Master Thesis

Swistry: P2P Live Streaming

Philip Frey
philip.frey@alumni.ethz.ch

Dept. of Computer Science
Swiss Federal Institute of Technology (ETH) Zurich
January-July, 2006

Prof. Dr. Roger Wattenhofer
Distributed Computing Group

Advisors: Stefan Schmid, Thomas Locher

Abstract

Today's Internet connections are getting faster and faster. Therefore it has become possible to use the Internet not only for conventional content such as websites but also for live media streams (e.g. music or video). Very often a lot of people want to have access to one specific resource, like an interesting soccer match, at the same time. In order to deliver the desired content to everyone at almost the same time at a high enough rate, a very powerful infrastructure is needed. With the rapidly increasing amount of people using the Internet to receive live media, it is no longer reasonable to base the stream distribution on the client-server paradigm; another way of spreading the content is needed.

This is where Swistry comes into play. We use a peer-to-peer (P2P) approach to deliver the data to the clients. They no longer directly connect to the source of the stream as it was in client-server systems, instead they are downloading the media data from other clients that are watching the same video stream. Every client in this system will therefore not only receive the media data but also forward it to some other clients that wish to receive it. As opposed to having each client connect to the same source, the big advantage of this solution is that we can support many more simultaneous viewers.

Contents

Chapter 1

Introduction

This chapter gives a short introduction to the Swistry network and states the motivation as well as the goal of this project.

1.1 Motivation

The way the Internet is used has changed a lot in the last few years. The main factors are the high bandwidth broadband connections that have become available to everyone. Nowadays we need special devices for a lot of tasks which can be done over the Internet in the future. Popular examples are the telephone or the stereo system. Not only does the Internet reduce the need for these physical devices but the services will become available on-demand for a lower price since no extra infrastructure is needed anymore.

This project focuses on the distribution of live media content such as TV broadcasts or radio over the Internet. Since broadband connections have become affordable for end-users, the number of people that want to access live media over the Internet has rapidly increased and will continue to increase in the future. Therefore the providers of media streams need a way to distribute their content to a lot of clients at the same time. In traditional systems the clients would connect to a powerful server which directly provides them with the audio or video data. The drawback of this approach is that the server has to be a very powerful and therefore expensive system with an extremely high upload capacity in order to be able to deliver the data in real time. Even the fastest connection has its limit and therefore it is not possible to distribute live content among an almost unlimited number of people. The scalability of a one-to-many system is very poor. Therefore we need to move to a many-to-many relation where everyone does not only profit but also contribute.

1.2 Idea and Goal of Swistry

The Swistry system is basically a peer-to-peer (P2P) overlay network in which only very few clients directly connect to the source. The idea is that upon joining the network, clients are assigned a number of neighbours from which they can get the media stream. These neighbours are just other clients that have joined earlier. In this way, everybody who receives data also has to forward it to its direct neighbours. The result is a mesh of peers exchanging missing data very much like it is done in BitTorrent [1], but with the difference that the data is time-sensitive live media.

We use three different kinds of hosts in the network. End-users who want to watch a video or listen to some audio are running the *Swistry peer software*. The content provider, which in older system was the server, is running a *Swistry source*. The third type of host is the *network entry point (NEP)* which is responsible for the assignment of initial neighbours to joining peers. One of the goals is to do the assignment (unlike it is done in BitTorrent) in a way in which peers that have approximately the same bandwidth become neighbours of each other. This will result in a homogeneous distribution of the peers locally as described in section 2.3.

1.3 Overview

The rest of this report is structured as follows: Chapter 2 explains the details of how we address the problem of distributing live content. The overlay network in general as well as protocol details are discussed. Before we implemented the real system, we ran some simulations which are presented in Chapter 3. Chapter 4 discusses previous propositions for this problem and other work which was helpful to our project. Chapter 5 contains some ideas which would probably improve the Swistry system further but which could not be implemented any more due to time constraints.

Chapter 2

Swistry Network

This chapter contains detailed information about the overlay network and architecture of Swistry. After an overview of the whole system, the individual components and communication protocols are presented. Furthermore, some difficulties imposed by the network infrastructure as well as the media data itself are addressed.

2.1 Challenges

P2P systems are much more complex than the old client-server networks. The new challenges are discussed in the following sections. We assume that the overlay is distributed in a sense that each peer only knows about his direct neighbourhood as it is in the Swistry network and that there is no one with a global view.

2.1.1 Topology Control

There is nobody in the network knowing how the overall topology looks like and each peer only has a local view and therefore only knows about his direct neighbours. This might lead to disconnected networks or bottleneck peers within the network as shown in Figure 2.1.

Furthermore it is possible that the network gets *clustered*. This means that there are a couple of heavily interconnected peers (which form so called *clusters*) that have almost no connection to the rest of the network. Without special care taken, these clusters are likely to become completely disconnected from each other in which case the data cannot be received and forwarded anymore. This is illustrated in Figure 2.2.

Another important factor for successful live streaming is that each peer has neighbours of roughly equal bandwidth. If a very slow peer is connected to very fast peers, he will hardly be able to provide them with any data but will still receive a lot from them. One of the goals is therefore that each peer should not have to send much more data than he is receiving from his neighbours.

In terms of the delay it is important that the diameter of the network be kept as low as possible so that also the peer with the furthest distance to the source is able to receive the stream live within a couple of seconds.

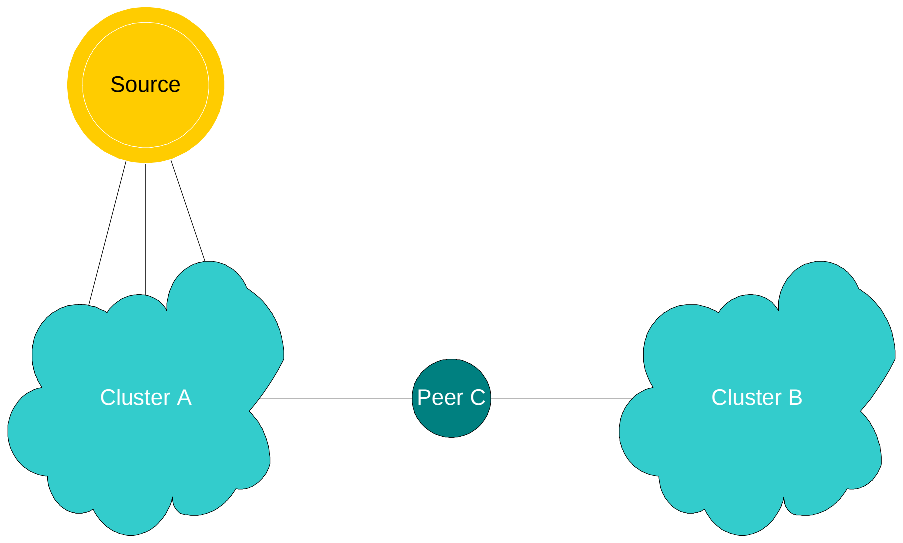


Figure 2.1: Peer C is a bottleneck node. If he leaves, the two clusters are no longer connected.

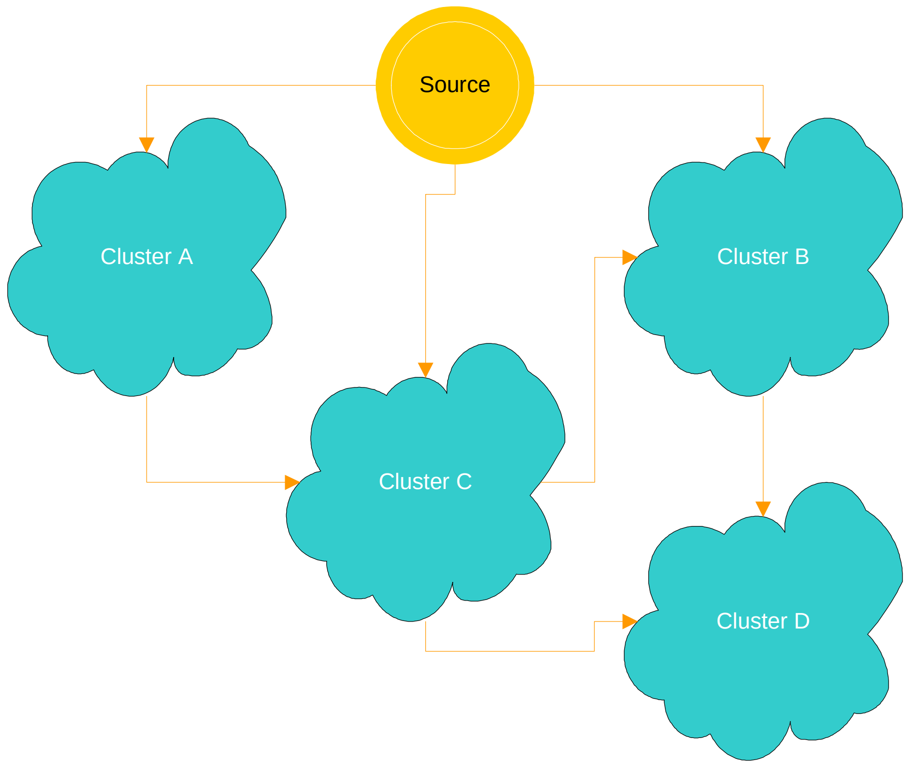


Figure 2.2: Peers inside a cluster are heavily connected but there are only very few inter-cluster connections.

2.1.2 Data Flow

Since especially the outbound bandwidth is still quite limited today, it is important to send as little duplicate data as possible. On the other hand it is important to get all the data in time in order to get a smooth stream.

As fair up- and download ratios should be achieved, it is crucial that the data available is evenly distributed. Otherwise it might happen that one host is providing data for all his neighbours and does not get much back in return. Nevertheless, care has to be taken not to impose too hard limits on the share ratio or else some peers might be given too little data with the result that the stream will no longer travel continuously through the network.

2.2 System Overview

As mentioned before, the Swistry network consists of three kinds of participants: the *peers* (end-users), the *source* and the *network entry point (NEP)*.

2.2.1 Topology

For every stream, there is an individual Swistry network. Each of these networks consists of one or more levels in which zero or more peers and *exactly one* source reside. The idea of the levels is to partition the peers according to the quality of the stream they want to receive and to make sure that all peers inside a specific level fulfill the minimum requirements to receive and forward the data at the necessary rate. We use this approach as a replacement for *multiple description coding (MDC)*.

On one such level there is only the quality available which is broadcast by the source of that level. The levels are not interconnected in any way but completely independent since streams on different levels have different bandwidth requirements.

To perform the assignment of a new peer to the appropriate level is the main task of the network entry point. An overview is given in Figure 2.3.

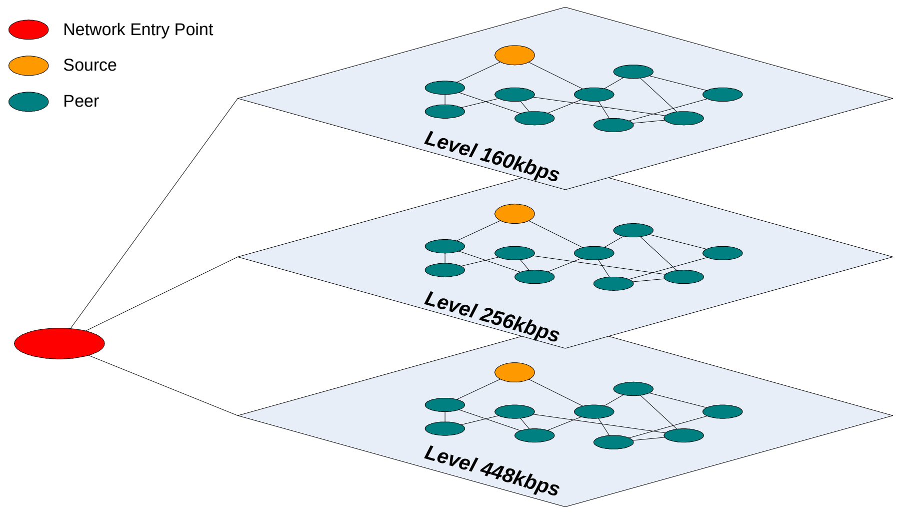


Figure 2.3: The global view of a Swistry network.

2.2.2 Layers vs. Multiple Description Coding

A goal of the Swistry project is to offer the possibility of varying the stream quality based on the available bandwidth. The most elegant solution would be to use a *multiple description code* which splits the full size stream into stripes of lower quality as described in Section 4.2. Each peer could then choose how many substreams he would like to receive. Although this sounds like a nice idea, it is not yet possible to do it efficiently with real time video. Therefore we had to look for an alternative and came up with the layered network topology. It offers some advantages but also some disadvantages. For example it is not possible for a peer to change his stream quality while watching it. He would have to leave the network and re-join with a different bandwidth when using the current implementation. It is possible to add such a mechanism though. On the other hand, this approach allows us to easily ban peers that are too slow to participate. There is no point in letting peers join that are not able to forward the data fast enough. If a peer that is too slow tries to get his initial neighbours from the network entry point, he will be informed that there is currently no level which he could join. This does not mean that there will never be such a level since a new source might join and host a lower level than the lowest existing one.

Our layered solution is nevertheless still quite flexible. Sources can join and leave as they wish. When an existing source leaves, all the peers of its level will automatically re-join at a lower level. They cannot join an upper level since they are already in the highest possible one according to their bandwidth.

Another advantage of the layered approach is that we can combine different media at the same time. For example, we chose an audio-only source for the lowest possible level in our experiment which allowed even extremely slow peers to at least listen to the stream (which might be quite OK if the stream is something like a real-time sports event).

2.2.3 Join Procedure

If a new peer wants to participate he has to browse the Swistry website where he will find a list of available streams with associated network entry point URLs. Each stream has such a dedicated network entry point with whom first contact has to be made in order to join. This procedure is quite similar to BitTorrent where you also have to visit some websites in order to get the torrent file which contains, among other things, the address of the tracker.

At the beginning of the join process, the peer requests a set of neighbours from the network entry point that will form his initial neighbourhood. Thereafter he will contact each and every one of them to notify them of his presence. The new peer is now part of the network until he leaves.

2.3 Intra-Level Network Structures

As mentioned before, it is helpful if the network is well structured. There is no point in wasting time trying to optimize every last bit of the topology though.

We assume that the joining peers all have different bandwidths (inbound and outbound). It is therefore important that the peers with the highest inbound and outbound bandwidths are closest to the source because all the other peers that do not have a direct connection to the source rely on them. In an optimal network we would have a linear decline of the peer's bandwidth with respect to the distance to the source (see Figure 2.4). The critical parameter here is the outbound bandwidth since in reality it is usually much lower than the inbound bandwidth.

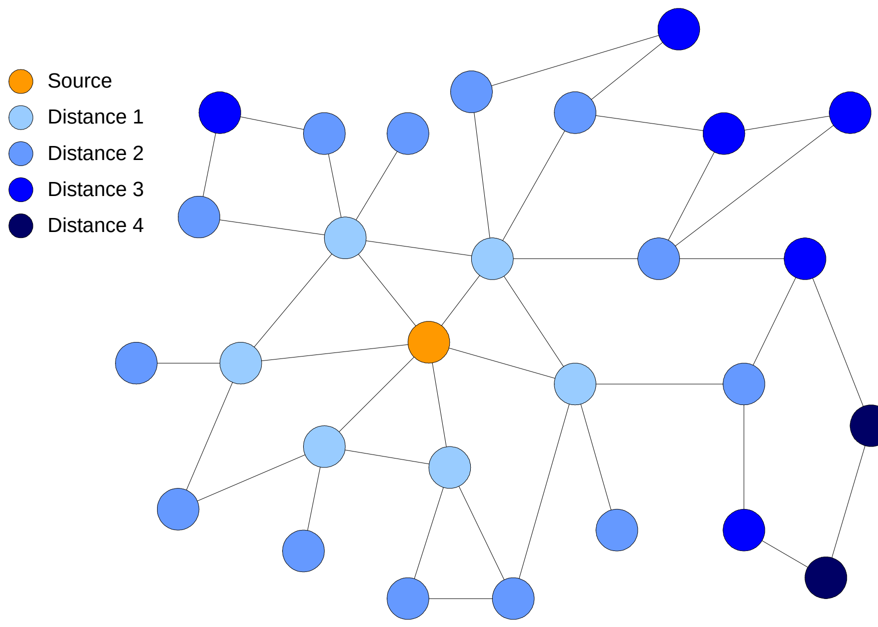


Figure 2.4: Peers with lower bandwidths are located further away from the source (in terms of hops).

2.3.1 Drawbacks of Tree Structures

Earlier P2P overlay networks used one or more trees as their primary (and often only) topology. A crucial fact of this approach is that each peer has one parent and some children. All of them are fixed. The source is the root of the tree and sends the stream to its children. If a peer does not get any data anymore, he knows who is to blame, but the tree has to be reconstructed in order to solve the problem. In the end, this topology turns out to have much more drawbacks than advantages which we will outline next.

Peers Leaving the Network

One of the most serious drawbacks is the fact that each peer (except for the leaves) is the source for at least one subtree. So if any peer inside the tree leaves the network, all subtrees below it get disconnected and experience an interruption of the stream until they are reconnected which is seldom a trivial task. The maintenance of this fix structure induces a large overhead to the network coordinator. The peers do not have the opportunity to quickly choose another neighbour as the provider of missing data but have to wait for a rebuild of the structure.

Unfair Share Ratio and Malicious Parent Nodes

Since each peer has exactly one predecessor and at least one child (except for the leaf nodes), each one of them has to send at least as much data as it receives. In case of a binary tree, each inner node has to have an outbound bandwidth that is at least twice as high as the bitrate of the stream. Currently available broadband connections are usually asynchronous the other way round, which means that end-users have much more download- than upload capacity. So we have a lot of users with enough inbound bandwidth to receive the stream but far too little outbound bandwidth to forward it to one or even more children.

In terms of outbound capacity needed, it is preferable to become a leaf node since they do not have to forward anything. This fact gives an incentive for everybody to be at the bottom of the tree. The only incentive trying to be at the top of the tree is that the closer a node is to the source, the smaller the probability of being in a disconnected subtree gets.

Another problem is the fact that a malicious parent could just decide not to forward any data anymore. Earlier implementations use various mechanisms to accuse such behavior at some sort of adjudicator. But a very malicious peer could now decide to accuse some innocent node of unfair behavior and the adjudicator does not have much choice to tell whether or not he is telling the truth but to base his decision on the number of similar accusations. Even if a parent that does not forward anything is found, quite some work is necessary in order to replace him. Nevertheless, these mechanisms are needed in tree structures because the children cannot spontaneously decide to switch neighbours as mentioned before.

Lots of Leaf Nodes

In a fully populated binary tree roughly half the nodes are leaves. This ratio gets even worse with n-ary trees where $n > 2$. One would think that a binary tree is the best choice. Unfortunately this tree is the worst one could choose in terms of depth. So we either accept that the delay between the source and a leaf node is high and that the probability for a stream interruption is large but keep the number of leaf nodes minimal (choose a binary tree) or we accept that there are a lot of peers in the system that do not forward anything and therefore do not contribute to the distribution of the data (choose n-ary tree with n large).

After all we come to the conclusion that trees are just not well suited for fair P2P networks.

2.3.2 Forests

Later projects suggest the use of forests which are essentially overlay structures based on several trees simultaneously. With this idea each node could be an inner node in one tree and a leaf node in all the other trees. Forests are a little better because each peer has at least some choice among the parents but it is still not optimal. The main drawback of this solution is that it is even more complicated to maintain such a structure. Especially when peers join and leave very often this results in a large overhead.

See Chapter 4 for examples of such structures.

2.3.3 The Swistry Fat Mesh

Since we have shown in the last section that trees are not what we want, we try to build a structure that could be described as a *fat mesh* and is sketched in Figure 2.4. Upon joining, peers are assigned a set of neighbours with similar bandwidths. It does not matter how they are connected exactly as long as the whole network is connected and as long as each node has a minimal number of neighbours.

