duration();
}


//**********************************************
// Return the length of the video in bytes
//**********************************************
u64 data_bank_client::get_video_length()
{
  return v_video_info_pack->get_video_size();
}


//**********************************************
// Return the skip distance of the video
//**********************************************
u64 data_bank_client::get_skip_distance()
{
  return v_video_info_pack->get_skip_distance();
}

//*****************************************************
// get the number of data blocks  in total
//*****************************************************
u32 data_bank_client::get_num_of_bb()
{
  return v_video_info_pack->get_bb_total();
}
```

data_bank_server.h

```
#ifndef DATA_BANK_SERVER
#define DATA_BANK_SERVER

//#include "../include/packet_video_init_resp.h"
#include "../include/stream_table.h"

#include "../include/types.h"
```

```
#include "../include/enum.h"

#include "../include/packet_status_resp.h"
#include "../include/packet_conn_req.h"
#include "../include/client_info.h"
#include "../include/packet_interact_skip.h"
#include <map>

// Constants
const u16 C_BB_TOTAL = 45 ;
const u16 C_DB_PER_BB = 100;
const u64 C_VIDEO_SIZE = 20690708; // bytes
const int C_VIDEO_DURATION = 137000; // msec

const u32 C_DB_TOTAL = C_BB_TOTAL * C_DB_PER_BB;
const u16 C_DB_SIZE = 4598; // byte
const u32 C_BB_SIZE = C_DB_SIZE * C_DB_PER_BB; // byte
const int C_BB_DURATION = C_VIDEO_DURATION / C_BB_TOTAL; // msec

const int C_VIDEO_HEADER_SIZE = 2; // amount of db's

const int C_RECOM_CA_SIZE = 45; // amount of bb's
const int C_SPEED_LOW = 40; // msec
const int C_SPEED_NORMAL = 30; // msec
const int C_SPEED_HIGH = 25; // msec

typedef std::map<u64, client_info*>::const_iterator iterator_client;

class data_bank_server
{
private:
  bool v_initialized;
  // Private variable defining to who and when the server should stream data
  stream_table* v_streams;

  // Map containing all logged in client
  std::map<u64, client_info*> v_clients;

public:
  // Constructor
  data_bank_server();
  // Destructor
  ~data_bank_server();

  // Initialize the class
  void initialize();

  // Get the next receiver in-line
  bool get_next_receiver(u64 video_id,
             u64& client_id,
                         int& wait_time,
             u16& next_bb,
             u16& next_db,
             speed_level& level,
             u64& ip_addr,
             u16& port);
```

```
  u64 get_server_id(u64 client_id);
  u64 get_video_id();

  // Fill conn_granted packet with info related to video
  void get_movie_info(u64 get_video_id, packet_conn_granted* p);

  // Update the status of a client given in the packet
  void update_status(packet_status_resp* pack);

  bool add_streams(u16* db_list, u64 video_id);

  // Add new stream to internal stream table
  bool add_new_stream(packet_snd_data_stream* pack);
  // Remove a stream from data bank
  void remove_stream(u64 client_id);
  // Update the contents of a given stream
  bool update_stream(packet_snd_data_stream* pack);

  // Ad newe client to internal client table
  bool add_new_client(packet_conn_req* pack);
  // Remove a client from data bank
  void remove_client(u64 client_id);

  // Set client posotion
  bool skip_stream(u64 client_id, u16 BB);

  // Find client which can attach a client into DB, BB
  bool find_clients(u16 BB, u16 DB, u64*& clients, int& client_length);

  // Get the ip address of a given client id
  u64 get_ip_address(u64 client_id) ;

  // Get the dcp port of a given client id
  u16 get_dcp_port(u64 client_id);

  // Attach a stream to some client
  void add_stream_to_client(u64 client_id, packet_snd_data_stream* pack);

  // Get attached clients of a given client
  packet_snd_data_stream* get_attached_clients(u64 client_id);

  // Get the current status of a client
  packet_status_resp* get_client_status(u64 client_id);

  u16 get_num_of_bb(u64 video_id);
  u32 get_size_of_bb(u64 video_id);
  u32 get_num_of_db(u64 video_id);
  u16 get_size_of_db(u64 video_id);
  u64 get_video_size(u64 video_id);
  u16 get_num_of_db_in_bb(u64 video_id);

};

#endif
```

data_bank_server.cpp

```
#include "stdafx.h"
#include "../include/data_bank_server.h"

//**********************************************
// Constructor
//**********************************************
data_bank_server::data_bank_server()
{
  v_streams = new stream_table();
  v_streams->initialize(NULL);
  v_initialized = true;
}

//**********************************************
// destructor
//**********************************************
data_bank_server::~data_bank_server()
{

}

//**********************************************
// Initialize
//**********************************************
void data_bank_server::initialize()
{
  return;
}

//**********************************************
// Get the next receiver in-line
//**********************************************
bool data_bank_server::get_next_receiver(u64 video_id,
                             u64& client_id,
                                     int& wait_time,
                             u16& next_bb,
                             u16& next_db,
                             speed_level& level,
                             u64& ip_addr,
                             u16& port)
{
  return v_streams->get_next_receiver(client_id,
                                      wait_time,
                                      next_bb,
                                      next_db,
                                      level,
                                      ip_addr,
                                      port);
}


//**********************************************
// Return dummy server-id
//**********************************************
```

```
u64 data_bank_server::get_server_id(u64 client_id)
{
  return 0;
}

//**********************************************
// Return dummy video id
//**********************************************
u64 data_bank_server::get_video_id()
{
  return 0;
}

//**********************************************
// Get movieinfo. Write data to packet.
//**********************************************
void data_bank_server::get_movie_info(u64 get_video_id, packet_conn_granted* p)
{
  //p->set_video_id(1);
  p->set_video_size(C_VIDEO_SIZE);
  //p->set_video_len(C_VIDEO_DURATION);
  p->set_bb_total(C_BB_TOTAL);
  p->set_db_total(C_DB_TOTAL);
  p->set_db_size(C_DB_SIZE);
  p->set_video_header_size(C_VIDEO_HEADER_SIZE);
  p->set_recon_ca_size(C_RECOM_CA_SIZE);
  p->set_speed_low(C_SPEED_LOW);
  p->set_speed_normal(C_SPEED_NORMAL);
  p->set_speed_high(C_SPEED_HIGH);
  p->set_sec_type(0);
  p->set_skip_distance(C_BB_DURATION);
//  p->set_video_title(NULL);
 }

//**********************************************
// Add_streams
//**********************************************
bool data_bank_server::add_streams(u16* db_list, u64 video_id)
{
  return true;
}

//**********************************************
// Update status of a given client
//**********************************************
void data_bank_server::update_status(packet_status_resp* pack)
{
  client_info* client = v_clients[pack->client_id];
  client->update_status(pack);
}

//*****************************************************
// Add new client.
//*****************************************************
bool data_bank_server::add_new_client(packet_conn_req* pack)
{
  client_info* client = new client_info(pack);
```

```
  v_clients[pack->client_id] = client;
  return true;
}

//****************************************************
// Add new client which should receive data from server.
//****************************************************
void data_bank_server::remove_client(u64 client_id)
{
  client_info* ci = v_clients[client_id];
  if(ci == NULL) {
    return;
  }
  else {
    v_clients.erase(client_id);
    delete ci;
  }
  return;
}

//****************************************************
// Add new client which should receive data from server.
//****************************************************
bool data_bank_server::add_new_stream(packet_snd_data_stream* pack)
{
 v_streams->add_stream(pack, C_SPEED_LOW, C_SPEED_NORMAL, C_SPEED_HIGH);
  return true;
}

//****************************************************
// delete client which are receiving data from server.
//****************************************************
void data_bank_server::remove_stream(u64 client_id)
{
  v_streams->remove_stream(client_id);
}

//****************************************************
// Update the propeties of a stream
//****************************************************
bool data_bank_server::update_stream(packet_snd_data_stream* pack)
{
  return v_streams->update_stream(pack);
}

//****************************************************
// Set the position of a given stream
//****************************************************
bool data_bank_server::skip_stream(u64 client_id, u16 BB)
{
  return v_streams->skip(client_id, BB);
}

//****************************************************
// Add new client which should receive data from server.
//****************************************************
u16 data_bank_server::get_num_of_bb(u64 video_id)
```

```
{
  return C_BB_TOTAL;
}

//****************************************************
//  get the size of a bb
//****************************************************
u32 data_bank_server::get_size_of_bb(u64 video_id)
{
  return C_BB_SIZE;
}

//****************************************************
// get the number of data blocks per in total
//****************************************************
u32 data_bank_server::get_num_of_db(u64 video_id)
{
  return C_DB_TOTAL;
}

//****************************************************
// get the size of a db
//****************************************************
u16 data_bank_server::get_size_of_db(u64 video_id)
{
  return C_DB_SIZE;
}

//****************************************************
// get the total length of the video
//****************************************************
u64 data_bank_server::get_video_size(u64 video_id)
{
  return C_VIDEO_SIZE;
}

//****************************************************
// get the total length of the video
//****************************************************
u16 data_bank_server::get_num_of_db_in_bb(u64 video_id)
{
  return C_DB_PER_BB;
}

//****************************************************
// Find client containing given data in connect area
//****************************************************
bool data_bank_server::find_clients(u16 BB, u16 DB, u64*& clients, int& client_length)
{
  iterator_client it;
  clients = (u64*) malloc(sizeof(u64) * v_clients.size());

  int next = 0;
  client_info* info;
  for (it = v_clients.begin(); it != v_clients.end(); it++) {
      info = it->second;
      // We should check if client contains the right data in the buffer
```

```
      clients[next] = info->get_client_id();
      next++;
  }
  if (next == 0) {
    client_length = 0;
    return false;
  }
  else {
    client_length = next;
    return true;
  }
}

//****************************************************
// Get the ip address of a client
//****************************************************
u64 data_bank_server::get_ip_address(u64 client_id)
{
  return v_clients[client_id]->get_client_ip4();
}

//****************************************************
// Get the DCP port of a client
//****************************************************
u16 data_bank_server::get_dcp_port(u64 client_id)
{
  return v_clients[client_id]->get_client_dcp_port();
}

//****************************************************
// Add a stream to a client
//****************************************************
void data_bank_server::add_stream_to_client(u64 client_id, packet_snd_data_stream* pack)
{
  client_info* client = v_clients[client_id];
  if (!client == NULL)
    client->add_stream(pack);
}

//****************************************************
// Get whatever clients are attached to a given client
//****************************************************
packet_snd_data_stream* data_bank_server::get_attached_clients(u64 client_id)
{
  packet_snd_data_stream* rv = NULL;
  client_info* client = v_clients[client_id];
  if (!client == NULL)
     rv = client->get_attached_clients();

  return rv;
}

//****************************************************
// Get the status of a given client
//****************************************************
packet_status_resp* data_bank_server::get_client_status(u64 client_id)
{
```

```
  packet_status_resp* rv = NULL;
  client_info* client = v_clients[client_id];
  if (!client == NULL)
     rv = client->get_client_status();

  return rv;
}
```

data_buffer.h

```
#ifndef DATA_BUFFER
#define DATA_BUFFER

#include <string.h>
#include "../include/data_container.h"

class data_buffer : public data_container {
private:
  u32 v_min_start_up_db;
  u16 v_db_size;

  u16* bb_map;
  u8* signal_map;

  u16 cbp;
  u16 pp;

  int v_num_bb;
  int v_num_db_in_bb;

  data_element** v_data;
  HANDLE v_semaphore_buffer;

public:
  data_buffer(int number_of_bb,
              int number_of_db_in_bb,
              u16 db_size,
              u32 min_start_up_db);
  void* read_db(u16 bb, u16 db, int& len);
  void write_db(u16 bb, u16 db, void* data, int len);

  u16 get_pp();
  void set_pp(u16 new_pp);

  size_t read_data(void* buf, size_t max_len, unsigned int& offset);
};

#endif
```

data_buffer.cpp

```
#include "stdafx.h"
#include "../include/data_buffer.h"


//*********************************************************
// Constructor
// Creates the new buffer and mallocs the needed space.
//*********************************************************
data_buffer::data_buffer(int number_of_bb,
                         int number_of_db_in_bb,
                         u16 db_size,
                         u32 min_start_up_db)
{
  // Save variables
  v_db_size = db_size;
  v_min_start_up_db = min_start_up_db;
  v_num_bb = number_of_bb;
  v_num_db_in_bb = number_of_db_in_bb;

  // Create semaphore
  v_semaphore_buffer = CreateSemaphore(NULL, 0, 10000, NULL);

  // Allocate memory
  bb_map = (u16*)malloc(sizeof(u16) * v_num_bb);
  ZeroMemory((void*)bb_map, (sizeof(u16) * v_num_bb));

  signal_map = (u8*)malloc(sizeof(u8) * v_num_bb);
  ZeroMemory((void*)signal_map, (sizeof(u8) * v_num_bb));

  v_data = (data_element**) malloc((v_num_bb * v_num_db_in_bb) * sizeof(data_element*));
  ZeroMemory((void*)v_data, v_num_bb * v_num_db_in_bb * sizeof(data_element*));

  // play pointer
  pp = 0;
  cbp = 0;
}


//*********************************************************
// Reads a specific db from the buffer if it exists.
//*********************************************************
void* data_buffer::read_db(u16 bb, u16 db, int& len)
{
  int pos = -1;
  for(int i = 0; i < v_num_bb; i++) {
    if( bb_map[i] == bb ) {
      pos = i;
      break;
    }
  }

  if( pos == -1 ) {
    len = 0;
    return NULL;
```

```
  }
  else {
    len = v_data[(pos * v_num_db_in_bb) + db]->data_len;
    return v_data[(pos * v_num_db_in_bb) + db]->data;
  }
}


//****************************************************************************
// Writes a specific db to the buffer (only if the right db is in the buffer).
//****************************************************************************
void data_buffer::write_db(u16 bb, u16 db, void* data, int len)
{
  int pos = -1;
  for(int i = 0; i < v_num_bb; i++) {
    if( bb_map[i] == bb ) {
      pos = i;
      break;
    }
  }

  if( pos == -1 ) {
    cbp = (cbp + 1) % v_num_bb;
    pos = cbp;
    bb_map[pos] = bb;
    signal_map[pos] = 0;
    ZeroMemory((void*)(&(v_data[pos * v_num_db_in_bb])),
               v_num_db_in_bb * sizeof(data_element*));
    System::Diagnostics::Debug::WriteLine("Start new BB: " + bb);
  }

  if( bb == 9 && db == 1 )
    System::Diagnostics::Debug::WriteLine("STOP");


  data_element* e = (data_element*) malloc(sizeof(data_element));
  e->data_len = len;
  e->data = data;
  v_data[(pos * v_num_db_in_bb) + db] = e;

  // Check if full BB then signal data
  int completed = 0;
  if( signal_map[pos] == 0 ) {
    completed = 1;
    for( int i = 0; i < v_num_db_in_bb; i++ ) {
      if( v_data[(pos * v_num_db_in_bb) + i] == NULL ) {
        completed = 0;
        break;
      }
    }
  }

  if( completed ) {
    signal_map[pos] = 1;
    ReleaseSemaphore(v_semaphore_buffer, 1, NULL);
    System::Diagnostics::Debug::WriteLine("Signal data ready for BB: " + bb_map[pos]);
  }
```

```
}


//****************************************************
// Returns and updates the current postition of the PP
//****************************************************
u16 data_buffer::get_pp()
{
  return pp;
}
void data_buffer::set_pp(u16 new_pp)
{
  pp = new_pp;
}


//**********************************************
// Reads data and returns it back to the buffer
//**********************************************
size_t data_buffer::read_data(void* buf, size_t max_len, unsigned int& offset)
{
  DWORD wait_result_sema = WaitForSingleObject(v_semaphore_buffer, INFINITE);

  int rv = 0;

  data_element* d;

  int pos = -1;
  for( int i = 0; i < v_num_bb; i++ ) {
    if( bb_map[i] == pp ) {
      pos = i;
      break;
    }
  }

  for( int i = 0; i < v_num_db_in_bb; i++ ) {
    d = (v_data[(pos * v_num_db_in_bb) + i]);

    if( d != NULL ) {
      //memcpy((void*)(((char*)buf)+(i * d->data_len)), d->data, d->data_len);
      memcpy((void*)(((char*)buf)+(i * 4598)), d->data, d->data_len);
      rv += d->data_len;
    }
  }

  System::Diagnostics::Debug::WriteLine("BB delivered to client: " + pp );

  offset = pp * v_num_db_in_bb * v_db_size;
  pp++;

  if( pp == 45 )
    System::Diagnostics::Debug::WriteLine("BB delivered to client: " + pp );

  return rv;
}
```

data_cache.h

```
#ifndef DATA_CACHE
#define DATA_CACHE

#include "../include/types.h"
#include "../include/request_queue.h"

class data_cache : public data_container {

private:
  u64 v_movie_id;
  u16 v_num_bb;
  u32 v_size_bb;
  u16 v_db_in_bb;
  u16 v_size_db;
  u64 v_video_size;

  void* read_data;

  request_queue* v_request_queue;
  data_element** v_cache_array;
public:
  data_cache(request_queue* queue,
             u64 movie_id,
             u16 num_bb,
             u32 size_bb,
             u16 num_db,
             u16 size_db,
             u64 video_size);

  void* read_db(u16 BB, u16 DB, int& len);
  void write_db(u16 bb, u16 db, void* data, int len);
  void write_bb(u16 bb, void* data, int len);
  void data_not_sent(u64 client_id, u16 bb, u16 db);
};

#endif
```

data_cache.cpp

```
#include "stdafx.h"
#include "../include/data_cache.h"


// ***************************************************
// Constructor
// ***************************************************
data_cache::data_cache(request_queue* queue,
                       u64 movie_id,
                       u16 num_bb,
                       u32 size_bb,
                       u16 num_db,
```

```
                       u16 size_db,
                       u64 video_size)
{
  v_request_queue = queue;
  v_movie_id = movie_id;
  v_num_bb = num_bb;
  v_size_bb = size_bb;
  v_db_in_bb = num_db / num_bb;
  v_size_db = size_db;
  v_video_size = video_size;

  read_data = NULL;

  v_cache_array = (data_element**)malloc(num_bb * sizeof(data_element*));
  ZeroMemory(v_cache_array, num_bb);
}


// ************************************************
// Read a db from cache
// ************************************************
void* data_cache::read_db(u16 bb, u16 db, int& len)
{
  // In this version the file is read directly from the disc.
  if( read_data == NULL ) {
    ifstream fs ("c:\\movie.mov", ios::in | ios::binary);
    read_data = malloc(v_video_size);

    if( !fs.read((char*)read_data, v_video_size) )
      System::Diagnostics::Debug::WriteLine("Could not read data from file.");
    fs.close();
  }

  // Calculate if we have reached end-of-stream
  if (bb == v_num_bb - 1 && db == v_db_in_bb - 1)
    len = v_video_size % v_size_db;
  else
    len = v_size_db;

  char* rv = ((char*)read_data) + ((bb * v_size_db * v_db_in_bb) + (db * v_size_db));
  return (void*) rv;
}


// ************************************************
// Write a db to cache
// ************************************************
void data_cache::write_db(u16 bb, u16 db, void* data, int len)
{
}


// ************************************************
// Write a bb to cache
// ************************************************
void data_cache::write_bb(u16 bb, void* data, int len)
{
```

```
  data_element* d = new data_element;
  d->data = data;
  d->data_len = len;

  v_cache_array[bb] = d;
}
```

data_container.h

```
#ifndef DATA_CONTAINER
#define DATA_CONTAINER

#include <stdlib.h>
#include "../include/types.h"


  struct data_element {
  int data_len;
  void* data;
};


class data_container {
 public:
  virtual void* read_db(u16 bb, u16 db, int& len) = 0;
  virtual void write_db(u16 bb, u16 db, void* data, int len) = 0;
};


#endif
```

global_functions.h

```
#ifndef GLOBAL_FUNCTIONS
#define GLOBAL_FUNCTIONS

//#include <sys/time.h>
//#include <time.h>
#include <stdlib.h>
#include <sys/timeb.h>
#include <time.h>


struct system_time {
  long sec;
  long msec;
};

void get_system_time(system_time* time);

bool operator<(const system_time t1, const system_time t2);
```

```
bool operator>(const system_time t1, const system_time t2);
system_time operator-(const system_time t1, const system_time t2);
system_time operator+(const system_time t1, const system_time t2);
bool operator==(const system_time t1, const system_time t2);
system_time operator+(const system_time t1, int usec);
int to_msec(const system_time t1);

#endif
```

global_functions.cpp

```
#include "stdafx.h"
#include "../include/global_functions.h"

//****************************************
// Get current time and create a timestamp
//****************************************
void get_system_time(system_time* time)
{
  _timeb timebuffer;
  _ftime64_s( &timebuffer );
  time->sec = timebuffer.time;
  time->msec = timebuffer.millitm;
}


//****************************************
// Compares (<) two timestamps
//****************************************
bool operator<(const system_time t1, const system_time t2)
{

  if (t1.sec == t2.sec) {
    return t1.msec < t2.msec;
  }
  else {
    return t1.sec < t2.sec;
  }
}


//****************************************
// Compares (>) two timestamps
//****************************************
bool operator>(const system_time t1, const system_time t2)
{

  if (t1.sec == t2.sec) {
    return t1.msec > t2.msec;
  }
  else {
    return t1.sec > t2.sec;
  }
}
```

```
//****************************************
// Addition of two timestamps
//****************************************
system_time operator+(const system_time t1, const system_time t2)
{
  struct system_time rv;
  int new_msec = t1.msec + t2.msec;

  // Calculate remainder
  int remainder = new_msec % 1000;
  rv.msec = remainder;

  // Calculate carry
  int carry = (int) new_msec / 1000;
  rv.sec = t1.sec + t2.sec + carry;
  return rv;
}


//****************************************
// Subtraction of two timestamps
//****************************************
system_time operator-(const system_time t1, const system_time t2)
{
  struct system_time rv;
  int new_msec = t1.msec - t2.msec;

  if (new_msec < 0) {
    rv.sec = t1.sec - t2.sec - 1;
    rv.msec = 1000 - abs(new_msec);
  }
  else {
    rv.sec = t1.sec - t2.sec;
    rv.msec = t1.msec - t2.msec;
  }
  return rv;
}


//****************************************
// Compare (==) two timestamps
//****************************************
bool operator==(const system_time t1, const system_time t2)
{
  return (t1.sec == t2.sec) && (t1.msec == t2.msec);
}


//****************************************
// Converts timestamp to msec.
//****************************************
int to_msec(const system_time t1)
{
  return ( (t1.sec*1000) + t1.msec);
}
```

```
//****************************************
// Add msec. to a timestamp
//****************************************
system_time operator+(const system_time t1, int msec)
{
  struct system_time rv;
  int new_msec = msec + t1.msec;

  // Calculate remainder
  int remainder = new_msec % 1000;
  rv.msec = remainder;

  // Calculate carry
  int carry = (int) new_msec / 1000;
  rv.sec = t1.sec + carry;
  return rv;
}
```

logic_client.h

```
#ifndef LOGIC_CLIENT
#define LOGIC_CLIENT

#include "../include/transport_handler.h"
#include "../include/data_bank_client.h"
#include "../include/stream_engine.h"
#include "../include/data_buffer.h"
#include "../include/packet_queue.h"
#include "../include/stream_engine_thread.h"
#include "../include/status_thread.h"
#include "../include/enum.h"
#include "../include/packet_conn_granted.h"
#include "../include/packet_status_req.h"
#include "../include/packet_conn_req.h"
#include "../include/packet_status_resp.h"
#include "../include/packet_data.h"
#include "../include/packet_stop_stream.h"
#include "../include/packet_disconn.h"
#include "../include/packet_stream_error.h"
#include "../include/packet_end_of_data.h"
#include "../include/packet_handler.h"
#include "../include/packet_interact_pause.h"
#include "../include/packet_interact_pause_stream.h"
#include "../include/packet_interact_resume.h"
#include "../include/packet_interact_skip.h"
#include "../include/packet_no_data.h"
#include "../include/packet.h"
#include "../include/packet_adjust_speed.h"
#include "../include/packet_req_backup_data.h"
#include "../include/packet_conn_closed.h"
#include "../include/packet_resend_data.h"
#include "../include/packet_conn_denied.h"
```

```
#include "../include/packet_snd_data_stream.h"

class logic_client {

 private:
  u64 v_video_id;
  size_t v_buf_size;

  transport_handler* v_transport_handler;
  data_bank_client* v_data_bank;
  stream_engine* v_stream_engine;
  data_buffer* v_data_buffer;
  packet_queue* v_packet_queue;

  DWORD  v_stream_thread_id;
  HANDLE v_stream_thread;

