Media Streaming Distribution Network with Network Coding

Prototype System Design, Implementation and Evaluation

J O H A N N E S   E R I K S S O N

Master of Science Thesis
Stockholm, Sweden 2006

ICT/ECS-2006-107

Media Streaming Distribution Network with Network Coding

Prototype System Design, Implementation and Evaluation

Master of Science Thesis
August 2006

Johannes Eriksson joherik@kth.se
ZTE Research and Development Lab
ZTE Corporation, Sweden AB
Stockholm, Sweden

Examiner:	Vladimir Vlassov vlad@it.kth.se
Supervisor:	Eric Sun sun.zheng1@zte.com.cn
Advisor:	Richard Wang wang_rz@yahoo.com

Master of Science Thesis
Stockholm, Sweden 2006

ICT/ECS-2006-107

Abstract

During the last years more and more network applications are developed to build up overlay distribution networks. Overlay distribution networks that fulfill the people's needs, such as sharing files with each other. A great example is to look at BitTorrent that millions of people are using today. There are also demands on good streaming services; people want to see the tv-shows they missed and follow lectures over the internet and preferably live. With other words lots and lots of data needs to be transferred over the network. It would be nice to optimize the transmission of data in these overlay networks and not only use plain routing. This can be achieved with Network Coding.

First of all this thesis is a part of a bigger project. Together in this project we have made research in different areas such as Network Coding, Peer Selection, transport protocols and different kinds of packet formats. After that a prototype was developed, a prototype for a media streaming distribution network that uses Network Coding.

The focus in this thesis has been to first of all implementing the communication parts such as the message structure and the packet transmission part, and second has been to implement the actual prototype application. Apart from that this thesis has made some tests on the prototype and after that an evaluation took place.

The evaluation told us that there is a lot of work to do in the future to get a prototype that could challenge the existing streaming services, but we are sure that the combination of an overlay network where every node are helping distribute information and using Network Coding are going to be a great solution for streaming media to a big crowd.

Vocabulary

Term	*Symbol*	*Definition*
Field size:		The bit size of the individual data values/symbols contained in the packet to be sent, e.g. 8 bits or 16 bits.
Generation:		The media stream to be sent via network coding is divided into "generations". A single generation contains packets that are all related to the same set of source (*h*) vectors. [2]
Global code:	*G*	The encoding vectors that are sent node to node.
Innovative info:		Incoming global code to a node that provides new information (non redundant) which will help create a full-rank matrix.
Jitter:		Uncontrollable latency variance.
Local code:		The temporary vector that is multiplied by the received incoming global code vectors at a node to create a new outgoing global vector.
NC dimension:	*h*	The dimension of the matrix created for network coding of the source stream. It should be equal to or less than the value of the "minimum cut" between the source and any receiver. The input stream is divided into *h* number of *x*-streams.
NC:		Network Coding: a method used to encourage and allow mixing of data at intermediate nodes, which helps to maximize the flow of data across the minimum cut of the topology. [8]
Neighbor:		A node that is only one “hop” away in the topology. The neighbors are the only nodes that you know of.
Node:		Can be a receiver, source or a receiver&source in the overlay network.
Parent:		A Parent is a node that distributes information to another node (Child).
Child:		A Child is a node that receives information from another node (Parent).
Source:		A Source is a node that helps distribute information in the overlay network. Also called Parent.
Receiver:		A Receiver is a node that receives information and uses it but doesn’t forward it to someone else.
Non-innovative info:		Incoming global code to a node that provides no new information (redundant).

Term	*Symbol*	*Definition*
Peer:		An active node in the overlay network that you're communicating with. Parent or a child.
Premature transmission:		Each node will transmit coded content after certain amount of time whether it receives full ranked content or not.
Rank:		The rank of a matrix is the number of the linearly independent rows or columns of a matrix. [9]
Server Rate:		Current output bandwidth at the server/source.
Source Rate:	r_s	The bandwidth required to output the total of the individual x streams ($r_s = h \cdot r_x$)
Time invariant NC:		The initial global encoding matrix (from the server) does not change throughout the lifetime of the transmission.
Time variant NC:		The initial global encoding matrix (from the server) changes periodically over time. This method provides a more "robust" network coding solution, as with periodic change comes a greater chance that the matrix has full rank and can be solved completely.
Tracker:		A standalone application that the nodes connects to, to get information about the network.
Full rank		A matrix that is non-reducible and has a rank that is equal to the number of rows; is also called a non-singular matrix. If a matrix is full rank, it can be solved.

1. Introduction

Large scale communication networks like Internet are used almost by everyone today. Nowadays users not only want to read information, they also want to get information in the form of video/audio streams. To meet the users' needs lots of different techniques are discussed and investigated by researchers all around the world.

1.1 Problem Statement

There are several problems you need to think about when building a prototype of a media streaming distribution network. The prototype that is working on every node in the network should have several choices for an area of usage. It could choose to act as a source node, receiver node or as both.

A source node only encodes the incoming data stream and forwards it to its children whereas a receiver node only decodes the incoming data stream and sends it out to the screen. As you probably already figured out the prototype collects all information, decodes it, send it to the screen and then encodes the streams and send them to its children nodes when it is working as a source and a receiver.

Of course the most common node acts as both a source and a receiver, because in our case clients don't wait for the whole video stream before they send it along, they send it along continuously. Working only as a source is not going to be that common for the clients but maybe the company that offers this video stream needs to have a few servers to help the network stream the video to all its users.

The choice for act only as a receiver should not be the clients' decision; the client should become only a receiver automatically if the client has a really bad upload capacity. Clients that are joining the network from a mobile phone or a bad ADSL connection are examples on clients that will only act as receivers.

When talking about encode data streams it means that all of the incoming data streams will be combined using network coding (NC) and network coding is also used when decoding the data streams. NC is the main technique that we are going to use when implementing this prototype and therefore it will steer some of the other parts of the prototype, parts such as peer selection and data processing.

Since there is usually more than one incoming data stream at the input of a node, it is necessary to acquire synchronization. Synchronization of the incoming data streams is a main requirement for a media distributed network. If you don't supply synchronization among these data streams there can be serious problems for real-time applications such in our case media streaming.

To acquire synchronization you will need a buffering model. You could tag every packet, related with the same source vector, with a specific number and then put all incoming packets in a single buffer sorted after the specific number. In other words, every node needs a mechanism that puts the incoming packets in the buffer in the right order.

As mentioned above there is several main issues that this thesis will have to dig deep into, issues such as NC, peer selection and data processing. Dig deep to reach the goal of this project, that is to design, implement and evaluate a prototype application to stream media over a peer selected network. This network should use network coding to make the packet transmission more efficient.

1.2 Related Work

There is lots of researching going on about network coding, most of the available research is theoretically like in [1, 3, 4], but more and more practical views are coming as you can see in [2]. Also most of the existing research is about NC in a file distributed way and not in a real time streaming way.

Anyhow, there is some available theoretically research in real time streaming using NC but to our best knowledge it seems that no one has researched and build a prototype in this area. You could also find several different approaches and theories to tackle the issues of peer selection [14, 15, 19] and data processing [5-7, 9-12] when building such a prototype.

1.3 Structure of thesis

First of all I want to explain that this thesis is a part of a big project where four people, included me, worked pretty much together. This means that much of the research were done together in the group and that led to that some of the subchapters in the background chapter were also done in a group-work.

Subchapter such as: 2.1 and 2.2.1 are group-work and subchapters such as: 2.2.2 and 2.3 are more or less my own parts.

The reminder of this thesis is structured as follow: Chapter 2 covers the research areas related to the thesis; Chapter 3 presents the analysis and design of the system prototype; Chapter 4 describes the implementation of the proposed prototype design in detail; the evaluation of the implemented prototype is discussed in Chapter 5; then the conclusions of the thesis is found in Chapter 6; and finally Chapter 8 covers the future work.

2. Background

To reach the goal of the thesis some real investigation is needed in certain areas. Areas such as peer selection and data processing are very important when talking about real-time media streaming in an overlay network. Important for sure, but one thing one needs to think of is that the peer selection and the data processing part must be investigated in the sense that network coding is used. Therefore it is necessary to dig deep into NC first, so one will understand all the issues and problems that may appear when trying to come up with a solution for peer selection and data processing.

2.1 Network Coding

There has been a lot of research on routing and making routing more efficient because plain routing basically just forwards packets. With network coding, nodes in an overlay network send out packets that are combinations of information from the previous incoming packets. This leads to possible throughput improvements and a more robust network as said in [1].

Combining incoming information requires some computations at the nodes in the network; this will not be a bottleneck these days because of the very powerful computers. Instead the network bandwidth will be the bottleneck and therefore it has become very important to try maximizing the bandwidth.

2.1.1 Overview

The main idea behind NC is that in some cases it would be more optimized to send a combination of information instead of sending all the information separate. This would lead to a reduced amount of network usage but at the same time it achieves the same result.

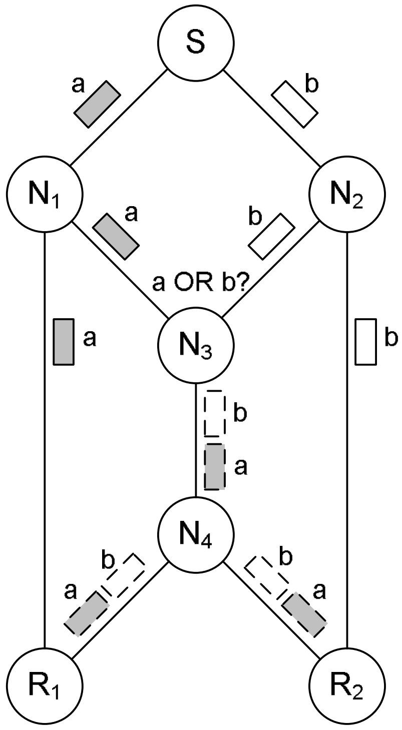


Figure 2.1: A possible scenario without network coding.

In Figure 2.1 the S is the source that is sending some information to the receivers (R). For the information to get there, it must travel through some nodes (N) in the network. The two letters in the figure, a and b, is the actual information that the receivers wants, it must have both a and b to get the "message".

Figure 2.1 also shows that when all links have maximum bandwidth usage and network coding is not used, N_3 is forced to choose if it should forward packet **a** or **b**. This means that depending on what N_3 chooses, R_1 and R_2 will be satisfied, but not at same time. If there are some bandwidth left for $N_{3,}$ N_3 could transmit both **a** and **b** and that would satisfy both R_1 and R_2 at the same time. But this scenario would also lead to that the bandwidth is not optimum used, because N_4 is going to send both **a** and **b** to both R_1 and R_2.

This problem could be solved if N_4 knows which packet to send where, because then N_4 only needs to send packet **b** to R_1 and packet **a** to R_2. Many of the problems could be solved if all facts are known for each node, but this also means that the topology has to be known for each node. Another way to solve these problems is to use NC which will be explained further down.

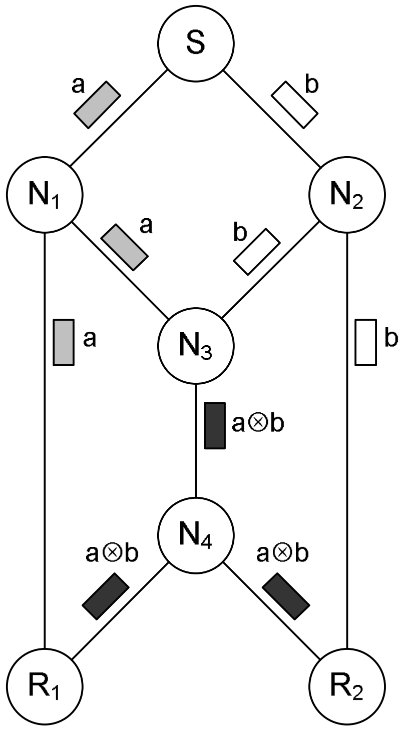


Figure 2.2: A possible scenario with network coding.

As shown in Figure 2.2, when all links have maximum bandwidth usage and network coding is used N_3 is forced to do a combination of **a** and **b** to be able to forward packet **a** and **b**. This means that R_1 and R_2 will be satisfied at the same time, because R_1 could use the **b** part of the combination and R_2 could use the **a** part of the combination.

The major problem with NC is that when a certain packet is not received at a node then would all packets be useless for that node. This is because it is impossible to resolve the original information with some unknown encoded data.

One way to do this problem smaller and also make the implementation more feasible is to divide the information into generations. Then if some part is lost would that only mean that the all the parts in that generation would be lost.

2.1.2 Basics

As said before NC is letting nodes encode the incoming information before sending it on, with help from some coding scheme. Among the simplest coding schemes is linear coding, which regards a block of data as a vector over a certain base field and allows a node to apply a linear transformation to a vector before passing it on.

Linear network coding is proved to be an optimum scheme for encode information according to this paper [2].

Linear equations are simple as said before, and the simplest linear system is one with two equations and two variables.

Solving three-variable, three-equation linear systems is more difficult, at least initially, than solving the two-variable systems because the computations involved are messier. The systematized method for solving the three-or-more-variables systems is called Gaussian elimination. Gaussian elimination is the most common method to solve linear equation systems and it is not complicated at all.

Theoretically NC consists of two different parts. The first part is the global encoding matrix (G_t) which could be fixed or randomly created. This is the entire idea behind NC which makes it possible for a node to forward some data, without first having received the complete generation. The other part of NC is the original information (X). These will then be multiplied which then generates the result (Y) according to the following formula:

$$Y = G_t \cdot X$$

The crucial thing is to choose a global encoding matrix that makes it possible to resolve the original information at a receiver later on. This is also based on the field size because if a field size that is large enough is chosen then would this matrix be solvable with high probability. But on the other hand the field size should be kept as low as possible to achieve the highest throughput possible and by that also keep the overhead as low as possible. One could say that field size is how much information that is available to represent each piece of the encoded data.

A solution to this problem would be to use finite fields that make it possible to have a fixed field size. After some reading [2]-[4] it seems that 2^8 or perhaps 2^{16} would be an appropriate fixed field size.

A finite field is a field with a finite field order (i.e., number of elements), also called a Galois field. The order of a finite field is always a prime or a power of a prime. For each prime power, there exists exactly one (with the usual caveat that "exactly one" means "exactly one up to an isomorphism") finite field GF(p^n) [5].

Another NC technique is random network coding (RNC), and the difference is that the global encoding vector is randomized instead of static and predefined. You will find more information about random network coding in [3].

2.1.3 Practical

Below is NC described in a more practical way.

The general procedure of NC for the source is as follows:

1. Start with the original information (X)
2. Create a random or fixed global encoding matrix (G_t)
3. Multiply the global encoding matrix (G_t) with the original information (X) to retrieve the encoded data (Y). ($Y = G_t * X$)
4. Then send both G_t and Y to the children of that source.

For a middle node would the procedure of NC be quite similar:

1. Start with receiving the global encoding matrix (G_t) and the encoded data (Y_r).
2. Create a random or fixed local encoding matrix (G_l).
3. Calculate a new global encoding matrix (G_n) by multiplying the old global encoding matrix with the local encoding matrix ($G_n = G_l * G_t$).
4. Calculate new encoded data (Y_n) by multiplying the old encoded data with the local encoding matrix ($Y_n = G_l * Y_r$).

For a receiver would the procedure of NC be somewhat related:

1. Start with receiving the global encoding matrix (G_t) and the encoded data (Y).
2. Solve X in the formula ($Y = G_t * X$).
3. Then if everything went well should the original information be resolved.

A practical and more mathematical example of NC could be seen in Appendix A. The example was developed after careful reading of this research paper [2].

2.2 Data Processing

The data processing part is divided up in three parts: buffering model, data format and encode/decode function for network coding. Every node in an overlay network needs a buffering model, a buffering model that consist of one or more buffers, size of the buffers, a flushing policy and much more.

Real time streaming makes it very important to know when and what to flush to its children. The data format part is also very important, how to build up the different message types that those different peer selection scenarios are using; what should be included and which protocol is going to be used when sending these messages.

Apart from these two issues every node in the network needs to have a network coding encode/decode function. This is the heart of the whole prototype; which mathematically areas are going to be used to succeed on combining, encoding and decoding the incoming data streams.