Let's assume in a first step for simplicity reasons that peers join and then stay until the end of the stream. We would like to build the network so that the faster peers are closer to the source than the slower ones. This means that each peer should be assigned initial neighbours that are about as fast as he is where the source is assumed to be the fastest node available. This assignment is performed by the network entry point as soon as a new peer establishes initial contact. In order to be able to do that, the network entry point needs to remember the peers that have joined before and are currently part of the network. The network entry point basically keeps an ordered list (in terms of outbound bandwidth) of available peers from which he chooses suitable initial neighbours for the new one.

Upon joining, peers will be informed about the maximum and minimum amount of neighbours they are supposed to have. We have imposed these limits for a few reasons which are discussed in the following section.

Lower Limit Motivation

The reason for the lower limit is that we want to force a certain minimal connectivity in the network. With a lower limit of let's say 10 neighbours it is not so likely anymore that the network gets disconnected if a peer is a bottleneck as shown in Figure 2.1 and loses one of the neighbours in a cluster. For that to happen, all but the leaving neighbour would have to reside in the same cluster which is rather unlikely.

If a peer should find himself in a situation with too few neighbours, he starts making contact with the neighbours of his direct neighbours (the peer's 2-neighbourhood). He basically asks each of his remaining neighbours about their neighbours and chooses the ones he did not know before. If, for whatever reason, all neighbours around him disappeared at the same time, he would go back to the network entry point and request a new set of initial neighbours. To prevent the propagation of dead peers, whenever the network entry point has found the suitable initial contacts for the peer, he first checks whether they are still alive using a custom ping mechanism which is described later on.

Upper Limit Motivation

The upper limit is to protect strong peers. Basically everybody wants to have a set of good neighbours and therefore the fast peers are always preferred. If such a strong peer would have to send a lot of packets to many neighbours, he would sooner or later not be able to deliver the packets in time anymore. As a reaction to that his neighbours would drop him and find a better one. The constraint on the maximum amount of neighbours allowed prevents such a wave reaction throughout the network. After all, the peers should not waste too much time on neighbourhood management but try to get the media data in time and forward it accordingly.

Available Peer Bandwidths and Level Assignment

Each joining peer has to tell the network entry point about his available inbound and outbound bandwidth. We first considered measuring the bandwidth of the new peer using some kind of bandwidth estimation procedure like the one described in [2]. The difficulty is to decide between what other host and the new peer to measure the bandwidth in order to get an accurate result. We might choose a reference host to which the new peer has a bad connection and therefore our system would place him in a quite low level although he might actually have a good enough connection to join an upper level. What we propose is that each user should measure his bandwidth by himself using one of the web based bandwidth meters and provide that information upon joining.

The network entry point uses the outbound bandwidth information from the peer to determine which level he is going to give him initial neighbours from. The peer gets assigned the level whose bandwidth requirement is the highest of the ones smaller than the peer's outbound bandwidth. This is the most natural way to do the assignment since everybody wants the highest possible quality that is available with respect to his bandwidth.

The only drawback of letting the peers provide us with the bandwidth information is that a malicious peer could specify an outbound and inbound bandwidth that is higher than the ones he really has. This and other questions related to incentives are discussed in Section 2.5.

Coping with Churn

Of course, the assumption,that peers join and then stay until the end of the stream, made at the beginning of Section 2.3.3 is not very realistic. In fact there are always some nodes leaving and some other ones joining. The behaviour is expected to be similar to a person zapping through TV channels. It might therefore be easily possible that peers change their neighbours quite frequently. Even peers that are currently in the network and do not have joining or leaving direct neighbours might get involved in neighbourhood changes. For example if a new peer wants to become a neighbour of a peer that has already reached the maximum amount of neighbours allowed. The full peer thereafter starts asking his direct neighbours if they are willing to accept the new neighbour although they were not chosen as initial neighbours. For details on the join process see Section 2.4.2.

One of the great dangers with regard to churn[1] is that the network might get disconnected if a lot of nodes join and leave all the time. In this case, a global view would be helpful to detect a possible weak spot in the current topology. Since we do not have that possibility (the network entry point does not necessarily remember all peers that have joined and he has no idea of how they are connected) we must do something based only on local information. As long as a peer keeps receiving the stream, he is fine—he can be pretty sure that he is still connected with the source over some path. He is getting into trouble if his neighbours stop forwarding the data stream. This is probably a sign that they are not receiving the stream anymore either. It would take too much time to ask the neighbours about their data flow and trying to reconnect as a cluster. The best thing to do in order to prevent a lag is to go back to the entry point and get some additional neighbours. Since some node in the cluster is the first one to realize the fact that the network has become disconnected (the last one that kept a connection to the rest of the network) he might get lucky and reconnect to a neighbour that still has a connection to the source before the rest of the cluster realizes it. Since not everybody starts starving at the same time, the network entry point will not get overrun even if each node of the cluster requests a new set of neighbours from him.

2.4 Protocols and Message Exchange

2.4.1 Initialising a Swistry Network

In order to start up a Swistry network for broadcasting a video stream, there are three steps that need to be taken:

1. First of all, the network entry point has to be configured and started. It will be listening on a specified port for incoming TCP connections.

   It is usually running on a dedicated host but it is also possible to run it on the same host as a source. In the latter case, the machine needs to be fast enough since sources usually require quite a lot of CPU time to encode the video stream.

2. Now it is time to set up and start the sources which are the origins of the streams. The media content is usually the same for each source but the qualities vary. The main function of the source is to act as a gateway between some media source like the *VLC media player* [3] or a *SHOUTcast* server [4] and the Swistry network. The source is responsible

[1]Peers are constantly joining and leaving.

for transforming the media stream into finite packets and forwarding them into the Swistry network. Whenever a source is joining, the network entry point creates a new level based on the bitrate specified by that source. If a level with that bitrate already exists, the source will be rejected.

After all sources have joined, the network entry point has set up all the levels and is now ready to accept peer connections and do the level assignments.

3. Since the core network parts are ready now, the IP address and chosen port number of the network entry point need to be published on the Swistry (or some other) website in order to inform the users that a new stream is online.

2.4.2 Join Protocol

The most important parameters for a peer to join the right network are the IP address and port number of the network entry point that is responsible for the desired stream as well as the bandwidth available.

Figure A.2 in Appendix A defines the protocol followed by a new peer and the network entry point.

The peer first establishes a TCP connection to the network entry point over which he sends a JOIN message that contains information about himself, such as his communication port and bandwidth. The network entry point then checks if there is a level for which the joining peer meets the minimum requirements. If there is no such level, he sends a PEERTOOSLOW message back informing him that he is not fast enough to join this stream. Otherwise he selects a predefined number of existing peers from the appropriate level and sends their IP addresses and port numbers back to the joining peer using an INITNEIGHBOURS message. To each initial neighbour candidate, a PING message is sent to see if he is still alive. If a peer fails to reply with a PONG message, he will be dropped.

Upon receipt of an INITNETIGHBOURS message, the peer starts contacting each of the reported initial neighbours using a HELLO message because they need to be aware of him. Otherwise they would not send him notifications about new packets.

Since there is an upper bound on allowed neighbours, some of the initial peers might already have reached it which means that they cannot directly accept the new peer. In this case, they ask all their direct neighbours if anyone has not yet reached the upper limit and is therefore willing to accept the new peer using a FORWARD message. This does not affect the distribution of the peers inside the topology much since everybody has neighbours that have similar bandwidths. On the other hand this forwarding mechanism induces a few random long range contacts which are perfectly desired because they reduce the diameter of the network.

We may encounter two scenarios if an initial contact is already full:

1. There exists a neighbour of the initial contact who is willing to accept the new peer: He replies to the FORWARD inquiry with an ACCEPTFORWARD message. The original peer then informs the new peer that he cannot directly accept him but that there is another peer to which he has been successfully redirected. This is done using the REDIRECT message. Since the new peer sends all HELLO messages simultaneously, special care has to be taken not to end up being redirected to the same neighbour in the 2-neighbourhood twice. Otherwise the new peer would end up with fewer neighbours.

2. All direct neighbours of the initial neighbour are full as well. So there is basically nobody who can accept another peer. The solution to this problem is for the initial neighbour to drop the worst of his current neighbours. He now has space for one additional neighbour

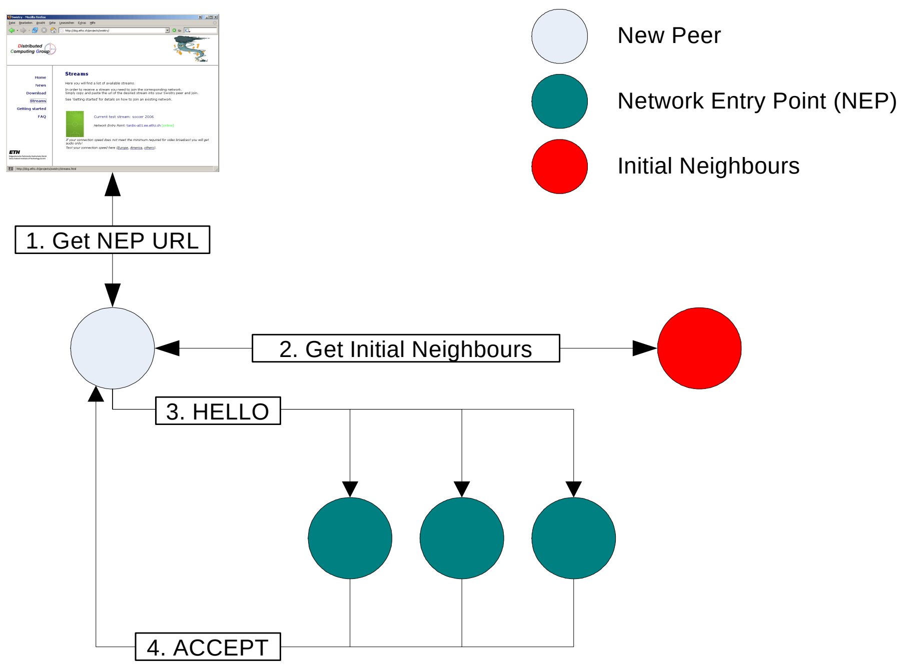


Figure 2.5: Successful join of a new peer.

and accepts the new peer directly and informs him by sending an ACCEPT message back. The new peer does not realise the occurrence of this situation but he does not care since it makes no difference to him anyway—all he wants is a neighbour.

The network entry point is not involved in this process and therefore has no idea who is connected to whom.

Figure 2.6 shows an example of the join procedure. The red node is the source, the grey one is the new peer and the green ones are peer that have joined earlier. Each peer is assigned the three initial neighbours that match his bandwidth (indicated by the number) the closest. There are no complications in steps 1 to 5 but in step 6, upon joining of a new peer with bandwidth 4, the peers with bandwidths 6 and 9 have too many neighbours. In step 6.1, peer 9 can forward the new peer to his neighbour with bandwidth 10 (scenario 1). Step 6.2 shows scenario 2 where peer 6 does not have a neighbour with spare capacity anymore. He therefore drops his worst (slowest) neighbour which is peer 5 and accepts peer 4 instead (although peer 4 is even slower than the one he has just dropped!).

2.4.3 Ping Protocol

Swistry uses its own TCP based implementation of ping. It is not important to the system whether a certain host machine is online but we want to know if it is still running the Swistry application. Therefore we cannot just use the ping implementation of Java. Another advantage of the custom ping protocol is that we can detect NAT or firewall issues if the target host is not able to accept our TCP connection attempt.

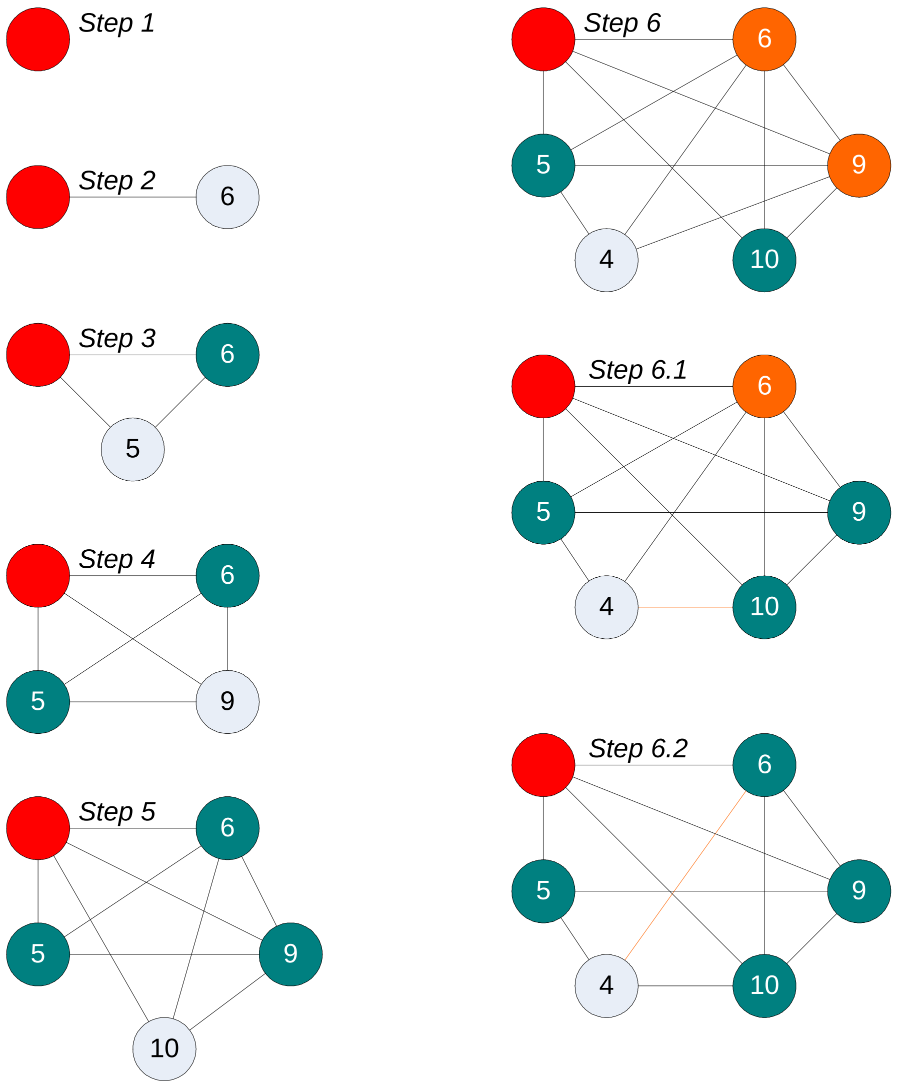


Figure 2.6: An example of the join procedure (init=3, min=2, max=4).

2.4.4 Pull-based Stream Propagation

The source keeps producing P2PPACKET messages which contain the actual stream data. Since Swistry is based on a pull-based data distribution mechanism, the source does not forward the generated packets directly to its neighbours. Instead, it sends NOTIFICATION messages in order to inform them that a certain packet has become available.

Each packet has a unique identifier and a fixed size. The NOTIFICATION message is basically a container for sending these packet IDs. Upon receipt of such a message, the receiver checks if he still needs that packet. If he does, he requests it using a REQUEST message that contains the packet's ID. In the third step, the sender of the NOTIFICATION delivers the P2PPACKET to the requesting node. As soon as the P2PPACKET has arrived, the receiver announces it by sending NOTIFICATION messages to his neighbours (which are no necessarily connected to the source as well) and forwards the packet in the same way. This is done until each peer has received the packet.

Of course there are always a lot of packets exchanged simultaneously since not everybody needs the same packets at the same time (as the delays vary). A simplified example of this process is shown in Figure 4.4.

The advantage of the pull-based approach is that there are very few duplicate packets sent and if some packet is lost for whatever reason, the affected peer can just re-request it from another neighbour. This leads to a lot of flexibility. The packet loss could be further minimised by using TCP instead of UDP. The drawback of TCP is the relatively large header.

The disadvantage is that the delay is increased. At least three times the propagation delay plus the transmission delay does a receiver have to wait for the actual stream data. If the propagation delay is low, this does not preponderate and thanks to the flexibility of choosing a provider, it is easily possible to request the packets from the neighbours with low delay.

At the moment, Swistry requests the packets in random order. This sometimes leads to a lot of peers requesting the packets from the same peer without exchanging a lot among themselves. This needs to be improved in future releases.

Another way to boost the performance a little is to initially push packets to a small number of neighbours in order to decrease the initial delay.

2.4.5 Leave Protocol

We differentiate between *active* and *passive* leaving. In passive leaving, a node just disappears. Be it because the user has killed the application, has lost his network connection etc. When leaving the Swistry network actively, all current neighbours and the network entry point will be informed; this is the expected behaviour.

Passive

Since the actual data is being forwarded using a pull-based approach running over UDP, the sender of the NOTIFICATION message might not realise that one of his neighbours has disappeared with the effect that he will continue sending waste UDP packets.

Currently there are two possibilities for a peer to realise that a neighbour has passively left the network:

1. Upon receipt of a HELLO message, the peer has to check whether or not he can accept the new one as neighbour due to the upper bound restriction. In order not to send any false rejects, he pings all his current neighbours to see who is still alive. At this point, all neighbours that do not respond with a PONG message will be dropped.

2. There is a thread called *ZombieKiller* which sends PING messages to the current neighbours in a predefined interval (usually every 30 seconds). When a neighbour has failed to reply with a PONG message for the third time in a row he will be dropped.

Active

When closing the application window or leaving the network manually over the menu, a BYE message is sent to all peers in the current neighbourhood and to the network entry point.

Upon receipt of such a BYE message, a confirmation has to be sent back. Otherwise a malicious peer could just flood the network with false BYE messages and everybody would drop all his neighbours.

A BYE message contains the IP address and the port number of the leaving peer and is sent using a TCP connection. The receiver now establishes a new TCP connection to the specified IP address and port number and sends a BYECONFIRM message containing his port number. When receiving such a confirmation message, the leaving peer responds with a BYEACK message if he is really leaving or with a BYENACK if he is not (this means that the original BYE message was not valid). The corresponding neighbour will only be dropped if the confirmation was successful (i.e. a BYEACK message was received).

2.4.6 Finding New Neighbours

In case a peer does not have enough neighbours anymore, he will need to find new ones. By *not enough* we mean less than the required lower bound.

In order to learn about new peers that again have a similar bandwidth, all remaining neighbours are asked to report their direct neighbours. A DIRECTREPORTNEIGHBOURS message is used for that purpose. The answers (the DIRECTNEIGHBOURS messages) will be collected and duplicate peers as well as already known peers are removed. The remaining ones form the set of candidates. Each of them is contacted using a HELLO message with the same protocol as when first joining until the upper bound of allowed neighbours is reached. Thereafter, the lonesome peer should have enough neighbours again.

It might happen though that a peer suddenly does not have any neighbours any more for whatever reason or that he still does not have the minimum amount of required contacts even with the new ones. One way to learn about further neighbours would be to ask the new ones again for their neighbours and so on. In our approach, the lonesome peer returns to the network entry point and runs the join protocol again to get a new set of initial neighbours. With that he should have enough neighbours. If he still does not, this can only mean that there are currently no more neighbours logged on (since the number of assigned initial neighbours is always at least as large as the minimum required if there are that many peers online) in which case the search is finished.

With our solution, the search for new contacts terminates in at most two steps with the drawback that it is not fully decentralised but for all practical purposes this is much more efficient.

2.5 Incentives and Information Hiding

For the Swistry network to have a high throughput, it is important that the peers with high bandwidths are close to the source. To be close to the source means to have a low delay and less dependency on other peers. Therefore everybody wants to be as close to the source as possible. It is not known which node is actually the source. The only indication might be that it does not request any packets at the moment. In future releases, this should be added in order to better protect and hide it.

Since we do not test the bandwidth of new peers at the network entry point but rather let them tell us what they have, a malicious peer might declare a bandwidth which is much higher than his real bandwidth. At the beginning this works fine for him. He will be close to the source. But since each peer has an upper limit of allowed contacts, the new peer will be dropped sooner or later if he can not deliver the data fast enough. Remember that upon receipt of a HELLO message, if a peer has no spare capacity to accept the new peer and all his direct neighbours are full as well, the worst existing neighbour will be dropped—in our case this will be the malicious node. The declared bandwidth is only important when joining at the network entry point. The peers have a different system to rank their neighbours based on real experience. So the only way to prevent being dropped is by delivering the data fast enough. If the malicious peer can do that then there is no problem at all and everybody is happy.