  DWORD  v_status_thread_id;
  HANDLE v_status_thread;

  int hatten;

  int v_state;

  u64 v_client_id;
  u64 v_server_id;

  void conn_granted(packet_conn_granted* pack);
  void conn_denied(packet_conn_denied* pack);
  void conn_closed(packet_conn_closed* pack);
  void recv_data_stream(packet_recv_data_stream* pack);


  void status_req(packet_status_req* pack);
  void snd_data_stream(packet_snd_data_stream* pack);
  void stop_stream(packet_stop_stream* pack);

  void data(packet_data* pack);
  void no_data(packet_no_data* pack);
  void end_of_data(packet_end_of_data* pack);

  int logic_clientmovie_info();

  void resend_data(packet_resend_data* pack);
  void adjust_speed(packet_adjust_speed* pack);

  bool check_data_reception(u64 snd_client_id,
    u16 bb,
    u16 db);

 public:
  logic_client(u64 video_id, transport_handler* th, packet_queue* pq);

  void packet_handler(packet* pack);

  int connect(size_t buf_size, struct login_data* client, u16 ccp_port, u16 dcp_port);
```

```
  size_t poll();

  size_t recv_data(void* buf, size_t max_len, unsigned int& offset);

  int disconnect();

  int pause();

  int resume();

  int skip(unsigned int distance);

  int movie_info();

  u64 get_video_duration();

  u64 get_video_length();

  u64 get_skip_distance();


};

#endif
```

logic_client.cpp

```
#include "stdafx.h"
#include "../include/logic_client.h"


//*******************************************************
// Constructor
// Sets the reference to transport_handler and packet_queue
//*******************************************************
logic_client::logic_client(u64 video_id, transport_handler* th, packet_queue* pq)
{
  v_video_id = video_id;
  v_stream_engine = NULL;

  v_transport_handler = th;
  v_packet_queue = pq;
  v_data_bank = new data_bank_client();
}


//*******************************************************
// Function called to connect client to server, waits for
// respond on connect request.
//*******************************************************
int logic_client::connect(size_t buf_size,
                          struct login_data* client,
                          u16 ccp_port,
```

```
                          u16 dcp_port)
{
  packet_conn_req* snd_p = new packet_conn_req(0,0);
  snd_p->set_video_id(1234);
  snd_p->set_ccp_port(ccp_port);
  snd_p->set_dcp_port(dcp_port);
  v_transport_handler->send_CCP(snd_p, 0);

  // Wait for respond on connect packet - always a packet_conn_granted in this version
  packet* recv_p = v_packet_queue->get_next_packet();
  packet_handler(recv_p);
  client->skip_distance = ((packet_conn_granted*)recv_p)->get_skip_distance();
  client->video_duration = ((packet_conn_granted*)recv_p)->get_video_duration();
  client->video_size = ((packet_conn_granted*)recv_p)->get_video_size();

  v_data_bank->initialize((packet_conn_granted*) recv_p);
  return 0;
}


//************************************************************
// Returns data from the buffer to the client.
//************************************************************
size_t logic_client::recv_data(void* buf, size_t max_len, unsigned int& offset)
{
  return v_data_buffer->read_data(buf, max_len, offset);
}


//************************************************************
// Returns data from the buffer to the client.
//************************************************************
int logic_client::disconnect()
{
  // If a stream engine exists it must be stopped
  if( v_stream_engine != NULL ) {
    TerminateThread(v_stream_thread, 0);

    // Delete objects
    delete(v_stream_engine);
  }

  // Delete status thread
  TerminateThread(v_status_thread, 0);

  packet_disconn* p = new packet_disconn(v_server_id, v_client_id);
  v_transport_handler->send_CCP(p, v_server_id);

  return 0;
}


//************************************************************
// Function is called duing a skip.
//************************************************************
int logic_client::skip(unsigned int distance)
{
```

```
  packet_interact_skip* p = new packet_interact_skip(v_server_id, v_client_id);
  v_data_buffer->set_pp(distance);
  p->set_target_block(distance);
  v_transport_handler->send_CCP(p, v_server_id);
  return 0;
}


//*******************************************************
// The packet handler is called upon reception of a packet.
//*******************************************************
void logic_client::packet_handler(packet* pack)
{
  switch( pack->get_packet_type() ) {
    // Server to client resp
    case 10:
      switch( pack->get_action_type() ) {
        case 10: // CONN_GRANTED
          conn_granted((packet_conn_granted*) pack);
          break;
        case 20: // CONN_DENIED
          break;
        case 30: // CONN_CLOSED
          break;
        case 50: // RECV_DATA_STREAM
          recv_data_stream((packet_recv_data_stream*) pack);
          break;
      }
    break;

    // Server to client req
    case 20:
      switch( pack->get_action_type() ) {
        case 10: // STATUS_REQ
          break;
        case 20: // SND_DATA_STREAM
          snd_data_stream((packet_snd_data_stream*) pack);
          break;
        case 30: // STOP_STREAM
          stop_stream((packet_stop_stream*) pack);
          break;
      }
    break;

    // Client recv. to client snd
    case 50:
      switch( pack->get_action_type() ) {
        case 10: // DATA
          data((packet_data*) pack);
          break;
        case 20: // NO_DATA
          break;
        case 30: // END_OF_DATA
          break;
      }
    break;
```

```
    // Client snd to client recv.
    case 60:
      switch( pack->get_action_type() ) {
        case 10: // RESEND_DATA
          resend_data((packet_resend_data*) pack);
          break;
        case 20: // AJUST_SPEED
          adjust_speed((packet_adjust_speed*) pack);
          break;
      }
    break;
  }
}


//*********************************************************
// Called upon reception of conn_granted, sets client and
// server id and creates the buffer. (Private function).
//*********************************************************
void logic_client::conn_granted(packet_conn_granted* pack)
{
  v_client_id = pack->get_client_id();
  v_server_id = pack->get_server_id();
  v_state = 1;

  v_data_buffer = new data_buffer(pack->get_recon_ca_size(),
                                pack->get_db_total() / pack->get_bb_total(),
                                pack->get_db_size(),
                                pack->get_video_header_size());

  // Startup the status thread
  status_params* args = (status_params*) malloc(sizeof(status_params));
  args->v_data_buffer = this->v_data_buffer;
  args->v_transport_handler = this->v_transport_handler;
  args->v_server_id = v_server_id;
  args->v_client_id = v_client_id;

  v_status_thread = CreateThread(NULL,
                         0,
                 (LPTHREAD_START_ROUTINE) status_thread,
                 (void*) args,
                 0,
                &v_status_thread_id);

}


//***********************************************************
// Called upon reception of recv_data_stream packet, sets
// from whom the client will recieve data. (Private function).
//***********************************************************
void logic_client::recv_data_stream(packet_recv_data_stream* pack)
{
  v_data_bank->set_recv_table(pack);
}
```

```
//*******************************************************
// Called upon reception of recv_data_stream packet, sets
// to whom the client must send. (Private function).
//*******************************************************
void logic_client::status_req(packet_status_req* pack)
{
  packet_status_resp* status_pack = new packet_status_resp(v_server_id, v_client_id);
  status_pack->set_pp(v_data_buffer->get_pp());
  v_transport_handler->send_CCP(status_pack, v_server_id);

  delete(status_pack);
}


//*******************************************************
// Called upon reception of snd_data_stream packet, sets
// to whom the client must send data and starts a stream-
// engine if one is not already running. (Private function).
//*******************************************************
void logic_client::snd_data_stream(packet_snd_data_stream* pack)
{
  v_data_bank->add_stream(pack);

  if( v_stream_engine == NULL ) {
    v_stream_engine = new stream_engine(v_data_buffer,
                                        v_video_id,
                                        v_client_id);

    stream_params* args = (stream_params*) malloc(sizeof(stream_params));
    args->v_data_bank = this->v_data_bank;
    args->v_transport_handler = this->v_transport_handler;
    args->v_stream_engine = this->v_stream_engine;;

    v_stream_thread = CreateThread(NULL,
                        0,
                 (LPTHREAD_START_ROUTINE) stream_thread,
                 (void*) args,
                 0,
                 &v_stream_thread_id);
  }
}


//*******************************************************
// Upon reception of stop_stream the stream-engine is
// informed to stop streaming. (Private function).
//*******************************************************
void logic_client::stop_stream(packet_stop_stream* pack)
{
  v_data_bank->stop_stream(pack);
}


//**************************************
// Returns aditional info about the movie
//**************************************
int logic_client::movie_info()
```

```
{
  return 0;
}


//***************************************
// Handles data packets
//***************************************
void logic_client::data(packet_data* pack)
{
  // Check if packet is ok before insert into buffer
  if( check_data_reception(pack->get_snd_client_id(),
                           pack->get_bb(),
                           pack->get_db()) ) {

    int data_len;
    void* data_pointer = pack->get_data(data_len);

    v_data_buffer->write_db(pack->get_bb(),
                            pack->get_db(),
                            data_pointer,
                            data_len);
  }
}

//***************************************
// Checks received data
//***************************************
bool logic_client::check_data_reception(u64 snd_client_id,
                                        u16 bb,
                                        u16 db)
{
  return true;
}

//***************************************
// Function to resend data to a client
//***************************************
void logic_client::resend_data(packet_resend_data* pack)
{
  v_data_bank->resend_block(pack->get_snd_client_id(),
                            pack->get_bb(),
                            pack->get_db());

  v_data_bank->adjust_speed(pack->get_snd_client_id(), HIGH);
}


//***************************************
// Handles speed adjustment
//***************************************
void logic_client::adjust_speed(packet_adjust_speed* pack)
{
  v_data_bank->adjust_speed(pack->get_snd_client_id(),
                            (speed_level) pack->get_speed_level());
}
```

```
//*****************************************
// Returns video duration from the databank
//*****************************************
u64 logic_client::get_video_duration()
{
  return v_data_bank->get_video_duration();
}

//*************************************************
// Returns the length of the video from the databank
//*************************************************
u64 logic_client::get_video_length()
{
  return v_data_bank->get_video_length();
}

//*************************************************
// Returns the video skip distance from the databank
//*************************************************
u64 logic_client::get_skip_distance()
{
  return v_data_bank->get_skip_distance();
}
```

logic_client_thread.h

```
#ifndef LOGIC_CLIENT_THREAD
#define LOGIC_CLIENT_THREAD

#include "../include/logic_client.h"
#include "../include/packet_queue.h"

struct logic_params
{
  packet_queue* v_packet_queue;
  logic_client* v_logic_client;
};

void logic_thread(void* args);

#endif
```

logic_client_thread.cpp

```
#include "stdafx.h"
#include "../include/logic_client_thread.h"

//********************************************************
// Thread which pops packets from the incomming queue and
```

```
// handes them to the logic.
//*****************************************************
void logic_thread(void* args)
{
  packet_queue* v_packet_queue = ((logic_params*)args)->v_packet_queue;
  logic_client* v_logic_client = ((logic_params*)args)->v_logic_client;

  int foo = 0;

  for(;;) {
    packet* pack = v_packet_queue->get_next_packet();
    v_logic_client->packet_handler(pack);
  }

  return;
}
```

logic_server.h

```
#ifndef LOGIC_SERVER
#define LOGIC_SERVER

#include "../include/vod_server.h"
#include "../include/transport_handler.h"
#include "../include/stream_engine.h"
#include "../include/data_cache.h"
#include "../include/stream_engine_server_thread.h"
#include "../include/request_queue.h"
#include "../include/enum.h"
#include "../include/packet_conn_granted.h"
#include "../include/packet_status_req.h"
#include "../include/packet_conn_req.h"
#include "../include/packet_status_resp.h"
#include "../include/packet_data.h"
#include "../include/packet_stop_stream.h"
#include "../include/packet_disconn.h"
#include "../include/packet_stream_error.h"
#include "../include/packet_end_of_data.h"
#include "../include/packet_handler.h"
#include "../include/packet_interact_pause.h"
#include "../include/packet_interact_pause_stream.h"
#include "../include/packet_interact_resume.h"
#include "../include/packet_interact_skip.h"
#include "../include/packet_no_data.h"
#include "../include/packet.h"
#include "../include/packet_adjust_speed.h"
#include "../include/packet_req_backup_data.h"
#include "../include/packet_conn_closed.h"
#include "../include/packet_resend_data.h"
#include "../include/packet_conn_denied.h"
#include "../include/packet_snd_data_stream.h"

class logic_server {
```

```
private:
 int hackvar;


 transport_handler* v_transport_handler;
 stream_engine* v_stream_engine;
 data_cache* v_data_cache;
 data_bank_server* v_data_bank;
 request_queue* v_request_queue;

 stream_params args;
 DWORD  v_stream_thread_id;
 HANDLE v_stream_thread;

 int v_state;

 // Handle status respond packet
 void status_resp(packet_status_resp* pack);
 // Handle connectio request packet
 void conn_req(packet_conn_req* pack);
 // Handle pausing of the clietns video player
 void interact_pause(packet_interact_pause* pack);
 // Handle pausing of the video stream
 void interact_pause_stream(packet_interact_pause_stream* pack);
 // Handle resending of data
 void resend_data(packet_resend_data* pack);
 // Handle adjustment of speed
 void adjust_speed(packet_adjust_speed* pack);
 // Handle disconnect packet
 void disconnect(packet_disconn* pack);
 // Handle skip packet
 void skip(packet_interact_skip* pack);

 // Start new stream engine
 void start_stream_engine(u64 video_id);

 // Attach a client to the local server
 void attach_client_to_server(packet_conn_req* pack);

 // Attach a client to a set of clients
 void attach_client_to_streamers(u64* sender_clients, int client_length, packet_conn_req* pack);

 // Fil a packet snd data stream packet
 void fill_packet_snd_data_stream(packet_snd_data_stream* pack, int spreading);
public:

 // Constructor
 logic_server(transport_handler* th);

 // Handle incoming packet
 void packet_handler(packet* pack);

 // Get the next data request. Called from server application
 int get_data_req(struct data_struct_req& data, data_type type);

 // Deliver data. Called from server application
```

```
  int deliver_data(data_class* data, data_type type);
};

#endif
```

logic_server.cpp

```
#include "stdafx.h"
#include "../include/logic_server.h"

//**********************************************
// Constructor
//**********************************************
logic_server::logic_server(transport_handler* th)
{
  v_stream_engine = NULL;
  v_state = 0;
  v_transport_handler = th;
  v_data_bank = new data_bank_server();
  v_request_queue = new request_queue();
  System::Diagnostics::Debug::WriteLine("Logic Created");

hackvar = 0;
}

//**********************************************
// Packet handler
//**********************************************
void logic_server::packet_handler(packet* pack)
{
  switch( pack->get_packet_type() ) {
    // Client to server resp
    case 30:
      switch( pack->get_action_type() ) {
        case 10: // STATUS_RESP
          status_resp((packet_status_resp*) pack);
          break;
      }
    break;

    // client to server req
    case 40:
      switch( pack->get_action_type() ) {
        case 10: // CONN_REQ
          conn_req((packet_conn_req*) pack);
          break;
        case 30: // INTERACT_PAUSE
          interact_pause((packet_interact_pause*) pack);
          break;
        case 40: // INTERACT_PAUSE_STREAM
          interact_pause_stream((packet_interact_pause_stream*) pack);
          break;
        case 60: // INTERACT_SKIP
          skip((packet_interact_skip*) pack);
```

```
          break;
        case 70: // DISCONNECT
          disconnect((packet_disconn*) pack);
          break;
      }
    break;

    // Client snd to client recv.
    case 60:
      switch( pack->get_packet_type() ) {
        case 10: // RESEND_DATA
          resend_data((packet_resend_data*) pack);
          break;
        case 20: // AJUST_SPEED
          adjust_speed((packet_adjust_speed*) pack);
          break;
      }
    break;
  }
}

//**********************************************
// Client to server resp
//**********************************************
void logic_server::status_resp(packet_status_resp* pack)
{
  v_data_bank->update_status(pack);
}

//**********************************************
// Handle connection request packet
//**********************************************
void logic_server::conn_req(packet_conn_req* pack)
{
  // We should add client to applicationtask queue to
  // validate client. Thi is not implemented. We simply
  // verify the client.

  // Extract client id coming from transport_handler
  u64 client_id = pack->client_id;
  u64 video_id = v_data_bank->get_video_id();

  // Create packet conn-granted packet
  packet_conn_granted* p;
  p = new packet_conn_granted(pack->client_id, v_data_bank->get_server_id(client_id));

  v_data_bank->get_movie_info(video_id, p);
  v_transport_handler->send_CCP(p, client_id);

  // Find appropriate clients
  u64* sender_clients = NULL;
  int client_length = 0;
  bool rv = v_data_bank->find_clients(0,0, sender_clients, client_length);
  if (rv)
    attach_client_to_streamers(sender_clients, client_length, pack);
  else
    attach_client_to_server(pack);
```

```
  // If stream engine has not been instantiated yet, do so.
  if( v_stream_engine == NULL )
    start_stream_engine(video_id);
}


//**********************************************
// Attach a client to server. The new or
// relocated client is given in the pack argument.
//**********************************************
void logic_server::attach_client_to_server(packet_conn_req* pack)
{
  // Add client to databank;
  // We should check if client already exists
  v_data_bank->add_new_client(pack);

 u64 client_id = pack->client_id;
 u64 server_id = v_data_bank->get_server_id(client_id);

  // Create packet packet_snd_data_stream to store in databank
  packet_snd_data_stream* stream_pack;
  stream_pack = new packet_snd_data_stream(client_id, server_id);

  fill_packet_snd_data_stream(stream_pack, 0);

  stream_pack->client_CCP_port = pack->get_ccp_port();
  stream_pack->client_DCP_port = pack->get_dcp_port();
  stream_pack->client_id = pack->client_id;
  stream_pack->client_ip = pack->client_ip;
  stream_pack->set_client_id(pack->client_id);
  stream_pack->set_client_ip4(pack->client_ip);
  stream_pack->set_client_port(pack->get_dcp_port());

  v_data_bank->add_new_stream(stream_pack);
}


//**********************************************
// Attach a client to some other client(s)
// The senders of data (clients, which the new or
// relocated client is to be attached to) is given
// in sender. The new or relocated client is given
// in the pack argument.
//**********************************************
void logic_server::attach_client_to_streamers(u64* sender_clients,
                                              int client_length,
                                              packet_conn_req* pack)
{
  // Add client to databank;
  // We should check if client already exists
  v_data_bank->add_new_client(pack);

  u64 client_id = pack->client_id;
  u64 server_id = v_data_bank->get_server_id(client_id);

  // Create packet containing senders of data
```

```
  if (client_length == 1) {
    packet_snd_data_stream* stream_pack;
    stream_pack = new packet_snd_data_stream(sender_clients[0], server_id);

    fill_packet_snd_data_stream(stream_pack, 0);
    stream_pack->client_CCP_port = pack->get_ccp_port();
    stream_pack->client_DCP_port = pack->get_dcp_port();
    stream_pack->client_id = pack->client_id;
    stream_pack->client_ip = pack->client_ip;
    stream_pack->set_client_ip4(pack->client_ip);
    stream_pack->set_client_port(pack->get_dcp_port());
    stream_pack->set_client_id(pack->client_id);

    // Update internal representation of network
    v_data_bank->add_stream_to_client(sender_clients[0], stream_pack);

    // Send packet
    v_transport_handler->send_CCP(stream_pack, sender_clients[0]);
  }
  else {
    for (int i = 0; i < client_length; i++) {
      packet_snd_data_stream* stream_pack;
      stream_pack = new packet_snd_data_stream(sender_clients[i], server_id);

      fill_packet_snd_data_stream(stream_pack, i+1);
      stream_pack->client_CCP_port = pack->get_ccp_port();
      stream_pack->client_DCP_port = pack->get_dcp_port();
      stream_pack->client_id = pack->client_id;
      stream_pack->client_ip = pack->client_ip;
      stream_pack->set_client_ip4(pack->client_ip);
      stream_pack->set_client_port(pack->get_dcp_port());
      stream_pack->set_client_id(pack->client_id);

      // Update internal representation of network
      v_data_bank->add_stream_to_client(sender_clients[i], stream_pack);

      // Send packet
      v_transport_handler->send_CCP(stream_pack, sender_clients[i]);
    }
  }
}

//**********************************************
// Fill packet_snd_data_stream with datablocks
//**********************************************
void logic_server::fill_packet_snd_data_stream(packet_snd_data_stream* pack, int spreading)
{
  int length ;
  void* db_list;
  int dbnum = 0;
  int num_of_dbs = v_data_bank->get_num_of_db_in_bb(0);

  switch (spreading) {
    case 0:
      pack->init(num_of_dbs);
      db_list = malloc(sizeof(u16) * num_of_dbs);
      pack->set_start_bb(0);
```

```
      pack->set_start_db(0);
      for (int i = 0; i < num_of_dbs; i++) {
        ((u16*)db_list)[i] = i;
      }
      pack->set_db_list((u16*) db_list);
      break;
    case 1:
      pack->init((int) num_of_dbs / 2);
      db_list = malloc(sizeof(u16) * (int) num_of_dbs / 2);
      pack->set_start_bb(0);
      pack->set_start_db(0);

      for (int i = 0; i < num_of_dbs; i = i+2) {
        ((u16*)db_list)[dbnum] = i;
        dbnum++;
      }
      pack->set_db_list((u16*) db_list);
      break;
    case 2:
      pack->init((int) num_of_dbs / 2);
      db_list = malloc(sizeof(u16) * (int) num_of_dbs / 2);
      pack->set_start_bb(0);
      pack->set_start_db(1);

      for (int i = 1; i < num_of_dbs ; i = i+2) {
        ((u16*)db_list)[dbnum] = i;
        dbnum++;
      }
      pack->set_db_list((u16*) db_list);
      break;
    default:
      break;
  }
}


//*********************************************
// Instantiate new stream engine, and start
// stream engine thread
//*********************************************
void logic_server::start_stream_engine(u64 video_id)
{
  v_data_cache = new data_cache(v_request_queue,
                                video_id,
                                v_data_bank->get_num_of_bb(video_id),
                                v_data_bank->get_size_of_bb(video_id),
                                v_data_bank->get_num_of_db(video_id),
                                v_data_bank->get_size_of_db(video_id),
                                v_data_bank->get_video_size(video_id));

  // Create new stream-engine object
  v_stream_engine = new stream_engine((data_container*)v_data_cache,
                                       video_id,
                                       false); // Don't start sending

  // We should request first portion of video data from server application
  // if this is not contained in data cache.
```

```
  // Params
  args.v_data_bank = this->v_data_bank;
  args.v_transport_handler = this->v_transport_handler;
  args.v_stream_engine = this->v_stream_engine;

  // Start stream engine thread
  v_stream_thread = CreateThread(NULL,
                         0,
                   (LPTHREAD_START_ROUTINE) stream_thread,
                     (void*) &args,
                 0,
                 &v_stream_thread_id);
}


//**********************************************
// Handle pause interaction
//**********************************************
void logic_server::interact_pause(packet_interact_pause* pack)
{
//  v_data_bank->pause_client(pack);
}

//**********************************************
// Handle pause stream
//**********************************************
void logic_server::interact_pause_stream(packet_interact_pause_stream* pack)
{
  //v_data_bank->pause_client(pack);
}


//**********************************************
// Client snd to client resv.
//**********************************************
void logic_server::resend_data(packet_resend_data* pack)
{

}


//**********************************************
// Adjust speed of client
//**********************************************
void logic_server::adjust_speed(packet_adjust_speed* pack)
{
  //v_data_bank->adjust_speed(pack->get_snd_client_id(),
  //  (speed_level) pack->get_speed_level());
}

//**********************************************
// Handle disconnect packet from client
// This procedure should collect all clients which
// are receiving data from the disconnecting client and
// relocate these.
// Furthermore all client which are streaming data to the
```

```
// disconnecting client should stop sending data.
//**********************************************
void logic_server::disconnect(packet_disconn* pack)
{
  packet_snd_data_stream* receiver = v_data_bank->get_attached_clients(pack->client_id);
  if (!receiver == NULL) {
    packet_status_resp* status = v_data_bank->get_client_status(receiver->get_client_id());
    // Set the next blocks to be received to what has been fetched from status packet.
    receiver->set_start_bb(status->get_pp());
    receiver->set_start_db(0);

    // Attach client to server. We should relocate client to better suited clients.
    v_data_bank->add_new_stream(receiver);

    v_transport_handler->close_CCP_connection(pack->client_id);
    v_data_bank->remove_stream(pack->client_id);
    v_data_bank->remove_client(pack->client_id);
  }
}

//**********************************************
// Handle disconnect packet from client
//**********************************************
void logic_server::skip(packet_interact_skip* pack)
{
  v_data_bank->skip_stream(pack->client_id, pack->get_target_block());

  // Check if we need to relocate any clients
  // These should be relocated know
}


//**********************************************
// Return next data request to server application
//**********************************************
int logic_server::get_data_req(struct data_struct_req& data,
                               data_type type)
{
  switch( type ) {
    // Client to server resp
  case VIDEO_DATA:
    v_request_queue->get_data_request(data);
    return 0;
  break;
  defualt:
    // Not implmentet in this version
  break;
  }

  return 1;
}

//**********************************************
// Deliver data from server application to protocol.
//**********************************************
int logic_server::deliver_data(data_class* data, data_type type)
{
```

```
  switch( type ) {
    // Client to server resp
    case VIDEO_DATA:
      v_data_cache->write_bb((u16)(data->get_transaction_id()), data->get_data(), data->get_data_length());
      return 0;
    break;
    defualt:
      // Not implmentet in this version
    break;
  }

  return 1;
}
```

logic_server_thread.h

```
#ifndef LOGIC_SERVER_THREAD
#define LOGIC_SERVER_THREAD

#include "../include/logic_server.h"
#include "../include/packet_queue.h"

struct logic_params
{
  packet_queue* v_packet_queue;
  logic_server* v_logic_server;
};

void logic_thread(void* args);