2.2.1 Buffering Model

2.2.1.1 Introduction

The buffering model specifies how the strategy of the two buffers works. The main task of the first buffer, called transmission buffer, is to synchronize the packets' arrivals and departures. This buffer contains the non decoded information.

The second buffer, called playback buffer, is to take care of the decoded information and store it in the right order for the actual playback.

In practice the capacities of different edges vary (depending on loss, congestion, competing traffic etc.), thereby must the transmission be synchronized. To get it synchronized in practice the packet must contain a field with information (generation number) about which generation a certain packet belongs to.

Packets that are related to same source vectors $X_1, \ldots, X_h$ are in same generation where h is the generation size. This field would be sufficient if it has a size that is one byte because same generation number could be reused over time.

When a packet arrives at a node on whichever edge, the packet is put into the transmission buffer sorted by the generation number. Then on first possible opportunity or after a while the information will be sent on the outgoing edges. Before sending the packet, it should be generated as a random linear combination of packets from the buffer within the actual generation.

The current generation will regularly be taken from the transmission buffer to the playback buffer. The information could be deleted after some time or saved for a certain time, depending on if the node should be able to resend the information at a later stage.

2.2.1.2 Absolute delay/Latency

The absolute delay (D_n) also known as latency is the time it takes for a packet to travel from the source (S) to a destination node (N_n) as shown in Figure 2.3.

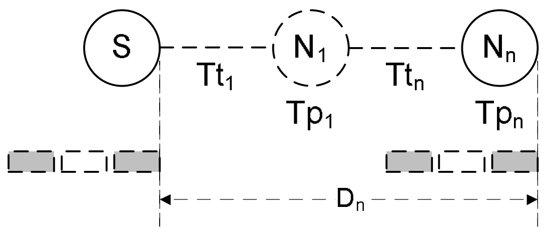


Figure 2.3: The absolute delay (D_n) in a schematic network.

Figure 2.3 also shows that the packet may travel through some other nodes (N_1, .., N_{n-1}) to get to the destination node (N_n) and then the absolute delay will increase. From that the absolute delay is the sum of the travel time between every node on the way to the destination, plus the sum of all nodes' processing time.

$$D_n = \sum_{i=1}^{n} (T_{p_i} + T_{t_i})$$

Where:

- T_p is the processing time at the node (time between receiving and transmitting).
- T_t is the travel time from the parent to the child.

The absolute delay could be minimized in two ways. The first is to keep the node close to the source i.e. have a peer selection strategy that creates a topology with low diameter.

The second way is to minimize the processing time by having a flushing strategy that prioritize low processing time.

The absolute delay is not as important as keeping a low delay spread, because if a node has a big absolute delay it just means that the node are experience a constant delay of the stream. This means that the node would get to see the information a bit later than a node with a smaller absolute delay.

2.2.1.3 Delay spread

The delay spread (D_{sn}) at a node (N_n) is the time difference in arrivals between the first packet (D_{fn}) and the last packet (D_{ln}) in one generation as shown in Figure 2.4.

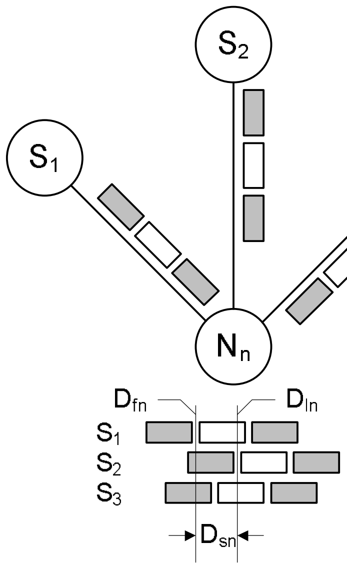


Figure 2.4: The definition of the delay spread (D_{sn}).

Figure 2.4 shows a node that getting incoming packets from three different parent-peers and the first incoming packet from each and one of the parent-peers belongs to the same generation. Every packet in the same generation is needed for decoding the packets and therefore the delay spread is an important criterion to determine the buffer size. Delay spread is then the time between the first packet from a certain generation arrives and the last one.

$$D_{s_{nx}} = D_{l_{nx}} - D_{f_{nx}} = \max\left(\sum_{i=1}^{n}(T_{p_i} + T_{t_i})\right) - \min\left(\sum_{i=1}^{n}(T_{p_i} + T_{t_i})\right)$$

Where:

- D_s is the actual delay spread.
- D_l is the absolute delay (see Chapter 2.2.1.2) for the last packet.
- D_f is the absolute delay (see Chapter 2.2.1.2) for the first packet.

This means that the delay spread is mostly dependant of the flushing strategy in the transmission buffers of preceding nodes (parents) between the source and the node. It also depends on the delay difference over different preceding edges. The delay spread will propagate because of the accumulated T_p. However if the flushing strategy in the transmission buffer flushes the incoming packets as soon as possible then would $T_p \approx 0$.

It is preferable to keep the delay spread as low as possible to assure that the transmission buffer could be as small as possible. There are two ways to minimize the delay spread.

The first is to choose a flushing strategy that makes the processing time at every node constant from the first to the last packet in the transmission buffer.

The second method is to assure that the preceding edges all have same absolute delay.

Ideally when every edge have same delay and every node have same processing time, the delay spread could be minimized by having a well designed peer selection strategy. This strategy would create a topology with receivers that all have its sources (parents) at the same distance from the source. This would then result in a topology with nodes that could have differences in absolute delay but still having a small delay spread.

2.2.1.4 Jitter

The jitter (D_j) is the variation in absolute delay over time from the source (S) to a destination node (N_n) as shown in Figure 2.5. Jitter is caused by network congestion, timing drift, or route changes [6].

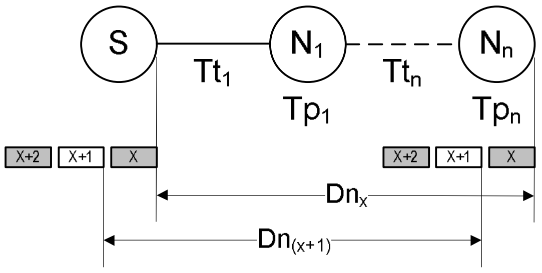


Figure 2.5: The jitter (D_j) in a schematic network.

Figure 2.5 shows that a packet (x) has an absolute delay of (Dn_x) and the next packet (x+1) has an absolute delay of ($Dn_{(x+1)}$), jitter is the time difference between these two delays.

$$D_{j_n} = D_{n_{(x+1)}} - D_{n_x} = \left(\sum_{i=1}^{n} (T_{p_i} + T_{t_i}) \right)_{(x+1)} - \left(\sum_{i=1}^{n} (T_{p_i} + T_{t_i}) \right)_{x}$$

Where:

- D_j is the actual jitter.
- Dn_{x+1} is the absolute delay (see Chapter 2.2.1.2) for the (x+1) packet.
- Dn_x is the absolute delay (see Chapter 2.2.1.2) for the (x) packet.

This means that if the jitter (D_j) is positive then the latter packet (#2) travels slower than the earlier packet (#1) and if the jitter is negative then the opposite will occur.

The jitter is dependent of the network steadiness and if the processing time at a preceding node differs over time. The problem is that it is hard to do something about the network more than choosing stable connections i.e. have a peer selection strategy that takes this into concern.

The processing time could be different over time if a particular node has a comprehensive workload at a certain time, which affects the data processing rate. But this is also hard to do something about more than try to prioritize the decoding and encoding process before others.

As said earlier it is hard to minimize the jitter but there is one way to deal with the problem and that is to have a jitter buffer, from now on it is called the playback buffer. This buffer intentionally delays the arriving packets so that the overlaying software (media player) experiences a clear connection with very little problems.

2.2.1.5 Transmission buffer

The main goal of the transmission buffer is to take care of the delay spread. This buffer could work in two different ways.

- By block decoding, that means that the node collects h or more packets and later on hopes to be able to invert G_t.

- By earliest decoding, that means that the node performs Gaussian elimination after each packet arrival. Then could the node detect and discard non-innovative packets as soon as possible. This would also lead to that the computational load for the node will be distributed over time.

Earliest decoding is the preferred method. This is based on the fact that it would also do the decoding faster after a complete generation has been received at the node, which means that the playback buffer could be a little bit smaller. Earliest decoding would also mean that the node knows which packets to transmit to its children and by that be able to only send innovative packets.

A well considerer flushing strategy should be implemented to prevent deadlock that could happen if every node in the network waits for new information and none of them has received full rank in the current generation.

This type of problem could be solved by using two different strategies. The first strategy is to send new information to the children when the first opportunity arises (when the outgoing queue is empty). The second strategy is to use premature transmission. This means that the node transmits new information to its children before it receives full rank in the transmission buffer based on a pre-set waiting time.

The first chance flushing strategy has two main advantages. The first is that the latency and the delay spread automatically will be kept small.

The second one is that the information will be sent in several ways which will lead to redundancy. Unfortunately the redundancy may also lead to a much larger network load. This happens because in the worst case scenario would every innovative packet, that is received at the parents, be sent to the receiver. This means that the receiver gets a complete generation from every parent and that is not optimum if the network load should be kept as low as possible.

However the network load could be smaller if the receiver sends a special packet back to the parent when it has got full rank. This would then stop the transmission from the parents to the receiver of that particular generation.

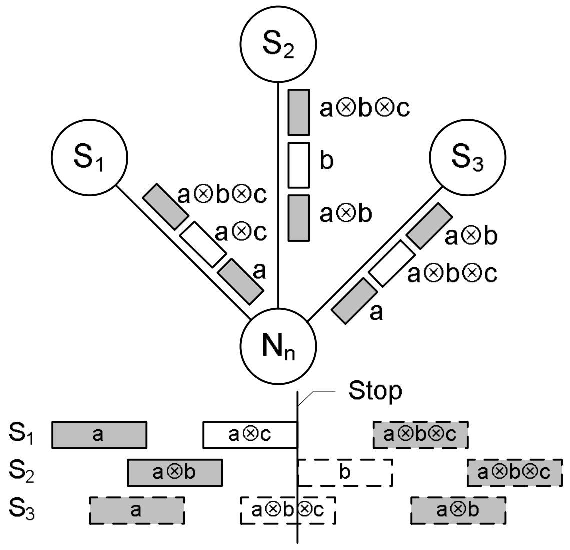


Figure 2.6: The first chance flushing strategy.

In Figure 2.6 the sources will send three packets each which will then lead to an unnecessary transmission of six packets (the packets with dotted lines). These are unnecessary because the node wants to resolve the original information, that is a, b and c, and it can do that by using only three packets (the packets with filled lines). Could the destination node (N_n) instead send acknowledge when it has full rank would the parents be able to stop sending the remaining packets. In this particular case, this would lead to that only two packets will be sent unnecessarily. This means that only five information packets will be sent instead of nine.

If one or two arbitrary connections would be lost should this only lead to the fact that the stopped message will be sent later in time, but the receiver could anyhow get full rank. The attentive reader could also see that for the current generation there is one of the sources (S_3) that is not useful at all because every packet from that source is non-innovative when received.

This is nothing to be concerned about for one particular generation. But if this repeats over time then should that source be dropped in favour for a new one that hopefully sends a higher degree of innovative packets.

The premature transmission flushing strategy has one crucial advantage. This is that it keeps the network load at a minimum. The problem is that it is not redundant, which means that if some receiver does not get full rank for one generation then that generation could not be saved. This would unfortunately lead to an interruption of the stream and as result poor quality for the user.

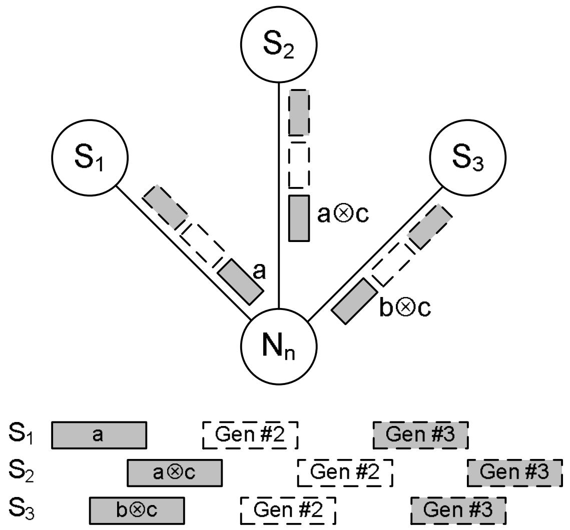


Figure 2.7: The premature transmission flushing strategy.

In Figure 2.7 the sources will send one packet each, which together leads to full rank at the destination node (N_n) and can therefore resolve the original information. This keeps the network load at a minimum, but if one of the sources is lost this would make it impossible to get full rank.

This is the case when the information is streamed in real time and there is no time to ask for missing packets, especially when the receiver does not know which packet that is missing and which one of the sources has it.

The buffer size of the transmission buffer is hard to calculate and know in advance. One method that possibly could help and dynamically finger point in the right direction is to measure the delay spread for some of the first generations received, and then calculate the average of them. This result could then be multiplied with a factor (z) to make it large enough.

The problem is that in an implementation of this buffer, the size should not be a certain time but it should be a number of generations. This could be determined by taking the previous result and divide it with the average time of the recently received packets and then the result will be the number of generations.

The main problem with this approach is that it would be much better to have a reasonable buffer size directly from the beginning. In this case the buffer size will be dynamically calculated after a certain time.

Another approach is to have a static transmission buffer size that is defined by simulation. This would then mean that the buffer would not be adapted to the actual circumstances, but it might still be sufficient to get a size near the required one as long it is not smaller than required.

If this approximation will be used, it seems reasonable to believe that the necessary size is dependent difference in distance of the parents from the source.

2.2.1.6 Playback buffer

The playback buffer should work in a rather straightforward way and the main task of this buffer is to take care of the jitter and the frame. The playback buffer contains the decoded information that will be played by the overlay software (media player). It will receive the information from the transmission buffer as fast as possible when the transmission buffer has gotten a generation with full rank that it could solve.

The basic idea with this “extra” buffer is to be able to have some time between arrivals of the first packet in a generation to the last in the transmission buffer. This buffer must have sufficient size to handle a certain playback time before it runs out just to be sure that the stream is complete.

The flushing strategy of the playback buffer is quite simple. It should flush or erase the information from the buffer when it has been played, and then the information thereby has served its purpose. This would be done at even intervals because the information stream rate is constant.

Would this buffer be empty at anytime, the procedure starts over and the overlaying software has to wait for this buffer to be refilled.

It is difficult to know how big the buffer size of the playback buffer should be. It is hard to know mostly because the main purpose of the playback buffer is to take care of the problems caused by jitter. As said before, jitter is caused by network congestion, timing drift, or route changes and these factors are impossible to know in advance. One method that possibly could help and dynamically finger point in the right direction is to measure the jitter for some of the first generations received and then calculate the average of them.

The main problem is also the same as it was for the transmission buffer, which is that it would be much better to have a reasonable buffer size directly from the beginning. But in this case the buffer size will be dynamically calculated after a certain time.

Another approach is to have a static playback buffer size that is defined by simulation. This would then mean that the buffer would not be adapted to the actual circumstances. But as said before it might still be sufficient to get a size near the required one as long it is not smaller than required.

If this approximation will be used, then it seems reasonable to believe that the necessary size is dependent difference in distance of the parents from the source. An example of buffer size is 5 seconds. That is the default buffer size in the windows media player [7].

2.2.2 Packet format

The packet format is very important to get an effective communication between nodes in the network. There are several known predefined transport protocols that could do the job for us, protocols like TCP, UDP, RTP and RTCP. The different protocols are good for different things, for example TCP is not good for streaming but UDP is. Why that is so, is going to be explained later in this chapter.

There is also one other way to go, and that is to build our own packet format on top of UDP and TCP. The benefit of that suggestion would be to minimize the overhead because many of the fields in RTP and RTCP are not needed for the prototype but on the other hand it is a little bit risky and time consuming. Another benefit of developing an own packet format is that it can include whatever that is needed, with fields exactly as big as they should be; with other words it provides flexibility.

2.2.2.1 TCP

Transport Control Protocol or TCP is defined in RFC 793 [10] and it is not good for streaming because it is a connection oriented protocol and that makes it not fast enough, plus it also comes with a lot of overhead. TCP is however really good for building up an overlay network topology because it is reliable and one can get a lot of feedback, plus it also guarantee delivery. So, TCP is great but it is not sure that it is enough for our purpose for the network, maybe some fields of our own needs to be applied on top of TCP.