Since the declaration of the bandwidth does not only influence the distance to the source but also determines the target level and therefore also the quality of the stream, there is a strong incentive to declare a higher bandwidth. As explained above, the result will be that if a peer is not able to forward the stream fast enough he will always be dropped first which again results in a poor watching experience because the search for new neighbours might take some time.

It is no problem if a new peer reports a slower bandwidth than the actual one since he will be perfectly able to forward the data but he might get put in a lower level where the stream quality is poorer. Everybody is free to join a lower level of course.

The basic idea of Swistry incentives is that most of the information is hidden. Nobody knows what the bandwidth declarations of the others are. So upon receipt of a HELLO message, the peer cannot know whether the new peer will be fast or slow—he just accepts him and drops him again if necessary. Everybody measures how reliably his neighbours provide him with new data and updates their ranking accordingly since this is what matters in the end.

2.6 Bandwidth Limitations and Requirements

A major problem in reality are the asynchronous data connections offered by European Internet providers. Assume that for a certain stream all Swistry peers are Swiss Cablecom broadband clients. They can choose one of the following connections (download/upload in Kbit/s): (300/100), (3000/300), (4000/400) or (6000/600).

Each of these connections (except maybe for the slowest one) are more than enough to receive a live stream in good quality. The issue is that not even the fastest one is capable of forwarding a very good stream since the download-upload ratio is 10:1 which is very bad for P2P systems. If it were the other way round, it would be natural to use trees since the outgoing data could be ten times the incoming so every node could accept ten children and would still be fast enough.

In an ideal scenario, end-users would have share ratios of 1:1 with a synchronous link of 1000 Kbit/s or more since the goal of a fair P2P system is to make peers upload as much as they download.

In our tests we were broadcasting streams with 220, 320 and 530 Kbit/s. For a great watching experience which would make people use the Internet instead of the TV, a minimum of around 800Kbit/s would be necessary which is not even possible with the fastest Cablecom product.

Internet connections in Asia are already much faster (by a factor of 100 on average) than the ones in Europe but it is probably only a question of time before connections with higher bandwidths and equal ratios become available over here.

In order to be sure that peers are able to forward the media streams fast enough, they need an upload bandwidth that is at least 1.3 times the stream bitrate in order to be assigned to that level by the network entry point. We then can be sure that they have enough bandwidth to process network maintenance messages and stream data at the same time as our simulations have shown.

2.7 Firewall and NAT Issues

Firewalls and Network Address Translation (NAT) boxes are installed in many homes today. The Swistry user can choose which port he wants to use for communication when configuring the client application. Since we are using TCP for maintenance and UDP for the actual stream data, the firewall must allow both protocols on that chosen communication port. On the NAT box, the port needs to be forwarded to the host on the internal network that is running the Swistry application.

We do not implement any mechanism to bypass NAT of firewall devices as it is done in the Skype network (see Section 4.3.2). Instead the user has to do that manually. If somebody does not have the privileges to change these things and does not have a port forwarded to his host which he can use, there is no way he can participate in the Swistry network. We have left out this mechanism primarily due to time constraints with the motivation that the users who really want and are allowed to participate can do it by manually configuring their network infrastructure accordingly.

2.8 Security Issues

In terms of network maintenance there are certain mechanisms that prevent malicious peers from doing much damage although the messages are neither signed nor encrypted. The stream data itself is not protected in any way.

2.8.1 Denial-of-Service

Each component is susceptible to denial-of-service attacks since the whole communication is based on message passing. As soon as a peer has to respond to an infinite amount of requests he will sooner or later stop working. This is not that much of a problem since the components are all threaded which means that the only work that is done by the main thread is the dispatching of the new message. It is thereafter delegated to the appropriate handler. Also malicious messages that could let the handlers crash are not terribly bad since the rest of the application is not affected by that. Of course there are certain mechanisms implemented that check if the message is valid and drop it in case it is not but there is always a chance that some messages might not be identified as malicious as we know from various other systems.

The most vulnerable part in the Swistry design is the network entry point since its IP address and port number are well known and there is currently no backup solution implemented. If he fails, new peers cannot join anymore. There are basically two possibilities to eliminate that weakness in future releases: One could either have a second network entry point and load balance the incoming requests or add another way for peers to join the existing network even when the network entry point is not available anymore. A peer should then be able to join using another peer instead of the network entry point. This idea is similar to the distributed tracker used in BitTorrent.

2.8.2 Malicious Peers

What happens with peers that join and do not contribute as requested is discussed in Section 2.5. A malicious peer could forward garbage instead of the real stream which the receiver would not realise and just forward it to the media player. But why forwarding garbage if it is as expensive as forwarding the real data? There is no incentive for that except for trying to interrupt the stream of the neighbours. Digital signatures could be used to assure data integrity.

Another way for adversaries to destroy the performance is to try to disconnect some peers from the network. It does not work to send fake DROPPED or BYE messages since they have to be confirmed by the respective sender. The only way to disconnect a peer from the network would be to repeatedly send HELLO messages from different IP addresses and ports until the victim has dropped all good neighbours. If the upper limit for the amount of allowed connections is high enough, this is no longer feasible since the peer forwards the new (malicious) peer to his neighbours and only if all neighbours are full as well he might drop a good neighbour. This attack would have to be performed such that the attackers can provide the victim with valid stream data during the attack. Otherwise, the victim would first drop the attackers again before dropping good nodes. The ranking of the neighbours is such that peers that have been reliably sending data for a long time have a high ranking and are very unlikely to be dropped for a new neighbour.

2.9 Components and Software Architecture

This section describes how the Swistry software architecture looks like without going into implementation details. For more information please refer to the JavaDoc and the source code.

The core component is the `Node`. It contains all functions for the neighbour management as well as the `Notification` and `P2PPacket` objects. As direct subclasses there are the `Peer` and the `Source` which both implement additional role specific methods as described in Section 2.9.2. The system discussion will be based on the underlying communication mechanisms.

2.9.1 Handlers and Tasks

In order to process different tasks in parallel, Swistry is highly threaded. All incoming requests are immediately dispatched and forwarded to the appropriate handler. The counterpart to the handlers are the tasks which actively initiate communication to one or more neighbours and send requests. Some tasks wait for the replies to their requests and some let the dispatcher catch it and forward it to the responsible handler.

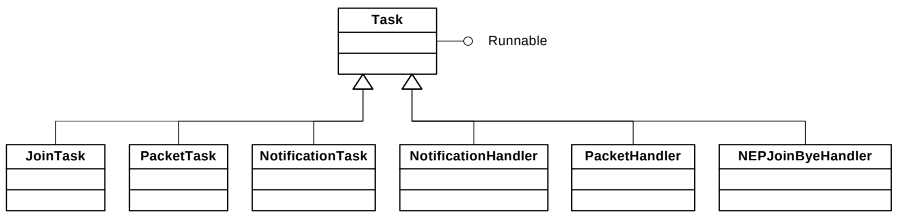


Figure 2.7: UML diagram showing some tasks and handlers.

2.9.2 Derived Core Components

The `Peer` as well as the `Source` have a UDP and a TCP communication interface which we will explain next.

TCP vs. UDP

TCP is used for the whole network maintenance and management whereas UDP is only used to receive and forward the streaming data. Real world experiments have shown that there is actually no need for UDP. The motivation was that it is possible to send much smaller packets with UDP. The overall performance is probably even better when using larger TCP packets if broadcasting on a high bitrate. There is a significant overhead in the three-way communication if the packets are too small. When packets are so small that there are more than 3 of them needed to be received per second, the stream is starting to lag significantly. Since we implement a re-requesting strategy in case a packet is late or lost, we could as well let TCP do that for us. Further extensive tests are needed to harden that assumption.

Data Flows

The `Node` has two mechanisms for handling incoming connections and packets: One for UDP and one for TCP. They listen for incoming data and dispatch it accordingly. The dispatch process starts the appropriate handler and immediately returns to listening for new data. Each handler is run as an individual Thread. With this approach we do not need to block while responding to a message.

Figure 2.8 illustrates the approach. The red arrows indicate the data flow. Since most handlers deal with network maintenance, they need access to the set of `Neighbour` objects which represents the current neighbourhood. Based on that they can decide to accept new peers, drop bad ones etc.

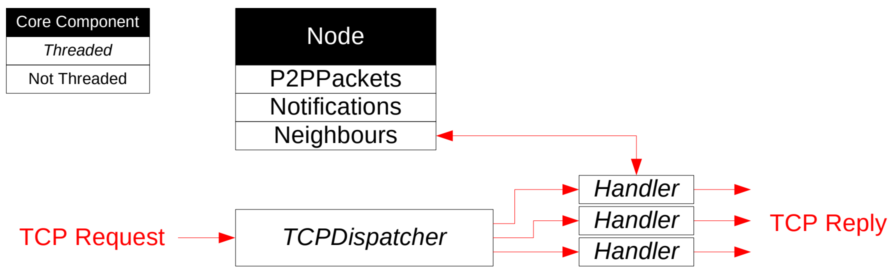


Figure 2.8: Illustration of the TCP handling mechanism in Swistry.

The UDP data flow is a little more complex since it involves more components than just the `Node` and some handlers. The `Source` and the `Peer` differ in that the `Source` does not send any requests. In future versions, it might request packets as well so that the other peers will not know if they are directly connected to the source of the stream or not.

Figure 2.9 shows the data flow inside a source. The actual stream data is received from the *AV Source* by the `InputAdapter` and forwarded to the `Packetizer`. The AV Source can be anything that is sending an audio or video stream. As soon as there is enough data available, a new `P2PPacket` is created out of it and added to the packet list of the `Node` (green trail). Upon

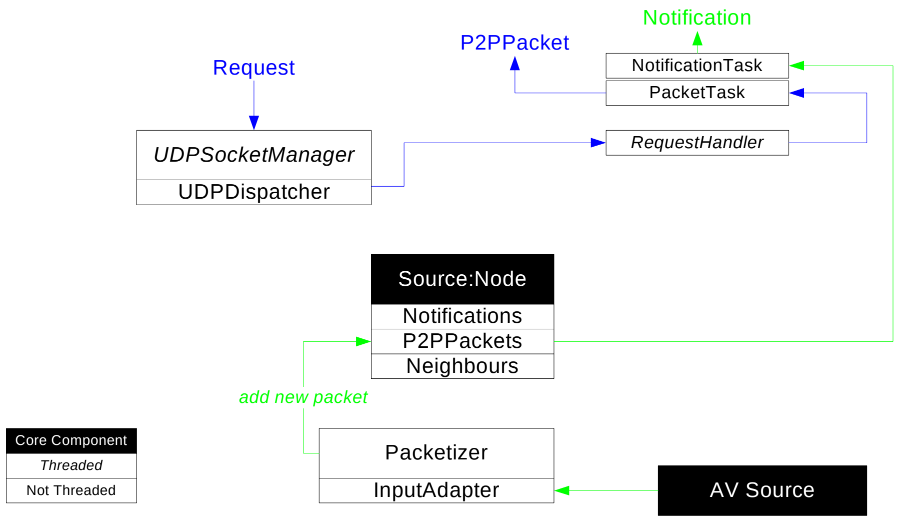


Figure 2.9: UDP data flow inside a source component.

adding a new packet, the source immediately starts a `NotificationTask` to inform his direct neighbours about it by broadcasting a notification message.

They react to it by sending a request for the announced packet. The dispatcher then starts a `RequestHandler` that checks if the received request is valid. If it is, a `PacketTask` is started that sends the desired packet to that neighbour (blue trail).

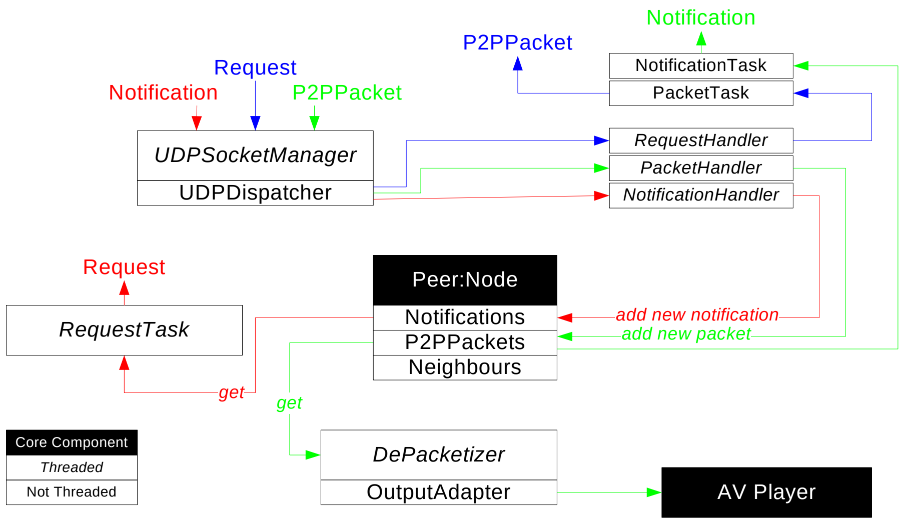


Figure 2.10: UDP data flow inside a peer component.

Figure 2.10 is very similar to Figure 2.9. The most important difference is that a peer has an *AV Player* instead of an *AV Source* and therefore also a `DePacketizer` and not a `Packetizer` that reassembles the chopped data parts again in the correct order so that the stream can be forwarded to the player.

When receiving a notification message from a neighbour, the UDP dispatcher starts the appropriate thread which is called `NotificationHandler`. He checks if the announced packet is of interest (according to its identification number) and adds it to the collection of the other received notifications if necessary (red trail).

The `RequestTask` component is responsible for getting available notifications out of the collection and sending appropriate request messages in order to get the actual packets.

If a neighbour requests a packet, the `RequestHandler` is started (same as in the source) that checks if the requested packet is still available and if the request is valid. If so, a `PacketTask` is started that sends back the desired packet (blue trail).

The third and last situation is the reception of a P2PPacket. As explained above, we do not sign the packets. In order to do a minimal check, it is verified that a packet with this id has actually been requested. If not, the packet is dropped. Otherwise it is added to the collection of `P2PPackets` at the `Peer` and a `NotificationTask` is started to announce it to the neighbours (except for the one that has originally sent it) (green trail).

The `PacketHandler` is also responsible for updating the rankings of the neighbours. It is increased by one with each packet that has arrived in time and decreased by one with every late packet. In addition to the ranking, the round trip time (RTT) is recorded. We measure the time elapsed between the moment when the request was sent and the time when the requested packet arrived because we now have a value which takes into account the bandwidth as well as the delay.

2.10 Media Input and Output

The source itself does not have any notion of the content it is broadcasting. Therefore the whole system is not dependent on any particular codecs or media formats. All it does is take an octet stream as input, transform it into finite packets and forward them. The receiver has no idea what it is receiving either. It just reassembles the received pieces into a continuous octet stream using the `DePacketizer`.

As provider for such an octet input stream, we used the VideoLAN client and various SHOUTcast stations. The interface between them and the Swistry source is implemented by the `InputAdapter` for which we make almost no assumptions. This gives us the freedom to use whatever we like as input. If we wanted to add RealPlayer support for example, all we need to do is write a RealPlayerInputAdapter that somehow connects to the RealPlayer and receives the media data. The incoming data is then forwarded to the `Packetizer` using its `addData(byte[] data, int dataSize)` method.

The same holds for the output stream. There we need to implement a matching `OutputAdapter` that is able to hand over the octet stream to the appropriate player. The interface of the `OutputAdapter` is quite slim as well. `dataAvailable(byte[] data, int len)` is the only method that must be implemented. It is used by the `DePacketizer` to forward the reassembled byte stream.

2.11 Media Codecs

The search for the perfect media codec is like looking for the holy grail. There are several possibilities but none of them is perfect.

It is a tradeoff between quality and bitrate. In order to get a great watching experience, the quality of the video and of the audio stream should be very high. Something like AC3 (Dolby

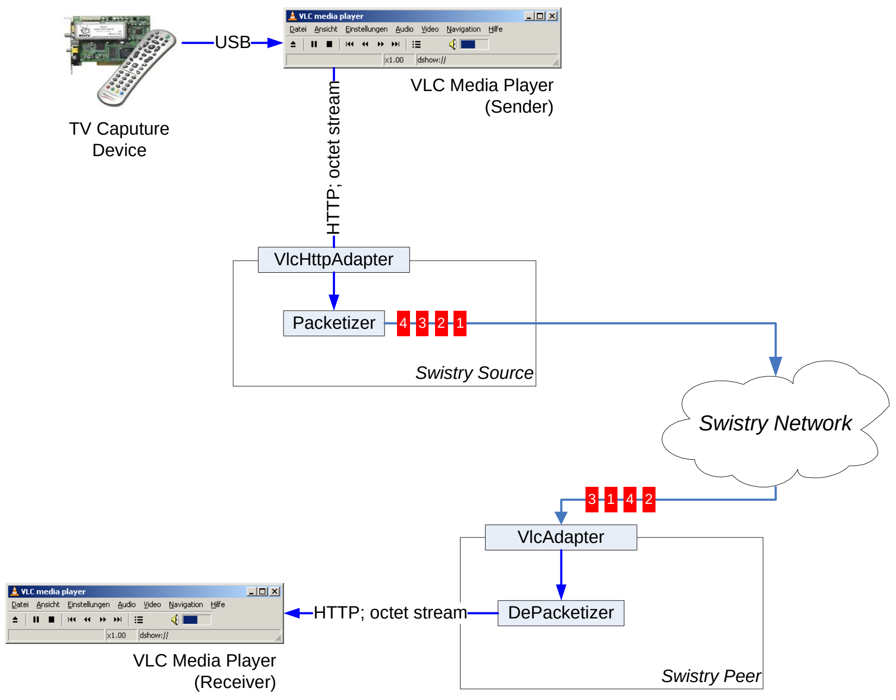


Figure 2.11: Media flow from a TV capture device to the VLC client of the peer.

Digital) for audio combined with high-definition video would be optimal. Unfortunately this is not realistic with today's Internet connections but it might be in a couple of years time.

The audio and the video stream are each encoded independently with respective codecs. In order to transport them, they are multiplexed into one byte stream. Unfortunately this multiplexing induces some overhead especially when broadcasting at low bitrates.

We chose the VideoLAN client as source and destination because it is easily possible to set it up for various encodings and multiplexers. In addition to that, it has a HTTP interface which we were using in the `VlcHttpAdapter` to get the byte stream. It is available for Linux, Windows and Mac and a lot of people already have it installed on their system.

Table 2.1 shows what combinations of codecs and multiplexers are supported by the VideoLAN client [2].

Finally we have chosen the latest H.264 video codec in combination with MP3 audio and muxed it into an MPEG-TS container. Our primary focus was on providing a reasonable video quality on low bitrates. The H.264 codec is basically an MPEG-4 codec which is usually used for live video conference with little movement. The audio stream was encoded using a standard MP3 encoding which provides good quality even on low bitrates. Figure 2.12 contains the details on what bitrates we used. The total values are the sum of the video and audio bitrate without the multiplexer overhead. In brackets, the actually transmitted bitrate is stated.

Since Swistry has no notion of the content, streams with future encodings can be broadcast without the need of changing the system. All that has to be done is to implement custom input- and output adapters as mentioned above.