#endif
```

logic_server_thread.cpp

```
#include "stdafx.h"
#include "../include/logic_server_thread.h"

//**********************************************
// Main procedure.
//**********************************************
void logic_thread(void* args)
{
  packet_queue* v_packet_queue = ((logic_params*)args)->v_packet_queue;
  logic_server* v_logic_server = ((logic_params*)args)->v_logic_server;

for(;;) {
packet* pack = v_packet_queue->get_next_packet();
v_logic_server->packet_handler(pack);
}
```

```
  return;
}
```

packet_adjust_speed.h

```
#ifndef PACKET_ADJUST_SPEED
#define PACKET_ADJUST_SPEED

#include "../include/packet.h"

#define c_adjust_speed_packet_type 60
#define c_adjust_speed_action_type 20
#define c_adjust_speed_payload_len 1

class packet_adjust_speed : public packet
{
 public:
  packet_adjust_speed(u64 recv_id, u64 snd_id);

  ~packet_adjust_speed();

  void set_speed_level(u8 speed_level);
  u8 get_speed_level();
};

#endif
```

packet_adjust_speed.cpp

```
#include "stdafx.h"
#include "../include/packet_adjust_speed.h"

//*****************************************************
// Constructor
//*****************************************************
packet_adjust_speed::packet_adjust_speed(u64 recv_id, u64 snd_id)
: packet(c_adjust_speed_packet_type, c_adjust_speed_action_type, recv_id, snd_id)
{
  v_payload = malloc(c_adjust_speed_payload_len);
  set_payload_len(c_adjust_speed_payload_len);
}

//*****************************************************
// Deconstructor
//*****************************************************
packet_adjust_speed::~packet_adjust_speed()
{
  free(v_payload);
}
```

```
//*****************************************************
// Get or set the speed level
//*****************************************************
void packet_adjust_speed::set_speed_level(u8 speed_level)
{
  set_value_u8(speed_level, v_payload, 0);
}
u8 packet_adjust_speed::get_speed_level()
{
  return get_value_u8(v_payload, 0);
}
```

packet_conn_closed.h

```
#ifndef PACKET_CONN_CLOSED
#define PACKET_CONN_CLOSED

#include "../include/packet.h"

#define c_conn_closed_packet_type 10
#define c_conn_closed_action_type 30
#define c_conn_closed_payload_len 256

class packet_conn_closed : public packet
{
 public:
  packet_conn_closed(u64 recv_id, u64 snd_id);

  ~packet_conn_closed();

  void set_message(char* message);
  char* get_message();
};

#endif
```

packet_conn_closed.cpp

```
#include "stdafx.h"
#include "../include/packet_conn_closed.h"

//*****************************************************
// Constructor
//*****************************************************
packet_conn_closed::packet_conn_closed(u64 recv_id, u64 snd_id)
: packet(c_conn_closed_packet_type, c_conn_closed_action_type, recv_id, snd_id)
{
  v_payload = malloc(c_conn_closed_payload_len);
  set_payload_len(c_conn_closed_payload_len);
}
```

```
//****************************************************
// Deconstructor
//****************************************************
packet_conn_closed::~packet_conn_closed()
{
  free(v_payload);
}

//****************************************************
// Get or set the error message
//****************************************************
void packet_conn_closed::set_message(char* message)
{
  strncpy(((char*)v_payload), message, 255);

  if(strlen(message) > 255 )
    ((char*)v_payload)[255] = '\0';
}
char* packet_conn_closed::get_message()
{
  return (char*)v_payload;
}
```

packet_conn_denied.h

```
#ifndef PACKET_CONN_DENIED
#define PACKET_CONN_DENIED

#include "../include/packet.h"

#define c_conn_denied_packet_type 10
#define c_conn_denied_action_type 20
#define c_conn_denied_payload_len 256

class packet_conn_denied : public packet
{
 public:
  packet_conn_denied(u64 recv_id, u64 snd_id);

  ~packet_conn_denied();

  void set_message(char* message);
  char* get_message();
};

#endif
```

packet_conn_denied.cpp

```
#include "stdafx.h"
#include "../include/packet_conn_denied.h"
```

```
//****************************************************
// Constructor
//****************************************************
packet_conn_denied::packet_conn_denied(u64 recv_id, u64 snd_id)
: packet(c_conn_denied_packet_type, c_conn_denied_action_type, recv_id, snd_id)
{
  v_payload = malloc(c_conn_denied_payload_len);
  set_payload_len(c_conn_denied_payload_len);
}

//****************************************************
// Deconstructor
//****************************************************
packet_conn_denied::~packet_conn_denied()
{
  free(v_payload);
}

//****************************************************
// Get or set the error message
//****************************************************
void packet_conn_denied::set_message(char* message)
{
  strncpy(((char*)v_payload), message, 255);

  if(strlen(message) > 255 )
    ((char*)v_payload)[255] = '\0';
}
char* packet_conn_denied::get_message()
{
  return (char*)v_payload;
}
```

packet_conn_granted.h

```
#ifndef PACKET_CONN_GRANTED
#define PACKET_CONN_GRANTED

#include "../include/packet.h"

#define c_conn_granted_packet_type 10
#define c_conn_granted_action_type 10
#define c_conn_granted_payload_len 308

class packet_conn_granted : public packet
{
 public:
  packet_conn_granted(u64 recv_id, u64 snd_id);

  ~packet_conn_granted();

  void set_client_id(u64 client_id);
  u64 get_client_id();
```

```
  void set_server_id(u64 server_id);
  u64 get_server_id();

  void set_video_size(u64 video_size);
  u64 get_video_size();

  void set_video_duration(u32 video_duration);
  u32 get_video_duration();

  void set_bb_total(u16 number_of_bb);
  u16 get_bb_total();

  void set_db_total(u16 number_of_db);
  u16 get_db_total();

  void set_db_size(u16 size_of_db);
  u16 get_db_size();

  void set_video_header_size(u32 video_header_size);
  u32 get_video_header_size();

  void set_recon_ca_size(u16 recon_ca_size);
  u16 get_recon_ca_size();

  void set_speed_low(u16 speed_low);
  u16 get_speed_low();

  void set_speed_normal(u16 speed_normal);
  u16 get_speed_normal();

  void set_speed_high(u16 speed_high);
  u16 get_speed_high();

  void set_sec_type(u16 sec_type);
  u16 get_sec_type();

  void set_skip_distance(u32 skip_distance);
  u32 get_skip_distance();

  void set_video_title(char* movie_title);
  char* get_video_title();
};

#endif
```

packet_conn_granted.cpp

```
#include "stdafx.h"
#include "../include/packet_conn_granted.h"


//**************************************************
//  Constructor
```

```
//**************************************************
packet_conn_granted::packet_conn_granted(u64 recv_id, u64 snd_id)
: packet(c_conn_granted_packet_type, c_conn_granted_action_type, recv_id, snd_id)
{
  v_payload = malloc(c_conn_granted_payload_len);
  set_payload_len(c_conn_granted_payload_len);
}

//**************************************************
//  Deconstructor
//**************************************************
packet_conn_granted::~packet_conn_granted()
{
  free(v_payload);
}

//**************************************************
//  Get/set client id
//**************************************************
void packet_conn_granted::set_client_id(u64 client_id)
{
  set_value_u64(client_id, v_payload, 0);
}
u64 packet_conn_granted::get_client_id()
{
  return get_value_u64(v_payload, 0);
}


//**************************************************
//  Get/set server id
//**************************************************
void packet_conn_granted::set_server_id(u64 server_id)
{
  set_value_u64(server_id, v_payload, 8);
}
u64 packet_conn_granted::get_server_id()
{
  return get_value_u64(v_payload, 8);
}

//**************************************************
//  Get/set video length in bytes
//**************************************************
void packet_conn_granted::set_video_size(u64 video_size)
{
  set_value_u64(video_size, v_payload, 16);
}
u64 packet_conn_granted::get_video_size()
{
  return get_value_u64(v_payload, 16);
}

//**************************************************
//  Get/set video duration in msec
//**************************************************
void packet_conn_granted::set_video_duration(u32 video_duration)
```

```
{
  set_value_u32(video_duration, v_payload, 24);
}
u32 packet_conn_granted::get_video_duration()
{
  return get_value_u32(v_payload, 24);
}

//***************************************************
//  Get/set total number of bb in the video
//***************************************************
void packet_conn_granted::set_bb_total(u16 number_of_bb)
{
  set_value_u16(number_of_bb, v_payload, 28);
}
u16 packet_conn_granted::get_bb_total()
{
  return get_value_u16(v_payload, 28);
}

//***************************************************
//  Get/set total number of db in the video
//***************************************************
void packet_conn_granted::set_db_total(u16 number_of_db)
{
  set_value_u16(number_of_db, v_payload, 30);
}
u16 packet_conn_granted::get_db_total()
{
  return get_value_u16(v_payload, 30);
}

//***************************************************
//  Get/set the size of the db (bytes)
//***************************************************
void packet_conn_granted::set_db_size(u16 size_of_db)
{
  set_value_u16(size_of_db, v_payload, 32);
}
u16 packet_conn_granted::get_db_size()
{
  return get_value_u16(v_payload, 32);
}

//***************************************************
//  Get/set the video_header_size, the DBs needed
//  before playback can begin duing startup and skip
//***************************************************
void packet_conn_granted::set_video_header_size(u32 video_header_size)
{
  set_value_u32(video_header_size, v_payload, 34);
}
u32 packet_conn_granted::get_video_header_size()
{
  return get_value_u32(v_payload, 34);
}
```

```
//**************************************************
//  Get or set the recommended ca size
//**************************************************
void packet_conn_granted::set_recon_ca_size(u16 recon_ca_size)
{
  set_value_u16(recon_ca_size, v_payload, 38);
}

u16 packet_conn_granted::get_recon_ca_size()
{
  return get_value_u16(v_payload, 38);
}

//**************************************************
//  Get or set the rate of the low speed level
//**************************************************
void packet_conn_granted::set_speed_low(u16 speed_low)
{
  set_value_u16(speed_low, v_payload, 40);
}
u16 packet_conn_granted::get_speed_low()
{
  return get_value_u16(v_payload, 40);
}

//**************************************************
//  Get or set the rate of the normal speed level
//**************************************************
void packet_conn_granted::set_speed_normal(u16 speed_normal)
{
  set_value_u16(speed_normal, v_payload, 42);
}
u16 packet_conn_granted::get_speed_normal()
{
  return get_value_u16(v_payload, 42);
}

//**************************************************
//  Get or set the rate of the high speed level
//**************************************************
void packet_conn_granted::set_speed_high(u16 speed_high)
{
  set_value_u16(speed_high, v_payload, 44);
}
u16 packet_conn_granted::get_speed_high()
{
  return get_value_u16(v_payload, 44);
}

//**************************************************
//  Get or set the security mechanisms type
//**************************************************
void packet_conn_granted::set_sec_type(u16 sec_type)
{
  set_value_u16(sec_type, v_payload, 46);
}
u16 packet_conn_granted::get_sec_type()
```

```
{
  return get_value_u16(v_payload, 46);
}

//**************************************************
//  Get or set the skip distance
//**************************************************
void packet_conn_granted::set_skip_distance(u32 skip_distance)
{
  set_value_u32(skip_distance, v_payload, 48);
}
u32 packet_conn_granted::get_skip_distance()
{
  return get_value_u32(v_payload, 48);
}

//**************************************************
//  Get or set the video title
//**************************************************
void packet_conn_granted::set_video_title(char* movie_title)
{
  strncpy(&(((char*)v_payload)[52]), movie_title, 256);

  if(strlen(movie_title) > 256 )
    ((char*)v_payload)[(52 + 256)] = '\0';
}
char* packet_conn_granted::get_video_title()
{
  return &(((char*)v_payload)[52]);
}
```

packet_conn_req.h

```
#ifndef PACKET_CONN_REQ
#define PACKET_CONN_REQ

#include "../include/packet.h"

#define c_conn_req_packet_type 40
#define c_conn_req_action_type 10
#define c_conn_req_payload_len 140

class packet_conn_req : public packet
{
 public:
  packet_conn_req(u64 recv_id, u64 snd_id);

  ~packet_conn_req();

  void set_ccp_port(u16 ccp_port);
  u16 get_ccp_port();

  void set_dcp_port(u16 dcp_port);
  u16 get_dcp_port();
```

```
  void set_video_id(u64 video_id);
  u64 get_video_id();

  void set_username(char* username);
  char* get_username();

  void set_password(char* password);
  char* get_password();
};

#endif
```

packet_conn_req.cpp

```
#include "stdafx.h"
#include "../include/packet_conn_req.h"

//*************************************************
//  Constructor
//*************************************************
packet_conn_req::packet_conn_req(u64 recv_id, u64 snd_id)
: packet(c_conn_req_packet_type, c_conn_req_action_type, recv_id, snd_id)
{
  v_payload = malloc(c_conn_req_payload_len);
  set_payload_len(c_conn_req_payload_len);
}

//*************************************************
//  deconstructor
//*************************************************
packet_conn_req::~packet_conn_req()
{
  free(v_payload);
}

//*************************************************
//  get or set the CCP port
//*************************************************
void packet_conn_req::set_ccp_port(u16 ccp_port)
{
  set_value_u16(ccp_port, v_payload, 0);
}
u16 packet_conn_req:: get_ccp_port()
{
  return get_value_u16(v_payload, 0);
}

//*************************************************
//  get or set the DCP port
//*************************************************
void packet_conn_req::set_dcp_port(u16 dcp_port)
{
  set_value_u16(dcp_port, v_payload, 2);
```

```
}
u16 packet_conn_req::get_dcp_port()
{
  return get_value_u16(v_payload, 2);
}

//*************************************************
//  get or set the video id
//*************************************************
void packet_conn_req::set_video_id(u64 video_id)
{
  set_value_u64(video_id, v_payload, 4);
}
u64 packet_conn_req::get_video_id()
{
  return get_value_u64(v_payload, 4);
}

//*************************************************
//  get or set the username
//*************************************************
void packet_conn_req::set_username(char* username)
{
  strncpy(&(((char*)v_payload)[12]), username, 64);

  if(strlen(username) > 64 )
    ((char*)v_payload)[(12 + 64)] = '\0';
}
char* packet_conn_req::get_username()
{
  return &(((char*)v_payload)[12]);
}

//*************************************************
//  get or set the password
//*************************************************
void packet_conn_req::set_password(char* password)
{
  strncpy(&(((char*)v_payload)[76]), password, 64);

  if(strlen(password) > 64 )
    ((char*)v_payload)[(76 + 64)] = '\0';
}
char* packet_conn_req::get_password()
{
  return &(((char*)v_payload)[76]);
}
```

packet.h

```
#ifndef PACKET
#define PACKET

#include <string.h>
```

```
//#include "winsock2.h"
#include <stdlib.h>
#include <iostream>
#include "../include/types.h"
#include "../include/global_functions.h"

#define c_header_len 28

class packet {

protected:
  void* v_header;

  void* v_payload;
  u32 v_payload_len;

  void* v_payload_data;
  u32 v_payload_data_len;

  system_time v_create_time;

  void set_value_u64(u64 value, void* target, int pos);
  u64 get_value_u64(void* source, int pos);

  void set_value_u32(u32 value, void* target, int pos);
  u32 get_value_u32(void* source, int pos);

  void set_value_u16(u16 value, void* target, int pos);
  u16 get_value_u16(void* source, int pos);

  void set_value_u8(u8 value, void* target, int pos);
  u8 get_value_u8(void* source, int pos);

  void set_value_s32(s32 value, void* target, int pos);
  s32 get_value_s32(void* source, int pos);

 public:
  packet(u16 packet_type, u16 action_type, u64 recv_id, u64 snd_id);

  packet(void* header, void* payload, u32 payload_len, void* payload_data, u32 payload_data_len);

  packet(void* header);

  ~packet();

  u8 get_packet_type();
  u8 get_action_type();

  void* get_header();

  void set_payload(void* payload, u32 payload_len);
  void* get_payload();

  void set_payload_len(u32 payload_len);
  u32 get_payload_len();

  void set_payload_data(void* payload_data, u32 payload_data_len);
```

```
  void* get_payload_data();

  void set_payload_data_len(u32 payload_data_len);
  u32 get_payload_data_len();

  long wait_time_usec();

  void print();

  unsigned int get_size();

  u64 get_recv_client_id();

  u64 get_snd_client_id();


  // Old client_struct_addr
  u64 client_ip;
  u16 client_DCP_port;
  u16 client_CCP_port;
  u64 client_id;
  SOCKET sd;
};

#endif
```

packet.cpp

```
#include "stdafx.h"
#include "../include/packet.h"

packet::packet(u16 packet_type, u16 action_type, u64 recv_id, u64 snd_id)
{
  v_create_time.sec = 0;
  v_create_time.msec = 0;

  get_system_time(&v_create_time);

  v_header = malloc(c_header_len);

  v_payload_len = 0;
  v_payload_data_len = 0;

  set_value_u8(1, v_header, 0); // Version number
  set_value_u8(0, v_header, 1); // Option
  set_value_u8(packet_type, v_header, 2); // Packet type
  set_value_u8(action_type, v_header, 3); // Action type
  set_value_u32(0, v_header, 4); // payload size
  set_value_u32(0, v_header, 8); // payload_data size
  set_value_u64(recv_id, v_header, 12);
  set_value_u64(snd_id, v_header, 20);
}

packet::packet(void* header,
```

```
               void* payload,
               u32 payload_len,
               void* payload_data,
               u32 payload_data_len)
{
  v_create_time.sec = 0;
  v_create_time.msec = 0;

  get_system_time(&v_create_time);

  v_header = header;

  v_payload = payload;
  v_payload_len = payload_len;

  v_payload_data = payload_data;
  v_payload_data_len = payload_data_len;
}

packet::packet(void* header)
{
  v_create_time.sec = 0;
  v_create_time.msec = 0;

  get_system_time(&v_create_time);

  v_header = header;

  v_payload_len = 0;
  v_payload_data_len = 0;
}


// Deconstructor
packet::~packet()
{
  free(v_header);
}


void* packet::get_header()
{
  return v_header;
}


u8 packet::get_packet_type()
{
  return get_value_u8(v_header, 2);
}


u8 packet::get_action_type()
{
  return get_value_u8(v_header, 3);
}
```

```
// set/get payload
void packet::set_payload(void* payload, u32 payload_len)
{
  v_payload = payload;
  set_payload_len(payload_len);
}

void* packet::get_payload()
{
  return v_payload;
}


// set/get payload len
void packet::set_payload_len(u32 payload_len)
{
  v_payload_len = payload_len;
  set_value_u32(v_payload_len, v_header, 4); // payload size
}

u32 packet::get_payload_len()
{
  return get_value_u32(v_header, 4);
}


// set/get payload data
void packet::set_payload_data(void* payload_data, u32 payload_data_len)
{
  v_payload_data = payload_data;
  set_payload_data_len(payload_data_len);
}

void* packet::get_payload_data()
{
  return v_payload_data;
}

// set/get payload_data len
void packet::set_payload_data_len(u32 payload_data_len)
{
  v_payload_data_len = payload_data_len;
  set_value_u32(v_payload_data_len, v_header, 8); // payload data size
}

u32 packet::get_payload_data_len()
{
  return get_value_u32(v_header, 8);
}


// Get and set 8 byte
void packet::set_value_u64(u64 value, void* target, int pos)
{
  *((u64*)&(((char*)target)[pos])) = htonl(value);
```

```
  if( sizeof(u64) == 4 )
    *((u64*)&(((char*)target)[(pos+4)])) = 0;
}

u64 packet::get_value_u64(void* source, int pos)
{
  return ntohl(*((u64*)(&((char*)source)[pos])));
}


// Get and set 4 byte (unsigned int)
void packet::set_value_u32(u32 value, void* target, int pos)
{
  *((u32*)&(((char*)target)[pos])) = htonl(value);
}

unsigned int packet::get_value_u32(void* source, int pos)
{
  return ntohl(*((u32*)(&((char*)source)[pos])));
}


// Get and set 2 byte (unsigned short)
void packet::set_value_u16(u16 value, void* target, int pos)
{
  *((u16*)(&((char*)target)[pos])) = htons(value);
}

u16 packet::get_value_u16(void* source, int pos)
{
  return ntohs(*((u16*)(&((char*)source)[pos])));
}



// Get and set 1 byte (unsigned short)
void packet::set_value_u8(u8 value, void* target, int pos)
{
  ((char*)target)[pos] = (char)value;
}

u8 packet::get_value_u8(void* source, int pos)
{
  return (u8)(((char*)source)[pos]);
}


// Get and set 4 byte (int)
void packet::set_value_s32(s32 value, void* target, int pos)
{
  *((s32*)(&((char*)target)[pos])) = htonl(value);
}

s32 packet::get_value_s32(void* source, int pos)
{
  return ntohl(*((s32*)(&((char*)source)[pos])));
}
```

```
// Print function
void packet::print()
{
  std::cout << "Header length: " << c_header_len << std::endl;
  for(unsigned int i = 0; i < c_header_len; i++ )
    std::cout << (int)((char*)v_header)[i] << std::endl;

  std::cout << std::endl;

  std::cout << "Payload length: " << v_payload_len << std::endl;
  for(unsigned int i = 0; i < v_payload_len; i++ )
    std::cout << (int)((char*)v_payload)[i] << std::endl;

  std::cout << std::endl;

  std::cout << "Payload data length: " << v_payload_data_len << std::endl;
  for(unsigned int i = 0; i < v_payload_data_len; i++ )
    std::cout << (int)((char*)v_payload_data)[i] << std::endl;
}


long packet::wait_time_usec()
{
  system_time current_time = {0,0};
  get_system_time(&current_time);

  long result = ((v_create_time.sec - current_time.sec) * 1000) +
                (v_create_time.msec - current_time.msec);

  if( result > 600000000 && result < 0 )
    result = -1;

  return result;
}


unsigned int packet::get_size()
{
  return (c_header_len + (unsigned int)v_payload_len + (unsigned int)v_payload_data_len);
}


u64 packet::get_recv_client_id()
{
  return get_value_u64(v_header, 10);
}


u64 packet::get_snd_client_id()
{
  return get_value_u64(v_header, 18);
}
```

packet_data.h

```
#ifndef PACKET_DATA
#define PACKET_DATA

#include "../include/packet.h"

#define c_data_packet_type 50
#define c_data_action_type 10
#define c_data_payload_len 5

class packet_data : public packet
{
 public:
  packet_data(u64 recv_id, u64 snd_id);

  ~packet_data();

  void set_bb(u16 bb);
  u16 get_bb();

  void set_db(u16 db);
  u16 get_db();

  void set_speed_level(u8 speed_level);
  u8 get_speed_level();

  void set_data(void* data, int len);
  void* get_data(int &len);
};

#endif
```

packet_data.cpp

```
#include "stdafx.h"
#include "../include/packet_data.h"

//**************************************************
//  constructor
//**************************************************
packet_data::packet_data(u64 recv_id, u64 snd_id)
: packet(c_data_packet_type, c_data_action_type, recv_id, snd_id)
{
  v_payload = malloc(c_data_payload_len);
  set_payload_len(c_data_payload_len);
}

//**************************************************
//  deconstructor
//**************************************************
packet_data::~packet_data()
{
```

```
  free(v_payload);
}

//**************************************************
// Get or set the bb number
//**************************************************
void packet_data::set_bb(u16 bb)
{
  set_value_u16(bb, v_payload, 0);
}
u16 packet_data::get_bb()
{
  return get_value_u16(v_payload, 0);
}

//**************************************************
// Get or set the db number
//**************************************************
void packet_data::set_db(u16 db)
{
  set_value_u16(db, v_payload, 2);
}
u16 packet_data::get_db()
{
  return get_value_u16(v_payload, 2);
}

//**************************************************
// Get or set the speed level
//**************************************************
void packet_data::set_speed_level(u8 speed_level)
{
  set_value_u8(speed_level, v_payload, 4);
}
u8 packet_data::get_speed_level()
{
  return get_value_u8(v_payload, 4);
}


//**************************************************
// Get or set the video data
//**************************************************
void packet_data::set_data(void* data, int len)
{
  set_payload_data(data, (u32)len);
}
void* packet_data::get_data(int &len)
{
  len = get_payload_data_len();
  return get_payload_data();
}
```

packet_disconn.h

```
#ifndef PACKET_DISCONN
#define PACKET_DISCONN

#include "../include/packet.h"

#define c_disconn_packet_type 40
#define c_disconn_action_type 70
#define c_disconn_payload_len 0

class packet_disconn : public packet
{
 public:
  packet_disconn(u64 recv_id, u64 snd_id);

  ~packet_disconn();
};

#endif
```

packet_disconn.cpp

```
#include "stdafx.h"
#include "../include/packet_disconn.h"

//**************************************************
// Constructor
//**************************************************
packet_disconn::packet_disconn(u64 recv_id, u64 snd_id)
: packet(c_disconn_packet_type, c_disconn_action_type, recv_id, snd_id)
{
  set_payload_len(c_disconn_payload_len);
}