2.2.2.2 UDP

User Datagram Protocol or UDP is a connectionless transport protocol defined in RFC 768 [11]. UDP provides a procedure for application programs to send messages to other programs with a minimum of protocol mechanism. UDP runs on top of IP (Internet Protocol) and it uses IP for transporting a message from a computer to another, and provide unreliable datagram delivery semantics of IP.

UDP is good for streaming because it is a very light-weight protocol with a very little overhead. It is even a better choice when talking about live streaming, because then the source just wants to send packets continuously and does not care of resending lost packets and things like that.

2.2.2.3 RTP

Real-time Transport Protocol or RTP is defined in RFC 3550 [12] and is running on top of UDP. RTP provides end-to-end network transport functions suitable for applications transmitting real-time data, such as audio, video or simulation data. RTP has no intentions to resend lost packets and it does not guarantee quality-of-service.

For our cause, RTP feels like a possible solution to our packet format but as said before an own format makes it more flexible and less overhead.

2.2.2.4 RTCP

Real-time Transport Control Protocol or RTCP is defined in RFC 3550 [12], the latest version of RTP's RFC. RTCP is RTP's control protocol and provides out-of-band control information for an RTP flow. RTCP task is not to transport any data itself but it periodically sends control packets to participants in a streaming media session. It gathers statistics on a connection and information such as bytes sent, packets sent, lost packets, jitter and round trip delay.

With other words, RTCP provides a media streaming service quality-of-service.

2.2.2.5 Summary

RTP and RTCP sounds really great for our casue of a media streaming service in a distributed overlay network, but it also seems like a little bit overkill to use these two great protocols for our first version of this prototype.

The time it takes to develop an own packet format on top of TCP and UDP probably not going to be as long as the time it takes to investigate RTP and RTCP's design. It also seems that the complexity of these protocols would make the implementation of these rather hard at a first glance.

However, there will for surtain come times when RTP and RTCP will be discussed again in this or another similar project. The most important thing for this project is to see that every thing with the NC and the communication betwenn nodes works, after that in some later version maybe it is time for RTP and RTCP.

2.3 Building a Peer-to-Peer Network

Peer-to-peer networks and peer selection are also two very big topics in the academic world today, mainly because of the increasing of file sharing between people around the world. Large peer-to-peer networks are building up today to distribute files, files that clients in the network want as fast as possible.

In our case we want to distribute real-time data streams to all of the clients in the network, and then the peer selection becomes even more important because a client must get parent-peers that provide a stable flow of information. This peer selection part of the report is explained best in a scenario based way. The scenarios will be explained by text and by some sequence diagrams.

2.3.1 Startup Process

The startup phase is a very critical phase where clients want to join the network. To do that the client needs to know some things about the network, things such as which nodes are its neighbors and which nodes are its peers. This can be done in several ways, one could have a centralized tracker that provides the clients with neighbors or peers as Bram Cohen talked about in [13] or one could use a more distributed way and use a gossip-like solution as they are talking about in [14] and [15].

Let's take a closer look on these two solutions below, but first we must mention that there are two different stages in the startup phase. One early stage when the first *h* clients are connecting to the network and one late stage when there are already more than *h* clients in the network. The two stages are going to be more explained in detail further down in this chapter.

2.3.1.1 Centralized Tracker

A centralized tracker could be a detached application that a client must connect to, to get information about the network. Then the client must know the address of the tracker in some way. Maybe the company that provides the streaming-service makes the address available on the internet or maybe it is hard coded in the user's application.

When the client connects to the tracker there are basically two scenarios that could happen, either the tracker gives the user a bunch of neighbors or a few parent-peers. The first goal for the client is to find some parent-peers that could provide the data stream, and the user will reach its goal in both of the scenarios. So, the question is: Which way is the best?

Let's say that the tracker chooses peers to every new client connecting to the network as shown in Figure 2.9 and also provides old clients with new peers if they need to change some of them. Then the tracker would be heavy loaded, especially when talking about big networks with thousands of clients. Not only heavy loaded in the sense of computations at the tracker, also in the sense of heavy communication to and from the tracker.

That leads us towards the first scenario: send a bunch of neighbors, possible peers, to the client as shown in Figure 2.8. If that is the case the tracker lays over the responsibility of choosing peers to the clients themselves and then minimizes the computational costs at the tracker. The communication with the tracker won't be that heavy either because now the clients don't need to contact the tracker every time they need a new peer; they just choose another one from the list of neighbors.

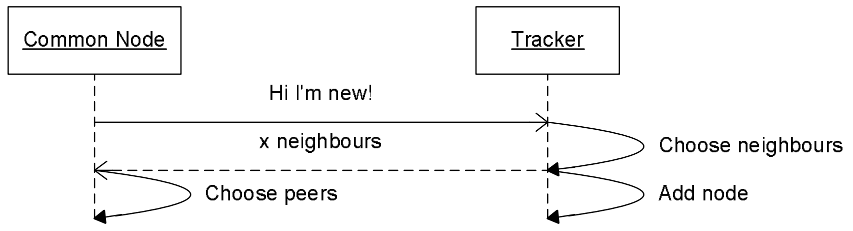


Figure 2.8: A scenario where the tracker provides neighbours to the node.

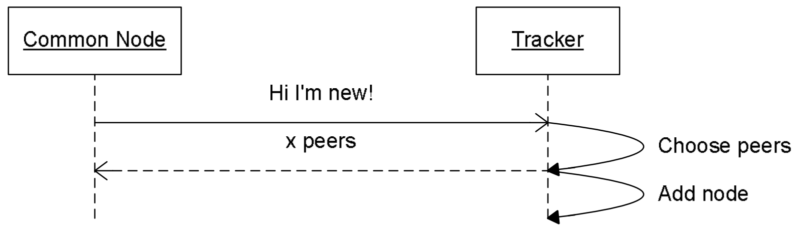


Figure 2.9: A scenario where the tracker provides peers to the node.

2.3.1.2 Distributed Algorithm

A distributed algorithm such as the PRO (Peer-to-peer Receiver-driven Overlay) protocol described in [14] doesn't use a tracker for getting information about the network. Instead it uses gossip through the network to collect information from lots of nodes, and then the client can choose the best parent-peers by itself. But how does a client start gossiping, one can wonder, in some way the new client must have a picture or some kind of view of the network so it know where to start.

2.3.1.3 Early Stage

Every node should have *h* parents, *h* is a fixed number and it is equals to the amount of subparts in a generation (X_1, X_2, …, X_h).

The early stage is not that common for clients because it is only for the *h* first clients and the main source node when connecting to the network. When the first client connects there is no need for a peer selection, because the new client just chooses everybody of its neighbors to be its parent-peers.

When a client, in this stage, gets a child-peer the client must check whether it has that child as a parent or not. If it does not have that neighbor as a parent it chooses that one to be a new parent, this is done because everybody should have *h* parents each and to avoid cycles in the network. When the *h* client connects to the network we are getting a full mesh topology of the network, main source node included.

For better understanding in the early stage please look at Appendix B.1 and B.2. There you will find some examples of nodes joining the network in the early stage.

2.3.1.4 Late Stage

This is the phase where almost every client is connecting to the network and it is a big difference compared to the early stage. When a client connects in this phase there is some kind of topology already built up, so the new client jumps right into it.

But how does the client know which nodes it going connect to? The answer to that is that the client needs some kind of a peer selection mechanism. There are several different mechanisms for peer selection and they will be described in the peer selection process below.

For better understanding in the late stage please look at Appendix C. There you will find an example of a node joining the network in the late stage.

2.3.2 Peer Selection Process

When building up an overlay network one have to think of building it for the right purpose. There are a lot of criteria that influence the chose of a peer selection mechanism.
Criteria such as:

- Low diameter – With a high diameter the stream needs to travel through many, many peers and the packet loss will increase.
- Structured network – If one has a structured network, the peers lies near (geographically) each other and minimizes the delay.
- Non-clustered - A clustered network would bring bottlenecks to the network. If a bottleneck breaks every stream on that cluster will be lost.
- Bandwidth utility - If one want to use a low diameter protocol it can be good to maximize the bandwidth utility to get as many children as possible.

- Delay (latency) – Of course one wants to minimize the delay, so the buffers don't need to be that big.
- Non-innovative messages – We also want to minimize the amount of non-innovative messages, because these messages don't contribute with useful information.
- Tracker load – If using a tracker in big networks it could be a bottleneck, therefore it is important to try to minimize the communication with the tracker and also the computation for the tracker.
- Fairness – Especially important when live media streaming is wanted, because then every node in the network needs to get the information at the same time. Each and every node should be treated similar.

Reach every one of these criteria with one peer selection mechanism is very hard, if not impossible. Anyhow, I will mention and discuss some possible peer selection mechanism below.

2.3.2.1 Random Peer Selection

When choosing peers, does the client or the tracker choose parent-peers, child-peers or both? The most effective way is to choose parent-peers, because then the client is sure that it will get information that it could send along to its future child-peers.

Okay, so now the client has parents that provide it with useful information but should it choose its child-peers now? It might work about okay but it would be better to let they choose parents by themselves because the new client doesn't know if the other client needs a parent or not.

With random peer selection it is very hard to say how it affects all the different criteria, just because it is random. It hardly going to become a structured network and it probably won't become that clustered either. Other criteria like low diameter, non-innovative messages and latency will be hard to minimize with a random peer selection.

Presumably the only two things one can grant is that fairness is provided and the work for the tracker won't reach a work-limit it can't handle.

2.3.2.2 PRO Protocol

As described before the PRO protocol tries to find the best parent-peers and connect to them. But how does one know which nodes are the best ones? The PRO protocol is designed for non-interactive streaming applications and its primary design goal is to maximize delivered bandwidth, so of course available bandwidth is something that contributes to the decision of choosing the best parent-peers.

PRO has two criteria to decide which nodes are the best ones, first, as mentioned, is available bandwidth and second the relative delay. Relative delay between two peers can be estimated with Global Network Positioning (GNP) [16, 17]. The available bandwidth is much harder to estimate, because then one has to make end-to-end measurements. This can be done with some probing technique [18], but it is not scalable because the available bandwidth changes in time; it means that probing must be used periodically.

In this case, a live media streaming network, the main criterion is to maximize the delivered bandwidth because in this project we want a stable flow of information with the bandwidth that the main source uses when streaming.

Okay, so this protocol maximizes the bandwidth utility and minimizes the delay but how about the other criteria mentioned? There is going to be a structured network because of the GNP and therefore there is a risk that it also will be some clusters in the network. There are probably going to be groups (clusters) in the network where nodes lie near to each other geographically. These groups might get only a few weak connections between each other and that is not good, because if a weak connection breaks all information from that cluster will be lost.

In this case it also will become fairly unfair for the nodes when they are connecting to the network. They will be treated differently depending on where in the world they are connecting from and how big bandwidth they can provide.

2.3.2.3 Low Diameter Protocol

Low diameter protocol is just what the name indicates; a protocol for minimize the diameter of the network [19]. First of all there is a tracker with a record of all the clients in the network, there is also a tracker cache. Some of the clients are in the cache and some aren't. When a new client is joining the network by contacting the tracker it chooses *h* parent-peers from the cache. After getting *h* parents the new client is ready for having child-peers and therefore automatically becomes a member of the tracker cache.

When a client, in the cache, has gotten x (a fix number) child-peers it becomes full and is automatically removed from the cache. A full client will be put in the cache again if one or some of the child-peers disconnects.

The number of child-peers a client should have before it gets full is hard to estimate when talking about streaming, because then it is important that the client can provide a stable flow of streaming media to all of its child-peers. In this case it had been nice for the clients to know their own available bandwidth, because then they had known how many child-peers they could provide streaming to.

Another thing is when choosing the parent-peers from the cache, should the clients choose parent-peers randomly or should they try to find the best ones? Probably the easiest thing is to choose randomly as mentioned before, because finding the best parent peers is not easy at all.

We already know that this protocol minimizes the diameter in the network and therefore also minimizes the packet loss, but how about the other criteria? Probably it is not going to be an especially structured network and nothing tells us that it going to be a clustered network either.

The delay is hard to estimate because in this case the underlying network topology is unknown and then we don’t know how the packets traverse, but one thing is for sure and that is that the packets don’t need to traverse especially many hops in the overlying network. Therefore the delay will probably be okay, not minimized but okay.

The other criteria is kind of hard to say something about, maybe the network will provide reasonable fairness if the nodes choose parent-peers from the cache randomly.

2.3.3 Leaving Process

The leaving process is just as it sounds, when clients are leaving the network. Of course there are many things the network must handle when clients leaves. The child peers of that specific client must get a new parent peer as shown in Appendix D and the tracker, if there is one, plus all of the clients’ neighbors must delete the client from their lists. All of these different issues are going to be explained later in this chapter but first we must describe the two different ways a client could leave the network.

The first and most common way is that a client leaves the network gracefully; it decides when to leave and tells everybody that needs to know that it is leaving so they are prepared. The other way, will hopefully not happen too often, is that a client non-graceful leaves the network. A non-graceful leave could be when a computer crashes and can’t keep the connections up and running or it could be when a clients’ ISP has some problem with the internet connection or something like that.

2.3.3.1 Graceful Leave

When a client decides to leave the network it tells all of its peers, both parents and children, and the tracker, if there is one, that it is leaving as shown in Figure 2.10. The tracker needs to know because it is necessary to delete the client from the client list so that if a new client joins it can’t get the leaving client as a neighbor. The parent peers wants to know so they don’t waste time on trying sending more data to that specific child peer. For telling the children there is basically two scenarios.

One idea is that the leaving client chooses a new parent peer to its children; the other idea is that the leaving client just tells the children that it is leaving and lets the children choose a new parent by themselves.

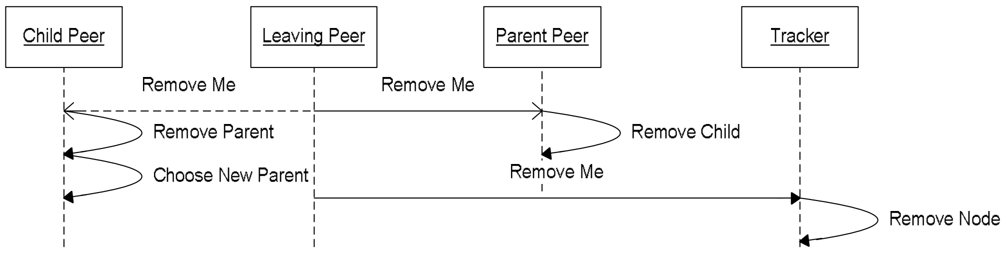


Figure 2.10: A possible scenario of a graceful leave.

Probably the best thing is to let the child-peer choose a new parent-peer; the peer selection mechanism takes care of how the child-peer is going to that.

2.3.3.2 Non-Graceful Leave

When a client leaves the network non-graceful all of that specific client's peers will notice it after some time as shown in Figure 2.11. Figure 2.11 also shows that all the peers, both child and parent, and the tracker are removing the leaving client from their internal lists. The clients' children will notice it by not getting anymore data packets and the clients' parents will notice it by not getting anymore "keep-alive" messages from the client if something like "keep-alive" messages has been implemented.

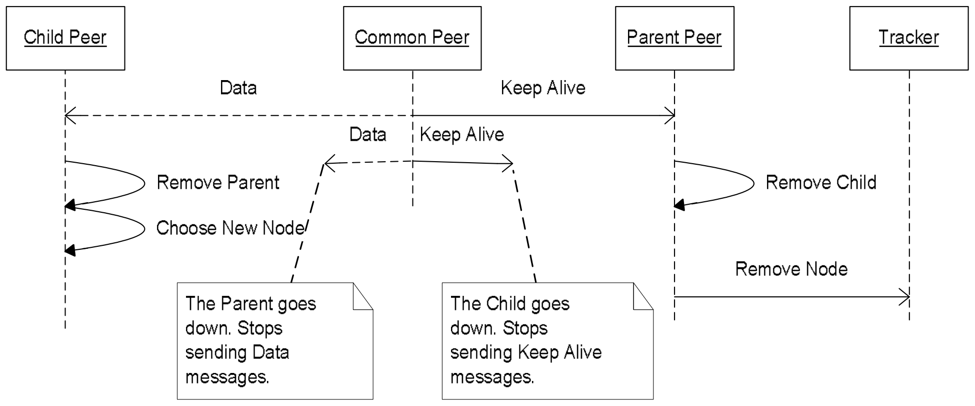


Figure 2.11: A possible scenario of a non-graceful leave.