[2]details about the codecs and multiplexers can be found at http://www.videolan.org/streaming/features.html

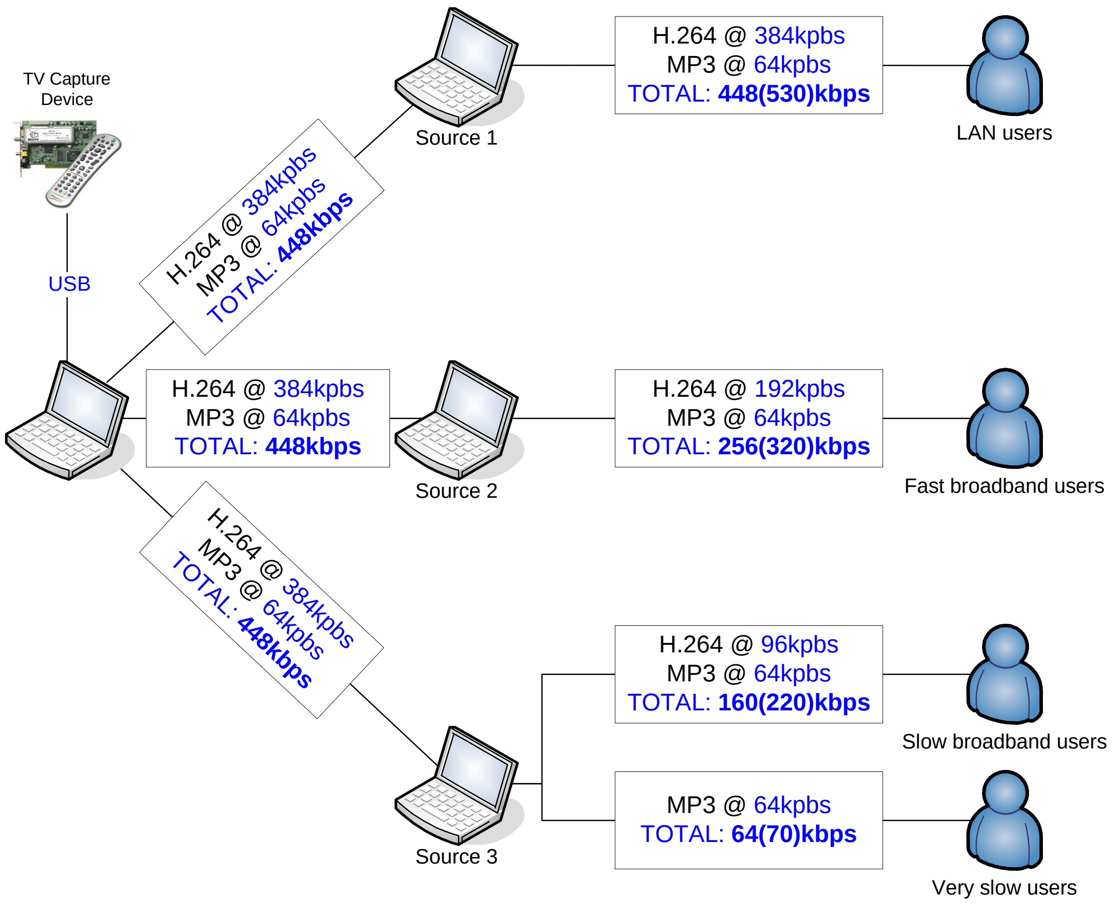


Figure 2.12: Setup of our streaming experiment.

		PS	TS	Ogg	ASF	MP4	MOV	Raw
Video formats	MPEG-1 video	Yes	Yes	Yes	No	No	No	Yes
	MPEG-2 video	Yes	Yes	Yes	No	No	No	Yes
	MPEG-4 video	Yes	Yes	Yes	Yes	Yes	Yes	Yes
	DivX 1/2/3 video	No	Yes	Yes	Yes	No	No	No
	WMV 1/2	No	Yes	Yes	Yes	No	No	No
	H/I 263	No	Yes	No	No	No	No	No
	MJPEG	No	Yes	Yes	Yes	No	No	No
	Theodora	No	No	Yes	No	No	No	No
Audio formats	MPEG 1/2/3	Yes	Yes	Yes	Yes	No	No	Yes
	AC3 (i.e. A52)	Yes	Yes	Yes	Yes	No	No	Yes
	MPEG-4 audio	No	Yes	No	No	Yes	Yes	No
	Vorbis/Speex	No	No	Yes	No	No	No	No
	FLAC	No	No	Yes	No	No	No	Yes

Table 2.1: Compatibility matrix of codecs and multiplexers supported by the VideoLAN client.

Chapter 3

Simulation and Experiments

This chapter discusses the initial simulations that were performed followed by the real-world experiments and conclusions we drew. The last part describes problems and difficulties we encountered during the project.

3.1 Simulation

Before we started implementing the real system we ran some simulations in order to see if our approach was feasible and would lead to good results in terms of performance and network characteristics.

The nodes in the simulation were built similarly to the ones used in Swistry. They had most of the functionality of the real nodes.

In order to be able to run the simulation faster, an agenda was implemented which contained the actions that needed to be taken by a certain peer at a certain time. This allowed us to simulate delays and bandwidth restrictions quite realistically. If each node would have been run as a thread, the Java VM would have decided what node runs when and the `sleep(long ms)` command would have been very inaccurate. In addition to that it was possible to simulate much more nodes when using an agenda then it would have been with threads.

3.1.1 Live vs. Fast Motion

The simulation could be done live or in fast motion. When doing it live, it was possible do dynamically add and remove peers and see what happened in terms of network connectivity and data flow. The time-lapse simulation was intended to give insight into how the system would change if run for some time.

The live simulation confirmed that the data flow is much better when the network is structured as described in Section 2.3. We compared our structure to a random peer distribution. The effect was that slow peers that were close to the source were not able to forward the stream fast enough and therefore much more duplicate packets were sent.

With the time-lapse simulation we investigated how our system would react to churn. We found out that there is always an upper limit of join and leave actions per time interval at which the network does not break apart with high probability. Generally speaking, the system is much more resilient to churn if the peers are allowed to have many neighbours.

3.1.2 Diameter

The interesting network characteristics are primarily its diameter and connectivity. A high diameter results in long delays for nodes that are far away from the source (in number of hops). If a stream broadcast claims to be live, it needs to be forwarded to everybody within couple of seconds. We therefore aim for a low diameter.

Figure 3.1 shows the average diameters on a network where nodes joined with uniformly distributed bandwidths in the range between 0 and 255. We ran 100 simulations for each amount of peers and took the average for each group. The number of peers was increased exponentially. It has to be emphasised here that the maximum number of allowed connections was limited to 6 which is very little. In reality, something like 30 would be used as an upper bound. In addition to that, 10000 peers is quite a lot and therefore rather improbable. So when turning to a realistic scenario with a few hundred peers and an upper bound between 20 and 30, the average diameter will be below 5 on average.

The second chart (3.2) shows the variance of the diameter in the individual simulations. When we ran the simulation with a lot of peers, the diameter variance was quite large. This is an effect of the local view of the peers and depends highly on the order in which they join. In addition to that, the threaded environment does not exactly represent reality because it is only pseudo simultaneous. This simulation did not take into account that peers are joining and leaving all the time.

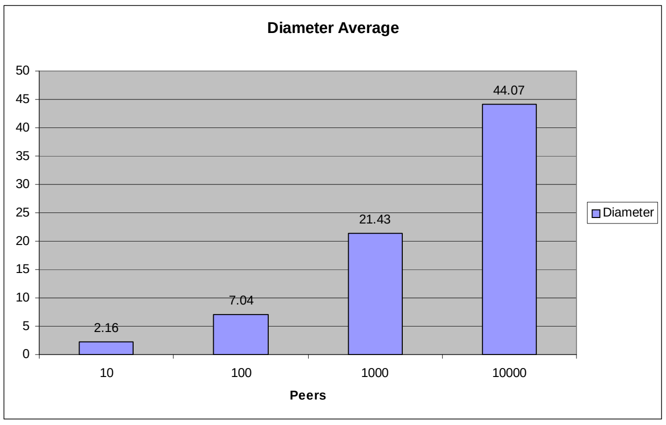


Figure 3.1: Results of network diameter simulation (lower bound: 4; upper bound: 6).

The diameter did not change dramatically when adding churn to the simulation. It grew at most by a factor of 2. Again, we chose an upper bound of 6 and a lower bound of 4. The problem with churn is that the network is likely to get disconnected as described in the following section.

3.1.3 Connectivity

In order to measure how heavily the peers are interconnected, we did min-cut calculations based on the algorithm presented in [5] which gave us intelligence on how weak the weakest point was. A min-cut of one means that there is a bottleneck edge: The network is not connected anymore if that edge disappears. The min-cut should never be smaller than the lower bound of required neighbours in order to keep the network stable. Since we do not have a superior node that knows where the weak spots are, it might nevertheless happen that the min-cut drops below the lower

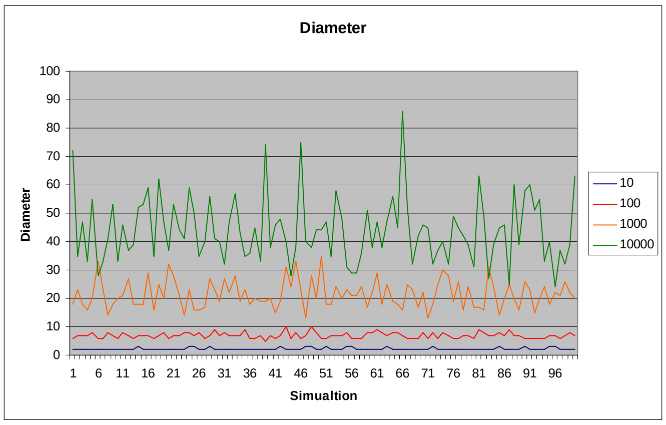


Figure 3.2: Variance of the diameter (lower bound: 4; upper bound: 6).

bound if there is a lot of churn. If the lower limit is high enough, the min-cut is expected not to get too small as our tests have shown. As explained in Section 2.3.3 it is rather unlikely that such bottleneck edges emerge. In static context where peers join and do not leave anymore, there were no bottleneck nodes.

Figure 3.3 shows the two possible scenarios when the network has become clustered. In this illustrating example it its assumed that the blue node is the only connection between the two clusters. In case a) the blue node has become a critical bottleneck since there is only one edge that connects the blue node to Cluster A and with that also Cluster B to Cluster A. In b) he has several connections to either cluster. The chance of the network being disconnected after a peer in Cluster A has left is much smaller.

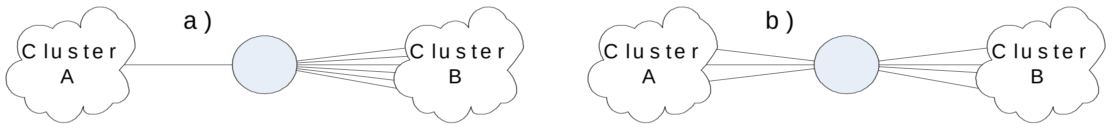


Figure 3.3: a) A bad but unlikely situation. b) The peer has several neighbours in each cluster.

3.2 Experiments

We ran several live tests with the Swistry network. First we used only nodes in the lab of the electrical engineering department of ETH. These nodes are directly connected over a switched network with 100Mbit. Therefore the delays were very small and the bandwidth was more than sufficient.

After these trials were successful, we expanded the network into the Internet by uploading our client software to the Swistry home[1]. We used world cup soccer matches in order to attract people. Unfortunately Swistry did not gain enough popularity during the test phase to run large

[1]http://dcg.ethz.ch/projects/swistry

networks. We logged only around 35 different external IP addresses during the whole test phase. There were never much more than 40 people logged in at the same time (including around 20 lab nodes). This might be because the initial release had still some problems which we could solve based on the users feedback we received. Also the GUI has been improved and the usage has become simpler with later versions which is very important in order to attract people.

The sources and some initial peers were running on the lab computers for the soccer live streaming. The complete setup of the experiment is depicted in Figure 2.12. We used a Hauppauge WinTV [6] device to receive the TV signal with VLC on a dedicated machine that also did a first encoding into H.264 video and MP3 audio. Details about our coding decision can be found in Section 2.11. The final encoding was done on three other machines that ran the source application of Swistry. They downloaded the encoded media stream from the first machine. In order to provide three different qualities, two of these second-level machines had to re-encode the stream with a lower bitrate. We decided to use three different video bitrates (all encoded with H.264) but only one quality for the audio stream because 64kbps MP3 stereo was enough for the soccer matches. Still the bandwidth requirements were quite high compared to what is available from today's Internet providers. A peer that wanted join had to have at least 30% more outbound bandwidth than what was required by the stream bitrate.

The available final encodings are listed in the following table:

Level/Encoding	**H.264 video**	**MP3 audio**	**Total (MPEG-TS)**	**Mux Overhead**
High	384kbps	64kbps	530kbps	18%
Medium	192kbps	64kbps	320kbps	25%
Low	96kbps	64kbps	220kbps	37%
Audio-Only	-	64kbps	64kbps	-

Table 3.1: The encodings we used to stream the soccer matches.

As can be seen in the last column, the multiplexer induces a large overhead when the bitrates are small. We did not have much choice of multiplexers because MPEG-TS is the only one that supports the H.264 encoding and even this combination is non-standard. Therefore it is quite tricky to get other players to run that specific VLC encoded data. Future streams might use *mencoder* and *mplayer* as media source and sink instead of VLC although they are not that widely spread.

The size of the UDP packets turned out to be a crucial parameter. It has to be chosen so that there are at most three packets needed per second. We recommend a size so that each packet contains roughly 500ms of media data. In our network, this worked like a charm even when the peers were outside the lab. The drawback of UDP is that the packets cannot have an arbitrary size but are limited to 65kB. For our high-quality stream, we used 33kB packets.

3.3 Problems

3.3.1 VideoLAN Client (VLC)

As already mentioned before, we had some issues with the VLC player although it is very comfortable to use and supports a lot of encodings, multiplexers and streaming possibilities.

A major problem was that when using the MPEG-TS mux, it did not send any stream specific information other than it was an octet stream. Therefore the receiver had to figure out by himself what multiplexer and codecs were used. This is where mplayer and Windows Media Player failed. Even the VLC player itself sometimes had difficulties in starting the playback. They claimed

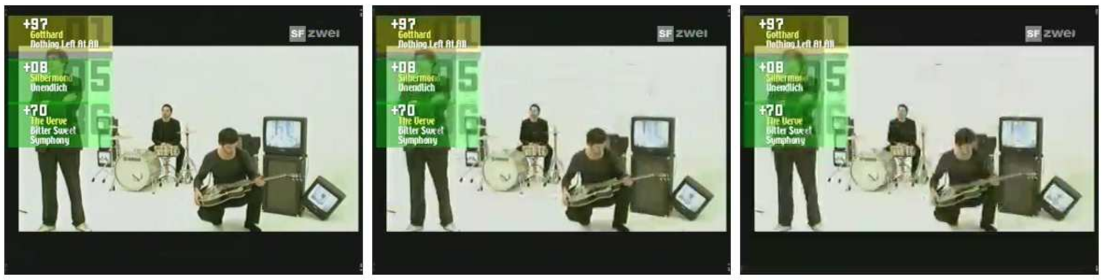


Figure 3.4: High, medium and low quality video sample.

that the data that was fed to them was garbage. The only way to make them start the playback was to stop and restart the it.

Very fast computers are needed in order to do the live video and audio encoding. We used IBM Thinkpad notebooks with Intel Pentium M 1.86GHz processors and 1GB RAM.

In future releases, a media player might be directly integrated into the Swistry peer application. The drawback of that is that we will lose some of the flexibility we have right now. Probably the best thing would be to provide a choice of watching the stream with the built in player *and* of forwarding the data to an external application using a custom `OutputAdapter`.

Another problem with VLC not giving any more specific information about the data stream was that we could not distinguish key frames from normal ones. Whenever a key frame was lost, the quality suffered much more than when loosing a normal frame. There was no meta information at all that we could use to prioritise the data.

3.3.2 Java Media Framework (JMF)

Before using VLC as media source, we intended to use the *Java Media Framework (JMF)* that provides multimedia support for Java applications. Unfortunately, the framework is quite outdated. The latest available version was released in May 2003.

There is a project called *Jffmpeg* which still adds support for new codecs to JMF. There is no real support for TV capture devices though and it is very cumbersome and slow. We therefore decided not to do the multimedia part in Java.

Chapter 4

Related Work

This chapter gives an overview of the streaming approaches that have been made earlier. Some of our ideas were motivated by the protocols and systems that are described next.

The very first P2P streaming systems only supported audio streams which were usually encoded in MP3 or OGG. Since MP3 streaming can be done in reasonable quality using as little as 64kbps there was not much need for a P2P solution. The popularity of SHOUTcast [4] which is based on a client-server architecture shows that audio streaming using direct connections to the sources never really was a problem.

Earlier P2P approaches were very often based on tree structures to distribute their content. In Section 2.3.1 we show why this is not an optimal solution with regard to today's bandwidth distributions.

4.1 Previous Overlay Propositions

This section gives an overview of related projects. They are ordered chronologically according to the year when they were published.

4.1.1 Distributed Video Streaming with Forward Error Correction

Instead of just retransmitting a lost packet, this paper [7] suggest the use of *Forward Error Correction (FEC)* which is basically an encoding of the data such that lost data can be reconstructed using the data that was received successfully. In addition to that, they work with more than one sender where each can be used simultaneously by the receiver in order to further minimise the probability of packet loss. The system is receiver-driven and based on two algorithms: rate allocation which determines the sending rate for each sender and packet allocation to assure that no senders send the same packet.

FEC induces a small overhead. Not all of the bandwidth can therefore be used for the stream data. It is a tradeoff between protection from data loss (redundancy) and efficient use of the available bandwidth.

The bandwidth of each sender is estimated using a technique which is based on the TCP-friendly protocol. Using the outcome of this estimation, data is requested from each sender starting at the one with the smallest packet loss until the sum of the incoming data rates meets the stream bandwidth.

4.1.2 BitTorrent

The BitTorrent system was invented by Bram Cohen in 2003 [1] and it has become very popular.

Shared files are split into finite pieces which are exchanged using a pull-based approach. In order to download a file using BitTorrent the user needs to have a so called *torrent* file which contains information where to start looking for peers that share the desired file. For each shared file there must be such a torrent file which each participant has to download (usually from a website). This file is then handed over to a BitTorrent client that initiates contact to a so called *tracker*. The job of the tracker is to remember who is currently sharing a certain file and to help the peers find each other inside the P2P network.

After having received an initial set of neighbours the new peer starts exchanging data with them. In the beginning he cannot upload anything to them because he does not have anything yet. They will tell him what data they have and the new peer responds to these notifications with request messages for certain parts of the data. As soon as he has received a few pieces he starts sending notifications to his neighbourhood as well and will soon start uploading whatever they request. The whole system is based an a tit-for-tat fairness policy. SHA-1 hashes are used to verify the integrity of the individual pieces.

If a peer has a neighbour that is never uploading anything to him, he will sooner or later stop responding to his data requests and remove him from his neighbourhood. By doing so, each peer tries to keep a set of neighbours with which he can exchange data as fast as possible in a fair way. In order to prevent cluster generation, peers accept random new neighbours in certain intervals and remove others.

When the data sharing starts, there is at least one peer which is known as the *seeder* that has the whole file stored locally. The individual pieces are requested at random to assure a fast distribution of the whole file. As soon as the seeder has sent each piece at least once he can leave the network and the peers will complete the download by exchanging the missing data among each other. In addition to the random requests, each peer tries to download the rarest pieces among their neighbours in order to prevent bottlenecks.

The major difference between BitTorrent and Swistry is that in the latter system there is no node that has all the data right from the beginning. The source keeps producing data in real-time which means that the amount of pieces that can be exchanged at any time is much smaller than in BitTorrent.

Another important difference is that in BitTorrent each piece is equally important. The file is only useful once all the pieces have been downloaded. In Swistry, the pieces that are closer to the current play time in the stream are much more important because there will be a lag if they arrive too late. This imposes the need for other packet requesting strategies than the ones used in BitTorrent.

An advantage of the real-time streaming as opposed to file sharing is that there is a maximum bitrate for downloading, given by the stream which is broadcast. So a fast peer can have slower neighbours without any disadvantages as long as they are at least as fast in terms of upload bandwidth as the stream bitrate. In BitTorrent a very fast peer would eventually choke the slower peers and reconnect with faster ones because what matters most to him is to have all pieces as early as possible.

Drawbacks

BitTorrent is not intended for live streaming. Each peer in the network tries to connect to the fastest available peers. There is no sense in doing that when forwarding a media stream since the bitrate of the stream determines the maximum throughput.

The system does not respect the order of the packets in any way because the whole data is available right from the beginning. This is not the case with live streams—the data is produced continuously. With BitTorrent, all packets are needed in order to get the file working. Swistry on the other hand is only interested in packets that contain media data required in the future with respect to the current playback position.

An attempt to exploit BitTorrent can be found in [8].

4.1.3 Pastry, Scribe & Splitstream

Splitstream [9] is based on Scribe [10] which is based on Pastry [11]. The goal of Splitstream is to provide video streaming using a tree overlay.