//**************************************************
// Deconstructor
//**************************************************
packet_disconn::~packet_disconn()
{
  free(v_payload);
}
```

packet_end_of_data.h

```
#ifndef PACKET_END_OF_DATA
#define PACKET_END_OF_DATA
```

```
#include "../include/packet.h"

#define c_end_of_data_packet_type 50
#define c_end_of_data_action_type 30
#define c_end_of_data_payload_len 0

class packet_end_of_data : public packet
{
 public:
  packet_end_of_data(u64 recv_id, u64 snd_id);

  ~packet_end_of_data();
};

#endif
```

packet_end_of_data.cpp

```
#include "stdafx.h"
#include "../include/packet_end_of_data.h"

//**************************************************
// Constructor
//**************************************************
packet_end_of_data::packet_end_of_data(u64 recv_id, u64 snd_id)
: packet(c_end_of_data_packet_type, c_end_of_data_action_type, recv_id, snd_id)
{
  set_payload_len(c_end_of_data_payload_len);
}

//**************************************************
// Deconstructor
//**************************************************
packet_end_of_data::~packet_end_of_data()
{

}
```

packet_interact_pause.h

```
#ifndef PACKET_INTERACT_PAUSE
#define PACKET_INTERACT_PAUSE

#include "../include/packet.h"

#define c_interact_pause_packet_type 40
#define c_interact_pause_action_type 30
#define c_interact_pause_payload_len 2

class packet_interact_pause : public packet
{
```

```
 public:
  packet_interact_pause(u64 recv_id, u64 snd_id);

  ~packet_interact_pause();

  void set_play_block(u16 play_block);
  u16 get_play_block();
};

#endif
```

packet_interact_pause.cpp

```
#include "stdafx.h"
#include "../include/packet_interact_pause.h"

//**************************************************
// Constructor
//**************************************************
packet_interact_pause::packet_interact_pause(u64 recv_id, u64 snd_id)
: packet(c_interact_pause_packet_type, c_interact_pause_action_type, recv_id, snd_id)
{
  v_payload = malloc(c_interact_pause_payload_len);
  set_payload_len(c_interact_pause_payload_len);
}

//**************************************************
// Deconstructor
//**************************************************
packet_interact_pause::~packet_interact_pause()
{
  free(v_payload);
}

//**************************************************
// Get or set the play block
//**************************************************
void packet_interact_pause::set_play_block(u16 play_block)
{
  set_value_u16(play_block, v_payload, 0);
}
u16 packet_interact_pause::get_play_block()
{
  return get_value_u16(v_payload, 0);
}
```

packet_interact_pause_stream.h

```
#ifndef PACKET_INTERACT_PAUSE_STREAM
#define PACKET_INTERACT_PAUSE_STREAM
```

```
#include "../include/packet.h"

#define c_interact_pause_stream_packet_type 40
#define c_interact_pause_stream_action_type 40
#define c_interact_pause_stream_payload_len 0

class packet_interact_pause_stream : public packet
{
 public:
  packet_interact_pause_stream(u64 recv_id, u64 snd_id);

  ~packet_interact_pause_stream();
};

#endif
```

packet_interact_pause_stream.cpp

```
#include "stdafx.h"
#include "../include/packet_interact_pause_stream.h"

//*************************************************
// Constructor
//*************************************************
packet_interact_pause_stream::packet_interact_pause_stream(u64 recv_id, u64 snd_id)
: packet(c_interact_pause_stream_packet_type, c_interact_pause_stream_action_type, recv_id, snd_id)
{
  set_payload_len(c_interact_pause_stream_payload_len);
}

//*************************************************
// Deconstructor
//*************************************************
packet_interact_pause_stream::~packet_interact_pause_stream()
{
  free(v_payload);
}
```

packet_interact_resume.h

```
#ifndef PACKET_INTERACT_RESUME
#define PACKET_INTERACT_RESUME

#include "../include/packet.h"

#define c_interact_resume_packet_type 40
#define c_interact_resume_action_type 50
#define c_interact_resume_payload_len 2
```

```
class packet_interact_resume : public packet
{
 public:
  packet_interact_resume(u64 recv_id, u64 snd_id);

  ~packet_interact_resume();

  void set_target_block(u16 target_block);
  u16 get_target_block();
};

#endif
```

packet_interact_resume.cpp

```
#include "stdafx.h"
#include "../include/packet_interact_resume.h"

//*************************************************
// Constructor
//*************************************************
packet_interact_resume::packet_interact_resume(u64 recv_id, u64 snd_id)
: packet(c_interact_resume_packet_type, c_interact_resume_action_type, recv_id, snd_id)
{
  v_payload = malloc(c_interact_resume_payload_len);
  set_payload_len(c_interact_resume_payload_len);
}

//*************************************************
// Deconstructor
//*************************************************
packet_interact_resume::~packet_interact_resume()
{
  free(v_payload);
}

//*************************************************
// Get or set the target block
//*************************************************
void packet_interact_resume::set_target_block(u16 target_block)
{
  set_value_u16(target_block, v_payload, 0);
}
u16 packet_interact_resume::get_target_block()
{
  return get_value_u16(v_payload, 0);
}
```

packet_interact_skip.h

```
#ifndef PACKET_INTERACT_SKIP
#define PACKET_INTERACT_SKIP

#include "../include/packet.h"

#define c_interact_skip_packet_type 40
#define c_interact_skip_action_type 60
#define c_interact_skip_payload_len 2

class packet_interact_skip : public packet
{
 public:
  packet_interact_skip(u64 recv_id, u64 snd_id);

  ~packet_interact_skip();

  void set_target_block(u16 target_block);
  u16 get_target_block();
};

#endif
```

packet_interact_skip.cpp

```
#include "stdafx.h"
#include "../include/packet_interact_skip.h"

//*************************************************
// Constructor
//*************************************************
packet_interact_skip::packet_interact_skip(u64 recv_id, u64 snd_id)
: packet(c_interact_skip_packet_type, c_interact_skip_action_type, recv_id, snd_id)
{
  v_payload = malloc(c_interact_skip_payload_len);
  set_payload_len(c_interact_skip_payload_len);
}

//*************************************************
// Deconstructor
//*************************************************
packet_interact_skip::~packet_interact_skip()
{
  free(v_payload);
}

//*************************************************
// Get or set the target block
//*************************************************
void packet_interact_skip::set_target_block(u16 target_block)
{
  set_value_u16(target_block, v_payload, 0);
```

```
}
u16 packet_interact_skip::get_target_block()
{
  return get_value_u16(v_payload, 0);
}
```

packet_no_data.h

```
#ifndef PACKET_NO_DATA
#define PACKET_NO_DATA

#include "../include/packet.h"

#define c_no_data_packet_type 50
#define c_no_data_action_type 20
#define c_no_data_payload_len 8

class packet_no_data : public packet
{
 public:
  packet_no_data(u64 recv_id, u64 snd_id);

  ~packet_no_data();

  void set_req_block_bb(u16 bb);
  u16 get_req_block_bb();

  void set_req_block_db(u16 db);
  u16 get_req_block_db();

  void set_next_block_bb(u16 bb);
  u16 get_next_block_bb();

  void set_next_block_db(u16 db);
  u16 get_next_block_db();

};

#endif
```

packet_no_data.cpp

```
#include "stdafx.h"
#include "../include/packet_no_data.h"

//**************************************************
// Constructor
//**************************************************
packet_no_data::packet_no_data(u64 recv_id, u64 snd_id)
: packet(c_no_data_packet_type, c_no_data_action_type, recv_id, snd_id)
```

```
{
  v_payload = malloc(c_no_data_payload_len);
  set_payload_len(c_no_data_payload_len);
}

//***************************************************
// Deconstructor
//***************************************************
packet_no_data::~packet_no_data()
{
  free(v_payload);
}

//***************************************************
// Get or set the request bb
//***************************************************
void packet_no_data::set_req_block_bb(u16 bb)
{
  set_value_u16(bb, v_payload, 0);
}
u16 packet_no_data::get_req_block_bb()
{
  return get_value_u16(v_payload, 0);
}

//***************************************************
// Get or set the request db
//***************************************************
void packet_no_data::set_req_block_db(u16 db)
{
  set_value_u16(db, v_payload, 2);
}
u16 packet_no_data::get_req_block_db()
{
  return get_value_u16(v_payload, 2);
}

//***************************************************
// Get or set the next bb
//***************************************************
void packet_no_data::set_next_block_bb(u16 bb)
{
  set_value_u16(bb, v_payload, 4);
}
u16 packet_no_data::get_next_block_bb()
{
  return get_value_u16(v_payload, 4);
}

//***************************************************
// Get or set the next db
//***************************************************
void packet_no_data::set_next_block_db(u16 db)
{
  set_value_u16(db, v_payload, 6);
}
u16 packet_no_data::get_next_block_db()
```

```
{
  return get_value_u16(v_payload, 6);
}
```

packet_queue.h

```
#ifndef PACKET_QUEUE
#define PACKET_QUEUE

#include <queue>
#include "../include/packet.h"
#include "../include/vod_exception.h"

using namespace std;

class packet_queue {

private:
  int foo;

  queue<packet*> v_queue;
  unsigned int v_max_size;
  unsigned int v_used_space;
  HANDLE v_mutex;
  HANDLE v_semaphore;

 public:
  packet_queue(unsigned int max_size);
  void insert_packet(packet* p);
  packet* get_next_packet();
};

#endif
```

packet_queue.cpp

```
#include "stdafx.h"
#include "../include/packet_queue.h"

//**************************************************
// Constructor
//**************************************************
packet_queue::packet_queue(unsigned int max_size)
{
  v_max_size = max_size;
  v_used_space = 0;

  v_mutex = CreateMutex(NULL, false, NULL);
  v_semaphore = CreateSemaphore(NULL, 0, 100000, NULL);
}
```

```
//*************************************************
// Insert a packet into queue
//*************************************************
void packet_queue::insert_packet(packet* p)
{
  DWORD wait_result = WaitForSingleObject(v_mutex, INFINITE);

  if( wait_result == WAIT_OBJECT_0 ) {
    if( v_max_size < v_used_space + p->get_size() ) {
      System::Diagnostics::Debug::WriteLine("Packet queue is full.");
    }
    else {
      v_queue.push(p);
      v_used_space += p->get_size();
    }
  }

  ReleaseMutex(v_mutex);
  ReleaseSemaphore(v_semaphore, 1, NULL);
}

//*************************************************
// Get the next packet awaiting in queue
//*************************************************
packet* packet_queue::get_next_packet()
{
  packet* rv = NULL;

  DWORD wait_result_sema = WaitForSingleObject(v_semaphore, INFINITE);

  if( wait_result_sema == WAIT_OBJECT_0 ) {

    DWORD wait_result = WaitForSingleObject(v_mutex, INFINITE);

    if( wait_result == WAIT_OBJECT_0 ) {

      if(!v_queue.empty()) {
        rv = v_queue.front();
        v_queue.pop();
        v_used_space -= rv->get_size();
      }

    ReleaseMutex(v_mutex);
    }
  }
  return rv;
}
```

packet_recv_data_stream.h

```
#ifndef PACKET_RECV_DATA_STREAM
#define PACKET_RECV_DATA_STREAM
```

```
#include "../include/packet.h"
#include "../include/vod_exception.h"

#define c_recv_data_stream_packet_type 10
#define c_recv_data_stream_action_type 50
#define c_recv_data_stream_payload_len 2 // Not including the DB's

class packet_recv_data_stream : public packet
{
 private:
  bool v_initialized;

  int calc_offset(int client_num);
 public:
  packet_recv_data_stream(u64 recv_id, u64 snd_id);

  ~packet_recv_data_stream();

  void init(u16 number_of_clients, u32 number_of_db_in_video);

  void set_client_id(int client_num, u64 client_id);
  u64 get_client_id(int client_num);

  void set_client_ip4(int client_num, s32 client_ip4);
  s32 get_client_ip4(int client_num);

  void set_client_port(int client_num, u16 client_port);
  u16 get_client_port(int client_num);

  void set_start_bb(int client_num, u16 start_bb);
  u16 get_start_bb(int client_num);

  void set_start_db(int client_num, u16 start_db);
  u16 get_start_db(int client_num);

  u16 get_number_of_db(int client_num);

  void set_db_list(int client_num, u16* db_list);
  u16 get_db(int client_num, int offset);
};

#endif
```

packet_recv_data_stream.cpp

```
#include "stdafx.h"
#include "../include/packet_recv_data_stream.h"

//*************************************************
// Constructor
//*************************************************
packet_recv_data_stream::packet_recv_data_stream(u64 recv_id, u64 snd_id)
: packet(c_recv_data_stream_packet_type, c_recv_data_stream_action_type, recv_id, snd_id)
```

```
{
  set_payload_len(0);
  v_initialized = false;
}

//*************************************************
// Deconstructor
//*************************************************
packet_recv_data_stream::~packet_recv_data_stream()
{
  if(v_payload_len > 0)
    free(v_payload);
}

//*************************************************
// Initialize packet
//*************************************************
void packet_recv_data_stream::init(u16 number_of_clients, u32 number_of_db_in_video)
{
  if(v_initialized)
    throw new packet_exception("Packet is already initialized");
  else {
    // Calculates the total size of the packet
    v_payload_len = c_recv_data_stream_payload_len +
                    (12 * number_of_clients) +
                    number_of_db_in_video;
    v_payload = malloc(v_payload_len);
    set_value_u16(number_of_clients, v_payload, 0);
    v_initialized = true;
  }
}

//*************************************************
// get/set client_id
//*************************************************
void packet_recv_data_stream::set_client_id(int client_num, u64 client_id)
{
  set_value_u64(client_id, v_payload, 0 + calc_offset(client_num));
}
u64 packet_recv_data_stream::get_client_id(int client_num)
{
  return get_value_u64(v_payload, 0 + calc_offset(client_num));
}

//*************************************************
// get/set client_ip
//*************************************************
void packet_recv_data_stream::set_client_ip4(int client_num, s32 client_ip4)
{
  set_value_s32(client_ip4, v_payload, 8 + calc_offset(client_num));
}
s32 packet_recv_data_stream::get_client_ip4(int client_num)
{
  return get_value_s32(v_payload, 8 + calc_offset(client_num));
}

//*************************************************
```

```
// get/set client_port
//**************************************************
void packet_recv_data_stream::set_client_port(int client_num, u16 client_port)
{
  set_value_u16(client_port, v_payload, 24 + calc_offset(client_num));
}

u16 packet_recv_data_stream::get_client_port(int client_num)
{
  return get_value_u16(v_payload, 24 + calc_offset(client_num));
}

//**************************************************
// get/set start_bb
//**************************************************
void packet_recv_data_stream::set_start_bb(int client_num, u16 start_bb)
{
  set_value_u16(start_bb, v_payload, 26 + calc_offset(client_num));
}

u16 packet_recv_data_stream::get_start_bb(int client_num)
{
  return get_value_u16(v_payload, 26 + calc_offset(client_num));
}

//**************************************************
// get/set start_db
//**************************************************
void packet_recv_data_stream::set_start_db(int client_num, u16 start_db)
{
  set_value_u16(start_db, v_payload, 28 + calc_offset(client_num));
}

u16 packet_recv_data_stream::get_start_db(int client_num)
{
  return get_value_u16(v_payload, 28 + calc_offset(client_num));
}

//**************************************************
// get number_of_db
//**************************************************
u16 packet_recv_data_stream::get_number_of_db(int client_num)
{
  return get_value_u16(v_payload, 30 + calc_offset(client_num));
}

//**************************************************
// get/set dbs
//**************************************************
void packet_recv_data_stream::set_db_list(int client_num, u16* db_list)
{
  for(int i = 0; i < (int)(get_value_u16(v_payload, 30 + calc_offset(client_num))); i++) {
    set_value_u16(db_list[i],
                  v_payload,
                  (32 + calc_offset(client_num)) + (i * sizeof(u16)));
  }
}
```

```
u16 packet_recv_data_stream::get_db(int client_num, int offset)
{
  return get_value_u16(v_payload, 32 + ((calc_offset(client_num)) * sizeof(u16)));
}

//*************************************************
// Calculate offset. Private.
//*************************************************
int packet_recv_data_stream::calc_offset(int client_num)
{
  int pos = 30;

  for( int i = 0; i < client_num; i++ ) {
    pos += (get_value_u16(v_payload, pos) * 2) + 30;
  }

  pos = pos - 30;
  return pos;
}
```

packet_req_backup_data.h

```
#ifndef PACKET_REQ_BACKUP_DATA
#define PACKET_REQ_BACKUP_DATA

#include "../include/packet.h"

#define c_req_backup_data_packet_type 40
#define c_req_backup_data_action_type 100
#define c_req_backup_data_payload_len 4

class packet_req_backup_data : public packet
{
 public:
  packet_req_backup_data(u64 recv_id, u64 snd_id);

  ~packet_req_backup_data();

  void set_bb(u16 bb);
  u16 get_bb();

  void set_db(u16 db);
  u16 get_db();
};

#endif
```

packet_req_backup_data.cpp

```
#include "stdafx.h"
```

```
#include "../include/packet_req_backup_data.h"

//**************************************************
// Constructor
//**************************************************
packet_req_backup_data::packet_req_backup_data(u64 recv_id, u64 snd_id)
: packet(c_req_backup_data_packet_type, c_req_backup_data_action_type, recv_id, snd_id)
{
  v_payload = malloc(c_req_backup_data_payload_len);
  set_payload_len(c_req_backup_data_payload_len);
}

//**************************************************
// Deconstructor
//**************************************************
packet_req_backup_data::~packet_req_backup_data()
{
  free(v_payload);
}

//**************************************************
// get or set bb
//**************************************************
void packet_req_backup_data::set_bb(u16 bb)
{
  set_value_u16(bb, v_payload, 0);
}

u16 packet_req_backup_data::get_bb()
{
  return get_value_u16(v_payload, 0);
}

//**************************************************
// get or set db
//**************************************************
void packet_req_backup_data::set_db(u16 db)
{
  set_value_u16(db, v_payload, 2);
}

u16 packet_req_backup_data::get_db()
{
  return get_value_u16(v_payload, 2);
}
```

packet_resend_data.h

```
#ifndef PACKET_RESEND_DATA
#define PACKET_RESEND_DATA

#include "../include/packet.h"
```

```
#define c_resend_data_packet_type 60
#define c_resend_data_action_type 10
#define c_resend_data_payload_len 4

class packet_resend_data : public packet
{
 public:
  packet_resend_data(u64 recv_id, u64 snd_id);

  ~packet_resend_data();

  void set_bb(u16 bb);
  u16 get_bb();

  void set_db(u16 db);
  u16 get_db();
};

#endif
```

packet_resend_data.cpp

```
#include "stdafx.h"
#include "../include/packet_resend_data.h"

//*************************************************
// Constructor
//*************************************************
packet_resend_data::packet_resend_data(u64 recv_id, u64 snd_id)
: packet(c_resend_data_packet_type, c_resend_data_action_type, recv_id, snd_id)
{
  v_payload = malloc(c_resend_data_payload_len);
  set_payload_len(c_resend_data_payload_len);
}

//*************************************************
// Deconstructor
//*************************************************
packet_resend_data::~packet_resend_data()
{
  free(v_payload);
}

//*************************************************
// Get or set the bb
//*************************************************
void packet_resend_data::set_bb(u16 bb)
{
  set_value_u16(bb, v_payload, 0);
}

u16 packet_resend_data::get_bb()
{
```

```
  return get_value_u16(v_payload, 0);
}

//***************************************************
// Get or set the db
//***************************************************
void packet_resend_data::set_db(u16 db)
{
  set_value_u16(db, v_payload, 2);
}

u16 packet_resend_data::get_db()
{
  return get_value_u16(v_payload, 2);
}
```

packet_snd_data_stream.h

```
#ifndef PACKET_SND_DATA_STREAM
#define PACKET_SND_DATA_STREAM

#include "../include/packet.h"
#include "../include/vod_exception.h"

#define c_snd_data_stream_packet_type 20
#define c_snd_data_stream_action_type 20
#define c_snd_data_stream_payload_len 32 // Not including the DB's

class packet_snd_data_stream : public packet
{
 private:
  bool v_initialized;

 public:
  packet_snd_data_stream(u64 recv_id, u64 snd_id);

  ~packet_snd_data_stream();

  void init(u16 number_of_db);

  void set_client_id(u64 client_id);
  u64 get_client_id();

  void set_client_ip4(s32 client_ip4);
  s32 get_client_ip4();

  void set_client_port(u16 client_port);
  u16 get_client_port();

  void set_start_bb(u16 start_bb);
  u16 get_start_bb();

  void set_start_db(u16 start_db);
```

```
  u16 get_start_db();

  u16 get_number_of_db();

  void set_db_list(u16* db_list);
  u16 get_db(int offset);
};

#endif
```

packet_snd_data_stream.cpp

```
#include "stdafx.h"
#include "../include/packet_snd_data_stream.h"

//****************************************************
// Constructor
//****************************************************
packet_snd_data_stream::packet_snd_data_stream(u64 recv_id, u64 snd_id)
: packet(c_snd_data_stream_packet_type, c_snd_data_stream_action_type, recv_id, snd_id)
{
  set_payload_len(0);
  v_initialized = false;
}

//****************************************************
// Deconstructor
//****************************************************
packet_snd_data_stream::~packet_snd_data_stream()
{
  if(v_payload_len > 0)
    free(v_payload);
}

//****************************************************
// Initialize packet
//****************************************************
void packet_snd_data_stream::init(u16 number_of_db)
{
  if(v_initialized)
    throw new packet_exception("Packet is already initialized");
  else {
    v_payload_len = c_snd_data_stream_payload_len + (number_of_db * 2);
    v_payload = malloc(v_payload_len);
    set_payload_len(v_payload_len);
    set_value_u16(number_of_db, v_payload, 30);
    v_initialized = true;
  }
}

//****************************************************
// get/set client_ip
//****************************************************
void packet_snd_data_stream::set_client_ip4(s32 client_ip4)
```

```
{
  set_value_s32(client_ip4, v_payload, 0);
}

s32 packet_snd_data_stream::get_client_ip4()
{
  return get_value_s32(v_payload, 0);
}

//**************************************************
// get/set client_port
//**************************************************
void packet_snd_data_stream::set_client_port(u16 client_port)
{
  set_value_u16(client_port, v_payload, 16);
}

u16 packet_snd_data_stream::get_client_port()
{
  return get_value_u16(v_payload, 16);
}

//**************************************************
// get/set client_id
//**************************************************
void packet_snd_data_stream::set_client_id(u64 client_id)
{
  set_value_u64(client_id, v_payload, 18);
}

u64 packet_snd_data_stream::get_client_id()
{
  return get_value_u64(v_payload, 18);
}

//**************************************************
// get/set start_bb
//**************************************************
void packet_snd_data_stream::set_start_bb(u16 start_bb)
{
  set_value_u16(start_bb, v_payload, 26);
}

u16 packet_snd_data_stream::get_start_bb()
{
  return get_value_u16(v_payload, 26);
}

//**************************************************
// get/set start_db
//**************************************************
void packet_snd_data_stream::set_start_db(u16 start_db)
{
  set_value_u16(start_db, v_payload, 28);
}

u16 packet_snd_data_stream::get_start_db()
```

```
{
  return get_value_u16(v_payload, 28);
}

//**************************************************
// get number_of_db
//**************************************************
u16 packet_snd_data_stream::get_number_of_db()
{
  return get_value_u16(v_payload, 30);
}

//**************************************************
// get/set dbs
//**************************************************
void packet_snd_data_stream::set_db_list(u16* db_list)
{
  for(int i = 0; i < (int)(get_value_u16(v_payload, 30)); i++)
    set_value_u16(db_list[i], v_payload, 32 + (i * sizeof(u16)));
}

u16 packet_snd_data_stream::get_db(int offset)
{
  return get_value_u16(v_payload, 32 + (offset * sizeof(u16)));
}
```

packet_status_req.h

```
#ifndef PACKET_STATUS_REQ
#define PACKET_STATUS_REQ

#include "../include/packet.h"

#define c_status_req_packet_type 20
#define c_status_req_action_type 10
#define c_status_req_payload_len 4

class packet_status_req : public packet
{
 public:
  packet_status_req(u64 recv_id, u64 snd_id);

  ~packet_status_req();

  void set_status_interval(u32 status_interval);
  u32 get_status_interval();
};

#endif
```

packet_status_req.cpp

```
#include "stdafx.h"
#include "../include/packet_status_req.h"

//**************************************************
// Constructor
//**************************************************
packet_status_req::packet_status_req(u64 recv_id, u64 snd_id)
: packet(c_status_req_packet_type, c_status_req_action_type, recv_id, snd_id)
{
  v_payload = malloc(c_status_req_payload_len);
  set_payload_len(c_status_req_payload_len);
}

//**************************************************
// Deconstructor
//**************************************************
packet_status_req::~packet_status_req()
{
  free(v_payload);
}

//**************************************************
// Get or set the interval
//**************************************************
void packet_status_req::set_status_interval(u32 status_interval)
{
  set_value_u32(status_interval, v_payload, 0);
}

u32 packet_status_req::get_status_interval()
{
  return get_value_u32(v_payload, 0);
}
```

packet_status_resp.h