The hard thing is now to decide when a peer should be declared "dead" and removed from the lists. There is a chance that the packets will start dropping in again after a while when the peer has recovered. When the child-peer of the leaving client has declared it dead it needs to choose a new parent-peer using the peer selection mechanism.

2.3.4 Bad or Change of Condition

Through this chapter about building peer-to-peer networks we have talked about the startup process and the different peer selection mechanisms, the thing left now is to talk about when peers are bad parents or when good peers becomes bad.

To understand when a peer is bad we need to define bad for a peer in a media streaming distributed network. There are several things that can make a parent-peer bad; to get an overview I list them here:

- Non-innovative messages - Too many non-innovative messages from the same parent-peer is not good. The child-peer needs to get lots of innovative messages to be able to decode the encoded data.
- Bandwidth - Does not get the required bandwidth from a certain parent-peer, the live streaming becomes just streaming and not live. It will also influence this node's child-peers and the delay will just grow and grow.
- Instability - When the connection, between a parent-peer and a child-peer, jumps up and down, the child-peer is in trouble and needs a new and better parent-peer. It is very important that the child-peers are getting a stable flow of information to keep the streaming live.

The chance of getting a bad parent-peer depends on which peer selection mechanism is used and the amount of bad luck. Before choosing parent-peer you don't know if a node is sending non-innovative packets or not, so in that sense it is all about bad luck.

If a client using random peer selection or choosing randomly from the tracker cache when using the low diameter protocol, the client doesn't know if the parent-peer's available bandwidth is enough for streaming. If it instead uses some peer selection mechanism that knows the available bandwidth of the neighbors it will not get the same problem. Not at first it won't, but you never know what is going to happen later on. There might happen something with the ISP or maybe the parent-peer is starting some other application that steals bandwidth.

Instability is maybe something one can check before choosing parent-peer with help from a tool like pchar [20] or a similar tool, but that is out of the scope of the project. If instability occur with some parent-peer just try to choose a new one.

The hard thing is to decide when a parent-peer is instable; how often it jumps up and down within a timeframe. It is also hard to decide how many non-innovative messages are too many, before the child-peer must choose a new parent.

The easiest thing would be to trigger the peer selection mechanism directly when a parent-peer turns bad, but probably it wouldn't be the most effective and best way to do. Too many changes in the network at the same time would probably make the traffic load much higher and interrupt the media streaming.

A more effective way would to have some kind of non-innovative message-counter; when a limit is reached, change parent-peer. Another way would be to have some timers as shown in Appendix E that triggers the peer selection mechanism, but in what period of time would be feasible?

These things are almost impossible to decide before one has implemented a prototype with a working peer selection mechanism, and then massive tests will decide how to work around these problems.

3. Design

This project was a group work where different parts where divided up among the group members. In the design chapter there going to be two big different parts; an overview of the whole system where I am not going into any details and a part, chapter 3.2, where I will explain in more detail what I have been doing.

3.1 System Overview

One could say that the system consists of 7 different parts:

- simulation program,
- prototype application,
- upper interface,
- logic layer ("black box"),
- lower interface,
- NS-2 packet transmission mechanism,
- prototype packet transmission part.

The two applications are both using the upper interface, logic layer and the lower interface but then of course the simulation application uses the NS-2 packet transmission mechanism and the prototype application uses the prototype packet transmission part. This is explained better with Figure 3.1.

The system design shown in Figure 3.1 is divided up into three layers, one upper layer (shown in yellow and blue in Figure 3.1) where the actual application will be implemented, one core logic layer (shown in orange in Figure 3.1) where all the logic will be implemented such as NC and peer selection, and finally one lower layer (shown in yellow and blue in Figure 3.1) where the transmission part will be implemented. This three layers is connected by and communicates through two interfaces (shown in green in Figure 3.1), the upper layer interface and the lower layer interface.

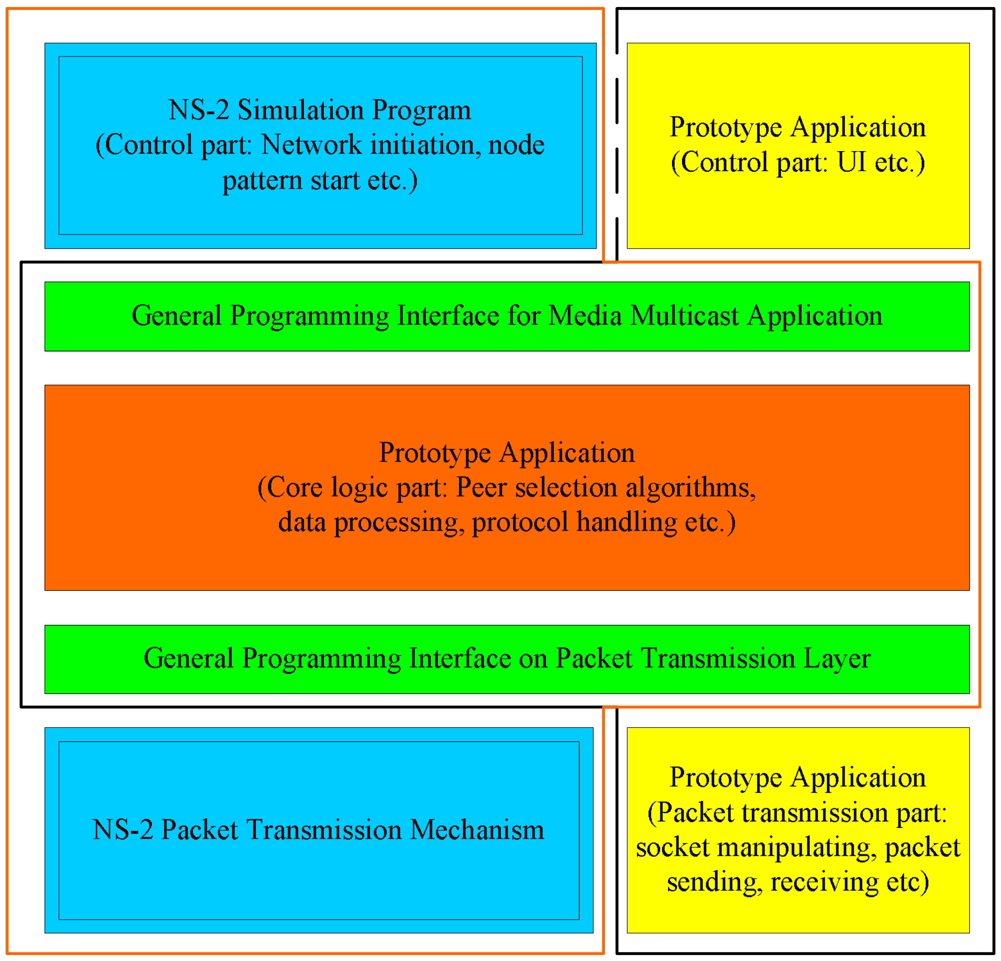

NS-2 Simulation Program
(Control part: Network initiation, node pattern start etc.)
Prototype Application
(Control part: UI etc.)
General Programming Interface for Media Multicast Application
Prototype Application
(Core logic part: Peer selection algorithms, data processing, protocol handling etc.)
General Programming Interface on Packet Transmission Layer
NS-2 Packet Transmission Mechanism
Prototype Application
(Packet transmission part: socket manipulating, packet sending, receiving etc)

Figure 3.1: Project Description

The upper interface, lower interface and the logic layer will be explained below, explained in a common sense not much into details because those were done in a group. The prototype application and the packet transmission part, shown in yellow in Figure 3.1, on the other hand will get a subchapter each and there they will be explained with more details and depth because that was developed by me.

3.1.1 Logic Layer

The logic layer is the heart of the system. It is here the peer selection, data processing and network coding takes place among other important things. Important things in a project like this become often secret because the companies will not reveal it to the competitors. Therefore the logic layer got the nickname “the black box”.

Even though I can not show any details of the logic layer one should know that this is the heart of the prototype and it is needed for building up a media streaming distribution network. To access the logic layer one should use the two interfaces; they are the way in and out of logic layer. The interfaces will be described below.

3.1.2 Upper Interface

The upper interface exists to make a connection between the actual prototype application and the logic layer. The connection is made by two part-interfaces, one for the information flow from the application through the interface to the logic layer and one for the other way around. The interface that the application uses to call the logic layer is called NcControlInterface and the one that the logic layer uses to call the application is called NcControlCallbackInterface as shown in Figure 3.2.

In Figure 3.2 generalization is shown with a solid line and a fat triangular arrow from a subclass (such as PrototypeApplication and NcController) to a superclass (such as NcControlCallbackInterface and NcControlInterface), this means that a subclass extends a superclass and it also implies inheritance from the superclass to subclass.

Figure 3.2 also shows composition, also known as composite aggregation, by an association line and a filled diamond, which means that an instance of the part (such as NcControlInterface or NcControlCallbackInterface) belongs to only one composite instance (such as PrototypeApplication or NcController) at a time.

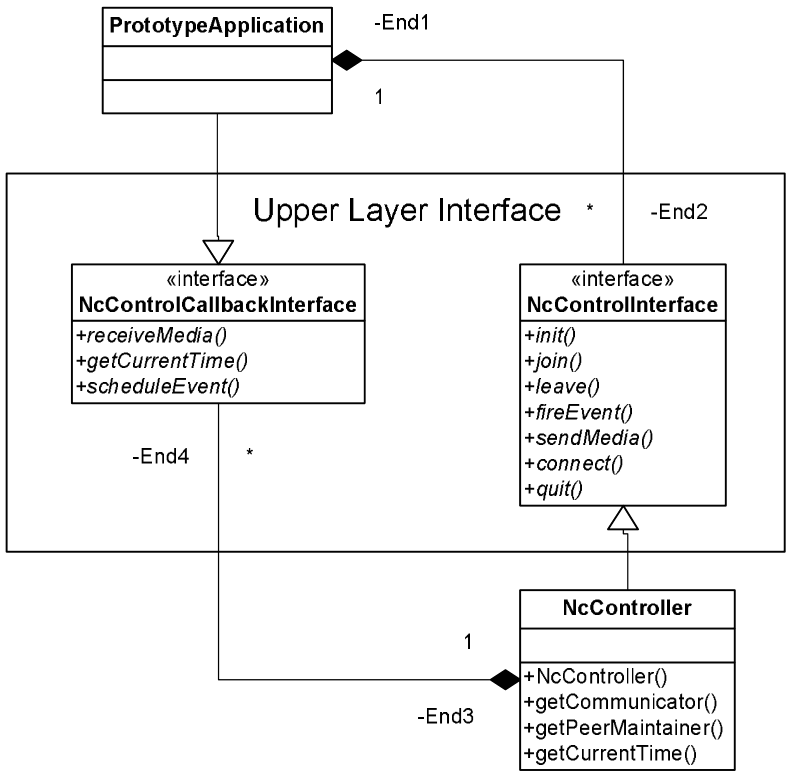


Figure 3.2: The upper layer interface with its connections.

3.1.3 Lower Interface

The lower interface exists to make a connection between the logic layer and the prototypes packet transmission part. This connection is also made by two part-interfaces, one for the information flow from the logic layer through the interface to the packet transmission part and one for the other way around. The one that the logic layer uses to call the packet transmission part is called NcCommunicationInterface and the one that the packet transmission part uses to call the application is called NcCommunicationCallbackInterface as illustrated in Figure 3.3.

Figure 3.3 shows that a Communicator extends a NcCommunicationCallbackInterface to take care of the receiving packets from the underlying transmission part and it also has an instance of the NcCommunicationInterface whenever it needs to send along a packet to the transmission part.

Figure 3.3 also shows that the PrototypeApplication extends the NcCommunicationInterface to be able to send the packet coming from the logic layer.

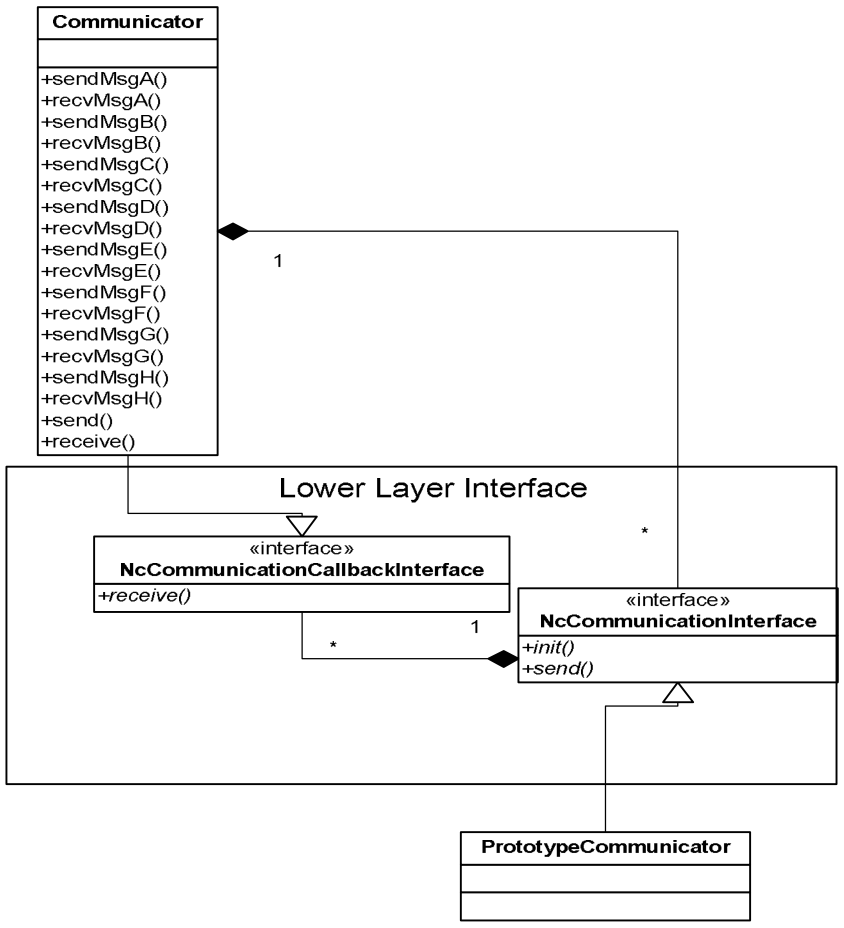


Figure 3.3: The lower layer interface with its connections.

3.2 Prototype Application

The prototype application is the main application where all starts. The application initiates a logic layer and calls one or more of the upper layer interface functions in an appropriate order. It also initiates a communicator for the packet transmission part, a SourceCommunicator or a PeerCommunicator, and the NodeProperties class. The NodeProperties class is for parse out neighbours from a configuration file and save it in two lists; the childList and the parentList. This is done before initiating the logic layer because the logic layer needs a communicator to work. There are two different types of a prototype: SourcePrototype and PeerPrototype. Of course a SourceCommunicator belongs to a SourcePrototype and a PeerCommunicator belongs to a PeerPrototype.

The prototype application is defined by three classes (as shown in Figure 3.4) such as Main, App and NodeProperties where NodeProperties is just a help-class. If interested on how this design works and what the classes do please read Chapter 4.3.1.