In every P2P network, we would like to have each peer contribute in the same way. Unfortunately this is not easily possible with just one tree. The stream is originated at the root of the tree and is propagated towards the leaves. This means that all the leaf nodes do not have to forward any data which is quite unfair for the interior nodes which usually have to forward more data than they receive.

Splitstream therefore suggests the use of more than just one tree for delivering a media stream. Each node can then be in a leaf position in all but one trees and will have to participate as an interior node in the last tree. Everybody can choose to which trees he will subscribe according to his available bandwidth. With this idea, the contribution of each peer will more or less match the amount of data he is receiving.

In order to expediently use the multi-tree architecture, the video stream that is broadcast by the source is split into several substreams which need to be combined in order to get a high-resolution picture in the end. The stream is split using a technique called *Multiple Description Coding (MDC)* which is described in Section 4.2. Figure 4.1 gives a simple example of how it works. The stream is originated at peer 1, split into two so called *stripes* (the substreams) and each of them is broadcast in its own tree. All nodes participate in both trees. Let's look at peer 5 for example: He receives stripe 2 from peer 8 and has to forward it to peers 2 and 3. So he forwards twice as much as he receives in this tree but since he is a leaf node in the tree for stripe 1 he does not have to forward anything there. After all he forwards as much as he receives.

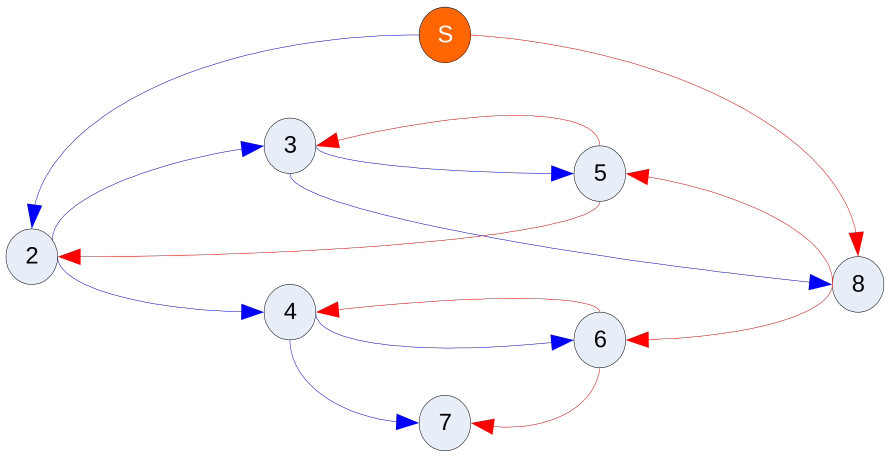


Figure 4.1: A Simple example of the Splitstream idea: Two stripes are broadcast simultaneously using two distinct trees.

Scribe provides Splitstream with the necessary multicast infrastructure. Any Scribe node can create a new multicast group or join some of the existing ones. These groups are used by Splitstream to forward the individual stripes of the stream. Each node can also send data to his group, provided he has the appropriate credentials which in Splitstream only the source has. For more details see [10].

Scribe again is built on top of another infrastructure called Pastry which is basically a self-organizing P2P location and routing substrate. The designers of Pastry state that *"Pastry performs application-level routing and object location in a potentially very large overlay network of nodes connected via the Internet." [11]*

The idea of Pastry is to use a generalized form of hypercube routing which is actually based on a ring structure. Each node in the Pastry network has a unique identifier and every message carries such an identifier as destination address or key. It is sent to the live node which is numerically closest to this identifier. Each node only knows about his direct neighbours. Upon receiving a message, the receiving node forwards it to a neighbour whose identifier shares a prefix with the destination address that is at least one digit longer than the one of the present node.

Drawbacks

A major drawback of this solution is the dependency on the underlying systems: Splitstream is based on Scribe which again is based on Pastry. This results in an unnecessary complex overlay structure with a very high maintenance overhead. The system relies on MDC which is only feasible for still images.

Splitstream is a push-based streaming approach. There is a source which sends the stream data to its successors and so forth. If a node has lost a piece there is no way for him to get it from a different source—he will just miss some frames of the respective stripe. This is not actually a problem as long as he has other stripes but losing some data in the last remaining stripe results in lag when watching the stream.

A direct comparison between Pastry-style and CAN-style[1] overlay networks can be found in [12].

4.1.4 Bullet

The Bullet approach [13] tries to combine the tree structure with an overlaid mesh in order to boost the bandwidth at which the stream is delivered. They argue that an additional distribution structure is needed because any bandwidth bottlenecks in the upper part of the tree limit the maximum bandwidth of the receivers lower down the tree.

The key idea of Bullet is to subdivide the stream into sequential blocks which *"are further subdivided into individual objects which are in turn transmitted to different points in the network. Nodes still receive a set of objects from their parents, but they are then responsible for locating peers that hold missing data objects." [13]* The goal is to spread the data in the network as uniformly as possible in order to avoid bottlenecks.

The obvious advantage is that each node has more than just one predecessor and can collect missing data items similarly to BitTorrent. There is no tracker or similar central instance that can tell the peers which neighbours they could ask for missing packets. The authors state that Bullet uses a *"scalable and efficient algorithm to enable nodes to quickly locate multiple peers capable of transmitting missing data items." [13]* This algorithm is called *RanSub* and addresses the challenge of locating disjoint content within the system.

[1]Overlays which use a numerical distance metric to route through a Cartesian hyper-space.

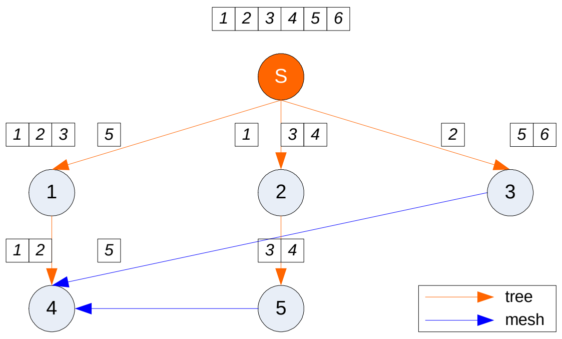


Figure 4.2: High-level view of the Bullet network which consists of a tree with a mesh overlaid.

Peers tell each other which packets they have by sending around sophisticated summaries which allow the receivers to send back requests for data they need. This is as well quite similar to the pull-based BitTorrent approach.

They make allowance to the fact that the important thing in streaming is to assure a continuous data flow and not to maximise the throughput. Therefore they suggest the use of the *TCP-friendly rate control* which is defined in RFC 3448 [14]. It is basically a congestion control mechanism that offers a lower variation of the throughput over time compared with normal TCP flows.

Drawbacks

As with all highly structured networks, there is a rather high maintenance and creation overhead. One has to pay special attention, when there is a lot of churn, not to spend too much time on maintaining the structure and therefore neglecting the actual data stream. Otherwise the watching experience will be rather unpleasant.

In addition to that, the systems needs some very complex and failure prone algorithms which might not perform all too well in real-life situations.

4.1.5 Peer-to-Peer Radio

A very simple solution to audio streaming which works rather well in practise and which has encountered some interest on the Internet. It can be found at [15].

The main focus of the project was on eliminating freeloaders, i.e. peers which do not contribute. The basic structure is a tree in which each peer can accuse other peers of being freeloaders. Upon joining, the new peer starts at the root of the tree and continues in a top-down approach trying to find a peer on its way to the leaf that can accept him as a new child.

To ensure the data integrity, each packet is signed by the source which induces a large overhead especially since the source already has quite some work to do with producing the packets and responding to new joining neighbours.

Another very similar system is *PeerCast* [16] which also supports P2P broadcasting. It is not very popular though.

Drawbacks

The major drawback of the Peer-to-Peer Radio is that it is based an very simple tree structure which needs a lot of maintenance and offers no flexibility at all.

4.1.6 Incentives-Compatible Peer-to-Peer Multicast

This paper [17] proposes a multiple tree overlay structure to broadcast a media stream. The authors focus on the problem of selfish behaviour among peers by presenting *“mechanisms that can distinguish nodes with selfish behaviour and reduce the quality of service experienced by these selfish nodes from their peers.”*

Drawbacks

Since the overlay network is based on a tree structure and since the data flow inside a tree is always top-down they need to rebuild the tree periodically so that a downloading peer that has a malicious parent can become an uploading peer and take revenge on the the former uploading peer by refusing to upload to him now. This of course introduces a large maintenance overhead.

Another problem of this approach lies in the decision which peer is really a malicious peer and needs to be punished or even excluded. It is usually quite hard to tell whether some peer makes false accusations.

Another feature that seems to be necessary here is the authentication of the path through which the data has come as well as authentication of the data itself. They suggest the use of one-way hash functions for this purpose, which add even more overhead.

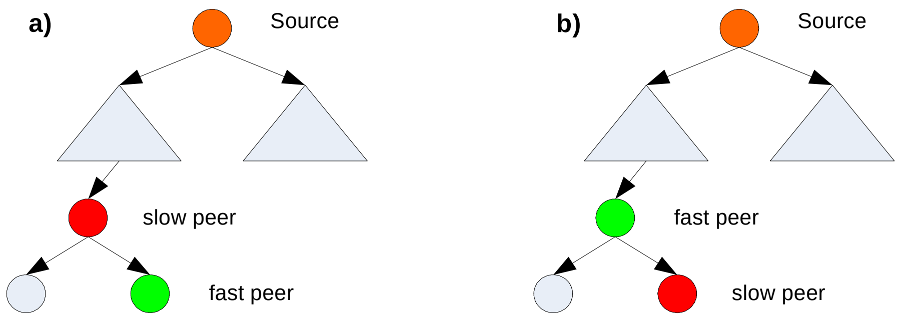


Figure 4.3: a) The fast peer receives the stream from the slow peer and there is nothing he can do about it. b) The tree has been rebuilt and the fast peer is now the parent of the slow node.

4.1.7 Chainsaw

This approach [18] is closer to our solution. The main difference between Chainsaw and Split-stream in terms of data distribution is that Chainsaw uses a pull-based approach instead of a push-based. This means that peers are notified of new packets by their neighbours and must explicitly request a packet from them in order to receive it. This approach has the advantage that we can minimize the amount of duplicate packets that are sent but at the same time have a mechanism to easily recover from lost or delayed packets—we can just re-request them from another neighbour. In addition to that, we do not need a global routing algorithm because each node only has to know which packets are available in his direct neighbourhood. In terms of network structure, they are completely different since Chainsaw does not use trees but a rather random overlay topology. We therefore do not need to perform routing updates if a node decides to leave the network.

A small drawback of this approach is that the stream first has to be divided into finite packets which at the receiver side have to be recomposed into a stream again.

The idea of the pull-based approach comes from BitTorrent [1]. Swistry differs from BitTorrent in that there is no seeder which has the complete data but rather a source that is producing a real-time stream. So the maximum download rate is limited by the bitrate of the stream. It makes no sense trying to be faster than that since there is not more data available at each point in time.

The source of the stream creates packets with increasing sequence numbers and announces them to their direct neighbours. The drawback of the data being produced live is that the window of interest (the number of currently relevant packets) is quite small if we try to distribute the stream with as little delay as possible.

Chainsaw suggests to request the available packets that are inside the window of interest at random. If packets were requested with some sort of priority (earlier packets would have higher priority than later ones) one could probably further improve the performance.

Chainsaw does not make any suggestions on how to choose each peer's neighbourhood which is vital to the good performance of the system. A strategy for requesting the packets other than just requesting them at random is not specified either.

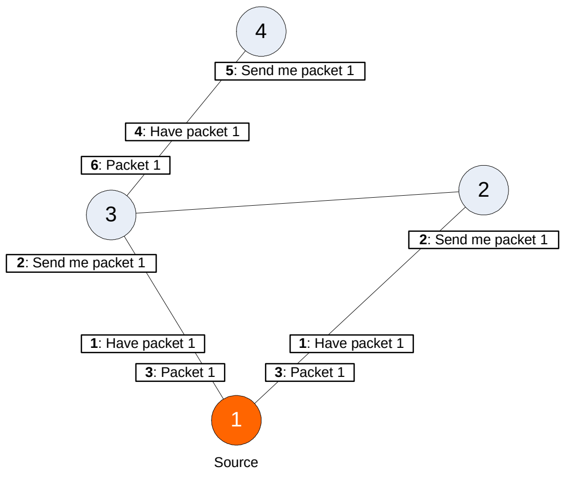


Figure 4.4: Chainsaw approach: Notification, request and finally packet delivery.

Drawbacks

Since there are three rounds of communication necessary in order to get a packet, the delay between the creation and the propagation is higher than in a push-based approach.

Another drawback is that each node has to keep old packets in case a neighbour needs them. If the delay between individual nodes is high, the rate of duplicate packets sent will increase because of several expired timeouts at the receiver side.

The lack of a global view might lead to a disconnected network if there are a lot of peers joining and leaving all the time and the data flow is not optimal either because the topology is not optimised in any way, on the contrary, it is completely random.

4.1.8 A Self-Repairing Peer-to-Peer System Resilient to Dynamic Adversarial Churn

This work [19] addresses the problem of maintaining desired network properties such as a low diameter and a low peer degree when facing a powerful adversary which constantly adds and removes peers. A stable P2P streaming system has to be somehow resilient to normal churn since we must expect users to join and leave quite frankly.

The authors suggest the use of a hypercube as network structure in which each node of the hypercube consists of several peers. Each peer has connections to other peers inside his node (short-range) and to nodes in adjacent hypercube nodes (long-range). They do not directly address the problem of streaming live content but describe a very stable P2P structure which should be used in future Swistry implementations to ensure connectivity and good network properties.

4.2 Multiple Description Coding

MDC [20] is a technique that, if well implemented, would bring some advantages for live video streaming. The idea of MDC is to split the original stream into several substreams which all contain a part of the original stream. If there is more than just one substream available they can be combined together in order to improve the quality. The original picture can only be restored if all substreams are available. The drawback of this approach is that it is quite hard to do this on real-time video.

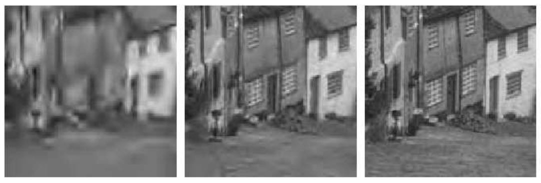


Figure 4.5: An example of image enhancement using MDC (images taken from [20]).

This technique is especially helpful in case of unreliable transport channels like the ones we find in the Internet (e.g. UDP). There is no need for retransmission which will cause additional delay in case some data from a substream arrives too late. The stream will just be reconstructed using the remaining available data with a small loss in media quality. It also allows a stream to be received in various qualities according to the available bandwidth. A slow peer can simply download fewer substreams than a fast peer. No extra infrastructure is needed for that. Nevertheless, in order to assure a certain robustness to the loss of parts of the substreams, MCD coding must sacrifice some compression efficiency. V.K.Goyal states in [20] that *"MDC coding is difficult because of conflicting requirements. If you design a good description at rate R_1 to send over Channel 1 and another good description at rate R_2 to send over Channel 2, there is no reason for the two descriptions together to be a good way to spend $R_1 + R_2$ total bits. Similarly, a good compression representation at rate $R_1 + R_2$ cannot easily be split into two useful descriptions."*

There are two well known ways to do MDC. The first uses a base layer (substream) which contains the basic stream information. The video stream can be watched if only the base layer is available but the quality is rather poor. In order to improve the quality and ultimately to reconstruct the original stream quality, a set of enhancement layers is used. Each of them enhances the video stream when combined with the base layer. So the more enhancement layers a peer can receive the better the final picture. Although, it is not possible to reconstruct any picture without the base layer. The advantage of this approach is that the encoding is simpler. It is used if there is a reliable channel over which the base layer can be sent and a set of unreliable channels which can be used for the enhancement layers.

The other way is to split the original stream into a set of equally important layers. The video stream can be reconstructed using at least one of the broadcast layers but it does not matter which one it is. A combination of several layers improves the picture quality as well. The drawback here is the more complicated encoding. All the details of the various possibilities to do MDC can be found in the original article by V.K.Goyal [20]. [21] contains information concerning video streaming with MDC.

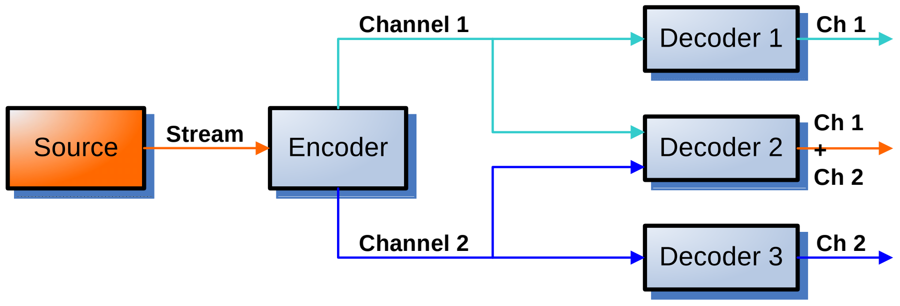


Figure 4.6: Example layout of an MDC architecture.

4.2.1 CoolStreaming/DONet

DONet stands for data-driven overlay network. The project [22] does not suggest any complicated topologies which have to be maintained but still tries to build an overlay network which is robust and resilient to churn. The implementation they released on the Internet in 2004 was known as *CoolStreaming*. A core concept of DONet is the data-centric design. There is nothing like a father-child relation nor is there any other form of given path through which the data flows. Whoever needs something, requests it from a neighbour that claims to possess it—again the BitTorrent approach.

The CoolStreaming implementation was fairly successful in Asian countries and it became quite popular. This is probably because they have much more synchronous high-bandwidth connections there which allow them to broadcast video streams in better quality. Modern versions of this software are still in use.

4.3 Firewall and NAT Traversal

Almost everybody is behind a firewall or a NAT router today. P2P systems have incoming as well as outgoing connections that are often initiated by a remote peer. Intrusion prevention systems like the aforementioned therefore try to block these unexpected connection attempts.

Since a P2P application should be as comfortable as possible for the end-user, a mechanism is needed that can circumvent at least the NAT boxes. Many solutions that address this problem have been proposed.

4.3.1 NUTSS: A SIP-based Approach to UDP and TCP Network Connectivity

Problems arise if peers are behind NAT boxes instead of being connected directly to the Internet. If the hosts want to communicate using UDP or if both hosts are behind NAT boxes, things get even more complicated. NUTSS [23] describes a way of how to establish TCP connections through NAT. It uses techniques like hole-punching and includes a STUN server to spoof IP addresses. [23] gives a detailed explanation of the system.

Since nowadays a lot of people use NAT boxes and firewalls, it has become vital that a user-friendly system can deal with these obstacles by itself.

4.3.2 An Analysis of the Skype Peer-to-Peer Internet Telephony Protocol

This paper [24] takes a closer look at how Skype [25] is handling NAT issues. They can communicate using either UDP or TCP. First they try to establish a UDP connection and if that fails, a TCP connection attempt is made. The first attempt is made using a random port and if that does not work, well known ports that are probably not blocked like 80 (HTTP) or 443 (HTTPS) are tried. If both peers are behind a NAT box, a so called *supernode* which is directly connected to the Internet is used as a communication proxy.

The idea is sketched in Figures 4.7 and 4.8.

In the first scenario, only peer 1 is behind a NAT box so there is no problem for him to establish a connection to peer 2. The opposite is not true: Peer 2 cannot directly connect to peer 1.

The second scenario is more complicated because now both peers are behind NAT boxes. It is therefore not possible for one side to directly connect to the other side. In step 1, peer 1 tries to establish a connection to peer 2 which is blocked by its NAT. This step is called *hole punching* because peer 1 has now enforced a port mapping in his NAT which can be used by peer 2 to connect to him. All he has to do now is use the proxy, to which both peers have a connection since he is not behind NAT, and tell him which port in his NAT is mapped. The proxy then forwards this information to peer 2 who can now, in step 3, establish a connection to peer 1 using that very port.

In reality it is not easy because the NAT box of peer 1 might delete the mapping again if the connection attempt of step 1 did not succeed. Details can be found in [24] or [23].