```
#ifndef PACKET_STATUS_RESP
#define PACKET_STATUS_RESP

#include "../include/packet.h"

#define c_status_resp_packet_type 30
#define c_status_resp_action_type 10
#define c_status_resp_payload_len 2

class packet_status_resp : public packet
{
 public:
  packet_status_resp(u64 recv_id, u64 snd_id);

  ~packet_status_resp();
```

```
  void set_pp(u16 bb);
  u16 get_pp();
};

#endif
```

packet_status_resp.cpp

```
#include "stdafx.h"
#include "../include/packet_status_resp.h"

//*************************************************
// Constructor
//*************************************************
packet_status_resp::packet_status_resp(u64 recv_id, u64 snd_id)
  : packet(c_status_resp_packet_type, c_status_resp_action_type, recv_id, snd_id)
{
  v_payload = malloc(c_status_resp_payload_len);
  set_payload_len(c_status_resp_payload_len);
}

//*************************************************
// Deconstructor
//*************************************************
packet_status_resp::~packet_status_resp()
{
  free(v_payload);
}

//*************************************************
// Get or set the playpointer
//*************************************************
void packet_status_resp::set_pp(u16 bb)
{
  set_value_u16(bb, v_payload, 0);
}

u16 packet_status_resp::get_pp()
{
  return get_value_u16(v_payload, 0);
}
```

packet_stop_stream.h

```
#ifndef PACKET_STOP_STREAM
#define PACKET_STOP_STREAM

#include "../include/packet.h"

#define c_stop_stream_packet_type 20
```

```
#define c_stop_stream_action_type 30
#define c_stop_stream_payload_len 12

class packet_stop_stream : public packet
{
 public:
  packet_stop_stream(u64 recv_id, u64 snd_id);

  ~packet_stop_stream();

  void set_client_id(u64 client_id);
  u64 get_client_id();

  void set_stop_bb(u16 stop_bb);
  u16 get_stop_bb();

  void set_stop_db(u16 stop_db);
  u16 get_stop_db();
};

#endif
```

packet_stop_stream.cpp

```
#include "stdafx.h"
#include "../include/packet_stop_stream.h"

//*************************************************
// Constructor
//*************************************************
packet_stop_stream::packet_stop_stream(u64 recv_id, u64 snd_id)
: packet(c_stop_stream_packet_type, c_stop_stream_action_type, recv_id, snd_id)
{
  v_payload = malloc(c_stop_stream_payload_len);
  set_payload_len(c_stop_stream_payload_len);
}

//*************************************************
// Deconstructor
//*************************************************
packet_stop_stream::~packet_stop_stream()
{
  free(v_payload);
}

//*************************************************
// Get or set client id
//*************************************************
void packet_stop_stream::set_client_id(u64 client_id)
{
  set_value_u64(client_id, v_payload, 0);
}

u64 packet_stop_stream::get_client_id()
```

```
{
  return get_value_u64(v_payload, 0);
}

//*************************************************
// Get or set stop bb
//*************************************************
void packet_stop_stream::set_stop_bb(u16 stop_bb)
{
  set_value_u16(stop_bb, v_payload, 8);
}

u16 packet_stop_stream::get_stop_bb()
{
  return get_value_u16(v_payload, 8);
}

//*************************************************
// Get or set stop db
//*************************************************
void packet_stop_stream::set_stop_db(u16 stop_db)
{
  set_value_u16(stop_db, v_payload, 10);
}

u16 packet_stop_stream::get_stop_db()
{
  return get_value_u16(v_payload, 10);
}
```

packet_stream_error.h

```
#ifndef PACKET_STREAM_ERROR
#define PACKET_STREAM_ERROR

#include "../include/packet.h"

#define c_stream_error_packet_type 40
#define c_stream_error_action_type 80
#define c_stream_error_payload_len 12

class packet_stream_error : public packet
{
 public:
  packet_stream_error(u64 recv_id, u64 snd_id);
  ~packet_stream_error();

  void set_client_id(u64 client_id);
  u64 get_client_id();

  void set_rate(u16 rate);
  u16 get_rate();

  void set_bb(u16 bb);
```

```
  u16 get_bb();
};

#endif
```

packet_stream_error.cpp

```
#include "stdafx.h"
#include "../include/packet_stream_error.h"


packet_stream_error::packet_stream_error(u64 recv_id, u64 snd_id)
  : packet(c_stream_error_packet_type, c_stream_error_action_type, recv_id, snd_id)
{
  v_payload = malloc(c_stream_error_payload_len);
  set_payload_len(c_stream_error_payload_len);
}


packet_stream_error::~packet_stream_error()
{
  free(v_payload);
}


void packet_stream_error::set_client_id(u64 client_id)
{
  set_value_u64(client_id, v_payload, 0);
}

u64 packet_stream_error::get_client_id()
{
  return get_value_u64(v_payload, 0);
}


void packet_stream_error::set_rate(u16 rate)
{
  set_value_u16(rate, v_payload, 8);
}


u16 packet_stream_error::get_rate()
{
  return get_value_u16(v_payload, 8);
}


void packet_stream_error::set_bb(u16 bb)
{
  set_value_u16(bb, v_payload, 10);
}

u16 packet_stream_error::get_bb()
```

```
{
  return get_value_u16(v_payload, 10);
}
```

request_queue.h

```
#ifndef REQUEST_QUEUE
#define REQUEST_QUEUE

#include <queue>
#include "../include/vod_server.h"

struct data_struct_req {
  unsigned long movie_id;
  unsigned long byte_offset;
  size_t len;
  long transaction_id;
};

using namespace std;

class request_queue {

private:
  queue<data_struct_req*> v_queue;

  unsigned int v_count_data;

  HANDLE v_mutex_data;
  HANDLE v_semaphore_data;

 public:
  request_queue();
  void add_data_request(data_struct_req* request);
  void get_data_request(struct data_struct_req& request);
  int count_data_request();
};

#endif
```

request_queue.cpp

```
#include "stdafx.h"
#include "../include/request_queue.h"


//**********************************************
// Constructor
//**********************************************
request_queue::request_queue()
{
```

```
  v_count_data = 0;

  v_mutex_data = CreateMutex(NULL, false, NULL);
  v_semaphore_data = CreateSemaphore(NULL, 0, 100, NULL);
}

//**********************************************
// Add request to queue
//**********************************************
void request_queue::add_data_request(data_struct_req* request)
{
  DWORD wait_result = WaitForSingleObject(v_mutex_data, INFINITE);

  if( wait_result == WAIT_OBJECT_0 ) {
    v_queue.push(request);
    v_count_data++;
  }

  ReleaseMutex(v_mutex_data);
  ReleaseSemaphore(v_semaphore_data, 1, NULL);
}

//**********************************************
// Get the next data request from queue
//**********************************************
void request_queue::get_data_request(struct data_struct_req& request)
{
  data_struct_req* temp = NULL;

  DWORD wait_result_sema = WaitForSingleObject(v_semaphore_data, INFINITE);

  if( wait_result_sema == WAIT_OBJECT_0 ) {

    DWORD wait_result = WaitForSingleObject(v_mutex_data, INFINITE);

    if( wait_result == WAIT_OBJECT_0 ) {

      if(!v_queue.empty()) {
        temp = v_queue.front();
        v_queue.pop();
        v_count_data--;

        memcpy((void*)&request, temp, sizeof(data_struct_req));
        delete(temp);
      }

      ReleaseMutex(v_mutex_data);
    }
  }
}

//**********************************************
// Return the number of awaiting requests.
//**********************************************
int request_queue::count_data_request()
{
  int rv;
```

```
  DWORD wait_result = WaitForSingleObject(v_mutex_data, INFINITE);

  if( wait_result == WAIT_OBJECT_0 )
    rv = v_count_data;

  ReleaseMutex(v_mutex_data);

  return rv;
}
```

socket_utils.h

```
#ifndef SOCKET_UTILS
#define SOCKET_UTILS

#include "../include/packet_data.h"


#include <sys/types.h>
#include "winsock2.h"
//#include <sys/select.h>
//#include <sys/time.h>
#include <errno.h>

#include <stdio.h>
#include <string.h>
#include <iostream>
//#include <unistd.h>
#include <fcntl.h>
#include "../include/vod_exception.h"

struct iovec {
  void *iov_base;     /* Pointer to data.  */
  size_t iov_len;     /* Length of data.  */
};



struct msghdr {
  void *msg_name;             /* Address to send to/receive from.  */
  int msg_namelen;      /* Length of address data.  */

  struct iovec *msg_iov;      /* Vector of data to send/receive into.  */
  size_t msg_iovlen;          /* Number of elements in the vector.  */

  void *msg_control;          /* Ancillary data (eg BSD filedesc passing). */
  size_t msg_controllen;      /* Ancillary data buffer length.  */
  int msg_flags;              /* Flags on received message.  */
};



SOCKET Socket(int family, int type, int protocol);
```

```
int Bind(SOCKET sockfd, const struct sockaddr* myaddr, int addrlen);

SOCKET Accept(SOCKET sockfd, struct sockaddr* myaddr, int* addrlen);

void Listen(SOCKET fd, int backlog);

int Close(SOCKET fd);

int Sendmsg(SOCKET sd, struct msghdr *msg, int flags);

int Recvmsg(SOCKET sd, struct msghdr *msg, int flags);

int Select(int, fd_set*, fd_set*, fd_set*, timeval*);

int Connect(SOCKET sockfd, const struct sockaddr* serveraddr, int addrlen);

size_t Send(SOCKET sockfd, const void *buf, size_t len, int flags);

int Read(SOCKET fd, void* buf, size_t len);

size_t Recvfrom(SOCKET sd,
        void* buf,
        size_t len,
int flags,
struct sockaddr* from,
int* fromlen);

void setnblck(SOCKET sd);

int set_so_reuseable(SOCKET sd);

char* produce_error(const char*);

#endif
```

socket_utils.cpp

```
#include "stdafx.h"
#include "../include/socket_utils.h"

// *********************************************
// Wrap the socket call
// *********************************************
SOCKET Socket(int family, int type, int protocol)
{
  int n = socket(family, type, protocol);
  if ( n < 0) {
    throw new transport_exception( produce_error("Error creating socket: "));
  }
  else {
    return n;
  }
}
```

```
// ********************************************
// Wrap the bind call
// ********************************************
int Bind(SOCKET sockfd, const struct sockaddr* myaddr, int addrlen)
{
  int n = bind(sockfd, myaddr, addrlen);
  if ( n < 0) {
    throw new transport_exception( produce_error("Error binding port: "));
  }
  else {
    return n;
  }
}

// ********************************************
// Wrap the accept call
// ********************************************
SOCKET Accept(SOCKET sockfd, struct sockaddr* myaddr, int* addrlen)
{
  SOCKET n = accept(sockfd, myaddr, addrlen);
  if (n == SOCKET_ERROR) {
    throw new transport_exception( produce_error("Error accepting new connection: "));
  }
  else {
    return n;
  }
}


// ********************************************
// Wrap the send  call
// ********************************************
size_t Send(SOCKET sockfd, const void *buf, size_t len, int flags)
{
  int n = send(sockfd, (const char*) buf,len, flags);
  if (n < 0) {
    throw new transport_exception( produce_error("Error sending data to remote host: "));
  }
  else {
    return n;
  }
}

// ********************************************
// Wrap the read call
// ********************************************
int Read(SOCKET sockfd, void *buf, size_t len)
{
  int n = recv(sockfd, (char*) buf, len, 0);

  if (n < 0) {
    Close(sockfd);
    //throw new transport_exception( produce_error("Error sending data to remote host: "));
  }
  else {
    return n;
```

```
  }
}

// ********************************************
// Wrap the recvfrom call
// ********************************************
size_t Recvfrom(SOCKET sd,
                void* buf,
                size_t len,
                int flags,
                struct sockaddr* from,
                int* fromlen)
{
  int n = recvfrom(sd, (char*) buf, len, flags, from, fromlen);
  return n;
}

// ********************************************
// Wrap the connect call
// ********************************************
int Connect(SOCKET sockfd, const struct sockaddr* serveraddr, int addrlen)
{
  int n = connect(sockfd, serveraddr, addrlen);

  if (n < 0) {
    throw new transport_exception( produce_error("Error connecting to remote host: "));
  }
  else {
    return n;
  }
}

// ********************************************
// Wrap the listen call
// ********************************************
void Listen(SOCKET fd, int backlog)
{
  if ( (listen(fd, backlog) < 0)) {
    throw new transport_exception( produce_error("Error listening on socket: "));
  }
  else {
    return;
  }
}

// ********************************************
// Wrap the close call
// ********************************************
int Close(SOCKET fd)
{
  int rv = closesocket(fd);

  if (rv < 0) {
    throw new transport_exception( produce_error("Error closing socket: "));
  }
  else {
    return rv;
```

```
  }
}

// ********************************************
// Wrap the select call
// ********************************************
int Select(int maxfdp1,
           fd_set* read_set,
           fd_set* write_set,
           fd_set* except_set,
           timeval* timeout)
{
  int n = select(maxfdp1,read_set, write_set, except_set, timeout);
  if (n < 0) {
    throw new transport_exception( produce_error("Error calling select: "));
  }
  else {
    return n;
  }
}

// ********************************************
// Wrap the sendmsg call
// ********************************************
int Sendmsg(SOCKET sd, struct msghdr *msg, int flags)
{
  int bufsize = 0;
  int offset = 0;
  char* buf = NULL;

  // Loop through all iovectors in msghdr
  for(int i = 0; i <  msg->msg_iovlen; i++) {
    bufsize += msg->msg_iov[i].iov_len;
  }

  buf = (char*) malloc(bufsize);

  for(int i = 0; i < msg->msg_iovlen; i++) {
    // Do expensive memcopy due to the unavailable sendmsg call
    memcpy(buf + offset, msg->msg_iov[i].iov_base, msg->msg_iov[i].iov_len);
    offset = offset + msg->msg_iov[i].iov_len;
  }

  return  sendto(sd, buf, bufsize, 0, (sockaddr*) msg->msg_name, msg->msg_namelen);
}

// ********************************************
// Wrap the recvmsg call
// ********************************************
int Recvmsg(SOCKET sd, struct msghdr *msg, int flags)
{
  int loop = msg->msg_iovlen;
  int readlen = 0;
  int read = 0;

  char hat;
  sockaddr sock;
```

```
  int socklen = sizeof(sock);

  // Loop through all iovectors in msghdr
  for(int i = 0; i <  loop; i++) {
    readlen += msg->msg_iov[i].iov_len;
  }

  void* buf = malloc(readlen);

  Recvfrom(sd, buf, readlen, 0, &sock, &socklen);

  int offset = 0;
  for(int i = 0; i <  loop; i++) {
    // Do expensive memcopy due to the unavailable recvmsg call
    memcpy(msg->msg_iov[i].iov_base, (void*)(((char*) buf)+offset), msg->msg_iov[i].iov_len);
    offset += msg->msg_iov[i].iov_len;
  }
  free(buf);

  return read;
}

// ************************************************
// Set a file descriptor nonblocking
// ************************************************
void setnblck(SOCKET sd)
{
  int opts;
  u_long iMode = 1;
  ioctlsocket(sd, FIONBIO, &iMode);

  int sndsize = 200000;
  int err = setsockopt(sd, SOL_SOCKET, SO_RCVBUF, (char *)&sndsize, (int)sizeof(sndsize));
  return;
}

// ************************************************
// Set a socket descriptor reusable
// ************************************************
int set_so_reuseable(SOCKET sd)
{
  int optval = 1;
  int rv = setsockopt(sd, SOL_SOCKET, SO_REUSEADDR, (char*) &optval, sizeof(optval));

  if (rv < -1)
    throw new transport_exception( produce_error("Error setting SO_REUSEADDR socket option: "));
  else
    return rv;
}

// ************************************************
// Read error from errno and return
// ************************************************
char* produce_error(const char* errmsg)
{
  char* err = strerror(errno);
  char* ptr = (char*) malloc(strlen(err) + strlen(errmsg) + 1);
```

```
  memcpy(ptr, errmsg, strlen(errmsg));
  memcpy(ptr + strlen(errmsg), err, strlen(err) + 1);

  return ptr;
}
```

stream_engine.h

```
#ifndef STREAM_ENGINE
#define STREAM_ENGINE

#include "../include/types.h"
#include "../include/enum.h"
#include "../include/packet.h"
#include "../include/packet_data.h"
#include "../include/data_container.h"

class stream_engine {

 private:
  data_container* v_data_container;
  u64 v_snd_client_id;
  u64 v_video_id;
  bool v_streaming;

 public:
  stream_engine(data_container* dc, u64 video_id, u64 client_id);
  stream_engine(data_container* dc, u64 video_id);

  u64 get_video_id();
  packet* stream(u64 recv_client_id, u16 bb, u16 db, speed_level speed);
};

#endif
```

stream_engine.cpp

```
#include "stdafx.h"
#include "../include/stream_engine.h"


//****************************************************
// Constructor
//****************************************************
stream_engine::stream_engine(data_container* dc,
     u64 video_id,
     u64 client_id)
{
  v_data_container = dc;
  v_snd_client_id = client_id;
```

```
  v_video_id = video_id;
}
//***************************************************
// Constructor
//***************************************************
stream_engine::stream_engine(data_container* dc,
                       u64 video_id)
{
  v_data_container = dc;
  v_snd_client_id = 0; // All data from the server has client-id 0 in this version
  v_video_id = video_id;
}

u64 stream_engine::get_video_id()
{
  return v_video_id;
}


//***************************************************
// Retrieve a specific packet
//***************************************************
packet* stream_engine::stream(u64 recv_client_id, u16 bb, u16 db, speed_level speed)
{
  // Retrive data
  int data_len = 0;
  void* data = v_data_container->read_db(bb, db, data_len);

  // Cache / buffer miss
  if( data == NULL )
    return NULL;

  // Building data packet
  packet_data* pack = new packet_data(recv_client_id, v_snd_client_id);
  pack->set_bb(bb);
  pack->set_db(db);
  pack->set_speed_level((u8) speed);
  pack->set_data(NULL, 0);
  pack->set_payload_data(data, data_len);

  return pack;
}
```

stream_engine_server_thread.h

```
#ifndef STREAM_ENGINE_SERVER_THREAD
#define STREAM_ENGINE_SERVER_THREAD

#include "../include/transport_handler.h"
#include "../include/stream_engine.h"
#include "../include/data_bank_server.h"

struct stream_params
```

```
{
  data_bank_server*  v_data_bank;
  stream_engine*     v_stream_engine;
  transport_handler* v_transport_handler;
};

void stream_thread(void* args);

#endif
```

stream_engine_server_thread.cpp

```
#include "stdafx.h"
#include "../include/stream_engine_server_thread.h"

//**************************************************
// Main procedure
//**************************************************
void stream_thread(void* args)
{
  data_bank_server* v_data_bank = ((stream_params*)args)->v_data_bank;
  stream_engine* v_stream_engine = ((stream_params*)args)->v_stream_engine;
  transport_handler* v_transport_handler = ((stream_params*)args)->v_transport_handler;

u64 recv_client_id;
int wait_time;
u16 next_bb;
u16 next_db;
speed_level speed;
u64 ip_addr;
u16 port;

  for(;;) {
wait_time = 0;

    // get the next receiver in line
if( v_data_bank->get_next_receiver(v_stream_engine->get_video_id(),
                                 recv_client_id,
                                 wait_time,
                                 next_bb,
                                 next_db,
                                 speed,
                                 ip_addr,
                                 port) ) {

      if( wait_time > 0 ) {
        //System::Diagnostics::Debug::WriteLine("Sleeping: " + wait_time);
  System::Threading::Thread::Sleep(wait_time);
      }
      if (!(next_bb > (v_data_bank->get_num_of_bb(0)) - 1)) {
  packet* pack = v_stream_engine->stream(recv_client_id,
                                          next_bb,
                                           next_db,
                                       speed);
```

```
        if(pack == NULL) {
          // If cache miss - inform data_bank
        }
        else {
     client_address v_client_address;
     v_client_address.client_ip = ip_addr;
    v_client_address.client_DCP_port = port;
       v_transport_handler->send_DCP(pack, &v_client_address);
        }
      }
    }
    else {
      delete v_stream_engine;
      v_stream_engine = NULL;
      break;
    }
  }
return;
}
```

stream_engine_thread.h

```
#ifndef STREAM_ENGINE_THREAD
#define STREAM_ENGINE_THREAD

#include "../include/transport_handler.h"
#include "../include/stream_engine.h"
#include "../include/data_bank_client.h"

struct stream_params
{
  data_bank_client*  v_data_bank;
  stream_engine*     v_stream_engine;
  transport_handler* v_transport_handler;
};

void stream_thread(void* args);

#endif
```

stream_engine_thread.cpp

```
#include "stdafx.h"
#include "../include/stream_engine_thread.h"


// ****************************************************
// Main streaming thread procedure
// ****************************************************
```

```
void stream_thread(void* args)
{
  data_bank_client* v_data_bank = ((stream_params*)args)->v_data_bank;
  stream_engine* v_stream_engine = ((stream_params*)args)->v_stream_engine;
  transport_handler* v_transport_handler = ((stream_params*)args)->v_transport_handler;

u64 recv_client_id;
int wait_time;
u16 next_bb;
u16 next_db;
speed_level speed;
u64 ip_addr;
u16 port;

for(;;) {
wait_time = 0;

    // get the next receiver in line
if( v_data_bank->get_next_receiver(0, recv_client_id,
                                 wait_time,
                                 next_bb,
                                 next_db,
                                 speed,
                                 ip_addr,
                                 port) ) {

  if( wait_time > 0 ) {
//        System::Diagnostics::Debug::WriteLine("Sleeping: " + wait_time);
        System::Threading::Thread::Sleep(wait_time);
      }

      if (!(next_bb > (v_data_bank->get_num_of_bb()) - 1)) {

        if( next_db == 0 || next_db == 1 )
          System::Diagnostics::Debug::WriteLine("Sender BB: " + next_bb);

        packet* pack = v_stream_engine->stream(recv_client_id,
                                           next_bb,
                                            next_db,
                                           speed);
        if(pack == NULL) {
          // Data could not be found in buffer.
          // We should send error packet to receiver.
          System::Diagnostics::Debug::WriteLine("Data was not found in buffer!");
        }
        else {
    client_address v_client_address;
  v_client_address.client_ip = ip_addr;
  v_client_address.client_DCP_port = port;
  v_transport_handler->send_DCP(pack, &v_client_address);
  }
      }
    }
else
break;
}
return;
```

```
}
```

stream_info.h

```
#ifndef STREAM_INFO
#define STREAM_INFO

#include "../include/packet_snd_data_stream.h"
#include "../include/global_functions.h"
#include "../include/enum.h"
#include <queue>


struct resend_block {
  u16 bb;
  u16 db;
};


class stream_info {
  // Client id associated to the given client
  u64 v_client_id;

  // Variable defining if we have started streaming
  bool started;

  // Timestamp indicating the last time a packet was sent
  system_time v_last_send_time;
  // Timestamp indicating the next time a packet should be sent
  system_time v_next_send_time;
  // The next buffer block a packet should be sent from
  u16 v_next_buffer_block;
  // The next data block to be sent
  u16 v_next_data_block;

  // last index of
  int v_last_index;

  // The amount of millisconds between consecutive packet dispatches
  int v_low_speed_interval;
  int v_normal_speed_interval;
  int v_high_speed_interval;

  // The speed level of the connected client
  speed_level v_speed;

  // The point in the movie where the stream should be stopped
  u16 v_stop_point_db;
  u16 v_stop_point_bb;

  // Queue containing all blocks to be resend
  std::queue<struct resend_block> v_resend_queue;

  // Get the next timestamp of data dispatch
```

```
  system_time update_send_time();

 public:
  // Packet containing information about the contents of the stream
  packet_snd_data_stream* v_pack;


  // Constructor
  stream_info(packet_snd_data_stream* pack, int min, int norm, int max);
  // Destructor
  ~stream_info();

  // Get the next block for sending
  bool get_next_block(u16& BB, u16& DB, speed_level& speed);

  // Add a resend block to internal resend queue
  void add_resend_block(u16 BB, u16 DB);

  // Adjust the speed
  void adjust_speed(speed_level new_speed);

  // Return the client id associated with the stream info.x
  u64 get_client_id();

  // Set the stop point of the stream
  void set_stop_point(u16 BB, u16 DB);

  // Get the stop point of the stream
  void get_stop_point(u16& BB, u16& DB);

  // Set the given point to play  from
  void set_play_point(u16 BB);

  // Set the next sending time
  void set_send_time(system_time t);

  // get the next sinding time
  system_time get_send_time();
};

#endif
```

stream_info.cpp

```
#include "stdafx.h"
#include "../include/stream_info.h"


//*********************************************
// Constructor
//*********************************************
stream_info::stream_info(packet_snd_data_stream* pack, int min, int norm, int max)
{
```

```
  v_pack = pack;
  v_next_buffer_block = 0;
  v_next_data_block = 0;
  v_client_id = pack->get_client_id();

  v_low_speed_interval = min;
  v_normal_speed_interval = norm;
  v_high_speed_interval = max;

  system_time now;
  system_time last = {0,0};
  get_system_time(&now);
  v_next_send_time = now;
  v_last_send_time = last;

  v_stop_point_bb = 44;
  v_stop_point_db = 99;

  v_last_index = 0;

  v_speed = HIGH;
  started = false;
}

//*********************************************
// Constructor
//*********************************************
stream_info::~stream_info()
{
  delete v_pack;