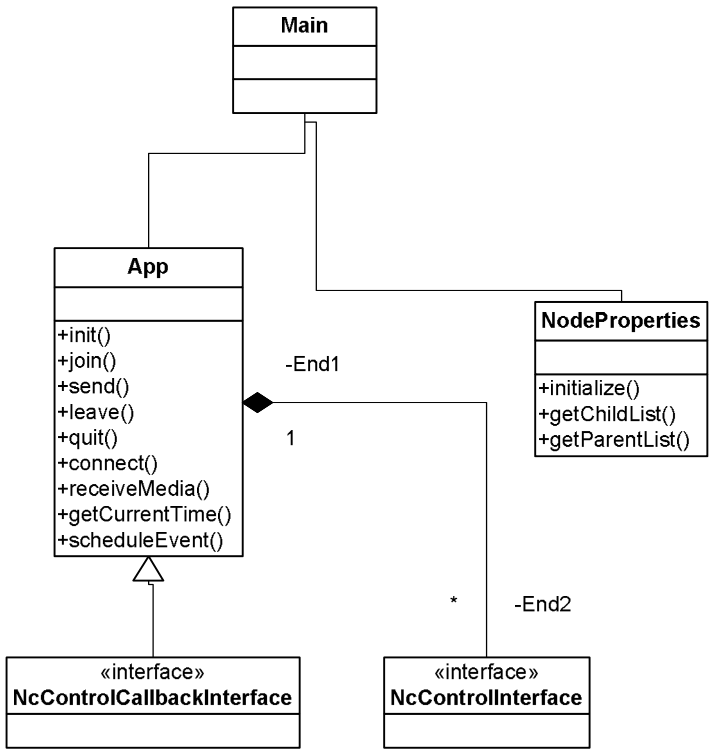


Figure 3.4: A stripped version of the prototype application design.

3.2.1 SourcePrototype

One SourcePrototype is needed in every media streaming network, it is the source that is actual does starts the sending of data. It is recommended that it is one source in the network before the first user joins. The SourcePrototype needs to be connected with a streaming service to become a real streaming source but that is out of the scope of this project.

Our SourcePrototype will instead divide up a file, the substance of the file does not matter, into generations and send them out on the network.

3.2.2 PeerPrototype

A PeerPrototype is the application a user needs to connect to the streaming network. The design is the same as the SourcePrototype, it calls the same functions of the upper layer interface. The difference you will find when the applications are using the upper layer interface to call the logic layer, is that the applications tells the logic layer if it is a peer or a source so it knows how it should react.

In the end a PeerPrototype and a SourcePrototype are doing the same, which is sending out generations of information to the neighbours. The difference is that one has to start the streaming and that is the work of the SourcePrototype.

3.3 Packet Transmission Part

This is the part where the actual sending and receiving of packets takes place. The raw packets with information come from the logic layer through the lower interface with a destination address. The packet transmission part's task is to create a socket connection, either a TCP socket connection or a UDP socket connection depending on the type of the message. When a connection is made the message should be sent to the destination and that is about it.

Of course some server sockets is also necessary, one must be prepared for incoming messages, both TCP and UDP messages. When receiving a packet this part makes a "call back" to the logic layer through the lower interface. Then it is up to the logic layer what to do with it, probably it first will be decoded.

To ease some things up server sockets, sockets and datagram sockets are going to be used. Server sockets are going to be multithreaded to optimize the performance of the handling of incoming messages. For a better understanding look at Figure 3.5 that shows a class-diagram of the packet transmission part.

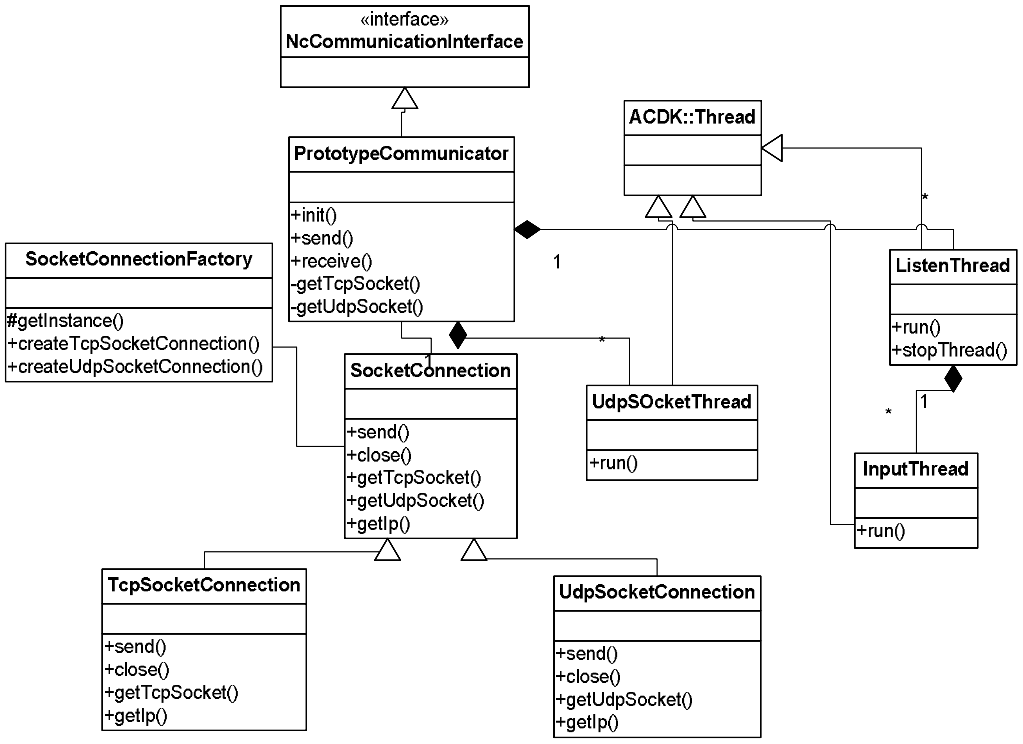


Figure 3.5: A stripped version of the packet transmission part design.

The class-diagram in Figure 3.5 contains eight classes such as PrototypeCommunicator, SocketConnectionFactory, SocketConnection, TcpSocketConnection, UdpSocketConnection, UdpSocketThread, ListenThread and InputThread where the PrototypeCommunicator is the heart of this packet transmission part. If interested on how this design works and what the different classes do please read Chapter 4.3.2.

4. Implementation

Before starting implementing I needed to study some C++, because it was rather new to me. First of all I started to learn about different data types, like strings, maps, queues, iterators and much more [21]. After that I went on reading about memory management, pointers and references, and then I got a little bit more confident in the C++ language.

I knew that the packet transmission part was going to be the hardest one, with the threading and the socket programming. So, I started looking for some external libraries that I could use, that included thread handling and socket wrapping. I read about and tried some libraries that I found, libraries like: datareel [22] and ACDK [23].

After some investigation I finally decided to go with ACDK because it was big enough (maybe little too big) and it had a build in script language that looked just like Java. I am much more convenient with using Java when I am implementing, so that made my decision easier. It also had a nice API that could help me find what I was looking for. There will be more information about ACDK below.

4.1 Artefaktur Component Development Kit – ACDK

ACDK is a big development framework that has C++ as its core implementation language [23]. It provides a very nice build in scripting language that is similar to Java as shown in Appendix F and ACDK C++ objects can be used directly via scripting. Apart from the ACDK scripting, you could use all features of C++ because ACDK is implemented in pure C++. That includes using C/C++ libraries, allocating objects on stack and using templates and so on.

ACDK is a framework with enhanced memory management features like garbage collection and debugging features. It also provides productive packages similar to JDK, packages like acdk::lang, acdk::net, acdk::io and acdk::util and so on.

One could say that ACDK is a combination of Java and C++, where Roger Rene Kommer [31] has tried to combine the advantages of both of the language.

Other things that are great with ACDK is that it provides multithreading and a acdk::net-library, which makes the implementation of a prototype like ours much easier.

To get a little bit more insight of how ACDK look likes, please take a look at Appendix F. There you will find a “Hello World!” example, written with ADCK’s build in scripting language.

4.2 Environment Details

Most of the time during the implementation I used a server at the office:

- Pentium dual core processor running at 3.2 GHz
- 2.0 GB internal memory
- Red Hat 9.0 [24] was working as operating system
- As editor Eclipse [25] with CDT (a plug-in for managing C/C++ projects [26]) was used.

Some of the time though I used my own Laptop:

- Pentium 4 running at 2.2 GHz
- 512 MB internal memory
- Fedora Core 3 as operating system [27]
- Eclipse as an editor.

One could say that Fedora is an updated version of the last version of Red Hat, meaning Red Hat 9.0.

4.3 Description

Of course my part of the system implementation was the prototype application and the packet transmission part as described in the design chapter, but also helping out implementing the logic layer. In the logic layer my task was to implement the whole message structure of the system, a message structure that we had come up with after the research in the beginning of the project. I can not show a detailed design of the structure and not describe it into a detailed level because restrictions from the company.

The things I can say are that it is eight different messages, with different information and purpose. Some of these messages are for building up the network and some of them are for the media streaming service. The hardest thing with this task was the bit mathematics when needed to encode and decode messages.

The implementation of the message structure was going on constantly, mostly because of the changing in the design but also when we decided to keep ACDK out of the logic layer completely I had to make some heavy changes in the code. The decision to keep ACDK out of the logic layer were based on that we wanted the logic layer (the heart) to be as clean and fast as possible.

4.3.1 Prototype Application

The implementation of the prototype application for this first version was rather basic so the major goal in implementing the prototype was to get the application to use the upper interface in the right way. We wanted the first version to be basic mostly because the logic layer was not completely implemented; it had no peer selection mechanism and no tracker. If the logic layer would be completely implemented, then the application just had to call the join-function in the upper layer to join and build up the network. Now, when not having a peer selection mechanism and a tracker it made the prototype little bit messier.

Instead of using the join-function I have used the connect function that connects to the neighbor to which the prototype sends along. Of course a user has more than one neighbor and therefore the prototype needs to call the connect function many times instead of just one call to the join function. Another drawback with this connect function is that the neighbor a user is connecting to, must also use the connect function and connect back to the user. This means that the neighbors need to be hard coded somewhere in the prototype.

Instead of hard coding the neighbors in the prototype I did it in configuration files, one configuration file to each of the users. Then I had the prototype application loop through to read and parse the configuration file so it could use the connect function. The name of the configuration file needs to be typed in as an argument when starting the application.

The prototype application was made in two different versions, one for a regular user (peer) and one for a source. Not much difference between these two prototypes because the source acts almost like a regular user and vice versa. The only thing that differs is that the source prototype must be able to start sending data not only forward incoming data.

4.3.2 Packet Transmission Part

The packet transmission part starts to act directly when the prototype application initiates a PrototypeCommunicator.

The first thing the PrototypeCommunicator does is that it starts the ListenThread and the UdpSocketThread. It starts these threads in the beginning because one wants to be sure of that the prototype is ready for incoming messages directly so that no messages will be lost.

The ListenThread is a server socket thread that listens for incoming TCP connections, if getting one it starts an InputThread to handle the request. This makes the ListenThread non-blocking and prepared for many incoming connections at the same time.

The InputThread handles the incoming packet, and when it has collected all bytes it makes a callback to the PrototypeCommunicator’s receive function. The same thing happens when the UdpSocketThread gets an UDP-packet; the UdpSocketThread makes a callback to the PrototypeCommunicator’s receive function.

When the receive function in PrototypeCommunicator has been called, as it will be every time the threads has collected a new message, the PrototypeCommunicator will also make a callback but this time to the logic layer through the lower layer interface.

Apart from receiving packets and making callbacks the PrototypeCommunicator waits for instructions from the lower layer interface, instructions about sending packets to a specific destination. The logic layer calls the PrototypCommunicator's send functions through the lower layer interface and it sends along the data, destination address and the destination port.

From this point the PrototypeCommunicator takes over the work by creating a SocketConnection, either an UdpSocketConnection or a TcpSocketConnection depending on the message type.

After that the PrototypeCommunicator calls the send function in the UdpSocketConnection or the TcpSocketConnection class and the SocketConnection sends the packet out on the socket towards the destination.

Finally the PrototypeCommunicator calls the close function in the SocketConnection class that closes the socket for further sending.

That is pretty much what the packet transmission part does; communicating with the neighbors in the overlay network and also communicating with the logic layer through lower layer interface.

4.4 Implementation Summary

4.4.1 Overview

In this Section the implementation of the whole prototype should be summarized, summarized in a way that the readers of this thesis will understand what happens in every step of the logic in the prototype. Every step means from the beginning when starting the application through the steps in the logic layer to the steps in the packet transmission part and then all the way back the other way. This will also be showed by a diagram.

Apart from the step-to-step tutorial of the prototype I am going to discuss a little bit about the chosen solution of the socket and thread problem.

4.4.2 Step-to-Step Tutorial

1. The PrototypeApplication calls the Upper Layer Interface's init function.
2. Logic Layer initializes a PrototypeCommunicator through the Lower Layer Interface.
3. The Communicator creates a ListenThread (TCP) and a UdpSocketThread.
4. The application makes a join(); wants to join the network.
5. Logic Layer sending a message to the Tracker through the Communicator.
6. The Communicator creates a TcpSocketConnection to the Tracker and sends the message to it.
7. ListenThread receives a response from Tracker and creates an InputThread.
8. The InputThread makes a callback to the Logic Layer through the Communicator.

9. The Logic Layer decodes the message, finds out that it is a message from the Tracker.
10. Start immediate the Peer Selection mechanism; choose parents.
11. Logic Layer sends messages to the parents through the Communicator.
12. The Communicator creates TcpSocketConnections to the parents and sends them a message.
13. ListenThread receives a message from the parents and starts several InputThreads.
14. The InputThreads makes callback to the Logic Layer.
15. The Logic Layer decodes the messages and finds out that it is a response from the parents.
16. Logic Layer sends messages to the parents to inform that it is ready for real data packets.
17. The Communicator creates UdpSocketConnections and sends along the messages to the parents.
18. The UdpSocketThread starts to receive lots and lots of data packets and makes callbacks to the Logic Layer.
19. The Logic Layer decodes the data packets and sends the pure media information to the application.
20. The Logic Layer also encodes the incoming information with NC and sends along the combination of the incoming packets to its children.
21. This continues until the application leaves the network by calling leave() or quit() or crashes.

It is recommended to watch Figure 4.1 to get a deeper understanding of what is happening.

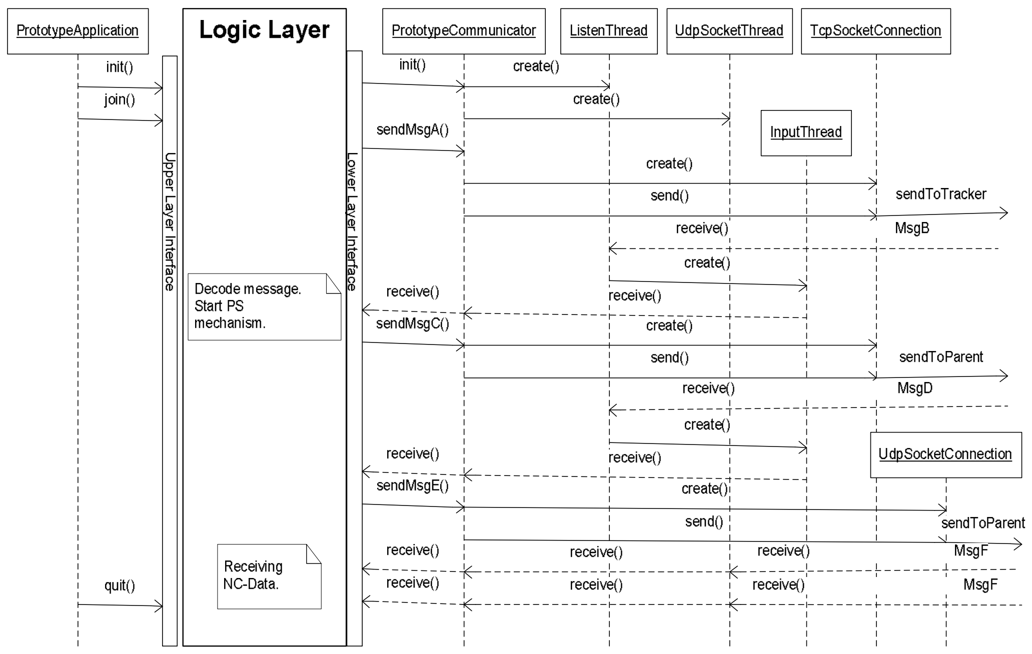


Figure 4.1: Step-by-step when a user is joining the network.

Under this time the ListenThread may receive request for information from future children, of course the logic layer must send a response. If it sends yes, the child will send another request over UDP. The UdpSocketThread receives it and makes a callback to the logic layer, and then the logic layer will start sending combined NC data packets to that specific child. This scenario is not shown in the figure above.