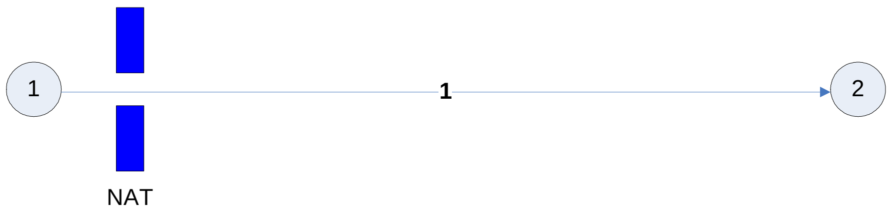


Figure 4.7: Only peer 1 is behind an NAT box.

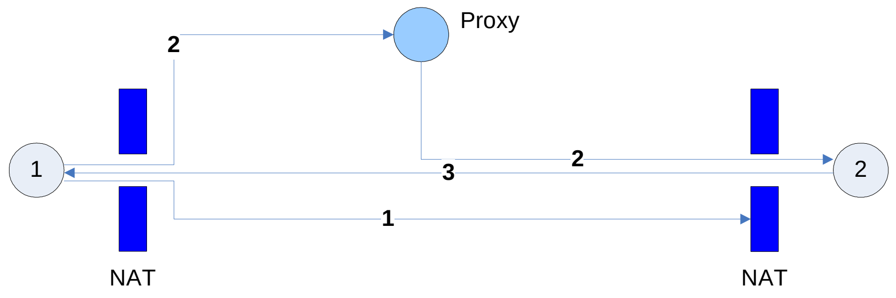


Figure 4.8: Both peers are behind NAT boxes and they are both connected to a proxy which is not behind a NAT box.

4.4 Further Helpful Work

4.4.1 Exploiting Internet Route Sharing for Large Scale Available Bandwidth Estimation

In Swistry and in some other P2P networks as well, it is important to have a mechanism to roughly estimate the available bandwidth. [2] gives some insight on how this can be achieved using an algorithm called BRoute which is based on a route sharing model. This is not yet implemented in the current version though.

Chapter 5

Future Work

At the moment, the individual levels within a Swistry network are not interconnected in any way. This is basically because each of them is broadcasting its own stream. The content is the same but the qualities differ. It might be nevertheless useful to interconnect them. Peers in upper levels that have enough bandwidth might help distributing data in lower ones. Care has to be taken not to mix up data from two different levels because the packets are part of different streams that do not need to be synchronized.

The Swistry users have to know or test by themselves what their available bandwidth is. The network entry point does not validate the reported values either. It would be safer and less cumbersome, if upon joining a bandwidth estimation process would determine and set the connection capacity for the new peers. In order to get accurate results, several connection tests with different end-points should be performed. The drawback of this approach can be seen in the Zattoo [26] application. If this bandwidth estimation procedure is not smart enough, it might erroneously assign false bandwidths. This would result in fast peers ending up in low levels with worse streaming qualities and vice versa. It should therefore still be possible to override the estimated values with user-specified ones.

In future releases, persistency should be added to the peer application. If a peer has been dropped because he was the worst available, he can just re-join with the current implementation and the node that has dropped him will not remember that he was a bad neighbour. A time restricted memory is needed because a node that was not uploading fast enough a while ago could have been overloaded. Such a node has to be given a chance to reconnect. Otherwise he might get banned by all peers forever which would completely prevent him from receiving the stream.

A hypercube on top of the fat mesh might render the topology more stable and more controllable. Also long range contacts could be included if necessary. Each peer would then be assigned a unique identification number upon joining. Based on that, a push-based initial distribution of new packets would become possible: As soon as the source creates a new packet, it sends it to a certain part of its direct neighbours. The rest will be informed using the notification mechanism. This approach will reduce the initial delay quite a bit.

As mentioned earlier, it is important that the data stream flows smoothly through the network. If switching from UDP to TCP for the stream distribution, a TCP-friendly algorithm should be used in order to reduce the variation of the throughput.

Currently, the users need to configure their NAT boxes manually. It would be easier though if Swistry had a mechanism similar to Skype, that is able to bypass these devices. Propositions can be found in [24] and [23].

The `DePacketizer` should be improved. More sophisticated buffering algorithms and timeouts would further reduce lags and improve the watching experience. If currently requested packets are delayed, the timeout till skipping these packets could be set with respect to the fill level of the buffer for example. A good `DePacketizer` strategy is particularly important when streaming over UDP with a lot of very small packets.

The neighbourhood criteria should be investigated further as well. At the moment we are choosing our neighbours based only on the bandwidth. A combination of bandwidth and delay might be more appropriate. The first step in this direction would be to implement the above mentioned bandwidth estimation process which could be expanded to take the delay into account.

Chapter 6

Acknowledgements

I would like to specially thank Prof. Dr. Roger Wattenhofer and my supervisors Stefan Schmid and Thomas Locher for all their support during the last six months. Without their help, Swistry would not exist today.

Bibliography

[1] B.Cohen. Incentives Build Robustness in BitTorrent. In *First Workshop on the Economics of Peer-to-Peer Systems*, 2003.

[2] N.Hu and P.Steenkiste. Exploiting Internet Route Sharing for Large Scale Available Bandwidth Estimation. In *INFOCOM'05*, 2005.

[3] VideoLAN media player. http://www.videolan.org.

[4] SHOUTcast. http://www.shoutcast.org.

[5] M.Stoer and F.Wagner. A Simple Min-Cut Algorithm. In *Journal of the ACM, 44(4):585-591*, 1997.

[6] Hauppauge. http://www.hauppauge.de.

[7] T.Nguyen and A.Zakhor. Distributed Video Streaming with Forward Error Correction. In *Proc. Packet Video Workshop, Pittsburgh, USA*, 2002.

[8] N.Liogkas and R.Nelson and E.Kohler and L.Zhang. Exploiting BitTorrent for Fun (but not Profit). In *5th International Workshop on Peer-To-Peer Systems (IPTPS)*, 2006.

[9] M.Castro and P.Druschel and A.Kermarrec and A.Nandi and A.Rowstron and A.Singh. SplitStream: High-Bandwidth Content Distribution in a Cooperative Environment. In *19th ACM Symposium on Operating Systems Principles*, 2003.

[10] M.Castro and P.Druschel and A.Kermarrec and A.Rowstron. Scribe: A Large-Scale and Decentralized Application-Level Multicast Infrastructure. In *IEEE Journal on Selected Areas in Communications (JSAC)*, 2002.

[11] A.Rowstron and P.Druschel. Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems. In *Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, 2001.

[12] M.Castro and M.Jones and A.Kermarrec and A.Rowstron and M.Theimer and H.Wang and A.Wolman. An Evaluation of Scalable Application-Level Multicast Built Using Peer-to-Peer Overlay Networks. In *IEEE INFOCOM*, 2003.

[13] D.Kostic and A.Rodriguez and J.Albrecht and A.Vahdat. Bullet: High Bandwidth Data Dissemination Using an Overlay Mesh. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003.

[14] M.Handley and J.Pahdye and S.Floyd and J.Widmer. TCP Friendly Rate Control (TFRC): Protocol Specification. In *Work in progress (Internet-Draft draft-ietf-tsvwg-tfrc-02.txt)*, 2001.

[15] M.Kaufmann. Peer-to-Peer Radio mit Erkennung von Freeloadern. 2003.

[16] PeerCast P2P Broadcasting. http://www.peercast.org.

[17] T.Ngan and D.Wallach and P.Druschel. Incentives-Compatible Peer-to-Peer Multicast. In *2nd Workshop on Economics of Peer-to-Peer Systems*, 2004.

[18] V.Pai and K.Kumar and K.Tamilmani and V.Sambamurthy and A.Mohr. Chainsaw: Eliminating Trees from Overlay Multicast. In *4th International Workshop on Peer-To-Peer Systems (IPTPS)*, 2005.

[19] F.Kuhn and S.Schmid and R.Wattenhofer. A Self-Repairing Peer-to-Peer System Resilient to Dynamic Adversarial Churn. In *4th International Workshop on Peer-To-Peer Systems (IPTPS)*, 2005.

[20] V.Goyal. Multiple Description Coding: Compression Meets the Network. In *IEEE Signal Processing Magazine*, 2001.

[21] S.Servetto and K.Nahrstedt. Video Streaming Over the Public Internet: Multiple Description Codes and Adaptive Transport Protocols. In *ICIP (3)*, 1999.

[22] X.Zhang and J.Liu and B.Li and T.Yum. CoolStreaming/DONet: A Data-Driven Overlay Network for Efficient Live Media Streaming. In *INFOCOM'05*, 2005.

[23] S.Guha and Y.Takeda and P.Francis. NUTSS: A SIP-based Approach to UDP and TCP Network Connectivity. In *SIGCOMM'04 Workshops*, 2004.

[24] S.Baset and H.Schulzrinne. An Analysis of the Skype Peer-to-Peer Internet Telephony Protocol. In *INFOCOM'06*, 2004.

[25] Skype. http://www.skype.com.

[26] Zattoo. http://www.zattoo.com.

[27] C.Zhang and A.Krishnamurthy and R.Wang. Brushwood: Distributed Trees in Peer-to-Peer Systems. In *4th International Workshop on Peer-To-Peer Systems (IPTPS)*, 2005.

[28] V.Padmanabhan and H.Wang and P.Chou. Resilient Peer-to-Peer Streaming. In *IEEE ICNP*, 2003.

[29] V.Venkataraman and P.Francis and J.Calandrino. Chunkyspread: Multi-tree Unstructured Peer-to-Peer Multicast. In *5th International Workshop on Peer-To-Peer Systems (IPTPS)*, 2006.

List of Figures

Appendix A

Protocols and Message Definitions

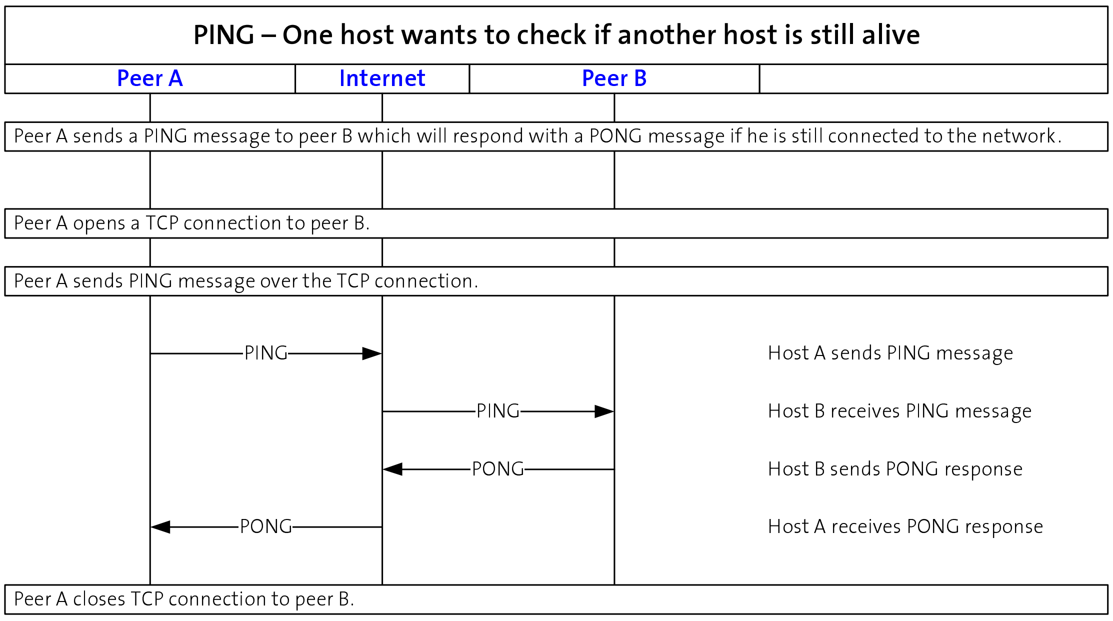

PING – One host wants to check if another host is still alive
Peer A
Internet
Peer B
Peer A sends a PING message to peer B which will respond with a PONG message if he is still connected to the network.
Peer A opens a TCP connection to peer B.
Peer A sends PING message over the TCP connection.
PING
Host A sends PING message
PING
Host B receives PING message
PONG
Host B sends PONG response
PONG
Host A receives PONG response
Peer A closes TCP connection to peer B.

Figure A.1: Ping protocol to test if a neighbour is still alive.

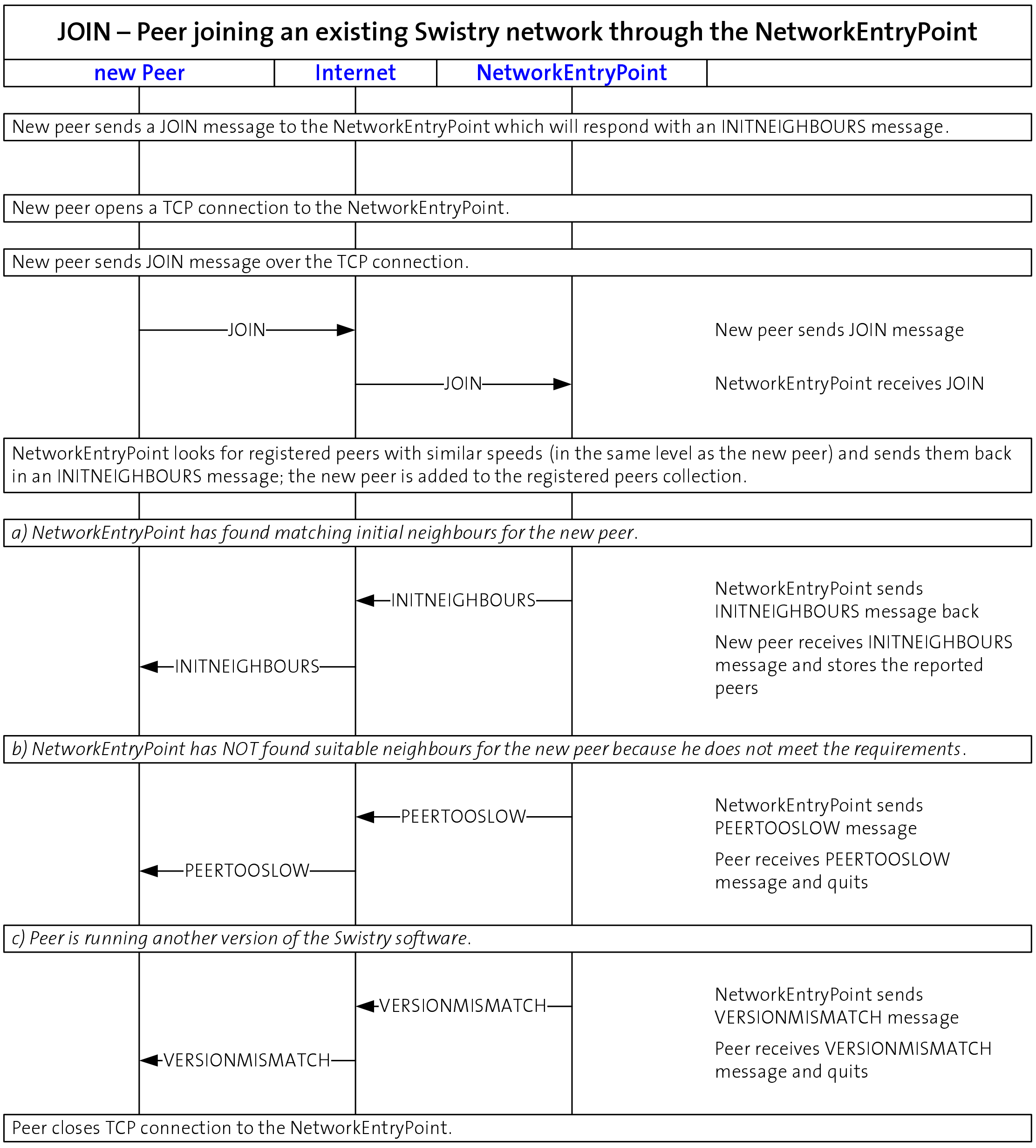
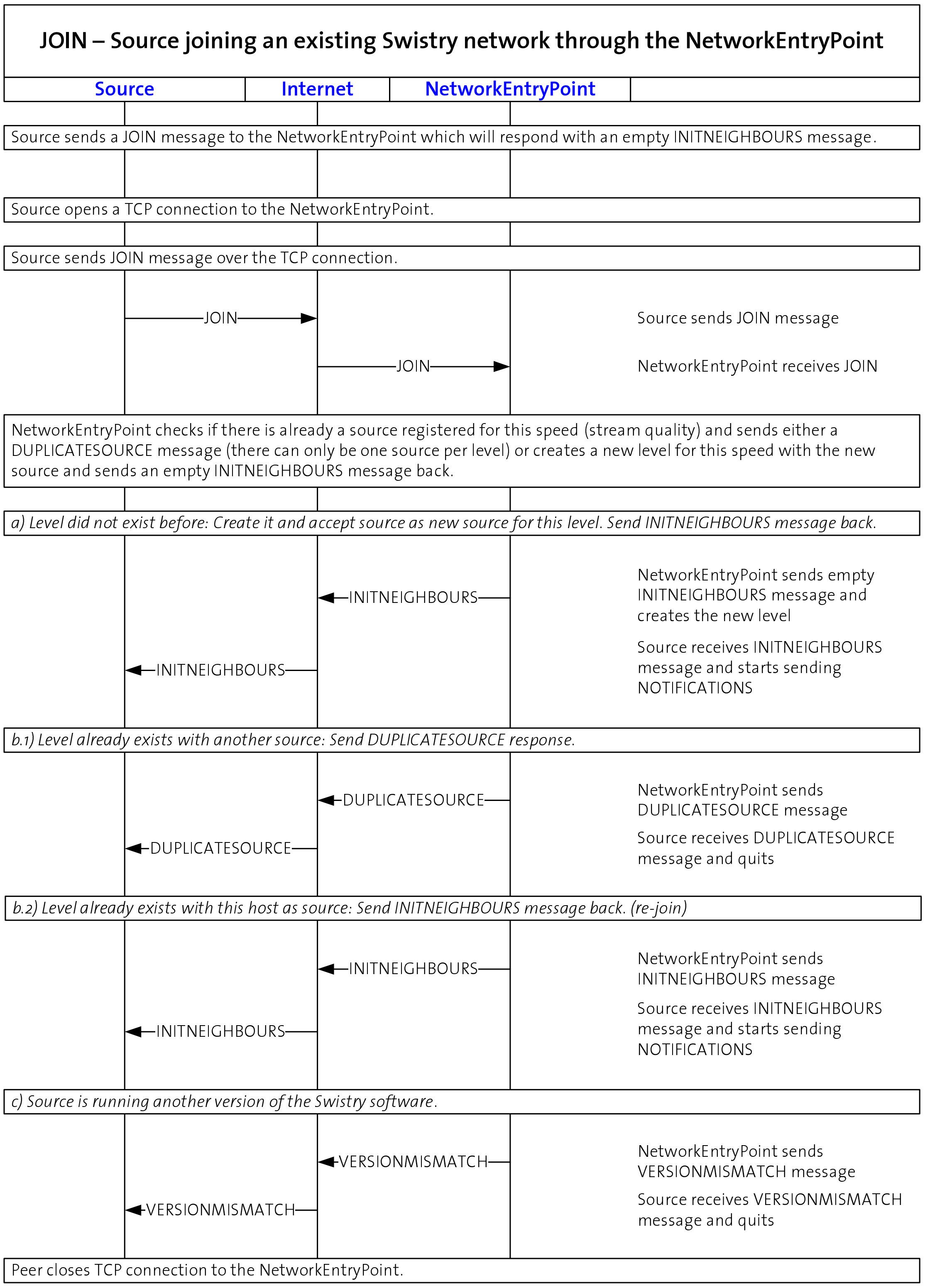

JOIN – Peer joining an existing Swistry network through the NetworkEntryPoint
new Peer
Internet
NetworkEntryPoint
New peer sends a JOIN message to the NetworkEntryPoint which will respond with an INITNEIGHBOURS message.
New peer opens a TCP connection to the NetworkEntryPoint.
New peer sends JOIN message over the TCP connection.
JOIN
New peer sends JOIN message
JOIN
NetworkEntryPoint receives JOIN
NetworkEntryPoint looks for registered peers with similar speeds (in the same level as the new peer) and sends them back in an INITNEIGHBOURS message; the new peer is added to the registered peers collection.
a) NetworkEntryPoint has found matching initial neighbours for the new peer.
INITNEIGHBOURS
NetworkEntryPoint sends INITNEIGHBOURS message back
INITNEIGHBOURS
New peer receives INITNEIGHBOURS message and stores the reported peers
b) NetworkEntryPoint has NOT found suitable neighbours for the new peer because he does not meet the requirements.
PEERTOOSLOW
NetworkEntryPoint sends PEERTOOSLOW message
PEERTOOSLOW
Peer receives PEERTOOSLOW message and quits
c) Peer is running another version of the Swistry software.
VERSIONMISMATCH
NetworkEntryPoint sends VERSIONMISMATCH message
VERSIONMISMATCH
Peer receives VERSIONMISMATCH message and quits
Peer closes TCP connection to the NetworkEntryPoint.