  // We should loop through all elements of v_resend_queue and delete these
}

//*********************************************
// update_send_time
//*********************************************
system_time stream_info::update_send_time()
{
  int time_increase = 0;

  switch (v_speed) {
  case LOW:
    time_increase = v_low_speed_interval;
    break;
  case NORMAL:
    time_increase = v_normal_speed_interval;
    break;
  case HIGH:
    time_increase = v_high_speed_interval;
    break;
  default:
    break;
  }
  v_next_send_time = v_next_send_time + time_increase;
  return v_next_send_time;
}
```

```
//*******************************************
// get_next_block
//*******************************************
bool stream_info::get_next_block(u16& BB, u16& DB, speed_level& speed)
{
  u16 latest_BB = v_next_buffer_block;
  u16 latest_DB = v_next_data_block;

  // Start checking if any resend blocks are queued.
  if (!v_resend_queue.empty()) {
    resend_block block;
    block = v_resend_queue.front();
    v_resend_queue.pop();
    BB = block.bb;
    DB = block.db;
    this->adjust_speed(HIGH);
    speed = v_speed;
    update_send_time();
    return true;
  }
  // Check if we have reached stop point.
  if (latest_BB > v_stop_point_bb ) {
    // We have crossed the stop point. Done sending data.
    return false;
  }

  if (!started) {
    // We are at beginning of stream. Send start block
    BB = v_pack->get_start_bb();
    DB = v_pack->get_start_db();
    v_next_data_block = v_pack->get_start_bb();
    v_next_buffer_block = v_pack->get_start_bb();
    speed = v_speed;
    update_send_time();
    started = true;
    return true;
  }
  else {
    // Normal data dispatching
    u16 number_of_dbs = v_pack->get_number_of_db();

    if (v_last_index == number_of_dbs - 1) {
      v_last_index = 0;
      v_next_buffer_block = latest_BB + 1;
    }
    else {
      v_last_index++;
    }

    v_next_data_block = v_pack->get_db(v_last_index);

    BB = v_next_buffer_block;
    DB = v_next_data_block;
    speed = v_speed;
    update_send_time();
    return true;
```

```
  }
}

//***********************************************
// Add a block to the resend queue
//***********************************************
void stream_info::add_resend_block(u16 BB, u16 DB)
{
  resend_block block = {BB,DB};
  v_resend_queue.push(block);
}

//***********************************************
// Adjust the speed of the stream
//***********************************************
void stream_info::adjust_speed(speed_level new_speed)
{
  v_speed = new_speed;
}

//***********************************************
// Return client associated with the stream_info
//***********************************************
u64 stream_info::get_client_id()
{
  return v_client_id;
}

//***********************************************
// Return client associated with the stream_info
//***********************************************
void stream_info::set_stop_point(u16 BB, u16 DB)
{
  v_stop_point_bb = BB;
  v_stop_point_bb = DB;
}

//***********************************************
// Return client associated with the stream_info
//***********************************************
void stream_info::get_stop_point(u16& BB, u16& DB)
{
  BB = v_stop_point_bb;
  DB = v_stop_point_bb;
}

//***********************************************
// Return client associated with the stream_info
//***********************************************
void stream_info::set_send_time(system_time t)
{
  v_next_send_time = t;
}

//***********************************************
// Set the current play point
//***********************************************
```

```
void stream_info::set_play_point(u16 BB)
{
  v_next_buffer_block = BB;
  v_last_index = -1;
}

//**********************************************
// Return client associated with the stream_info
//**********************************************
system_time stream_info::get_send_time()
{
  return v_next_send_time;
}
```

stream_table.h

```
#ifndef STREAM_TABLE
#define STREAM_TABLE

#include <map>
#include "../include/global_functions.h"
#include "../include/enum.h"
#include "../include/types.h"
#include "../include/packet_snd_data_stream.h"
#include "../include/packet_stop_stream.h"
#include "../include/stream_info.h"
#include "../include/packet_conn_granted.h"

typedef std::map<struct system_time, stream_info*>::const_iterator CIT;

class stream_table {

  bool v_initialized;
  // Internal representation of the packet structure containing information
  // about the video
  packet_conn_granted* v_pack;

  // Internal map representing the streams dispatched
  std::map<u64, stream_info*> v_streams;

  std::map<struct system_time, stream_info*> v_wait_times;

 public:
  // Constructor
  stream_table();
  // Destructor
  ~stream_table();

  // Initializer
  void initialize(packet_conn_granted* pack);

  // Add a stream to the stream_table
  void add_stream(packet_snd_data_stream* pack, int min, int norm, int max);
```

```
  // Delete a given stream from stream table
  void stop_stream(packet_stop_stream* pack);

  // Get next receiver in line. This will return the next receiver
  // in terms of the minimum delay before the receiver should receive
  // the next data block. The delay is given in wait_time. Note that
  // this may be negative.
  bool get_next_receiver(u64& client_id,
                         int& wait_time,
 u16& next_bb,
 u16& next_db,
 speed_level& speed,
 u64& ip_addr,
 u16& port);

  // Adjust the speed of a given client.
  bool adjust_speed(u64 client_id, speed_level new_speed);

  // Resend a block to a client.
  bool resend_block(u64 client_id, u16 BB, u16 DB);

  // Remove stream
  void remove_stream(u64 client_id);

  // / Update contents of a stream
  bool update_stream(packet_snd_data_stream* pack);

  // Skip to a given point in a stream
  bool skip(u64 client_id, u16 BB);

};

#endif
```

stream_table.cpp

```
#include "stdafx.h"
#include "../include/stream_table.h"

using namespace std;

//*********************************************
// constructor
//*********************************************
stream_table::stream_table()
{
  v_initialized = false;
  v_pack = NULL;
}

//*********************************************
// Initializer
//*********************************************
void stream_table::initialize(packet_conn_granted* pack)
```

```
{
  if (! v_initialized) {
    v_pack = pack;
    v_initialized = true;
  }
  else {
    throw new stream_table_exception("Error: stream_table already initialized");
  }
}


//*********************************************
// destructor
//*********************************************
stream_table::~stream_table()
{

  delete v_pack;
  v_streams.clear();
  v_wait_times.clear();
  // We should clean up all references to stream_info classes in maps
}

//*********************************************
// Add a stream
//*********************************************
void stream_table::add_stream(packet_snd_data_stream* pack, int min, int norm, int max)
{
  if (v_initialized) {
    stream_info* stream = new stream_info(pack, min, norm, max);

    // First packet should be sent immidiately
    system_time now;
    get_system_time(&now);
    stream->set_send_time(now);

    // Insert stream into maps
    v_streams[pack->get_client_id()] = stream;
    v_wait_times[now] = stream;
  }
  else {
    throw new stream_table_exception("Error: stream_table not initialized!");
  }
}

//*********************************************
// stop stream
//*********************************************
void stream_table::stop_stream(packet_stop_stream* pack)
{
  if (v_initialized) {
    u64 client_id = pack->get_client_id();
    stream_info* si = v_streams[client_id];
    if (pack->get_stop_bb() == 0 && pack->get_stop_db() == 0) {
      // Delete entry from maps
      v_wait_times.erase(si->get_send_time());
      v_streams.erase(client_id);
```

```
      delete si;
    }
    else {
      si->set_stop_point(pack->get_stop_bb(), pack->get_stop_db());
    }
  }
  else {
    throw new stream_table_exception("Error: stream_table not initialized!");
  }
}

//*********************************************
// get_next_receiver
//*********************************************
bool stream_table::get_next_receiver(u64& client_id,
 int& wait_time,
 u16& next_bb,
 u16& next_db,
 speed_level& speed_level,
 u64& ip_addr,
 u16& port)
{
  if (v_initialized) {
    wait_time = 0;
    client_id = 0;
    next_bb = 0;
    next_db = 0;

    system_time now;
    get_system_time(&now);

    system_time t1 = {0,0};

    CIT i = v_wait_times.lower_bound(t1);
    stream_info* info;

    if (i == v_wait_times.end()) {
      return false;
    }
    else {
      info = i->second;
      system_time next_send_time = info->get_send_time();
      if (!info->get_next_block(next_bb, next_db, speed_level)) {
      // There is no more data to send. Delete stream_info instance
      v_wait_times.erase(i->first);
      v_streams.erase(client_id);
      delete info;
      wait_time = 0;
        return true;
      }
      else {
       client_id = info->get_client_id();
      wait_time = to_msec(next_send_time - now);
        ip_addr = info->v_pack->get_client_ip4();
        port = info->v_pack->get_client_port();
      // Remove data from the wait times map
      v_wait_times.erase(i->first);
```

```
      // Insert data again with updated send time
      v_wait_times[info->get_send_time()] = info;
      return true;
      }
    }
  }
  else {
    throw new stream_table_exception("Error: stream_table not initialized!");
  }
}


//**************************************************
// Adjust the speed of a given client
//**************************************************
bool stream_table::adjust_speed(u64 client_id, speed_level new_speed)
{
  if (v_initialized) {
    stream_info* info =  v_streams[client_id];
    if (info == NULL) {
      return false;
    }
    else {
      info->adjust_speed( new_speed);
      return true;
    }
  }
  else {
    throw new stream_table_exception("Error: stream_table not initialized!");
  }
}

//**************************************************
// Resend a data block (enqueues) the data block
// into the resend queue of the client.
//**************************************************
bool stream_table::resend_block(u64 client_id, u16 BB, u16 DB)
{
  if (v_initialized) {
    stream_info* info =  v_streams[client_id];
    if (info == NULL) {
      return false;
    }
    else {
      info->add_resend_block( BB, DB);
      return true;
    }
  }
  else {
    throw new stream_table_exception("Error: stream_table not initialized!");
  }
}

//**************************************************
// Remove stream
//**************************************************
void stream_table::remove_stream(u64 client_id)
```

```
{
  stream_info* si = NULL;
  si = v_streams[client_id];
  if (si != NULL) {
  v_wait_times.erase(si->get_send_time());
  v_streams.erase(client_id);
  delete si;
  }
}

//**********************************************
// Update stream
//**********************************************
bool stream_table::update_stream(packet_snd_data_stream* pack)
{
  stream_info* si = v_streams[pack->client_id];
  if (si == NULL)
    return false;
  else
    return true;
}

//**********************************************
// Skip to a given point in the video
//**********************************************
bool stream_table::skip(u64 client_id, u16 BB)
{
  stream_info* si = v_streams[client_id];
if (si == NULL)
    return false;
  else
   si->set_play_point(BB);
}
```

thread_client.h

```
#ifndef THREAD_CLIENT
#define THREAD_CLIENT

#include "../include/transport_handler.h"
#include "../include/stream_engine.h"
#include "../include/data_bank_client.h"
#include "../include/logic_client.h"
#include "../include/packet_queue.h"

struct thread_params
{
  data_bank_client*  v_data_bank;
  stream_engine*     v_stream_engine;
  transport_handler* v_transport_handler;
  packet_queue*      v_packet_queue;
  logic_client*      v_logic_client;
};
```

```
void receive_thread(void* args);

void stream_thread(void* args);

void logic_thread(void* args);

#endif
```

thread_client.cpp

```
#include "stdafx.h"
#include "../include/thread_client.h"

void receive_thread(void* args)
{
  packet_queue* v_packet_queue = ((thread_params*)args)->v_packet_queue;
  transport_handler* v_transport_handler = ((thread_params*)args)->v_transport_handler;

  packet* pack = NULL;
struct client_address addr;

for(;;) {
v_transport_handler->receive(pack, addr);

u64 client_id = 1234;
if (pack == NULL) {
v_transport_handler->close_CCP_connection(client_id);
break;
}
  else {
v_packet_queue->insert_packet(pack);
}
}

return;
}


void stream_thread(void* args)
{
  data_bank_client* v_data_bank = ((thread_params*)args)->v_data_bank;
  stream_engine* v_stream_engine = ((thread_params*)args)->v_stream_engine;
  transport_handler* v_transport_handler = ((thread_params*)args)->v_transport_handler;

u64 recv_client_id;
int wait_time;
u16 next_bb;
u16 next_db;
speed_level speed;
u64 ip_addr;
u16 port;

for(;;) {
wait_time = 0;
```

```
if( v_data_bank->get_next_receiver(v_stream_engine->get_video_id(),
recv_client_id,
wait_time,
next_bb,
next_db,
speed,
ip_addr,
port) ) {

if( wait_time == 0 ) {
packet* pack = v_stream_engine->stream(recv_client_id,
next_bb,
next_db,
speed);

client_address v_client_address;
v_client_address.client_ip = ip_addr;
v_client_address.client_DCP_port = port;
v_transport_handler->send_DCP(pack, &v_client_address);
}
else {
System::Threading::Thread::Sleep(wait_time / 1000);
}
}
else
break;
}

return;
}


void logic_thread(void* args)
{
  packet_queue* v_packet_queue = ((thread_params*)args)->v_packet_queue;
  logic_client* v_logic_client = ((thread_params*)args)->v_logic_client;

for(;;) {
packet* pack = v_packet_queue->get_next_packet();
v_logic_client->packet_handler(pack);
}

  return;
}
```

transport_handler.h

```
#ifndef TRANSPORT_HANDLER
#define TRANSPORT_HANDLER

#include "../include/packet_data.h"

#include "../include/vod_exception.h"
```

```
#include "../include/socket_utils.h"
#include "../include/packet.h"
#include "../include/types.h"
//#include <arpa/inet.h>
#include <string.h>
//#include <netinet/in.h>
#include <map>
#include <errno.h>
//#include "winsock2.h"


enum NODE_TYPE{
  CLIENT = 0,
  SERVER = 1
};


struct client_address {
u64 client_ip;
  u16 client_DCP_port;
  u16 client_CCP_port;
  u64 client_id;
  SOCKET sd;
};

typedef std::map<int, client_address*>::const_iterator CI;

class transport_handler
{
  int foo;

  void fill_addr_client_values(packet*&, client_address* client_addr);

  // Variable indicating if the class is initialized
  bool v_initialized;

  // The type of the transport_handler instance
  NODE_TYPE v_type;

  // Socket descriptor for the CCP socket
  SOCKET v_CCP_sd;
  // SOcket descriptor for the DCP socket
  SOCKET v_DCP_sd;

  // value indicating the CCP port either opened
  // locally on the server or connected to remotely from the client
  u16 v_CCP_port;
  // Value indicating the DCP port opened by the transport_handler
  u16 v_DCP_port;

  // Hash containing socket descriptors and client id's
  std::map<int, client_address*> v_clients;

  // number indicating the highest socket number
  int v_high_sock;

  // variable indicating the highest fd which is ready for reading.
```

```
  int v_high_sock_ready;

  // fd set containing list of all socket descriptors used
  fd_set v_sock_list;

  // Struct containing information about the
  // local opened port used by DCP. Used on client and server
  struct sockaddr_in v_local_DCP_addr;

  // Struct containing information about the local opened port
  // used by CCP. Used by server to open a specific port locally,
  // and by client to bind a local port.
  struct sockaddr_in v_local_CCP_addr;

  // Struct containing the remote CCP server. Used only by client.
  struct sockaddr_in v_remote_CCP_addr;

  //initializer functions.
  void initialize_DCP();
  void initialize_CCP();

  // Function resetting all file descriptors in fd_set
  void reset_descriptors();

  // Generate new unique id from socket descriptor
  u64 generate_id(int);

  // Handle new connecting CCP client
  void handle_new_CCP_req();

  // Read a  incoming DCP packet
  void read_DCP_packet(packet*&, client_address&);

  // Read a  incoming DCP packet
  void read_CCP_packet(packet*&, SOCKET sd);

 public:
  // constructors
  transport_handler(u16, u16);
  transport_handler(u16, u16, u16, char*);
  // destructor
  ~transport_handler();

  // Initialize the class
  void initialize();
  // send data through the DCP channel. May throw a transport_exception*.
  bool send_DCP(packet*, client_address*);
  // send data through the CCP channel. May throw a transport_exception*.
  bool send_CCP(packet*, u64);
  // receive. May throw a transport_exception*.
  u64 receive(packet*&);
  // Close an open connection. May throw a transport_exception*.
  void close_CCP_connection(u64);

};

#endif
```

transport_handler.cpp

```
#include "stdafx.h"
#include "../include/transport_handler.h"

using namespace std;

//***************************************************
//  Constructor.
//  Initializes the transport_handler as server
//***************************************************
transport_handler::transport_handler(u16 local_DCP_port, u16 local_CCP_port)
{
  foo = 0;

  v_initialized = false;
  v_DCP_port = local_DCP_port;
  v_CCP_port = local_CCP_port;
  v_type = SERVER;

  v_high_sock = 0;
  v_high_sock_ready = 0;

  FD_ZERO(&v_sock_list);

  // initialize variables to zero
  ZeroMemory(&v_local_DCP_addr, sizeof(v_local_DCP_addr));
  ZeroMemory(&v_local_CCP_addr, sizeof(v_local_CCP_addr));
  ZeroMemory(&v_remote_CCP_addr, sizeof(v_remote_CCP_addr));

  v_local_CCP_addr.sin_family = AF_INET;
  v_local_CCP_addr.sin_addr.s_addr = htonl(INADDR_ANY);
  v_local_CCP_addr.sin_port = htons( local_CCP_port);

  v_local_DCP_addr.sin_family = AF_INET;
  v_local_DCP_addr.sin_addr.s_addr = htonl( INADDR_ANY);
  v_local_DCP_addr.sin_port = htons( local_DCP_port);

}

//***************************************************
//  Constructor.
//  Initializes the transport_handler as client
//***************************************************
transport_handler::transport_handler(u16 local_DCP_port,
                                     u16 local_CCP_port,
                                     u16 remote_CCP_port,
                                     char* remote_CCP_ip)
{
  foo = 0;

  v_initialized = false;
  v_DCP_port = local_DCP_port;
  v_CCP_port = remote_CCP_port;
  v_type = CLIENT;
```

```
  v_high_sock = 0;
  v_high_sock_ready = 0;

  FD_ZERO(&v_sock_list);

  // initialize variables to zero
  ZeroMemory(&v_local_DCP_addr, sizeof(v_local_DCP_addr));
  ZeroMemory(&v_local_CCP_addr, sizeof(v_local_CCP_addr));
  ZeroMemory(&v_remote_CCP_addr, sizeof(v_remote_CCP_addr));

  v_remote_CCP_addr.sin_family = AF_INET;
  v_remote_CCP_addr.sin_port = htons(remote_CCP_port);
  v_remote_CCP_addr.sin_addr.S_un.S_addr = inet_addr(remote_CCP_ip);

  v_local_CCP_addr.sin_family = AF_INET;
  v_local_CCP_addr.sin_addr.s_addr = htonl(INADDR_ANY);
  v_local_CCP_addr.sin_port = htons(local_CCP_port);

  v_local_DCP_addr.sin_family = AF_INET;
  v_local_DCP_addr.sin_addr.s_addr = htonl(INADDR_ANY);
  v_local_DCP_addr.sin_port = htons(local_DCP_port);
}




//*****************************************************
//  Destructor
//*****************************************************
transport_handler::~transport_handler()
{
  try {
    Close(v_CCP_sd);
    Close(v_DCP_sd);
  WSACleanup();
    // Loop through all connected clients and close socket descriptor
    CI a = v_clients.begin();
    struct client_address* client;

    while (a != v_clients.end()) {
      client = a->second;
      a++;
      Close(client->sd);
    }
  }
  catch(transport_exception* ex) {
    throw ex;
  }
}

//*****************************************************
//  Initializer function
//*****************************************************
void transport_handler::initialize()
{
  if (!v_initialized) {
```

```
    WSADATA wsaData;

  int iResult = WSAStartup(MAKEWORD(2,2), &wsaData);

  if (iResult == NO_ERROR) {
  initialize_DCP();
  initialize_CCP();
  v_initialized = true;
  }
  else
  v_initialized = false;
  }
  else
    throw new transport_exception("Error: transport_handler was already initialized.");
}

//*****************************************************
//  Initialize the Data Communication Client
//*****************************************************
void transport_handler::initialize_DCP()
{
  try {
    // Open cdp socket
    v_DCP_sd = Socket(AF_INET, SOCK_DGRAM, 0);
    // set the socket nonblocking
    setnblck(v_DCP_sd);

    Bind(v_DCP_sd, (struct sockaddr*) &v_local_DCP_addr, sizeof(v_local_DCP_addr)) ;
  }
  catch (transport_exception* ex) {
    throw ex;
  }
}

//*****************************************************
//  Initialize the Control Communication server
//*****************************************************
void transport_handler::initialize_CCP()
{
  try {
    // Open socket
    v_CCP_sd = Socket(AF_INET, SOCK_STREAM,0);
    // bind socket to local port
    set_so_reuseable(v_CCP_sd);
    Bind(v_CCP_sd, (struct sockaddr*) &v_local_CCP_addr, sizeof(v_local_CCP_addr)) ;
    if (v_type == CLIENT) {
      // Connect to server
      Connect(v_CCP_sd, (const sockaddr*) &v_remote_CCP_addr, sizeof(v_remote_CCP_addr));
      setnblck(v_CCP_sd);
    }
    else {
      setnblck(v_CCP_sd);
      // Listen with a queue lenght of 5
      Listen(v_CCP_sd, 5);
    }
  }
  catch (transport_exception* ex) {
```

```
    closesocket(v_CCP_sd);
    throw ex;
    }
}


//**********************************************************
// Reset file descriptors in fd set
//**********************************************************
void transport_handler::reset_descriptors()
{
  FD_ZERO(&v_sock_list);
  FD_SET(v_CCP_sd, &v_sock_list);
  FD_SET(v_DCP_sd, &v_sock_list);

  v_high_sock_ready = 0;
  v_high_sock = 0;

  client_address* client;

  for (CI  i = v_clients.begin(); i != v_clients.end(); ++i){
    client = i->second ;
    FD_SET(client->sd, &v_sock_list) ;
  }
}


//**********************************************************
// Close a socket descriptor associated with a given client id
//**********************************************************
void transport_handler::close_CCP_connection(u64 client_id)
{
  int sd;
  client_address* client;

  try {
    // Retrieve socket handler from client id
    client = v_clients[client_id];
    v_clients.erase(client_id);
    FD_CLR(client->sd, &v_sock_list);
    Close(client->sd);
    delete client;
  }
  catch (transport_exception* ex) {
    cout << "Caught exception: " << ex->get_message() << endl;
  }
}

//**********************************************************
// Send data through CCP
//**********************************************************
bool transport_handler::send_CCP(packet* pack, u64 client_id)
{
  if (v_initialized) {
    size_t rv;
    SOCKET sd;
```

```
    client_address* client;
    void* header;
    // We malloc an iovec of size 3 (maximum size)
    iovec* msg_parts = (iovec*) malloc(sizeof(iovec) * 3);

    try {
      if (v_type == CLIENT)
      sd = v_CCP_sd;
      else {
      client = v_clients[client_id];
      sd = client->sd;
      }
      // Send data
      // Create msghdr struct
      msghdr msg;

      // Fill socket addr indicating receiver of packet
      msg.msg_name = NULL;
      msg.msg_namelen = 0;

      // Fill msg_iov field
      // Set iocvector in msghdr to msg_parts
      msg.msg_iov = msg_parts;
      msg.msg_iovlen = 0;

      // Fill vector
      msg_parts[0].iov_base = pack->get_header();
      msg_parts[0].iov_len = c_header_len;
      msg.msg_iovlen++;
      if (pack->get_payload_len() > 0) {
      msg_parts[1].iov_base = pack->get_payload();
       msg_parts[1].iov_len = pack->get_payload_len();
       msg.msg_iovlen++;
      }
      if (pack->get_payload_data_len() > 0) {
        msg_parts[2].iov_base = pack->get_payload_data();
      msg_parts[2].iov_len = pack->get_payload_data_len();
      msg.msg_iovlen++;
      }

      msg.msg_control = (void*) NULL;
      msg.msg_controllen = 0;
      msg.msg_flags = 0;

      // Send message
    Sendmsg(sd, &msg, 0);
      delete msg_parts;


    }
    catch (transport_exception* ex) {
      delete msg_parts;
      throw ex;
    }
  }
  else
    throw new transport_exception("Error: transport_handler send was
```

```
                                  called before the class was inititalized.");
}

//************************************************************
// Send data through DCP
//************************************************************
bool transport_handler::send_DCP(packet* pack, client_address* client)
{

  if (v_initialized) {

    // We malloc an iovec of size 3 (maximum size)
      iovec* msg_parts = (iovec*) malloc(sizeof(iovec) * 3);
      ZeroMemory( (void*) msg_parts, sizeof(iovec) * 3 );
    try {
      // Create msghdr struct
      msghdr msg;
      ZeroMemory( (void*) &msg, sizeof(msg) );

      // Fill socket addr indicating receiver of packet
      sockaddr_in receiver;
      receiver.sin_family = AF_INET;
      receiver.sin_port   = htons(client->client_DCP_port);
      receiver.sin_addr.S_un.S_addr = client->client_ip;
      msg.msg_name = (void*) &receiver;
      msg.msg_namelen = sizeof(receiver);