Another thing that is not showed above is when the user chooses to leave or quit, then a special message will be sent to the tracker, parents and to all the children.

4.4.3 Discussion

Every time a message should be sent over TCP the Communicator creates a new TcpocketConnection, wouldn’t it be better to use some of the old TcpSocketConnections (sockets)? Maybe in some cases, but the main thing that made this design to what it is was that if a node has a lot of peers there will be many TCP sockets up all the time and that is rather demanding for the network.

It would be especially demanding for the tracker that communicates with all the nodes in the network.

Okay so it would be demanding, but a drawback of this design is that if a node gets peers that lay nowhere near it (geographical). Then creating a TCP connection would probably take a long time.

In this scenario it would be nice to keep the TCP connection up if the users need to communicate with each other again.

5. Evaluation

This part of the thesis presents results of evaluation of the implemented prototype. Unfortunately it is a little bit hard because of the “black box”, but some measurements will be displayed. Measurements such as: memory consumption, correctness and CPU usage in idle state and working state. Because of the “black box” this chapter is going to be intended on the two parts described in chapter 4, the prototype application and the packet transmission part. For the prototype application and the transmission part the evaluation would mostly be about the integration of these two parts and the “black box”.

5.1 Test-bed Platform

5.1.1 A Peer Prototype

Most of the computers that are running as peers are Pentium(R) D 2.8 GHz with 1.00 GB RAM. The operating system is Microsoft Windows XP SP2.

To be able to run the PeerPrototype, Microsoft Virtual PC [28] was installed and on that specific virtual PC the operating system openSUSE 10.0[29] were running. This was needed because the prototype is build for Linux in this first version, it will be translated to work on Windows in later versions.

One of these computers did not need to install Microsoft Virtual PC because openSUSE 10.0 were already installed directly on the computer.

In the test phase a laptop was also used. A Pentium 4 2.2 GHz with 512 MB RAM. The operating system is Fedora Core 3.0. In addition to this, ACDK was installed on every workstation.

The computers are connected in a little office network through a Gigabit switch and every computer (including the source) in the network is equipped with a Gigabit network interface.

5.1.2 A Source Prototype

The source prototype is running on a Pentium dual core processor running at 3.2 GHz with 2.00 GB internal memory. The operating system is Red Hat 9.0.

Of course ACDK was installed on this computer to, because every prototype peer or source needs it to work.

5.2 Correctness

To test the correctness of this prototype several test were done, test with different numbers of computers in the overlay network. All the tests included the source sending a video-file to the receiver/receivers, and then the receiver/receivers checked whether the amount of bytes sent were equal to the bytes received. For further correctness checking the receiver/receivers also played the video-file in a media player.

5.2.1 One receiver

First of all I tested the prototype with one main source and just one receiver (see Figure 5.1). This means that it is a test in the early stage (see chapter 2.3.1.3). Because of that this test is not testing network coding but it tests the encode/decode-part of the message structure and the packet transmission part.

The file sent from the source: germancoastguard.mpg, 2 576 388 Bytes.

Source	Peer
Bytes sent:	Bytes received:
2 576 388	2 576 388

Table 5.1: Shows the result of the first basic test.

Everything worked out fine and the bytes sent was equal to the bytes received. Every byte was in order and the receiver could watch the video-clip without problems.

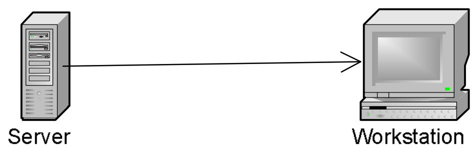


Figure 5.1: A topology with one source and one receiver.

5.2.2 Two receivers

This test included one main source and two receivers (peers), in other words an overlay network with three nodes. Worth mention is that it was a full mesh topology (see Figure 5.2) in this test case and that means that the network coding part and some other part in the logic layer were tested because the peers were not only receiving but also they helped distribute the video-clip to the other peer.

The file sent from the source: germancoastguard.mpg, 2 576 388 Bytes.

Source	Peer #1	Peer #2
Bytes sent:	Bytes received:	Bytes received:
2 576 388	2 458 988	2 457 388
2 576 388	2 476 388	2 437 988
2 576 388	2 459 588	2 458 788

Table 5.2: Shows the result of the second test.

This time everything worked out okay, the receivers did almost get the whole video-clip, just over 95 %. It is really hard to say what the problem is because it depends on so many things in the logic layer. At this time the logic layer is being tuned and therefore, before it is finished, it may lead to some issues like this.

One could think that the last 5 % is packet loss in the network, but in this office the computers are running on a Gigabit LAN and everyone is connected to each other through the Gigabit switch so probably the packet loss is not the problem.

The video-clip worked as it should; there was only a small amount of interrupts in the picture.

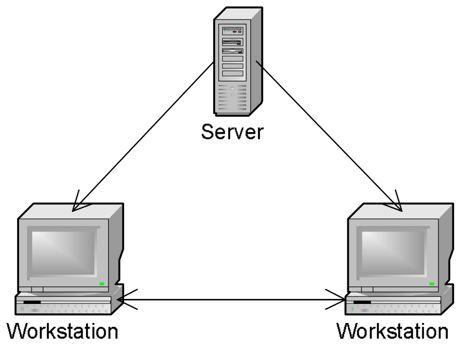


Figure 5.2: A full mesh topology, every peer has two parents.

5.2.3 Five Receivers

This test included one main source and five receivers (peers), in other words an overlay network with six nodes. The topology of the overlay network is randomized. One generation is divided up into 3 pieces (h = 3) and that means that all of the five peers must have three parents and some random number of children. With this configuration it does not become a full mesh topology (see Figure 5.3) and then sets more pressure on the logic layer and especially the network coding part.

The file sent from the source: germancoastguard.mpg, 2 576 388 Bytes.

Source	Peer #1	Peer #2
Bytes sent:	Bytes received:	Bytes received:
2 576 388	2 368 988	2 367 988
2 576 388	2 278 788	2 358 588
2 576 388	2 436 388	2 296 988
Peer #3	**Peer #4**	**Peer #5**
Bytes received:	Bytes received:	Bytes received:
2 368 988	2 368 988	2 368 988
2 456 588	2 356 788	2 288 688
2 288 988	2 458 388	2 478 988

Table 5.3: Shows the result of the third test.

This time with five receivers it worked out okay, a little bit worse than the times before. Almost 92 % of the video-clip arrived at the receivers; one thing that might seem a little bit strange is that in the first test all of the receivers but one got the same amount of bytes.

One thing that might be worth mention is that in this topology not every receiver has the same distance to the main source, that will put the NC to the test and it was not 100 % but it went okay for being the first version.

The vide-clip worked pretty okay, some more interrupts than the last test but okay.

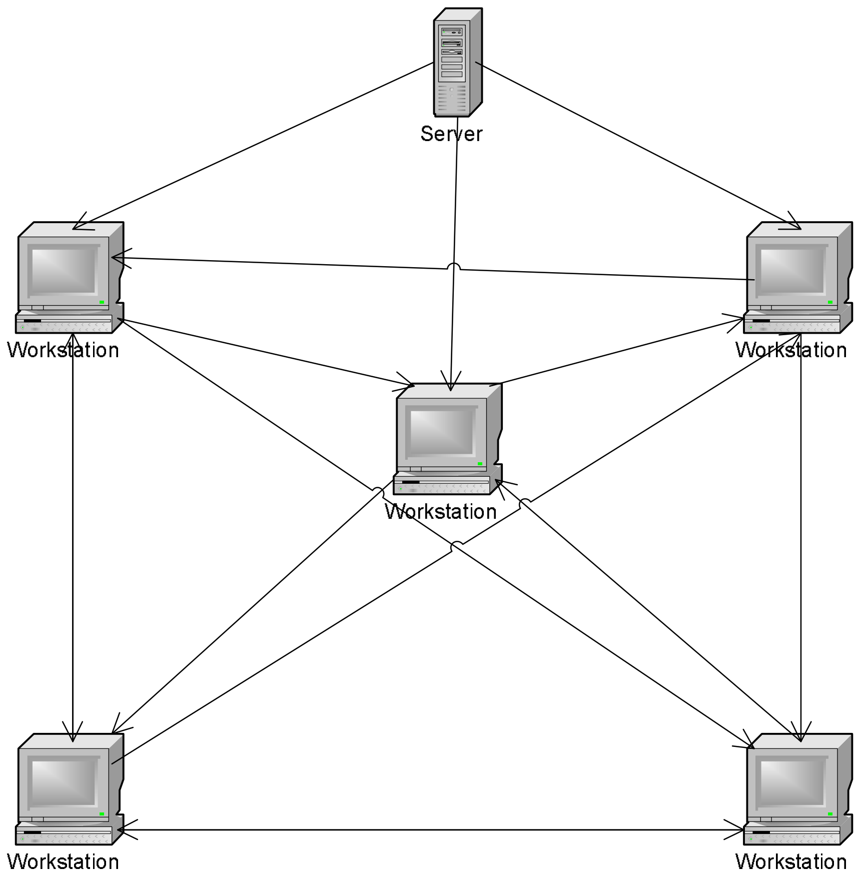


Figure 5.3: A topology where every peer has three parents.

Another similar test was done, but this time is h equals to five. This means a full mesh topology (see Figure 5.4) in an overlay network with six nodes where one node is the main source node. This was done because it is interesting to see whether the result is closer to what it should be than the last test.

The file sent from the source: germancoastguard.mpg, 2 576 388 Bytes.

Source	Peer #1	Peer #2
Bytes sent:	Bytes received:	Bytes received:
2 576 388	1 274 388	2 267 388
2 576 388	1 880 788	1 988 788
2 576 388	1 534 388	1 566 388
Peer #3	**Peer #4**	**Peer #5**
Bytes received:	Bytes received:	Bytes received:
2 930 388	1 869 388	1 286 388
2 043 988	1 505 588	1 877 188
1 566 388	1 566 388	1 545 388

Table 5.4: Shows the result of the last test.

This result was not good at all, a little bit too random and the receivers did not get much data at all. One thing that probably contributed to the result was the full mesh topology. In a full mesh topology there are lots of cycles and in this test, with five peers, the cycles and the non-innovative packets took over.

Since the logic layer is being tuned at this point it is hard to configure the network right for this particular topology. Some of the problems could depend on the configuration of the network and when getting information from the tuning I think the same tests will give better results.

Figure 5.4 illustrates the randomized topology with five workstations and a server as main source node all connected with each other in a full mesh topology.

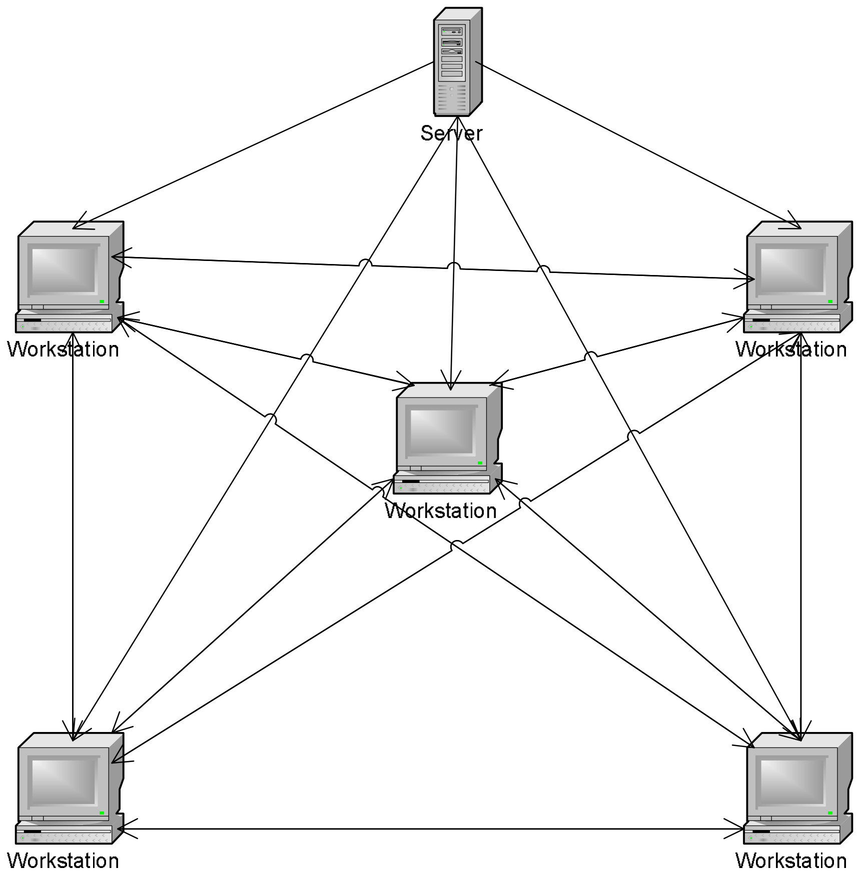


Figure 5.4: A full mesh topology, every peer has five parents.

5.2.4 Correctness Summary

After some investigation of the result one could say that the packet transmission part works exactly as it should, sends and receives all the packets. Still there is something wrong because the result were not 100 % but it were okay and a few interrupts on a video streaming service is acceptable.

One should know that the logic layer is big and complex so it is really hard to track down the issues, actually bug searching and tuning the logic layer might be another master thesis project.

Another problem was that the same test did not give the same result over and over again, it felt kind of random. That is not a good thing but this problem probably arises from the other problem where the receivers did not get the whole video-clip.

Different configuration gave of course different results but it is hard to see which configuration is the best one for this prototype. Huge testing needs to be done on the logic layer to get some real answers, preferably with some good network simulator. Actually this is already prepared by my group mates; the simulation of the logic layer starts whenever the company decides to.

Apart from the problems I must say that this first version of the prototype was okay for this first version, the results were good but too random. The prototype worked somewhat better than the project group expected.

5.3 Performance

5.3.1 Memory Consumption

When the first test was running the memory consumption increased over time; somewhere there was a memory leak. The memory leak was not to be found in the prototype application neither the packet transmission part because these two parts uses ACDK and ACDK has a build in garbage collector that takes care of that.

After some investigation the memory leaks were found, it showed to be more than one. Some of the leaks were found in my message structure, were I had used objects and forgot to delete them after. This is a rather common problem when using C/C++. The leaks were filled and the tests could continue.

After that the tests showed that the memory consumption differed a lot between different operating system. The computers with openSUSE had memory consumption on 15.5 MB and the computer with Fedora had 31.4 MB and last computer with Red Hat had memory consumption on 33.1 MB.

The overall memory consumption seems okay; ~30 MB is not that much for today’s computers.

5.3.2 CPU Usage

The CPU usage of the computers differed a lot, most because of the virtual PCs. This solution with the virtual PCs is not the best one, but it is convenient and easy to use.

However the virtual PCs used ~30 % of the CPU and that was approximately 10 times worse than a similar computer running openSUSE directly. This computer used only ~3 % (80 MHz) of the CPU and that is a really surprisingly and good result. The last workstation, the laptop, with Fedora used ~8 % (176 MHz) of the CPU which is a okay result.

The overall CPU usage seems okay; ~80-180 MHz is not too much for today’s computers.

5.4 Integration with Logic Layer

5.4.1 Prototype Application

The integration with the logic layer is really easy and smooth for the prototype application. The only thing the application needs to do is to call the init function and after that call the join function or the connect function.

On the other hand no peer selection mechanism was implemented in this version and that made it messier, as told before (see ch. 4.3.1). Still it is a smooth integration, especially because of the nice design of the upper layer interface and the complexity of the logic layer that takes care of the most things.

The smooth integration makes time for implementing a lot of features to the application; probably the next version will have a nice GUI. Worth mention is that probably the integration with a media player (in a later version) will take more time and effort for the programmers.

5.4.2 Packet Transmission Part

The integration with the logic layer is exactly as it was for the prototype application, meaning smooth and easy. The Packet transmission part must be ready to send packets whenever the logic layer says so and also it must be ready to receive messages, both over TCP and UDP.

The design of the lower layer interface is very general, as is the upper layer interface, and that is great for integration purpose. If someone wants to make an own packet transmission part and not working on the existing version that would be no problem. Just plug it in. The only thing it has to think of is to handle the send request from the logic layer and making callbacks when receiving messages.

The same pertain to the prototype application where the general upper layer interface makes it easy test different type of applications. Just plug it in.