Figure A.2: Join protocol followed by the peers upon joining a new Swistry network.



JOIN – Source joining an existing Swistry network through the NetworkEntryPoint

Source sends a JOIN message to the NetworkEntryPoint which will respond with an empty INITNEIGHBOURS message.

Source opens a TCP connection to the NetworkEntryPoint.

Source sends JOIN message over the TCP connection.

Source sends JOIN message

NetworkEntryPoint receives JOIN

NetworkEntryPoint checks if there is already a source registered for this speed (stream quality) and sends either a DUPLICATESOURCE message (there can only be one source per level) or creates a new level for this speed with the new source and sends an empty INITNEIGHBOURS message back.

a) Level did not exist before: Create it and accept source as new source for this level. Send INITNEIGHBOURS message back.

NetworkEntryPoint sends empty INITNEIGHBOURS message and creates the new level

Source receives INITNEIGHBOURS message and starts sending NOTIFICATIONS

b.1) Level already exists with another source: Send DUPLICATESOURCE response.

NetworkEntryPoint sends DUPLICATESOURCE message

Source receives DUPLICATESOURCE message and quits

b.2) Level already exists with this host as source: Send INITNEIGHBOURS message back. (re-join)

NetworkEntryPoint sends INITNEIGHBOURS message

Source receives INITNEIGHBOURS message and starts sending NOTIFICATIONS

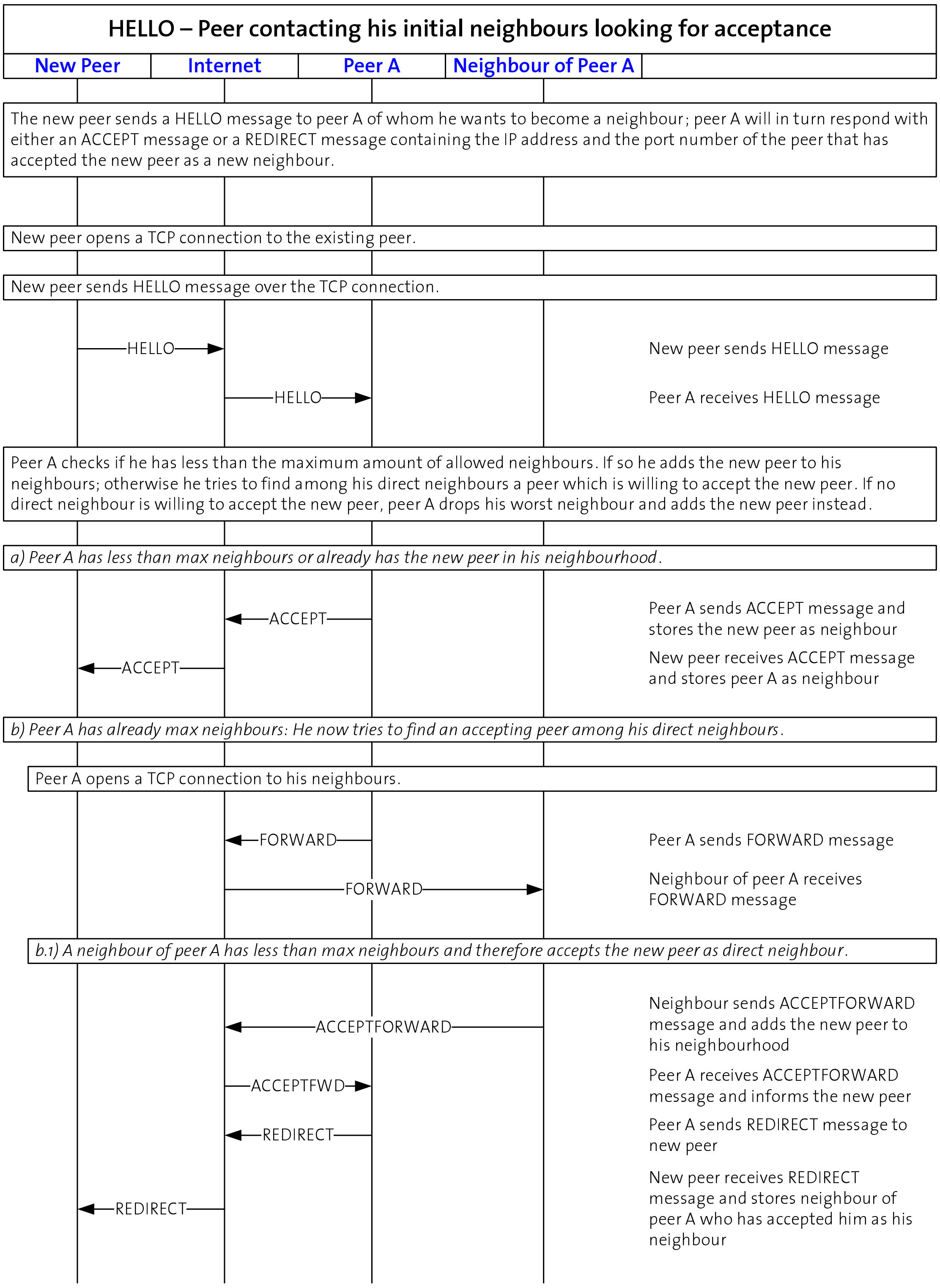
c) Source is running another version of the Swistry software.

NetworkEntryPoint sends VERSIONMISMATCH message

Source receives VERSIONMISMATCH message and quits

Peer closes TCP connection to the NetworkEntryPoint.

Figure A.3: Join protocol followed by the source upon joining a new Swistry network.



HELLO – Peer contacting his initial neighbours looking for acceptance

The new peer sends a HELLO message to peer A of whom he wants to become a neighbour; peer A will in turn respond with either an ACCEPT message or a REDIRECT message containing the IP address and the port number of the peer that has accepted the new peer as a new neighbour.

New peer opens a TCP connection to the existing peer.

New peer sends HELLO message over the TCP connection.

New peer sends HELLO message

Peer A receives HELLO message

Peer A checks if he has less than the maximum amount of allowed neighbours. If so he adds the new peer to his neighbours; otherwise he tries to find among his direct neighbours a peer which is willing to accept the new peer. If no direct neighbour is willing to accept the new peer, peer A drops his worst neighbour and adds the new peer instead.

a) Peer A has less than max neighbours or already has the new peer in his neighbourhood.

Peer A sends ACCEPT message and stores the new peer as neighbour

New peer receives ACCEPT message and stores peer A as neighbour

b) Peer A has already max neighbours: He now tries to find an accepting peer among his direct neighbours.

Peer A opens a TCP connection to his neighbours.

Peer A sends FORWARD message

Neighbour of peer A receives FORWARD message

b.1) A neighbour of peer A has less than max neighbours and therefore accepts the new peer as direct neighbour.

Neighbour sends ACCEPTFORWARD message and adds the new peer to his neighbourhood

Peer A receives ACCEPTFORWARD message and informs the new peer

Peer A sends REDIRECT message to new peer

New peer receives REDIRECT message and stores neighbour of peer A who has accepted him as his neighbour

Figure A.4: Hello protocol which is used to register with new neighbours.

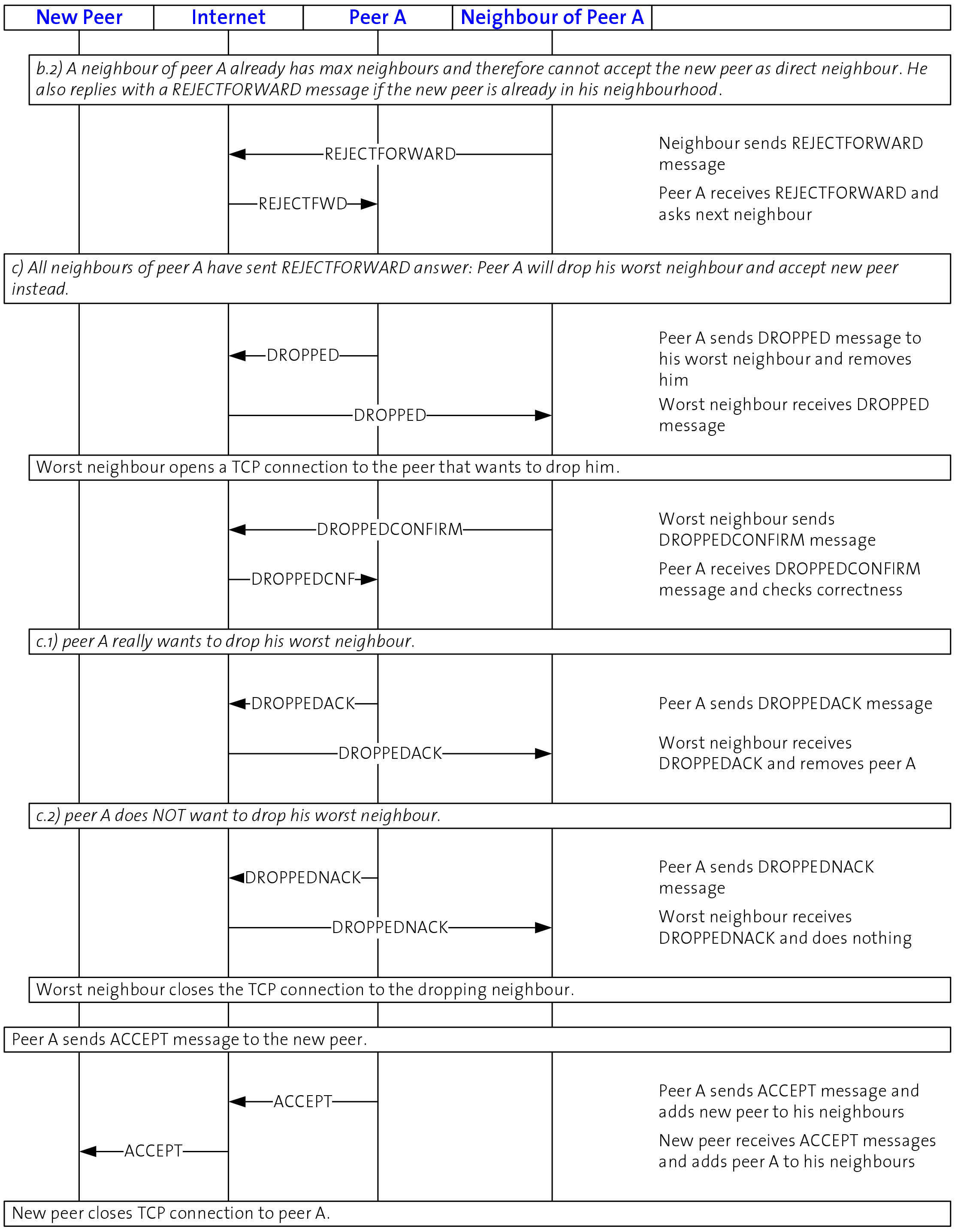

New Peer
Internet
Peer A
Neighbour of Peer A
b.2) A neighbour of peer A already has max neighbours and therefore cannot accept the new peer as direct neighbour. He also replies with a REJECTFORWARD message if the new peer is already in his neighbourhood.
REJECTFORWARD
REJECTFWD
Neighbour sends REJECTFORWARD message
Peer A receives REJECTFORWARD and asks next neighbour
c) All neighbours of peer A have sent REJECTFORWARD answer: Peer A will drop his worst neighbour and accept new peer instead.
DROPPED
DROPPED
Peer A sends DROPPED message to his worst neighbour and removes him
Worst neighbour receives DROPPED message
Worst neighbour opens a TCP connection to the peer that wants to drop him.
DROPPEDCONFIRM
DROPPEDCNF
Worst neighbour sends DROPPEDCONFIRM message
Peer A receives DROPPEDCONFIRM message and checks correctness
c.1) peer A really wants to drop his worst neighbour.
DROPPEDACK
DROPPEDACK
Peer A sends DROPPEDACK message
Worst neighbour receives DROPPEDACK and removes peer A
c.2) peer A does NOT want to drop his worst neighbour.
DROPPEDNACK
DROPPEDNACK
Peer A sends DROPPEDNACK message
Worst neighbour receives DROPPEDNACK and does nothing
Worst neighbour closes the TCP connection to the dropping neighbour.
Peer A sends ACCEPT message to the new peer.
ACCEPT
ACCEPT
Peer A sends ACCEPT message and adds new peer to his neighbours
New peer receives ACCEPT messages and adds peer A to his neighbours
New peer closes TCP connection to peer A.

Figure A.5: Hello protocol which is used to register with new neighbours (continued).

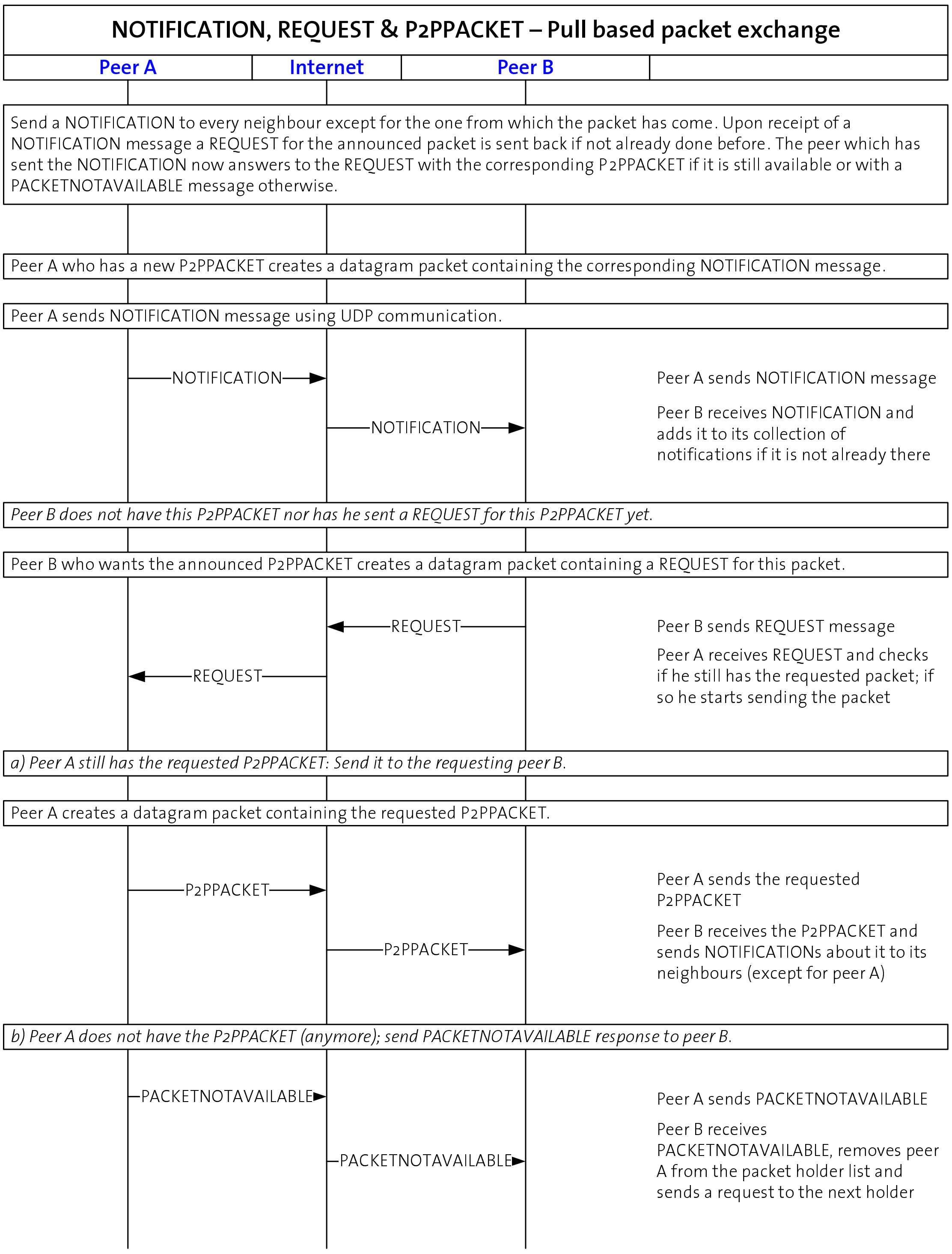

NOTIFICATION, REQUEST & P2PPACKET – Pull based packet exchange
Peer A
Internet
Peer B
Send a NOTIFICATION to every neighbour except for the one from which the packet has come. Upon receipt of a NOTIFICATION message a REQUEST for the announced packet is sent back if not already done before. The peer which has sent the NOTIFICATION now answers to the REQUEST with the corresponding P2PPACKET if it is still available or with a PACKETNOTAVAILABLE message otherwise.
Peer A who has a new P2PPACKET creates a datagram packet containing the corresponding NOTIFICATION message.
Peer A sends NOTIFICATION message using UDP communication.
NOTIFICATION
NOTIFICATION
Peer A sends NOTIFICATION message
Peer B receives NOTIFICATION and adds it to its collection of notifications if it is not already there
Peer B does not have this P2PPACKET nor has he sent a REQUEST for this P2PPACKET yet.
Peer B who wants the announced P2PPACKET creates a datagram packet containing a REQUEST for this packet.
REQUEST
REQUEST
Peer B sends REQUEST message
Peer A receives REQUEST and checks if he still has the requested packet; if so he starts sending the packet
a) Peer A still has the requested P2PPACKET: Send it to the requesting peer B.
Peer A creates a datagram packet containing the requested P2PPACKET.
P2PPACKET
P2PPACKET
Peer A sends the requested P2PPACKET
Peer B receives the P2PPACKET and sends NOTIFICATIONs about it to its neighbours (except for peer A)
b) Peer A does not have the P2PPACKET (anymore); send PACKETNOTAVAILABLE response to peer B.
PACKETNOTAVAILABLE
PACKETNOTAVAILABLE
Peer A sends PACKETNOTAVAILABLE
Peer B receives PACKETNOTAVAILABLE, removes peer A from the packet holder list and sends a request to the next holder

Figure A.6: Packet exchange protocol: The P2PPackets exchanged contain the actual stream data.