      // Fill msg_iov field
      // Set iocvector in msghdr to msg_parts
      msg.msg_iov = msg_parts;
      msg.msg_iovlen = 0;

      // Fill vector
      msg_parts[msg.msg_iovlen].iov_base = pack->get_header();
      msg_parts[msg.msg_iovlen].iov_len = c_header_len;
      msg.msg_iovlen++;
      if (pack->get_payload_len() > 0) {
      msg_parts[msg.msg_iovlen].iov_base = pack->get_payload();
      msg_parts[msg.msg_iovlen].iov_len = pack->get_payload_len();
      msg.msg_iovlen++;
      }
      if (pack->get_payload_data_len() > 0) {
      msg_parts[msg.msg_iovlen].iov_base = pack->get_payload_data();
      msg_parts[msg.msg_iovlen].iov_len = pack->get_payload_data_len();
      msg.msg_iovlen++;
      }

      msg.msg_control = (void*) NULL;
      msg.msg_controllen = 0;
      msg.msg_flags = 0;

      // Send message
      int res = Sendmsg(v_DCP_sd, &msg, 0);
      delete msg_parts;
    }
    catch (transport_exception* ex) {
      delete msg_parts;
```

```
      throw ex;
    }
  }
  else
    throw new transport_exception("Error: transport_handler.send was called before
                                   the class was inititalized.");
}


//***********************************************************
// Receive data from socket and return data wrapped in a packet
//***********************************************************
u64 transport_handler::receive(packet* &pack)
{
  SOCKET sd;
  pack = NULL;

  client_address client_addr;

  if (v_initialized) {
    try {
      for (;;){ // Loop until data has arrived
      reset_descriptors();
      // Perform select call on all socket descriptors with no time-out
      v_high_sock_ready = Select(v_high_sock + 1, &v_sock_list, NULL, NULL, NULL);
      if (FD_ISSET(v_CCP_sd, &v_sock_list)) {
        if (v_type == SERVER) {
       // Handle new connecting CCP client. This can only happen as server
       handle_new_CCP_req();
      }
      else {
        // client got control message from server
        read_CCP_packet(pack, v_CCP_sd);
          if( pack != NULL )
            fill_addr_client_values(pack, &client_addr);
        return 0;
      }
    }

    if (FD_ISSET(v_DCP_sd, &v_sock_list)) {
      // Read incoming DCP packet
      read_DCP_packet(pack, client_addr );
      fill_addr_client_values(pack, &client_addr);
      return 0;
    }

    // Loop through all connected clients to see if socket descriptor is set.
     CI a = v_clients.begin();
    struct client_address* client;

    while (a != v_clients.end()) {
      //we extract the file descriptor
      client = a->second ;
      // we increment the iterator as it might be deleted
      if (FD_ISSET(client->sd, &v_sock_list)){
        client_addr.client_id = client->client_id;
        client_addr.client_ip = client->client_ip;
```

```
        read_CCP_packet(pack, client->sd);
if(pack != NULL)
  fill_addr_client_values(pack, &client_addr);

return client->client_id;
      }
      a++ ;
    }

    }
  }
    catch (transport_exception* ex) {
      throw ex;
    }
  }

  else
    throw new transport_exception("Error: transport_handler.receive called
                                   before the class was inititalized.");
}


//*************************************************************
// Read new packet from DCP port
//*************************************************************
void transport_handler::read_DCP_packet(packet* &pack, struct client_address &client)
{
  size_t n_bytes;
  void* header = (void*) malloc(c_header_len);
  struct sockaddr_in sender;
  int len = sizeof(sockaddr_in);
  void* payload_data = NULL;
  void* payload = NULL;
  try {
    // First we must peek to see how much data awaits on socket
    // This is done, as calling receive on an UDP socket WILL
    // discard all data belonging to the datagram awaiting in
    // socket no matter how much of the data is read. Thus calling
    // receive MUST be done once on all data awaiting to be read.

    n_bytes = Recvfrom(v_DCP_sd,
                       header,
                       c_header_len,
                       MSG_PEEK,
                       (struct sockaddr*) &sender,
                       &len);

    pack = new packet(header);

    int bytes_to_read = c_header_len +
                        pack->get_payload_len() +
                        pack->get_payload_data_len();

    // Now we construct msg header for the recvmsg call
    msghdr msg;
    // We malloc an iovec of size 3 (maximum size)
    iovec* msg_parts = (iovec*) malloc(sizeof(iovec) * 3);
```

```
    // Fill socket addr to indicate sender of packet
    sockaddr_in sender;
    ZeroMemory( (void*) &sender, sizeof(sender) );
    msg.msg_name = (void*) &sender;
    msg.msg_namelen = sizeof(sender);

    // Fill msg_iov field
    // Set iocvector in msghdr to msg_parts
    msg.msg_iov = msg_parts;
    msg.msg_iovlen = 0;

    // Fill vector

    // Set packet header
    msg_parts[msg.msg_iovlen].iov_base = header;
    msg_parts[msg.msg_iovlen].iov_len = c_header_len;
    msg.msg_iovlen++;

    // Set payload header
    if (pack->get_payload_len() > 0) {
      msg_parts[msg.msg_iovlen].iov_base = malloc(pack->get_payload_len());
      msg_parts[msg.msg_iovlen].iov_len = pack->get_payload_len();
      payload = msg_parts[msg.msg_iovlen].iov_base;
      msg.msg_iovlen++;
    }

    if (pack->get_payload_data_len() > 0) {
      msg_parts[msg.msg_iovlen].iov_base = malloc(pack->get_payload_data_len());
      msg_parts[msg.msg_iovlen].iov_len = pack->get_payload_data_len();
      payload_data = msg_parts[msg.msg_iovlen].iov_base;
      msg.msg_iovlen++;
    }

    msg.msg_control = (void*) NULL;
    msg.msg_controllen = 0;
    msg.msg_flags = 0;

    n_bytes = Recvmsg(v_DCP_sd, &msg, 0);
    pack->set_payload(payload, pack->get_payload_len());
    pack->set_payload_data(payload_data, pack->get_payload_data_len());
  }
  catch (transport_exception* ex) {
        throw ex;
  }
}

//************************************************************
// Read new packet from CCP port
//************************************************************
void transport_handler::read_CCP_packet(packet* &pack, SOCKET sd)
{

  size_t n_bytes;
  void* header = (void*) malloc(c_header_len);
  try {
    n_bytes = Read(sd, header, c_header_len);
```

```
    if (n_bytes == 0) // Got eof - channel should be closed
      pack = NULL;
    else {
      pack = new packet(header);

      // Read payload
      if (pack->get_payload_len() > 0) {
void* payload = (void*) malloc(pack->get_payload_len());
n_bytes = Read(sd, payload, pack->get_payload_len());
pack->set_payload(payload, n_bytes);
      }
      if (pack->get_payload_data_len() > 0) {
void* payload_data = (void*) malloc(pack->get_payload_data_len());
n_bytes = Read(sd, payload_data, pack->get_payload_data_len());
pack->set_payload_data(payload_data, n_bytes);
      }
    }
  }
  catch (transport_exception* ex) {
    throw ex;
  }
}

//***********************************************************
// Handle new connection on CCP port
//***********************************************************
void transport_handler::handle_new_CCP_req()
{
  SOCKET new_sd;
  struct sockaddr_in new_client;
  int addr_len;

  if (v_type == SERVER) {
    try {
      // Accept the new incoming connection and set socket non blocking
      addr_len = sizeof(new_client);
      new_sd = Accept(v_CCP_sd, (struct sockaddr *) &new_client, &addr_len);
      setnblck(new_sd);
      // Generate client id and store this in hash with new file descriptor
      u64 client_id = generate_id(new_sd);
      struct client_address* client = (client_address*) malloc(sizeof(client_address));
      client->client_ip = new_client.sin_addr.S_un.S_addr;
      client->client_CCP_port = new_client.sin_port;
      client->client_id = client_id;
      client->sd = new_sd;
      v_clients[client_id] = client;
      System::Diagnostics::Debug::WriteLine("Connected client" + client_id);
    }
    catch (transport_exception* ex) {
      throw ex;
    }
  }
  else
    throw new transport_exception("Error: got new connection on DCP channel as client.");
}
```

```
//***********************************************************
// Generate new client id from feile descriptor
//***********************************************************
u64 transport_handler::generate_id(int fd)
{
  return fd;
}


void transport_handler::fill_addr_client_values(packet* &pack,
                                                client_address* client_addr)
{
  pack->client_ip = client_addr->client_ip;
  pack->client_DCP_port = client_addr->client_DCP_port;
  pack->client_CCP_port = client_addr->client_CCP_port;
  pack->client_id = client_addr->client_id;
  pack->sd = client_addr->sd;
}
```

transport_handler_thread.h

```
#ifndef TRANSPORT_HANDLER_THREAD
#define TRANSPORT_HANDLER_THREAD

#include "../include/transport_handler.h"
#include "../include/packet_queue.h"

struct receive_params
{
  packet_queue*      v_packet_queue;
  transport_handler* v_transport_handler;
};

void receive_thread(void* args);

#endif
```

transport_handler_thread.cpp

```
#include "stdafx.h"
#include "../include/transport_handler_thread.h"


//*********************************************
// Main procedure
//*********************************************
void receive_thread(void* args)
{
  packet_queue* v_packet_queue = ((receive_params*)args)->v_packet_queue;
  transport_handler* v_transport_handler = ((receive_params*)args)->v_transport_handler;
```

```
  packet* pack = NULL;

for(;;) {
u64 client_id = v_transport_handler->receive(pack);

if (pack == NULL) {
v_transport_handler->close_CCP_connection(client_id);
break;
}
  else {
v_packet_queue->insert_packet(pack);
}
}

return;
}
```

vod_client.h

```
#ifndef VOD_CLIENT
#define VOD_CLIENT

#include "../include/define.h"
#include "../include/types.h"
#include "../include/logic_client.h"
#include "../include/logic_client_thread.h"
#include "../include/packet_queue.h"
#include "../include/transport_handler.h"
#include "../include/transport_handler_thread.h"
#include "../include/packet_conn_req.h"

// Struct used to clients ports
struct client_addr {
  int dcp_port;
  int ccp_port;
};

// Struct containing servers address
struct server_addr {
  char* ip_address;
  int port;
};

// Struct containing clients login-data
struct login_data {
  char user_name[64];
  char password[64];
  unsigned int video_size;
  unsigned int video_duration;
  unsigned int skip_distance;
};

// Struct containing info needed for the client
```

```
struct video_info {
  unsigned long skip_distance;
  unsigned long video_duration;
  unsigned long video_length;
};


class vod_client {
 private:
  size_t       v_buf_size;

  client_addr* v_client_addr;
  server_addr* v_server_address;

  logic_client*      v_logic_client;
  transport_handler* v_transport_handler;
  packet_queue*      v_packet_queue;

  DWORD  v_logic_thread_id;
HANDLE v_logic_thread;

  DWORD  v_receive_thread_id;
HANDLE v_receive_thread;

  bool connect_status();
  void clean_up();

  logic_params args1;
  receive_params args2;

 public:
  // Constructor
  vod_client(size_t buf_size, struct client_addr* client);

  // Connect to server
  int connect(unsigned long video_id, struct server_addr* address, struct login_data* login);

  // Receive data from client's buffer
  size_t recv_data(void* buf, size_t max_len, unsigned long& offset);

  // Disconnect from server
  int disconnect();

  // User interaction, puase
  int pause();

  // User interaction, resume
  int resume();

  // User interaction, skip
  int skip(unsigned long distance);

  // Returns movie info
  void video_info(struct video_info* info);
};

#endif
```

vod_client.cpp

```
#include "stdafx.h"
#include "../include/vod_client.h"


//**************************************************
//  Constructor.
//  Sets the buffer size and client address
//**************************************************
vod_client::vod_client(size_t buf_size, struct client_addr* address)
{
  v_buf_size = buf_size;
  v_client_addr = new client_addr;
  memcpy(v_client_addr, address, sizeof(client_addr));
}

//**************************************************
//  Connects the client to the server.
//**************************************************
int vod_client::connect(unsigned long movie_id,
                        struct server_addr* address,
                        struct login_data* login)
{
  v_server_address = new server_addr;
  memcpy(v_server_address, address, sizeof(server_addr));

  // Create packet-queue
  v_packet_queue = new packet_queue(c_packet_queue_size_client);

  // Create transport handler
  v_transport_handler = new transport_handler(v_client_addr->dcp_port,
      v_client_addr->ccp_port,
      address->port,
      address->ip_address);
  v_transport_handler->initialize();

  // Create logic
  v_logic_client = new logic_client((u64)movie_id,
                                    v_transport_handler,
                                    v_packet_queue);

  // Create transport-handler thread
  args2.v_packet_queue = v_packet_queue;
  args2.v_transport_handler = v_transport_handler;
  v_receive_thread = CreateThread(NULL,
  0,
  (LPTHREAD_START_ROUTINE) receive_thread,
  (void*) &args2,
  0,
  &v_receive_thread_id);

  // Tries to login
  v_logic_client->connect(v_buf_size, login,
  v_client_addr->ccp_port,
  v_client_addr->dcp_port);
```

```
  // Create client-logic thread
  args1.v_packet_queue = v_packet_queue;
  args1.v_logic_client = v_logic_client;
  v_logic_thread = CreateThread(NULL,
0,
(LPTHREAD_START_ROUTINE) logic_thread,
(void*) &args1,
0,
&v_logic_thread_id);

  return 0;
}

//****************************************************
//  Receive data from protocol
//****************************************************
size_t vod_client::recv_data(void* buf, size_t max_len, unsigned long& offset)
{
  size_t rv = 0;

  if(v_logic_client != NULL) {
    rv = v_logic_client->recv_data(buf, max_len, (unsigned int&)offset);

    if( !this->connect_status() )
      clean_up();
  }

  return rv;
}

//****************************************************
//  Disconnect client
//****************************************************
int vod_client::disconnect()
{
  int rv = v_logic_client->disconnect();
  clean_up();

  return rv;
}

//****************************************************
//  Pause client
//****************************************************
int vod_client::pause()
{
  return 0;
}

//****************************************************
//  Resume client
//****************************************************
int vod_client::resume()
{
  return 0;
}
```

```
//**************************************************
//  Skip
//**************************************************
int vod_client::skip(unsigned long distance)
{
  return v_logic_client->skip(distance);
}

//**************************************************
//  Retrieve video info
//**************************************************
void vod_client::video_info(struct video_info* info)
{
  info->skip_distance = v_logic_client->get_skip_distance();
  info->video_duration = v_logic_client->get_video_duration();
  info->video_length = v_logic_client->get_video_length();
}

//**************************************************
//  Get the connectec status
//**************************************************
bool vod_client::connect_status()
{
  return true;
}

//**************************************************
//  Clean up
//**************************************************
void vod_client::clean_up()
{
  // Send cancel to threads
  TerminateThread(v_logic_thread, 0);
  TerminateThread(v_receive_thread, 0);

  // Delete objects
  delete(v_logic_client);
  delete(v_packet_queue);
  //delete(v_transport_handler);
}
```

vod_exception.h

```
#ifndef VOD_EXCEPTION
#define VOD_EXCEPTION

#include <string.h>
#include <stdlib.h>

class vod_exception
{
 private:
  char* v_err_msg;
```

```
 public:
  vod_exception(const char*);
  const char* get_message();
};


class transport_exception : public vod_exception
{
 public:
  transport_exception(const char*);
};


class packet_exception : public vod_exception
{
 public:
  packet_exception(const char*);
};


class packet_queue_exception : public vod_exception
{
 public:
  packet_queue_exception(const char*);
};

class data_bank_exception : public vod_exception
{
 public:
  data_bank_exception(const char*);
};

class stream_table_exception : public vod_exception
{
 public:
  stream_table_exception(const char*);
};

#endif
```

vod_exception.cpp

```
#include "stdafx.h"
#include "../include/vod_exception.h"

vod_exception::vod_exception(const char* msg)
{
  v_err_msg = (char*) malloc(strlen(msg) + 1);
  strcpy(v_err_msg, msg);
}

const char* vod_exception::get_message()
{
  return (const char*) v_err_msg;
```

```
}


// Transport exception
transport_exception::transport_exception(const char* msg): vod_exception(msg)
{

}


// Packet exception
packet_exception::packet_exception(const char* msg): vod_exception(msg)
{

}


// Packet_queue exception
packet_queue_exception::packet_queue_exception(const char* msg): vod_exception(msg)
{

}

// Packet_queue exception
stream_table_exception::stream_table_exception(const char* msg): vod_exception(msg)
{

}

// data bank exception
data_bank_exception::data_bank_exception(const char* msg): vod_exception(msg)
{

}
```

vod_server.h

```
#ifndef VOD_SERVER
#define VOD_SERVER

#include "../include/define.h"
#include "../include/types.h"
#include "../include/enum.h"
#include "../include/logic_server.h"
#include "../include/logic_server_thread.h"
#include "../include/packet_queue.h"
#include "../include/transport_handler.h"
#include "../include/transport_handler_thread.h"
#include "../include/data_container.h"
#include "../include/request_queue.h"

struct login_struct_req {
  long video_id;
  char user_name[32];
```

```
  char password[32];
  long transaction_id;
};

enum event_type {
  x,
  y
};

struct event_struct {
  event_type event;
};

class vod_server {
 private:
  logic_params args1;
  receive_params args2;

  logic_server*      v_logic_server;
  transport_handler* v_transport_handler;
  packet_queue*      v_packet_queue;

  DWORD  v_logic_thread_id;
HANDLE v_logic_thread;

  DWORD  v_receive_thread_id;
HANDLE v_receive_thread;

  int v_data_port;
  int v_control_port;

  request_queue* v_request_queue;

public:
  vod_server(int data_port, int control_port);

  int open();

  int close();

  void poll(int& video_data,
            int& video_data_hp,
            int& sec_data,
            int& sec_data_hp,
            int& login_req,
            int& events);

  int get_data_req(struct data_struct_req& data,
                   data_type type);

  int get_login_req(struct login_struct_req& user);

  int get_event(struct event_struct& event);

  int get_video_info_req(unsigned long& video_id);

  int deliver_data(data_class* data, data_type type);
```

```
};

#endif
```

vod_server.cpp

```
#include "stdafx.h"

//*******************************************************
// Constructor
//*******************************************************
vod_server::vod_server(int data_port, int control_port)
{
  v_data_port    = data_port;
  v_control_port = control_port;
}

//*******************************************************
// Open connection and start listening for incoming clients
//*******************************************************
int vod_server::open()
{
  // Create objects
  v_packet_queue = new packet_queue(c_packet_queue_size_server);
  v_transport_handler = new transport_handler(v_data_port, v_control_port);
  v_transport_handler->initialize();
  v_logic_server = new logic_server(v_transport_handler);

  // Create thread logic
  args1.v_packet_queue = v_packet_queue;
  args1.v_logic_server = v_logic_server;
  v_logic_thread = CreateThread(NULL,
0,
(LPTHREAD_START_ROUTINE) logic_thread,
(void*) &args1,
0,
&v_logic_thread_id);

  // Create thread transport_handler
  args2.v_packet_queue = v_packet_queue;
  args2.v_transport_handler = v_transport_handler;
  v_receive_thread = CreateThread(NULL,
  0,
  (LPTHREAD_START_ROUTINE) receive_thread,
  (void*) &args2,
  0,
  &v_receive_thread_id);

  return 0;
}

//*******************************************************
// Close server. Terminate all threads and close all open
// socket despcriptors.
```

```
//*******************************************************
int vod_server::close()
{
  // Logic - including stream engine
  TerminateThread(v_logic_thread, 0);
  delete v_receive_thread;

  // Transport handler - including socket
  TerminateThread(v_receive_thread, 0);
  delete v_transport_handler;

  // Additional objects
  delete v_packet_queue;
  delete v_request_queue;

  return 0;
}

//*******************************************************
// Poll call
//*******************************************************
void vod_server::poll(int& movie_data,
                      int& movie_data_hp,
                      int& sec_data,
                      int& sec_data_hp,
                      int& login_req,
                      int& events)
{
  movie_data = v_request_queue->count_data_request();
  movie_data_hp = 0;
  sec_data = 0;
  sec_data_hp = 0;
  login_req = 0;
  events = 0;
}

//*******************************************************
// Get next data request
//*******************************************************
int vod_server::get_data_req(struct data_struct_req& data,
                             data_type type)
{
  return v_logic_server->get_data_req(data, type);
}

//*******************************************************
// Get next login request
// Not implemented in this version.
//*******************************************************
int vod_server::get_login_req(struct login_struct_req& user)
{
  return 0;
}

//*******************************************************
// Get next event
// Not implemented in this version.
```

```
//**********************************************************
int vod_server::get_event(struct event_struct& event)
{
  return 0;
}

//**********************************************************
// Get next movie info request
// Not implemented in this version.
//**********************************************************
int vod_server::get_video_info_req(unsigned long& movie_id)
{
  return 0;
}

//**********************************************************
// Deliver data
//**********************************************************
int vod_server::deliver_data(data_class* data, data_type type)
{
  return v_logic_server->deliver_data(data, type);
}
```

Appendix C

Client application source files

VOD_Client.cpp

```
// VOD_Client.cpp : main project file.

#include "stdafx.h"
#include "Form1.h"

using namespace VOD_Client;

[STAThreadAttribute]
int main(array<System::String ^> ^args)
{
// Enabling Windows XP visual effects before any controls are created
Application::EnableVisualStyles();
Application::SetCompatibleTextRenderingDefault(false);

// Create the main window and run it
Application::Run(gcnew Form1());
return 0;
}
```

Form1.h

```
#pragma once


namespace VOD_Client {

using namespace System;
using namespace System::ComponentModel;
using namespace System::Collections;
using namespace System::Windows::Forms;
using namespace System::Data;
using namespace System::Drawing;
  using namespace System::IO;
  using namespace System::Text;

/// <summary>
/// Summary for Form1
///
```

```
/// WARNING: If you change the name of this class, you will need to change the
///          'Resource File Name' property for the managed resource compiler tool
///          associated with all .resx files this class depends on.  Otherwise,
///          the designers will not be able to interact properly with localized
///          resources associated with this form.
/// </summary>
public ref class Form1 : public System::Windows::Forms::Form
{
public:


    Form1(void)
{
InitializeComponent();
    }



protected:
/// <summary>
/// Clean up any resources being used.
/// </summary>
~Form1()
{
if (components)
{
delete components;
}
}

private: System::Windows::Forms::Button^  button1;
private: System::Windows::Forms::Button^  button2;
  private: System::Windows::Forms::Button^  button3;
  private: System::Windows::Forms::GroupBox^  groupBox1;
  private: System::Windows::Forms::GroupBox^  groupBox2;
  private: System::Windows::Forms::Label^  label1;
  private: System::Windows::Forms::TextBox^  textBox2;
  private: System::Windows::Forms::Label^  label2;
  private: System::Windows::Forms::TextBox^  textBox1;
  private: System::Windows::Forms::TextBox^  textBox4;
  private: System::Windows::Forms::Label^  label4;
  private: System::Windows::Forms::TextBox^  textBox3;
  private: System::Windows::Forms::Label^  label3;

  private: System::Windows::Forms::Label^  label6;
  private: System::Windows::Forms::TextBox^  textBox5;

  private: System::Windows::Forms::TextBox^  textBox6;
  private: System::Windows::Forms::Button^  button6;
  private: System::Windows::Forms::Button^  button5;
  private: System::Windows::Forms::Button^  button4;
  private: System::Windows::Forms::Label^  label10;
  private: System::Windows::Forms::TextBox^  textBox9;
  private: System::Windows::Forms::TextBox^  textBox8;
  private: System::Windows::Forms::TextBox^  textBox7;
  private: System::Windows::Forms::Label^  label9;
  private: System::Windows::Forms::Label^  label8;
```

```
  private: System::Windows::Forms::Label^  label7;
  private: System::Windows::Forms::Label^  label5;
  private: System::Windows::Forms::RichTextBox^  richTextBox1;
  protected:

private:
vod_client* client_protocol;
    login_data* login;
    data_thread* dt;
    unsigned long skip_distance;

  private: AxQTOControlLib::AxQTControl^  axQTControl1;
  private: System::Windows::Forms::Timer^  timer1;
  private: System::ComponentModel::IContainer^  components;