A drawback to have the interfaces so general is that the logic layer becomes very big and complex. As more general the interfaces becomes as more complex the logic layer gets. Still it makes it flexible and portable.

6. Conclusions & Future Works

During this thesis project I have researched, designed, implemented and evaluated a prototype of a media streaming distributed network with network coding. This chapter summarizes the work and provides some conclusions.

6.1 Research Areas

This section includes conclusions about the three big research areas: Network Coding (NC), Data Processing and Peer Selection.

After researching about NC it is easy to understand why that is such a big topic in the academic and the industry world today. It is a very popular technique that optimizes the transmission of data through an overlay network.

The hard thing with NC is to know exactly how it should be used in a network with certain needs and rules or how to configure the network to get out the most of the NC's benefits. With configure I mean which peer selection mechanism is going to be used and which transport protocol is going to be used and so on.

Probably NC will continue to be a big research topic and more communication application will use the technique in the future.

From the data processing part we choose to make our own packet format running on top of TCP and UDP. I think it was a wise chose, the tests and results showed me that the message structure and the packet transmission part worked really well.

I also learned that there are lots of things to think about when transfer data over an overlay network. Things such as: delay, loss, jitter and so on. It is real complex to take care of all of these issues and for that it might be a good idea to use a protocol like RTP in the future that will help you with that.

Peer selection is a very interesting and huge topic. There are no books that say how you should design your peer-to-peer network because there are so many different scenarios that may lead you in different directions.

In this thesis when all nodes in the peer-to-peer network should help each other to stream some data, I think the most important criterion is that every node should be treated the same. Due to administrative issues and planning problems we didn't choose any peer selection mechanism at this point. In future versions I think it is going to be a randomized peer selection in some way that will fulfill fairness in the network.

6.2 Design

For the prototype application I can conclude that the design weren't hard at all, it is a very small application. The nice designed interfaces made my parts easier but as said before, the logic layer became really complex.

This design could easily be bigger and better, I think I nice little GUI would be nice for future version.

The design of the packet transmission part I think is a great solution, the things that are needed is sending and receiving packets. This is done over a socket connection and this design makes it possible to just add a socket connection if you don’t want to use TCP or UDP.

6.3 Implementation

After the prototype was implemented I think that using ACDK was really great and help me a lot with the threads and the sockets, but I also thinks that it would be better for the prototype to not be depended on an external library. Another drawback with ACDK was that it was kind of hard to get it working as it should.

So, maybe in future versions the prototype will be re implemented with a more pure C++ code with raw C sockets and pthreads. Apart from that the packet transmission seems to work really well.

6.4 Evaluation

After testing the prototype I think it was really hard to evaluate it because of the complexity of the logic layer and for the reason that I couldn’t talk about it.

The NC part, combine/encode/decode information, seemed to work fine but the result showed that more and bigger tests are needed to get a deeper understanding. I think huge simulation tests are needed on a fixed topology first to understand how to configure the NC. Then it probably is going to be easier to make real tests with a real network and get better results.

Apart from that the evaluation showed that the prototype used an okay amount of the CPU and the memory. It also showed that there are a lot of things to work on for the next versions before having a media streaming prototype that will outclass the already existing streaming services.

6.5 Future Works

There a lot of things to work on in the future to get a prototype that can challenge the streaming services existing out there. First of all more and larger tests are needed to get more accurate results and maybe also test different packet formats and transport protocols. I think also it would be nice to implement the packet transmission part without ACDK and instead with raw C sockets and pthreads. After that the next thing would be to design, implement and evaluate one or a few possible peer selection mechanisms.

When having all that you probably needing a tracker application, that will inform the joining nodes about the network topology and things like that.

After that when having a prototype that joins and builds up a network and contributes to stream the media to its neighbors, then maybe some investigation is needed about the media quality. A Technique like PET [30] might be a good solution to get better quality on the streaming media.

Apart from the quality issue the prototype must be connected to some media player so a user can get the picture at all.

There are several other features of the prototype that are left out in this thesis, features that are needed if thinking of making this prototype commercial. Features like: a useful GUI, video quality, security and much more.

7. References

[1] Network Coding: An Overview by Axel Davidian
Seminar on Topics in Communications Engineering at Munich University of Technology, January 2005.

[2] Practical network coding by P. A. Chou, Y. Wu, and K. Jain, 51st Allerton Conf. Communication, Control and Computing, 2003.

[3] A random linear network coding approach to multicast by T. Ho, M. Médard, R. Koetter, D. Karger, M. Effros, J. Shi, B. Leong,
IEEE Transactions on Information Theory, 2003.

[4] On Average Throughput and Alphabet Size in Network Coding by Chandra Chekuri, Christina Fragouli, and Emina Soljanin, IEEE Transaction on Information Theory, 2005.

[5] Finite Field by Wolfram Research,
http://mathworld.wolfram.com/FiniteField.html

[6] Measuring Delay, Jitter, and Packet Loss with Cisco IOS SAA and RTTMON by Cisco Systems, Inc. White paper, Document ID: 24121

[7] Reducing Broadcast Delay by Bill Birney, Microsoft Corporation April 2003,
http://www.microsoft.com/windows/windowsmedia/howto/articles/BroadcastDelay.aspx

[8] The Network Coding Homepage by R. Koetter, National Science Foundation under Grant No: CCR-0325673 October 2003,
http://tesla.csl.uiuc.edu/~koetter/NWC/

[9] Wolfram Mathworld by Eric Weisstein at Wolfram Research August 2006,
http://mathworld.wolfram.com/MatrixRank.html

[10] Transmission Control Protocol (TCP), RFC 793 by J. Postel at Information Sciences Institute University of Southern California, September 1981

[11] User Datagram Protocol (UDP), RFC 768 by J. Postel at Information Sciences Institute University of Southern California, August 1980

[12] Real-time Transport Protocol (RTP), RFC 3550 by H. Schulzrinne, S. Casner, R. Frederick and V. Jacobson at Network Working Group, July 2003

[13] Incentives Build Robustness in BitTorrent by Bram Cohen May 2003, http://www.bittorrent.org/bittorrentecon.pdf.

[14] A Framework for Architecting Peer-to-Peer Receiver-driven Overlays by Reza Rejaie and Shad Stafford, ACM 2004.

[15] Algebraic Gossip: A Network Coding Approach to Optimal Multiple Rumor Mongering by Supratim Deb, Muriel Médard and Clifford Choute, IEEE Transaction on Information Theory, 2004.

[16] SCoLE: Scalable Cooperative Latency Estimation by Michal Szymaniak, Guillaume Pierre and Maarten van Steen,

10th International Conference on Parallel and Distributed Systems, July 2004.

[17] Predicting Internet Network Distance with Coordinates-Based Approaches by T. S. Eugene Ng and Hui Zhang, Proceedings of the IEEE INFOCOM 2002.

[18] Measurement-Based Optimization Techniques for Bandwidth-Demanding Peer-to-Peer Systems by T. S. Eugene Ng, Yang-hua Chu, Sanjay G. Rao, Kunwadee Sripanidkulchai and Hui Zhang, Proceedings of the IEEE INFOCOM 2003.

[19] Building Low-Diameter P2P Networks by Gopal Pandurangan, Prabhakar Raghavan and Eli Upfal, IEEE Journal on Selected Areas in Communications, vol. 21, pp. 905-1002, 2003.

[20] Pchar: A tool for measuring Internet Path Characteristics by Bruce a. Mah, http://www.kitchenlab.org/www/bmah/Software/pchar/

[21] C/C++ Reference by Nate Kohl, http://www.cppreference.com/

[22] DataReel Open Source by DataReel Software Development, http://www.datareel.com/

[23] Artefaktur Component Development Kit by Roger Rene Kommer, http://acdk.sourceforge.net/

[24] Red Hat Operating System by Red Hat, Inc. http://www.redhat.com/

[25] Eclipse SDK by The Eclipse Foundation, http://www.eclipse.org/

[26] Eclipse C/C++ Development Tooling by The Eclipse Foundation, http://www.eclipse.org/cdt/

[27] Fedora Core Operating System by Red Hat, Inc. http://fedora.redhat.com/

[28] Microsoft Virtual PC by Microsoft Corporation, http://www.microsoft.com/windows/virtualpc/default.mspx

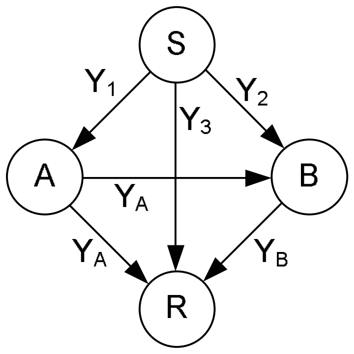
[29] openSUSE Operating System by Novell, Inc. http://en.opensuse.org/Welcome_to_openSUSE.org

[30] Priority Encoding Transmission (PET): A New, Robust and Efficient Video Broadcast Technology by Bernd Lamparter, Malik Kalfane, Andres Albanese and Michael Luby, August 1995

[31] Dipl.Ing. Roger René Kommer, Kassel in Germany, http://www.artefaktur.com

Appendix A: Practical example of NC

This is the network layout for the example with one source, two middle nodes and one receiver:



Start at the source (S) with generating a global encoding matrix (G_t) and multiply that with the original information (X):

$$Y = \begin{bmatrix} \alpha_{11}=5 & \alpha_{12}=7 & \alpha_{13}=3 \\ \alpha_{21}=2 & \alpha_{22}=8 & \alpha_{23}=9 \\ \alpha_{31}=7 & \alpha_{32}=6 & \alpha_{33}=5 \end{bmatrix} \cdot \begin{bmatrix} X_1=3 \\ X_2=5 \\ X_3=6 \end{bmatrix} = G_t \cdot X = \begin{bmatrix} 68 \\ 100 \\ 81 \end{bmatrix} \Rightarrow Y_1 = 68, Y_2 = 100, Y_3 = 81$$

Create a new local encoding matrix and multiply that with the received global encoding matrix (G_t) at node A. The result is the new global encoding vector (G_A):

$$G_A = [\beta_1 = 8 \quad \beta_2 = 6 \quad \beta_3 = 7] \cdot \begin{bmatrix} \alpha_{11} & \alpha_{12} & \alpha_{13} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} = [40 \quad 56 \quad 24] \Rightarrow G_{A1} = 40, G_{A2} = 56, G_{A3} = 24$$

Combine the received information with the local encoding matrix to get the new encoded data (Y_A) at node A:

$$Y_A = [Y_1 \quad 0 \quad 0] \cdot \begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix} = 544$$

Create a new local encoding matrix and multiply that with the received global encoding matrix (G_t) at node B. The result is the new global encoding vector (G_B):

$$G_B = [\gamma_1 = 8 \quad \gamma_2 = 2 \quad \gamma_3 = 4] \cdot \begin{bmatrix} \alpha_{11} & \alpha_{12} & \alpha_{13} \\ G_{A1} & G_{A2} & G_{A3} \\ 0 & 0 & 0 \end{bmatrix} = [96 \quad 176 \quad 120] \Rightarrow G_{B1} = 96, G_{B2} = 176, G_{B3} = 120$$

Combine the received information with the local encoding matrix to get the new encoded data (Y_B) at node B:

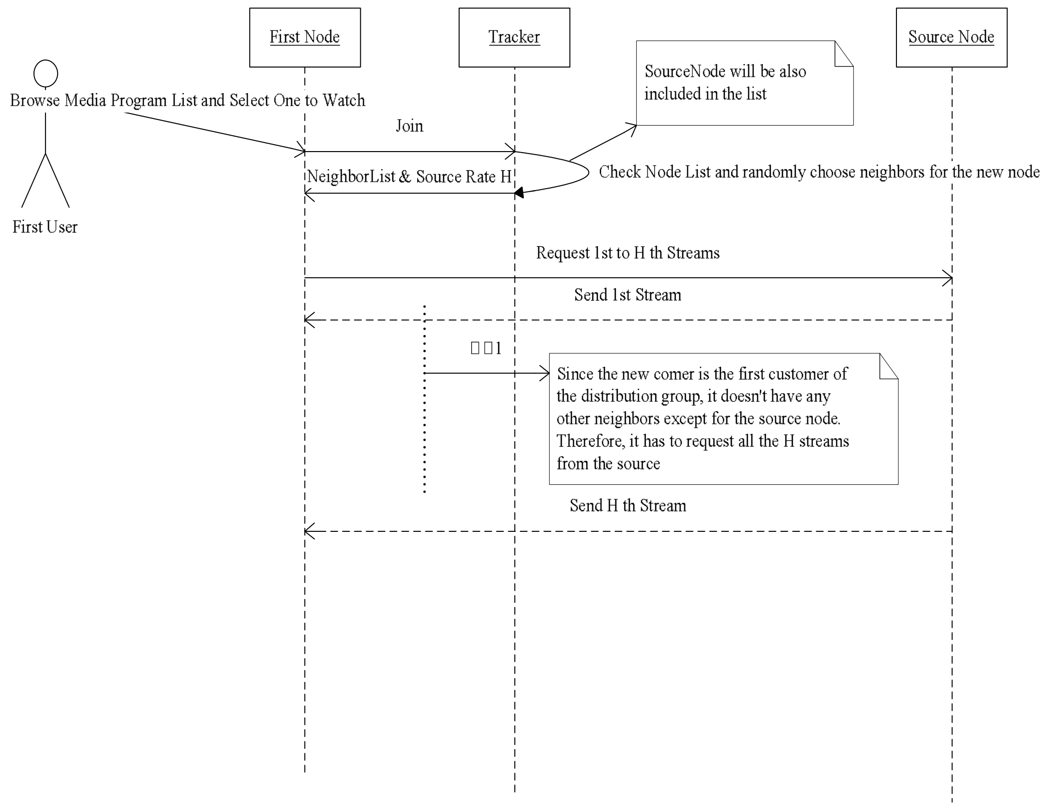
$$Y_B = [Y_2 \quad Y_A \quad 0] \cdot \begin{bmatrix} \gamma_1 \\ \gamma_2 \\ \gamma_3 \end{bmatrix} = 1888$$

Solve the received matrix which consists of information from the source (S) and both of the nodes (A and B). Then will the result be the original information sent from the source:

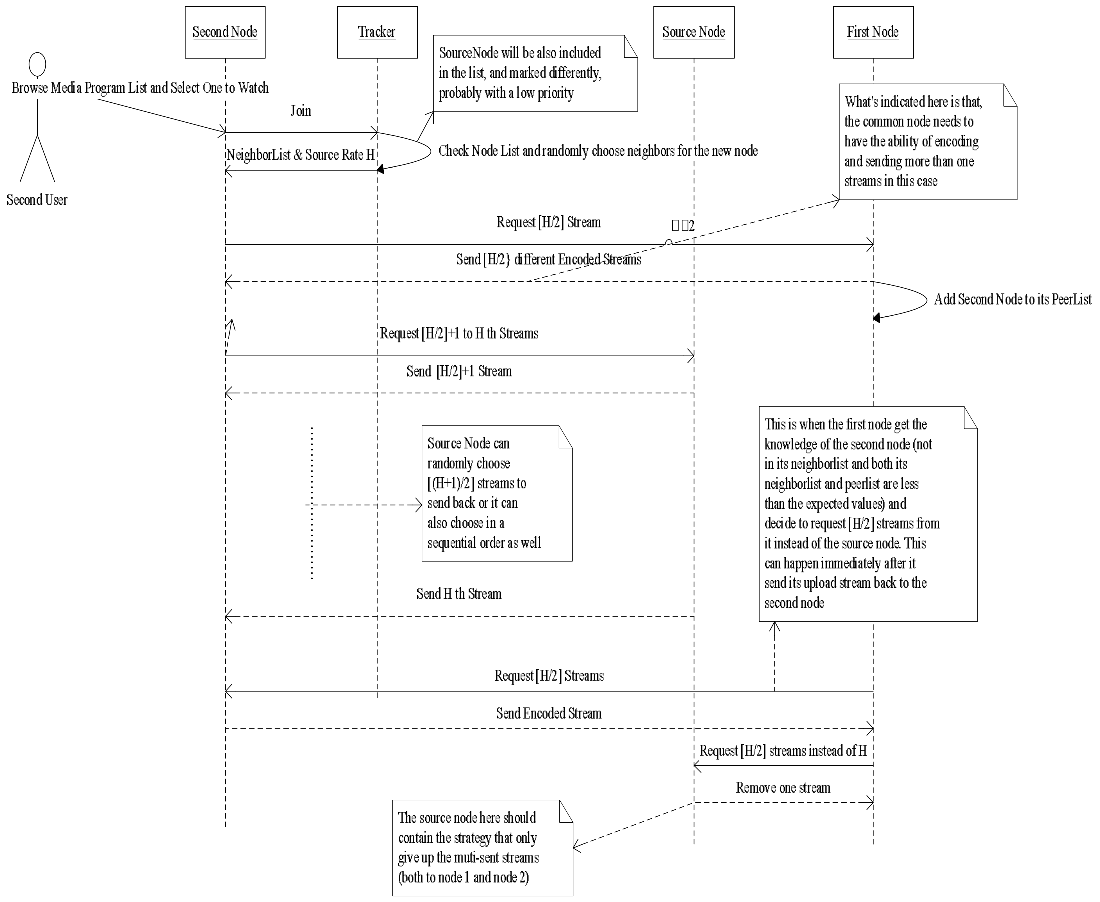
$$\begin{bmatrix} \alpha_{11} & \alpha_{12} & \alpha_{13} \\ G_{A1} & G_{A2} & G_{A3} \\ G_{B1} & G_{B2} & G_{B3} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} Y_3 \\ Y_A \\ Y_B \end{bmatrix} \Rightarrow x_1 = 3, x_2 = 5, x_3 = 6$$

Appendix B: Examples of the early stage.