    /// <summary>
/// Required designer variable.
/// </summary>


#pragma region Windows Form Designer generated code
/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
void InitializeComponent(void)
{
      this->components = (gcnew System::ComponentModel::Container());
      System::ComponentModel::ComponentResourceManager^  resources =
      (gcnew System::ComponentModel::ComponentResourceManager(Form1::typeid));
      this->button1 = (gcnew System::Windows::Forms::Button());
      this->button2 = (gcnew System::Windows::Forms::Button());
      this->button3 = (gcnew System::Windows::Forms::Button());
      this->groupBox1 = (gcnew System::Windows::Forms::GroupBox());
      this->label6 = (gcnew System::Windows::Forms::Label());
      this->textBox5 = (gcnew System::Windows::Forms::TextBox());
      this->textBox4 = (gcnew System::Windows::Forms::TextBox());
      this->label4 = (gcnew System::Windows::Forms::Label());
      this->textBox3 = (gcnew System::Windows::Forms::TextBox());
      this->label3 = (gcnew System::Windows::Forms::Label());
      this->textBox2 = (gcnew System::Windows::Forms::TextBox());
      this->label2 = (gcnew System::Windows::Forms::Label());
      this->textBox1 = (gcnew System::Windows::Forms::TextBox());
      this->label1 = (gcnew System::Windows::Forms::Label());
      this->groupBox2 = (gcnew System::Windows::Forms::GroupBox());
      this->label10 = (gcnew System::Windows::Forms::Label());
      this->textBox9 = (gcnew System::Windows::Forms::TextBox());
      this->textBox8 = (gcnew System::Windows::Forms::TextBox());
      this->textBox7 = (gcnew System::Windows::Forms::TextBox());
      this->label9 = (gcnew System::Windows::Forms::Label());
      this->label8 = (gcnew System::Windows::Forms::Label());
      this->label7 = (gcnew System::Windows::Forms::Label());
      this->textBox6 = (gcnew System::Windows::Forms::TextBox());
      this->button6 = (gcnew System::Windows::Forms::Button());
      this->button5 = (gcnew System::Windows::Forms::Button());
      this->button4 = (gcnew System::Windows::Forms::Button());
      this->label5 = (gcnew System::Windows::Forms::Label());
```

```
this->richTextBox1 = (gcnew System::Windows::Forms::RichTextBox());
this->axQTControl1 = (gcnew AxQTOControlLib::AxQTControl());
this->timer1 = (gcnew System::Windows::Forms::Timer(this->components));
this->groupBox1->SuspendLayout();
this->groupBox2->SuspendLayout();
(cli::safe_cast<System::ComponentModel::ISupportInitialize^  >
(this->axQTControl1))->BeginInit();
this->SuspendLayout();
//
// button1
//
this->button1->Location = System::Drawing::Point(6, 20);
this->button1->Name = L"button1";
this->button1->Size = System::Drawing::Size(75, 23);
this->button1->TabIndex = 1;
this->button1->Text = L"Play";
this->button1->UseVisualStyleBackColor = true;
this->button1->Click += gcnew System::EventHandler(this, &Form1::button1_Click);
//
// button2
//
this->button2->Enabled = false;
this->button2->Location = System::Drawing::Point(6, 49);
this->button2->Name = L"button2";
this->button2->Size = System::Drawing::Size(75, 23);
this->button2->TabIndex = 2;
this->button2->Text = L"Pause";
this->button2->UseVisualStyleBackColor = true;
this->button2->Click += gcnew System::EventHandler(this, &Form1::button2_Click);
//
// button3
//
this->button3->Location = System::Drawing::Point(96, 149);
this->button3->Name = L"button3";
this->button3->Size = System::Drawing::Size(75, 23);
this->button3->TabIndex = 3;
this->button3->Text = L"Connect";
this->button3->UseVisualStyleBackColor = true;
this->button3->Click += gcnew System::EventHandler(this, &Form1::button3_Click);
//
// groupBox1
//
this->groupBox1->Controls->Add(this->label6);
this->groupBox1->Controls->Add(this->textBox5);
this->groupBox1->Controls->Add(this->button3);
this->groupBox1->Controls->Add(this->textBox4);
this->groupBox1->Controls->Add(this->label4);
this->groupBox1->Controls->Add(this->textBox3);
this->groupBox1->Controls->Add(this->label3);
this->groupBox1->Controls->Add(this->textBox2);
this->groupBox1->Controls->Add(this->label2);
this->groupBox1->Controls->Add(this->textBox1);
this->groupBox1->Controls->Add(this->label1);
this->groupBox1->Location = System::Drawing::Point(509, 12);
this->groupBox1->Name = L"groupBox1";
this->groupBox1->Size = System::Drawing::Size(182, 190);
this->groupBox1->TabIndex = 4;
```

```
this->groupBox1->TabStop = false;
this->groupBox1->Text = L"Settings";
//
// label6
//
this->label6->AutoSize = true;
this->label6->Location = System::Drawing::Point(6, 154);
this->label6->Name = L"label6";
this->label6->Size = System::Drawing::Size(53, 13);
this->label6->TabIndex = 11;
this->label6->Text = L"Movie ID:";
//
// textBox5
//
this->textBox5->Location = System::Drawing::Point(65, 151);
this->textBox5->Name = L"textBox5";
this->textBox5->Size = System::Drawing::Size(25, 20);
this->textBox5->TabIndex = 10;
this->textBox5->Text = L"1";
this->textBox5->TextAlign = System::Windows::Forms::HorizontalAlignment::Right;
//
// textBox4
//
this->textBox4->Location = System::Drawing::Point(96, 72);
this->textBox4->Name = L"textBox4";
this->textBox4->Size = System::Drawing::Size(80, 20);
this->textBox4->TabIndex = 7;
this->textBox4->Text = L"10000";
this->textBox4->TextAlign = System::Windows::Forms::HorizontalAlignment::Right;
//
// label4
//
this->label4->AutoSize = true;
this->label4->Location = System::Drawing::Point(6, 75);
this->label4->Name = L"label4";
this->label4->Size = System::Drawing::Size(82, 13);
this->label4->TabIndex = 6;
this->label4->Text = L"Client TCP Port:";
//
// textBox3
//
this->textBox3->Location = System::Drawing::Point(96, 97);
this->textBox3->Name = L"textBox3";
this->textBox3->Size = System::Drawing::Size(80, 20);
this->textBox3->TabIndex = 5;
this->textBox3->Text = L"11000";
this->textBox3->TextAlign = System::Windows::Forms::HorizontalAlignment::Right;
//
// label3
//
this->label3->AutoSize = true;
this->label3->Location = System::Drawing::Point(6, 100);
this->label3->Name = L"label3";
this->label3->Size = System::Drawing::Size(84, 13);
this->label3->TabIndex = 4;
this->label3->Text = L"Client UDP Port:";
//
```

```
// textBox2
//
this->textBox2->Location = System::Drawing::Point(96, 46);
this->textBox2->Name = L"textBox2";
this->textBox2->Size = System::Drawing::Size(80, 20);
this->textBox2->TabIndex = 3;
this->textBox2->Text = L"20000";
this->textBox2->TextAlign = System::Windows::Forms::HorizontalAlignment::Right;
//
// label2
//
this->label2->AutoSize = true;
this->label2->Location = System::Drawing::Point(6, 49);
this->label2->Name = L"label2";
this->label2->Size = System::Drawing::Size(63, 13);
this->label2->TabIndex = 2;
this->label2->Text = L"Server Port:";
//
// textBox1
//
this->textBox1->Location = System::Drawing::Point(76, 20);
this->textBox1->Name = L"textBox1";
this->textBox1->Size = System::Drawing::Size(100, 20);
this->textBox1->TabIndex = 1;
this->textBox1->Text = L"192.168.15.145";
this->textBox1->TextChanged += gcnew System::
EventHandler(this, &Form1::textBox1_TextChanged);
//
// label1
//
this->label1->AutoSize = true;
this->label1->Location = System::Drawing::Point(6, 23);
this->label1->Name = L"label1";
this->label1->Size = System::Drawing::Size(54, 13);
this->label1->TabIndex = 0;
this->label1->Text = L"Server IP:";
//
// groupBox2
//
this->groupBox2->Controls->Add(this->label10);
this->groupBox2->Controls->Add(this->textBox9);
this->groupBox2->Controls->Add(this->textBox8);
this->groupBox2->Controls->Add(this->textBox7);
this->groupBox2->Controls->Add(this->label9);
this->groupBox2->Controls->Add(this->label8);
this->groupBox2->Controls->Add(this->label7);
this->groupBox2->Controls->Add(this->textBox6);
this->groupBox2->Controls->Add(this->button6);
this->groupBox2->Controls->Add(this->button5);
this->groupBox2->Controls->Add(this->button4);
this->groupBox2->Controls->Add(this->button2);
this->groupBox2->Controls->Add(this->button1);
this->groupBox2->Enabled = false;
this->groupBox2->Location = System::Drawing::Point(12, 351);
this->groupBox2->Name = L"groupBox2";
this->groupBox2->Size = System::Drawing::Size(491, 80);
this->groupBox2->TabIndex = 5;
```

```
this->groupBox2->TabStop = false;
this->groupBox2->Text = L"Control";
//
// label10
//
this->label10->AutoSize = true;
this->label10->Location = System::Drawing::Point(351, 25);
this->label10->Name = L"label10";
this->label10->Size = System::Drawing::Size(71, 13);
this->label10->TabIndex = 13;
this->label10->Text = L"Remain. time:";
//
// textBox9
//
this->textBox9->Location = System::Drawing::Point(426, 22);
this->textBox9->Name = L"textBox9";
this->textBox9->ReadOnly = true;
this->textBox9->Size = System::Drawing::Size(59, 20);
this->textBox9->TabIndex = 12;
this->textBox9->TextAlign = System::Windows::Forms::HorizontalAlignment::Right;
//
// textBox8
//
this->textBox8->Location = System::Drawing::Point(149, 22);
this->textBox8->Name = L"textBox8";
this->textBox8->ReadOnly = true;
this->textBox8->Size = System::Drawing::Size(59, 20);
this->textBox8->TabIndex = 11;
this->textBox8->TextAlign = System::Windows::Forms::HorizontalAlignment::Right;
//
// textBox7
//
this->textBox7->Location = System::Drawing::Point(286, 22);
this->textBox7->Name = L"textBox7";
this->textBox7->ReadOnly = true;
this->textBox7->Size = System::Drawing::Size(59, 20);
this->textBox7->TabIndex = 10;
this->textBox7->TextAlign = System::Windows::Forms::HorizontalAlignment::Right;
//
// label9
//
this->label9->AutoSize = true;
this->label9->Location = System::Drawing::Point(214, 25);
this->label9->Name = L"label9";
this->label9->Size = System::Drawing::Size(66, 13);
this->label9->TabIndex = 9;
this->label9->Text = L"Current time:";
//
// label8
//
this->label8->AutoSize = true;
this->label8->Location = System::Drawing::Point(354, 25);
this->label8->Name = L"label8";
this->label8->Size = System::Drawing::Size(0, 13);
this->label8->TabIndex = 8;
//
// label7
```

```
//
this->label7->AutoSize = true;
this->label7->Location = System::Drawing::Point(87, 25);
this->label7->Name = L"label7";
this->label7->Size = System::Drawing::Size(56, 13);
this->label7->TabIndex = 7;
this->label7->Text = L"Total time:";
//
// textBox6
//
this->textBox6->Location = System::Drawing::Point(243, 52);
this->textBox6->Name = L"textBox6";
this->textBox6->Size = System::Drawing::Size(24, 20);
this->textBox6->TabIndex = 6;
this->textBox6->Text = L"1";
this->textBox6->TextAlign = System::Windows::Forms::HorizontalAlignment::Center;
//
// button6
//
this->button6->Location = System::Drawing::Point(410, 49);
this->button6->Name = L"button6";
this->button6->Size = System::Drawing::Size(75, 23);
this->button6->TabIndex = 5;
this->button6->Text = L"Stop";
this->button6->UseVisualStyleBackColor = true;
this->button6->Click += gcnew System::EventHandler(this, &Form1::button6_Click);
//
// button5
//
this->button5->Location = System::Drawing::Point(273, 49);
this->button5->Name = L"button5";
this->button5->Size = System::Drawing::Size(75, 23);
this->button5->TabIndex = 4;
this->button5->Text = L"Skip >>";
this->button5->UseVisualStyleBackColor = true;
this->button5->Click += gcnew System::EventHandler(this, &Form1::button5_Click);
//
// button4
//
this->button4->Enabled = false;
this->button4->Location = System::Drawing::Point(162, 49);
this->button4->Name = L"button4";
this->button4->Size = System::Drawing::Size(75, 23);
this->button4->TabIndex = 3;
this->button4->Text = L"<< Skip";
this->button4->UseVisualStyleBackColor = true;
this->button4->Click += gcnew System::EventHandler(this, &Form1::button4_Click);
//
// label5
//
this->label5->AutoSize = true;
this->label5->Location = System::Drawing::Point(516, 212);
this->label5->Name = L"label5";
this->label5->Size = System::Drawing::Size(62, 13);
this->label5->TabIndex = 11;
this->label5->Text = L"Information:";
//
```

```
    // richTextBox1
    //
    this->richTextBox1->Location = System::Drawing::Point(517, 230);
    this->richTextBox1->Name = L"richTextBox1";
    this->richTextBox1->Size = System::Drawing::Size(163, 193);
    this->richTextBox1->TabIndex = 10;
    this->richTextBox1->Text = L"";
    //
    // axQTControl1
    //
    this->axQTControl1->Enabled = true;
    this->axQTControl1->Location = System::Drawing::Point(11, 13);
    this->axQTControl1->Name = L"axQTControl1";
    this->axQTControl1->OcxState = (cli::safe_cast<System::Windows::Forms::AxHost::State^>
    (resources->GetObject(L"axQTControl1.OcxState")));
    this->axQTControl1->Size = System::Drawing::Size(491, 338);
    this->axQTControl1->TabIndex = 12;
    //
    // timer1
    //
    this->timer1->Tick += gcnew System::EventHandler(this, &Form1::timer1_Tick);
    //
    // Form1
    //
    this->AutoScaleDimensions = System::Drawing::SizeF(6, 13);
    this->AutoScaleMode = System::Windows::Forms::AutoScaleMode::Font;
    this->ClientSize = System::Drawing::Size(701, 443);
    this->Controls->Add(this->axQTControl1);
    this->Controls->Add(this->label5);
    this->Controls->Add(this->richTextBox1);
    this->Controls->Add(this->groupBox2);
    this->Controls->Add(this->groupBox1);
    this->Name = L"Form1";
    this->Text = L"Client";
    this->groupBox1->ResumeLayout(false);
    this->groupBox1->PerformLayout();
    this->groupBox2->ResumeLayout(false);
    this->groupBox2->PerformLayout();
    (cli::safe_cast<System::ComponentModel::ISupportInitialize^  >
    (this->axQTControl1))->EndInit();
    this->ResumeLayout(false);
    this->PerformLayout();

  }


#pragma endregion

  private: System::Void button3_Click(System::Object^  sender, System::EventArgs^  e)
  {
    client_addr c;
    c.dcp_port = Convert::ToInt32(textBox3->Text);
    c.ccp_port = Convert::ToInt32(textBox4->Text);
    client_protocol = new vod_client(1000, &c);

    server_addr s;
    s.port = Convert::ToInt32(textBox2->Text);
```

```
  pin_ptr<const wchar_t> wch = PtrToStringChars(textBox1->Text);
  size_t sizeInBytes = ((textBox1->Text->Length + 1) * 2);
  char *ch = (char *)malloc(sizeInBytes);
  size_t convertedChars = 0;
  wcstombs_s(&convertedChars,
                ch, sizeInBytes,
                wch, sizeInBytes);
  s.ip_address = ch;

  login = new struct login_data;

  client_protocol->connect(Convert::ToInt32(textBox5->Text), &s, login);

  dt = new data_thread(client_protocol, login->video_size);

  groupBox1->Enabled = false;
  groupBox2->Enabled = true;
};


private: System::Void button2_Click(System::Object^  sender, System::EventArgs^  e)
{
  axQTControl1->Movie->Pause();
};

private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e)
{
  if( axQTControl1->URL == "" ) {
    System::Diagnostics::Process^ p = System::Diagnostics::Process::GetCurrentProcess();
    System::String^ filname = "c:\\video" + System::Convert::ToString(p->Id) + ".mov";
    axQTControl1->URL = filname;
    axQTControl1->Movie->TimeScale = 1000;
    textBox8->Text = Convert::ToString(axQTControl1->Movie->Duration);
    skip_distance = login->skip_distance;
  }

  axQTControl1->Movie->Play(1);
  timer1->Enabled = true;
};

private: System::Void button4_Click(System::Object^  sender, System::EventArgs^  e)
{
};

private: System::Void button6_Click(System::Object^  sender, System::EventArgs^  e)
{
  timer1->Enabled = false;
  axQTControl1->Movie->Stop();
  axQTControl1->URL = "";

  delete dt;

  client_protocol->disconnect();
  delete client_protocol;

  groupBox1->Enabled = true;
  groupBox2->Enabled = false;
```

```
      System::Diagnostics::Process^ p = System::Diagnostics::Process::GetCurrentProcess();
      System::String^ path = "c:\\video" + System::Convert::ToString(p->Id) + ".mov";
      //System::IO::File::Delete(path);
    };

    private: System::Void button5_Click(System::Object^  sender, System::EventArgs^  e)
    {
      int skip_bb = Convert::ToInt32(textBox6->Text);
      client_protocol->skip(skip_bb);
      timer1->Enabled = false;
      axQTControl1->Movie->Pause();
      axQTControl1->Movie->Time = skip_bb * skip_distance;
    };

    private: System::Void timer1_Tick(System::Object^  sender, System::EventArgs^  e)
    {
      textBox7->Text = Convert::ToString(axQTControl1->Movie->Time);
      textBox9->Text = Convert::ToString((axQTControl1->Movie->Duration) -
 (axQTControl1->Movie->Time));
      textBox7->Refresh();
      textBox9->Refresh();
    }

    public: System::Void TriggerPlay()
    {
      axQTControl1->Movie->Pause();
    }
  private: System::Void textBox1_TextChanged(System::Object^  sender, System::EventArgs^  e) {
           }
};
}
```

data_thread.h

```
#ifndef DATA_THREAD
#define DATA_THREAD

//#include <iostream>
//#include <fstream>

class data_thread {

private:
  DWORD  thread_id;
  HANDLE thread;

  vod_client* v_client;
  unsigned int v_movie_size;


public:
  data_thread(vod_client* client, unsigned int movie_size);
```

```
  ~data_thread();
};

#endif
```

data_thread.cpp

```
#include "stdafx.h"
#include "data_thread.h"


void data_retrive(void* args)
{
  vod_client* vc = (vod_client*)args;

  System::Diagnostics::Process^ p = System::Diagnostics::Process::GetCurrentProcess();
  System::String^ filname = "c:\\video" + System::Convert::ToString(p->Id) + ".mov";
  System::IO::FileStream^ fs = System::IO::File::Open(filname,
                                                      System::IO::FileMode::Open,
                                                      System::IO::FileAccess::Write,
                                                      System::IO::FileShare::ReadWrite);

  void* buffer = malloc(2000000);
  unsigned long offset;

  for(;;) {
    // Receive data
    offset = 0;
    size_t len = vc->recv_data(buffer, 2000000, offset);

    // Write data to file
    fs->Position = offset;
    for( int i = 0; i < len; i++ )
      fs->WriteByte(((char*)buffer)[i]);

    fs->Flush();

    System::Diagnostics::Debug::WriteLine("Writing data to client, length: " +
   len + " - offset: " + offset );
 }

return;
}


data_thread::data_thread(vod_client* client, unsigned int movie_size)
{
  System::Diagnostics::Process^ p = System::Diagnostics::Process::GetCurrentProcess();
  System::String^ path = "c:\\video" + System::Convert::ToString(p->Id) + ".mov";
  System::IO::FileStream^ fsa  = System::IO::File::Create(path);
  fsa->SetLength(movie_size);
  fsa->Close();

  thread = CreateThread(NULL,
```

```
0,
(LPTHREAD_START_ROUTINE) data_retrive,
(void*) client,
0,
&thread_id);
}


data_thread::~data_thread()
{
  //fs->Flush();
  //fs->Close();

  TerminateThread(thread, 0);
}
```

Appendix D

Server application source files

VOD_Server.cpp

```
// VOD_Server.cpp : main project file.

#include "stdafx.h"
#include "Form1.h"

using namespace VOD_Server;

[STAThreadAttribute]
int main(array<System::String ^> ^args)
{
// Enabling Windows XP visual effects before any controls are created
Application::EnableVisualStyles();
Application::SetCompatibleTextRenderingDefault(false);

// Create the main window and run it
Application::Run(gcnew Form1());
return 0;
}
```

Form1.h

```
#pragma once

namespace VOD_Server {

using namespace System;
using namespace System::ComponentModel;
using namespace System::Collections;
using namespace System::Windows::Forms;
using namespace System::Data;
using namespace System::Drawing;
using namespace System::Threading;
  using namespace System::IO;

/// <summary>
/// Summary for Form1
```

```
///
/// WARNING: If you change the name of this class, you will need to change the
///          'Resource File Name' property for the managed resource compiler tool
///          associated with all .resx files this class depends on.  Otherwise,
///          the designers will not be able to interact properly with localized
///          resources associated with this form.
/// </summary>
public ref class Form1 : public System::Windows::Forms::Form
{
  private:
    vod_server* my_server;
  private: System::Windows::Forms::Label^  label1;
  private: System::Windows::Forms::Label^  label2;
  private: System::Windows::Forms::TextBox^  tbxLocalCCPport;
  private: System::Windows::Forms::TextBox^  tbxLocalDCPport;


           data_thread* dt;

  public:
Form1(void)
{
InitializeComponent();
//
//TODO: Add the constructor code here
//
}

protected:
/// <summary>
/// Clean up any resources being used.
/// </summary>
~Form1()
{
if (components)
{
delete components;
}
}
private: System::Windows::Forms::Button^  button1;
private: System::Windows::Forms::Button^  button2;
protected:

private:
    /// <summary>
/// Required designer variable.
/// </summary>
System::ComponentModel::Container ^components;

#pragma region Windows Form Designer generated code
/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
void InitializeComponent(void)
{
      this->button1 = (gcnew System::Windows::Forms::Button());
```

```
this->button2 = (gcnew System::Windows::Forms::Button());
this->label1 = (gcnew System::Windows::Forms::Label());
this->label2 = (gcnew System::Windows::Forms::Label());
this->tbxLocalCCPport = (gcnew System::Windows::Forms::TextBox());
this->tbxLocalDCPport = (gcnew System::Windows::Forms::TextBox());
this->SuspendLayout();
//
// button1
//
this->button1->Location = System::Drawing::Point(123, 114);
this->button1->Name = L"button1";
this->button1->Size = System::Drawing::Size(75, 23);
this->button1->TabIndex = 0;
this->button1->Text = L"Start";
this->button1->UseVisualStyleBackColor = true;
this->button1->Click += gcnew System::EventHandler(this, &Form1::button1_Click);
//
// button2
//
this->button2->Enabled = false;
this->button2->Location = System::Drawing::Point(15, 114);
this->button2->Name = L"button2";
this->button2->Size = System::Drawing::Size(75, 23);
this->button2->TabIndex = 1;
this->button2->Text = L"Stop";
this->button2->UseVisualStyleBackColor = true;
this->button2->Click += gcnew System::EventHandler(this, &Form1::button2_Click);
//
// label1
//
this->label1->AutoSize = true;
this->label1->Location = System::Drawing::Point(12, 19);
this->label1->Name = L"label1";
this->label1->Size = System::Drawing::Size(78, 13);
this->label1->TabIndex = 2;
this->label1->Text = L"Local CCP port";
//
// label2
//
this->label2->AutoSize = true;
this->label2->Location = System::Drawing::Point(12, 59);
this->label2->Name = L"label2";
this->label2->Size = System::Drawing::Size(76, 13);
this->label2->TabIndex = 3;
this->label2->Text = L"LocalDCP port";
//
// tbxLocalCCPport
//
this->tbxLocalCCPport->Location = System::Drawing::Point(109, 16);
this->tbxLocalCCPport->Name = L"tbxLocalCCPport";
this->tbxLocalCCPport->Size = System::Drawing::Size(100, 20);
this->tbxLocalCCPport->TabIndex = 4;
this->tbxLocalCCPport->Text = L"20000";
//
// tbxLocalDCPport
//
this->tbxLocalDCPport->Location = System::Drawing::Point(109, 56);
```

```
      this->tbxLocalDCPport->Name = L"tbxLocalDCPport";
      this->tbxLocalDCPport->Size = System::Drawing::Size(100, 20);
      this->tbxLocalDCPport->TabIndex = 5;
      this->tbxLocalDCPport->Text = L"21000";
      //
      // Form1
      //
      this->AutoScaleDimensions = System::Drawing::SizeF(6, 13);
      this->AutoScaleMode = System::Windows::Forms::AutoScaleMode::Font;
      this->ClientSize = System::Drawing::Size(267, 170);
      this->Controls->Add(this->tbxLocalDCPport);
      this->Controls->Add(this->tbxLocalCCPport);
      this->Controls->Add(this->label2);
      this->Controls->Add(this->label1);
      this->Controls->Add(this->button2);
      this->Controls->Add(this->button1);
      this->Name = L"Form1";
      this->Text = L"Server";
      this->ResumeLayout(false);
      this->PerformLayout();

    }
#pragma endregion

private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
button1->Enabled = false;

       int localDCPport = Convert::ToInt32(tbxLocalDCPport->Text);
       int localCCPport = Convert::ToInt32(tbxLocalCCPport->Text);

my_server = new vod_server(localDCPport, localCCPport);
      my_server->open();

      //dt = new data_thread(my_server);

button2->Enabled = true;
    }

    private: System::Void button2_Click(System::Object^  sender, System::EventArgs^  e) {
  button2->Enabled = false;

    my_server->close();
      delete my_server;

  button1->Enabled = true;
}
};
}
```

Appendix E

Screendump of applications