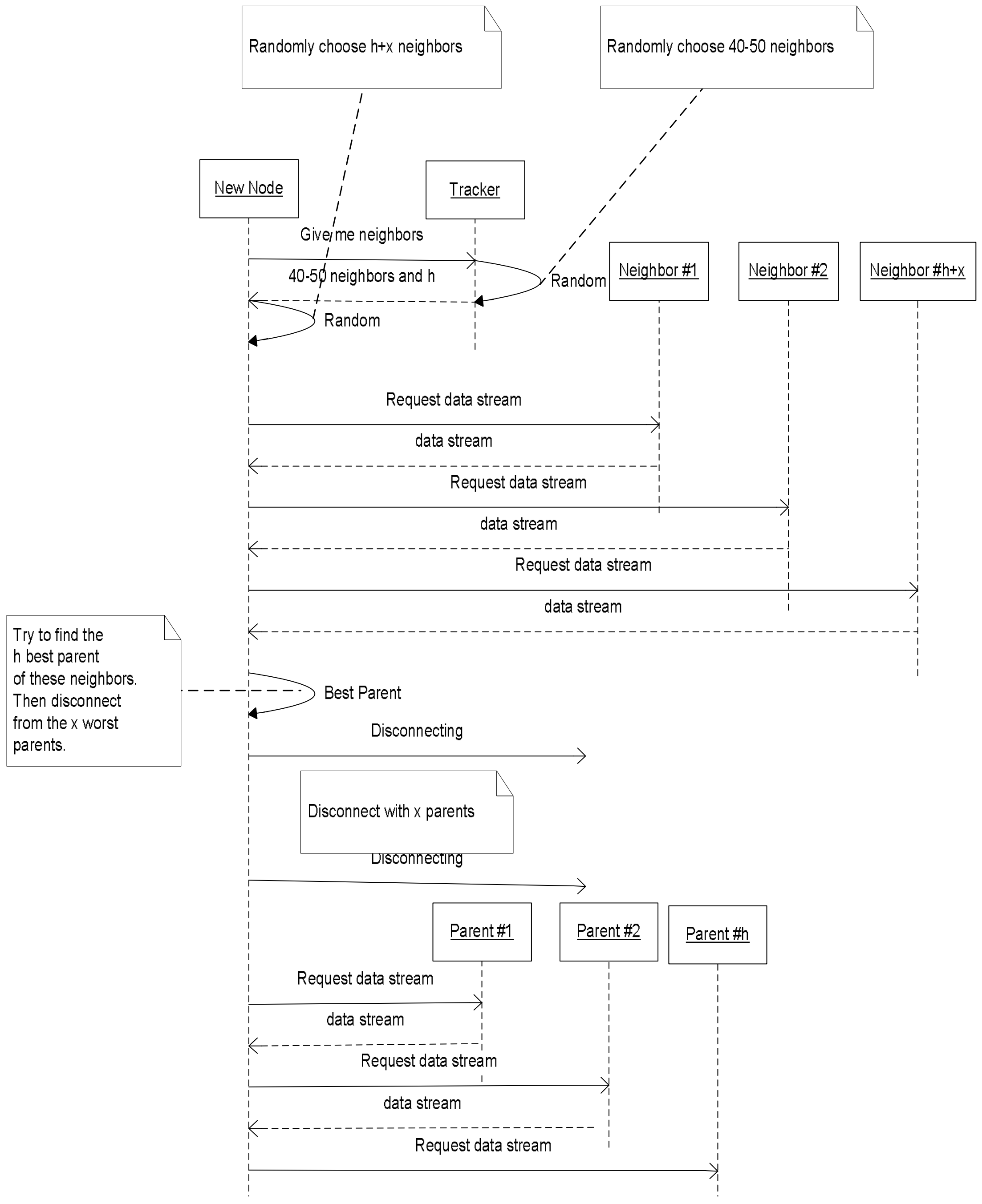
Appendix B.1: The first node is joining the network.


First Node
Tracker
Source Node
SourceNode will be also included in the list
Browse Media Program List and Select One to Watch
Join
NeighborList & Source Rate H
Check Node List and randomly choose neighbors for the new node
First User
Request 1st to H th Streams
Send 1st Stream
Since the new comer is the first customer of the distribution group, it doesn't have any other neighbors except for the source node. Therefore, it has to request all the H streams from the source
Send H th Stream

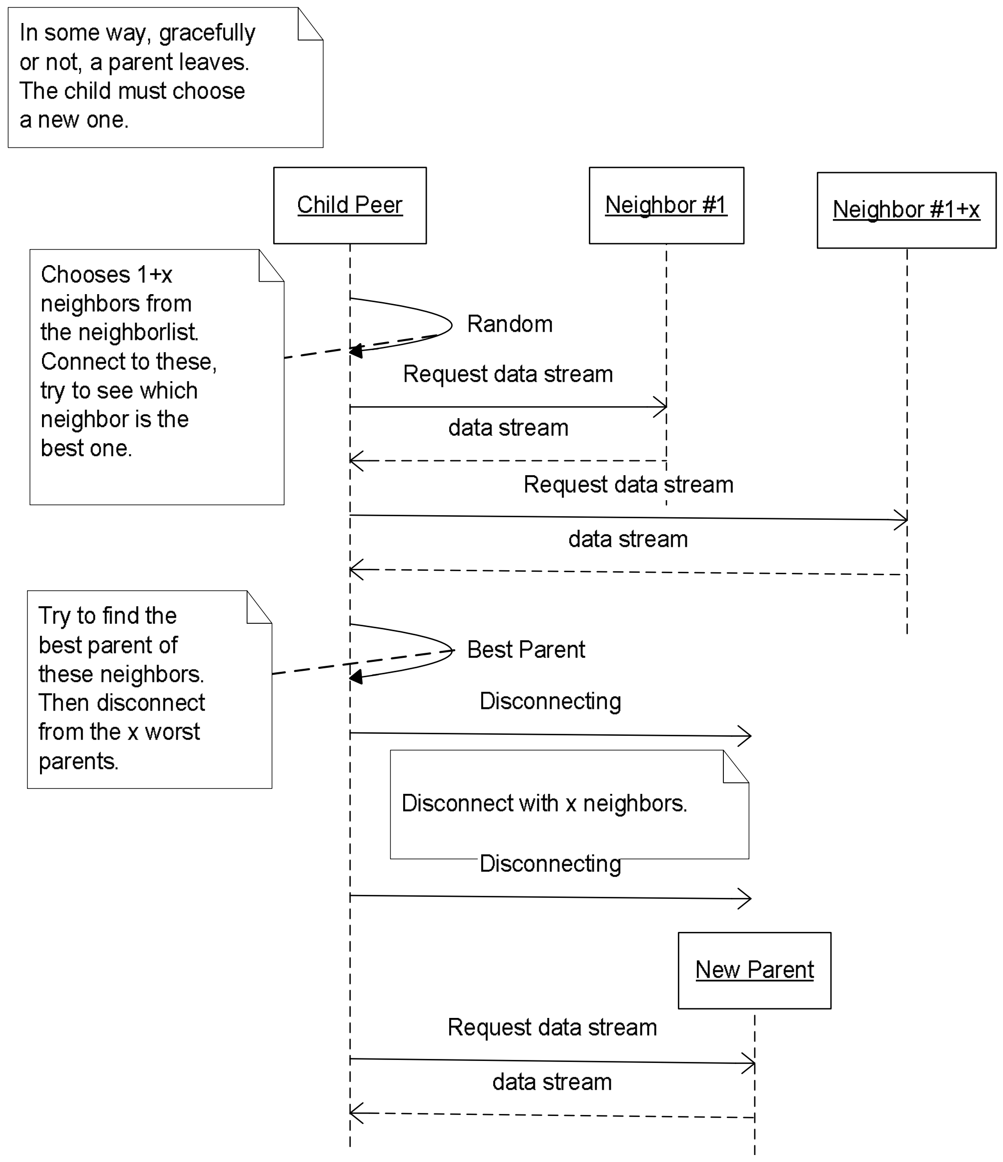
Appendix B.2: The second node is joining the network.


Second Node
Tracker
Source Node
First Node
SourceNode will be also included in the list, and marked differently, probably with a low priority
Browse Media Program List and Select One to Watch
Second User
Join
NeighborList & Source Rate H
Check Node List and randomly choose neighbors for the new node
What's indicated here is that, the common node needs to have the ability of encoding and sending more than one streams in this case
Request [H/2] Stream
Send [H/2} different Encoded Streams
Add Second Node to its PeerList
Request [H/2]+1 to H th Streams
Send [H/2]+1 Stream
Source Node can randomly choose [(H+1)/2] streams to send back or it can also choose in a sequential order as well
This is when the first node get the knowledge of the second node (not in its neighborlist and both its neighborlist and peerlist are less than the expected values) and decide to request [H/2] streams from it instead of the source node. This can happen immediately after it send its upload stream back to the second node
Send H th Stream
Request [H/2] Streams
Send Encoded Stream
Request [H/2] streams instead of H
Remove one stream
The source node here should contain the strategy that only give up the muti-sent streams (both to node 1 and node 2)

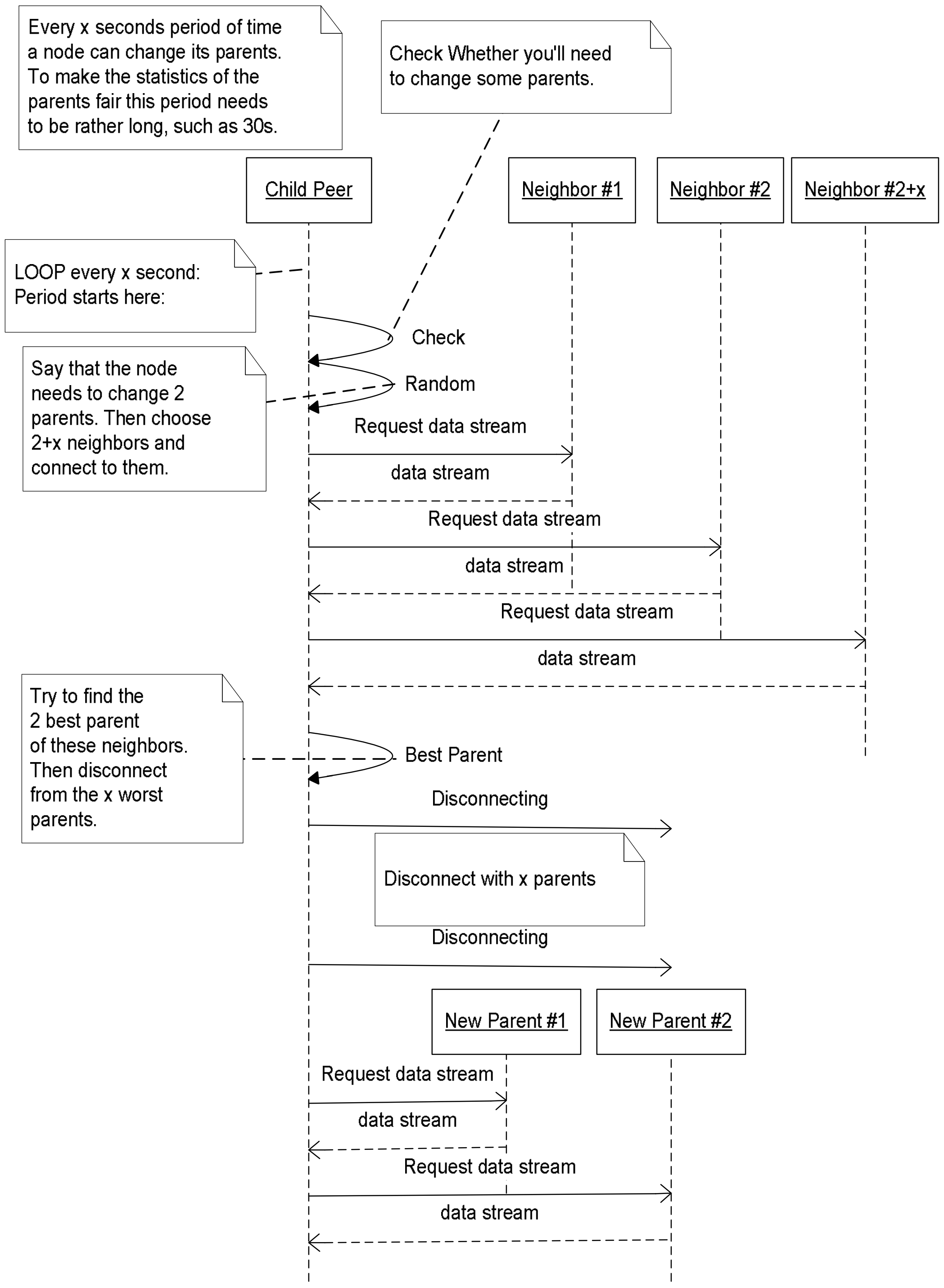
Appendix C: An example of the late stage.



Appendix D: An example of the leaving phase.


In some way, gracefully or not, a parent leaves. The child must choose a new one.
Child Peer
Neighbor #1
Neighbor #1+x
Chooses 1+x neighbors from the neighborlist. Connect to these, try to see which neighbor is the best one.
Random
Request data stream
data stream
Request data stream
data stream
Try to find the best parent of these neighbors. Then disconnect from the x worst parents.
Best Parent
Disconnecting
Disconnect with x neighbors.
Disconnecting
New Parent
Request data stream
data stream

Appendix E: An example of the bad/change condition phase.


Every x seconds period of time a node can change its parents. To make the statistics of the parents fair this period needs to be rather long, such as 30s.
Check Whether you'll need to change some parents.
Child Peer
Neighbor #1
Neighbor #2
Neighbor #2+x
LOOP every x second: Period starts here:
Check
Say that the node needs to change 2 parents. Then choose 2+x neighbors and connect to them.
Random
Request data stream
data stream
Request data stream
data stream
Request data stream
data stream
Try to find the 2 best parent of these neighbors. Then disconnect from the x worst parents.
Best Parent
Disconnecting
Disconnect with x parents
Disconnecting
New Parent #1
New Parent #2
Request data stream
data stream
Request data stream
data stream

Appendix F: A “Hello World!” example, written with ACDK’s scripting language.

```
------------------------------------------------------------------------------------------
#include <acdk.h>
#include <acdk/lang/System.h>

using namespace acdk::lang;

// Minimal example, which just says Hello World!
class MiniAcdkSample
{
public:
  static int acdkmain(RStringArray args)
  {
    System::out->println("Hello World!"); // Java-
like? ;)
    return 0;
  }
};

int
main(int argc, char* argv[], char** envptr)
{
  return
acdk::lang::System::main(MiniAcdkSample::acdkmain,
argc, argv, envptr);
}
------------------------------------------------------------------------------------------
--------------
```