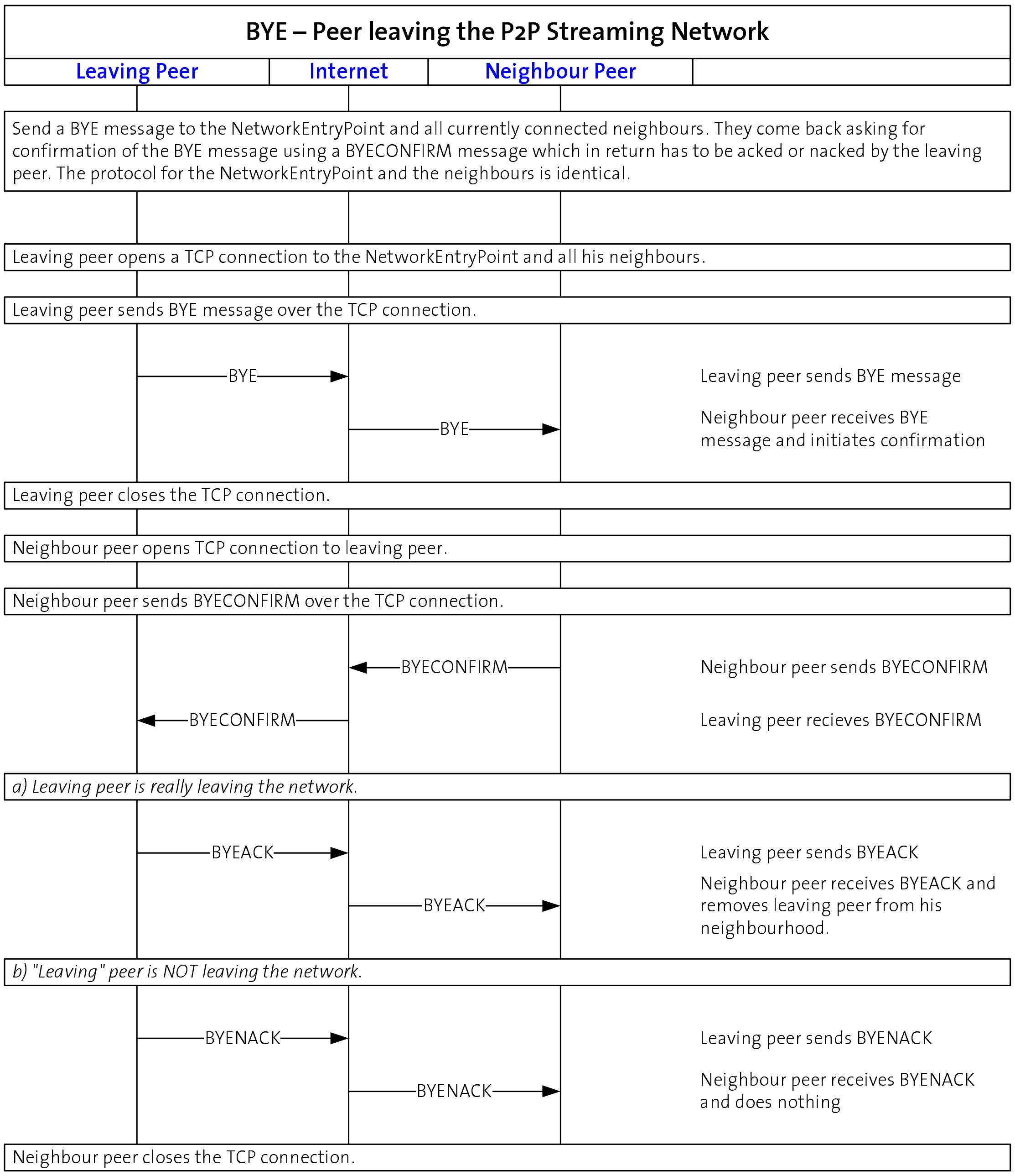
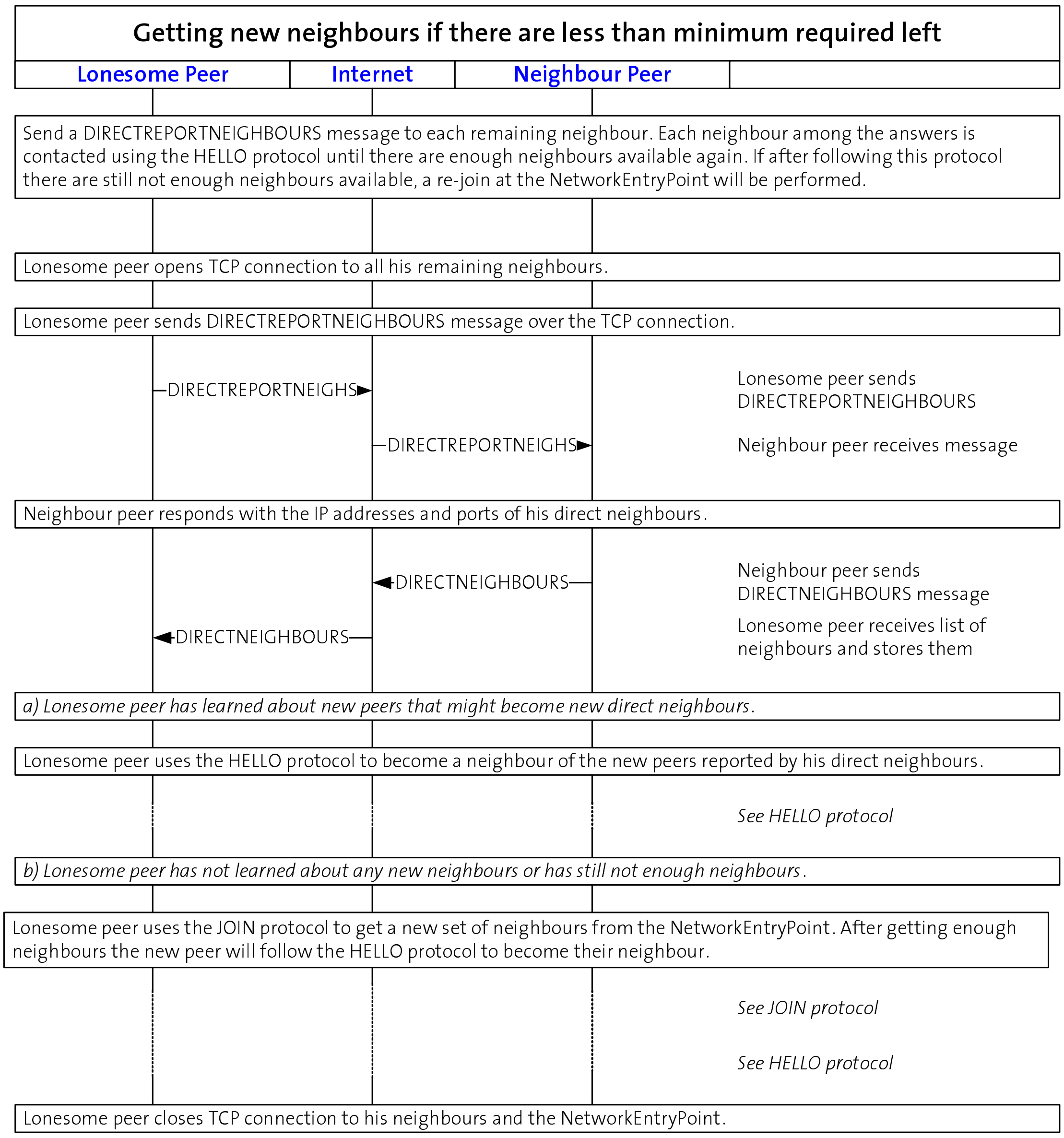

BYE – Peer leaving the P2P Streaming Network
Leaving Peer
Internet
Neighbour Peer
Send a BYE message to the NetworkEntryPoint and all currently connected neighbours. They come back asking for confirmation of the BYE message using a BYECONFIRM message which in return has to be acked or nacked by the leaving peer. The protocol for the NetworkEntryPoint and the neighbours is identical.
Leaving peer opens a TCP connection to the NetworkEntryPoint and all his neighbours.
Leaving peer sends BYE message over the TCP connection.
BYE
BYE
Leaving peer sends BYE message
Neighbour peer receives BYE message and initiates confirmation
Leaving peer closes the TCP connection.
Neighbour peer opens TCP connection to leaving peer.
Neighbour peer sends BYECONFIRM over the TCP connection.
BYECONFIRM
BYECONFIRM
Neighbour peer sends BYECONFIRM
Leaving peer recieves BYECONFIRM
a) Leaving peer is really leaving the network.
BYEACK
BYEACK
Leaving peer sends BYEACK
Neighbour peer receives BYEACK and removes leaving peer from his neighbourhood.
b) "Leaving" peer is NOT leaving the network.
BYENACK
BYENACK
Leaving peer sends BYENACK
Neighbour peer receives BYENACK and does nothing
Neighbour peer closes the TCP connection.

Figure A.7: Bye protocol followed by all nodes that leave the network (be it source or peer) in order to tell their neighbours that they have left.



Send a DIRECTREPORTNEIGHBOURS message to each remaining neighbour. Each neighbour among the answers is contacted using the HELLO protocol until there are enough neighbours available again. If after following this protocol there are still not enough neighbours available, a re-join at the NetworkEntryPoint will be performed.

Lonesome peer opens TCP connection to all his remaining neighbours.

Lonesome peer sends DIRECTREPORTNEIGHBOURS message over the TCP connection.

Lonesome peer sends DIRECTREPORTNEIGHBOURS

Neighbour peer receives message

Neighbour peer responds with the IP addresses and ports of his direct neighbours.

Neighbour peer sends DIRECTNEIGHBOURS message

Lonesome peer receives list of neighbours and stores them

a) Lonesome peer has learned about new peers that might become new direct neighbours.

Lonesome peer uses the HELLO protocol to become a neighbour of the new peers reported by his direct neighbours.

See HELLO protocol

b) Lonesome peer has not learned about any new neighbours or has still not enough neighbours.

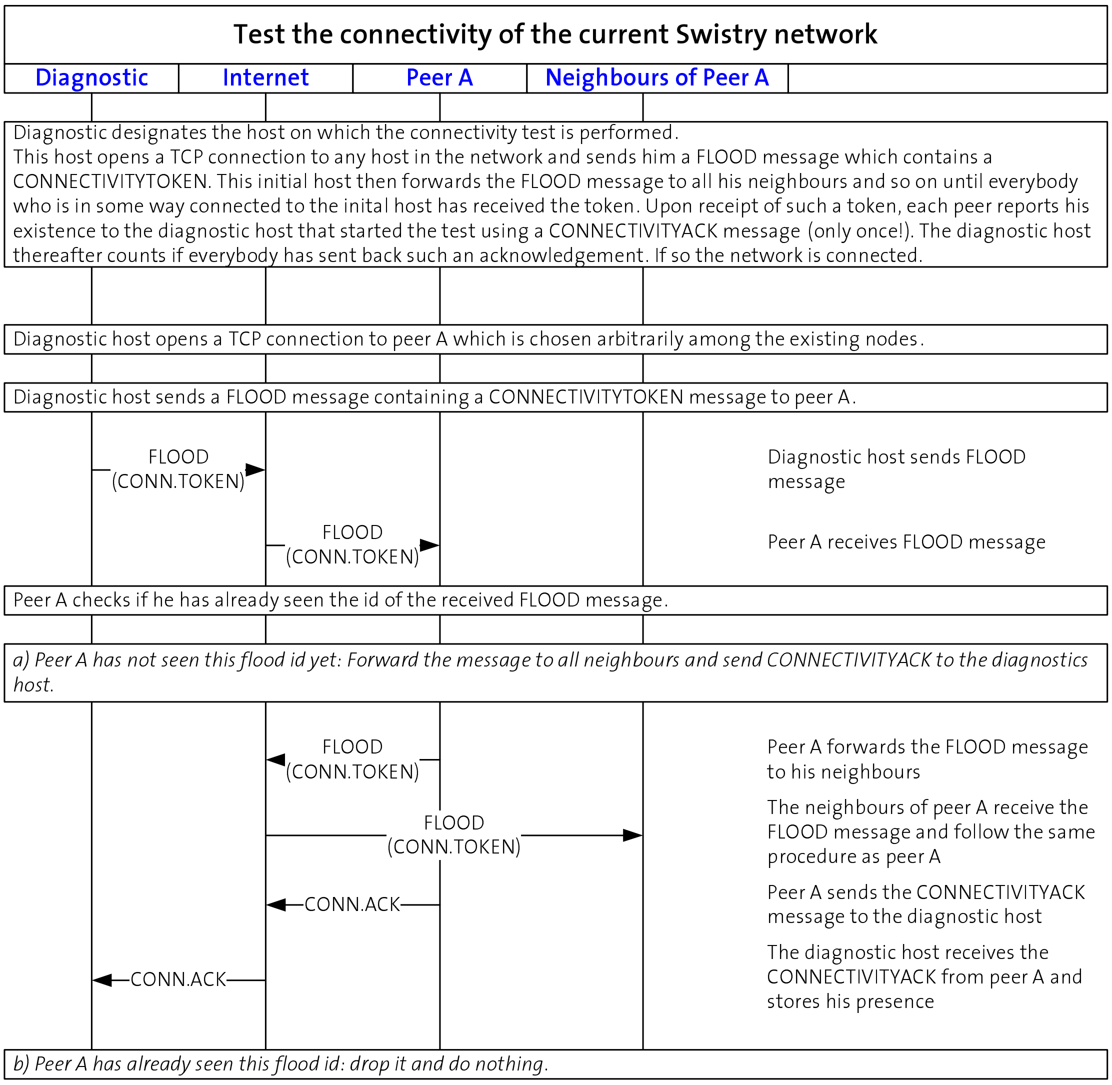
Lonesome peer uses the JOIN protocol to get a new set of neighbours from the NetworkEntryPoint. After getting enough neighbours the new peer will follow the HELLO protocol to become their neighbour.

See JOIN protocol

See HELLO protocol

Lonesome peer closes TCP connection to his neighbours and the NetworkEntryPoint.

Figure A.8: Protocol followed by nodes that want to learn about their 2-neighbourhood by asking their direct neighbours.



Diagnostic designates the host on which the connectivity test is performed.
This host opens a TCP connection to any host in the network and sends him a FLOOD message which contains a CONNECTIVITYTOKEN. This initial host then forwards the FLOOD message to all his neighbours and so on until everybody who is in some way connected to the inital host has received the token. Upon receipt of such a token, each peer reports his existence to the diagnostic host that started the test using a CONNECTIVITYACK message (only once!). The diagnostic host thereafter counts if everybody has sent back such an acknowledgement. If so the network is connected.

Diagnostic host opens a TCP connection to peer A which is chosen arbitrarily among the existing nodes.

Diagnostic host sends a FLOOD message containing a CONNECTIVITYTOKEN message to peer A.

Diagnostic host sends FLOOD message

Peer A receives FLOOD message

Peer A checks if he has already seen the id of the received FLOOD message.

a) Peer A has not seen this flood id yet: Forward the message to all neighbours and send CONNECTIVITYACK to the diagnostics host.

Peer A forwards the FLOOD message to his neighbours

The neighbours of peer A receive the FLOOD message and follow the same procedure as peer A

Peer A sends the CONNECTIVITYACK message to the diagnostic host

The diagnostic host receives the CONNECTIVITYACK from peer A and stores his presence

b) Peer A has already seen this flood id: drop it and do nothing.

Figure A.9: Connectivity test protocol to test if the Swistry network is still connected.

Message Definitions:

Join Peer

JOIN
<*byte*:messageType><*int*:peerPort><*int*:peerOutboundSpeed(kbps)><*byte*:PEER><*String*:*version*>

INITNEIGHBOURS
<*byte*:messageType><*int*:lowerNeighbourBound><*int*:uppderNeighbourBound><*byte*:*streamType*>
<*int*:nNeighbours>{<*InetAddress*:neighbourIP><*int*:neighbourPort>}

PEERTOOSLOW
<*byte*:messageType>

VERSIONMISMATCH
<*byte*:messageType><*String*:requiredVersion>

Join Source

JOIN
<*byte*:messageType><*int*:sourcePort><*int*:levelSpeed(kbps)><*byte*:SOURCE><*byte*:streamType><*String*:version>

INITNEIGHBOURS (a)
<*byte*:messageType><*int*:lowerNeighbourBound><*int*:uppderNeighbourBound><*byte*:streamType>
<*int*:0>

INITNEIGHBOURS (b.2)
<*byte*:messageType><*int*:lowerNeighbourBound><*int*:uppderNeighbourBound><*byte*:*streamType*>
<*int*:nNeighbours>{<*InetAddress*:neighbourIP><*int*:neighbourPort>}

DUPLICATESOURCE
<*byte*:messageType>

VERSIONMISMATCH
<*byte*:messageType><*String*:requiredVersion>

Ping

PING
<*byte*:PING>

PONG
<*byte*:PONG>

Figure A.10: Exact definitions of the messages exchanged in the described protocols.

Hello

HELLO
<*byte*:messageType><*InetAddress*:newPeerIP><*int*:newPeerPort>

ACCEPT
<*byte*:messageType><*InetAddress*:acceptingPeerIP><*int*:acceptingPeerPort><*int*:packetSize><*int*:bitrate>

REDIRECT
<*byte*:messageType><*InetAddress*:acceptingPeerIP><*int*:acceptingPeerPort><*int*:packetSize><*int*:bitrate>

FORWARD
<*byte*:messageType><*InetAddress*:newPeerIP><*int*:newPeerPort>

ACCEPTFORWARD
<*byte*:messageType><*InetAddress*:acceptingPeerIP><*int*:acceptingPeerPort>

REJECTFORWARD
<*byte*:messageType><*InetAddress*:rejectingPeerIP><*int*:rejectingPeerPort>

DROPPED
<*byte*:messageType><*InetAddress*:droppingPeerIP><*int*:droppingPeerPort>

DROPPEDCONFIRM
<*byte*:messageType><*InetAddress*:droppedPeerIP><*int*:droppedPeerPort>

DROPPEDACK
<*byte*:messageType><*InetAddress*:droppingPeerIP><*int*:droppingPeerPort>

DROPPEDNACK
<*byte*:messageType><*InetAddress*:droppingPeerIP><*int*:droppingPeerPort>

Packet Exchange

NOTIFICATION
<*byte*:messageType><*long*:packetId><*InetAddress*:packetHolderIP><*int*:packetHolderPort>

REQUEST
<*byte*:messageType><*long*:packetId><*InetAddress*:requestingPeerIP><*int*:requestingPeerPort>

P2PPACKET
<*byte*:messageType><*InetAddress*:packetSenderIP><*int*:packetSenderPort><*long*:packetId><*int*:packetSizeBytes>
<*byte[]:data*>

PACKETNOTAVAILABLE
<*byte*:messageType><*long*:packetId><*InetAddress*:packetNotHolderIP><*int*:packetNotHolderPort>

Figure A.11: Exact definitions of the messages exchanged in the described protocols (continued).

Bye

BYE
<*byte*:messageType><*InetAddress*:leavingPeerIP><*int*:leavingPeerPort>

BYECONFIRM
<*byte*:messageType><*int*:leftPeerPort>

BYEACK
<*byte*:messageType>

BYENACK
<*byte*:messageType>

Report Neighbours

DIRECTREPORTNEIGHBOURS
<*byte*:messageType>

DIRECTNEIGHBOURS
<*byte*:messageType><*int*:numberOfPeers>{<*InetAddress*:candidatePeerIP><*int*:candidatePeerPort>}

Connectivity Test

FLOOD
<*byte*:messageType><*int*:floodId><*msg*:messageToBeFlooded>

CONNECTIVITYTOKEN
<*byte*:messageType><*InetAddress*:diagnosticIp><*int*:diagnosticPort>

CONNECTIVITYACK
<*byte*:messageType><*InetAddress*:ackSendingHostIp><*int*:ackSendingHostPort>

Figure A.12: Exact definitions of the messages exchanged in the described protocols (continued).

Appendix B

Tutorial

This chapter contains a detailed tutorial on how to configure and start a new Swistry network. First, an overview of the required steps is given followed by instructions on how to start the individual components.

B.1 Starting Order

1. *the* ***network entry point (NEP)***
2. *at least one* ***source***
3. *any amount of* ***peers***

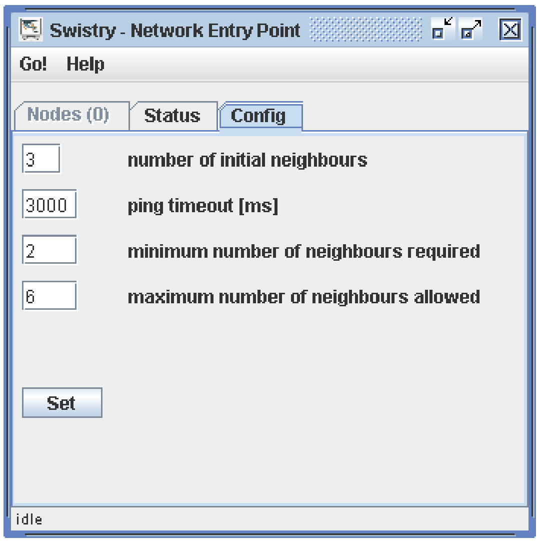
B.2 Setting up the Network Entry Point (NEP)

First, you need to configure the NEP appropriately. This step is only necessary when creating a new Swistry network for broadcasting a new stream though.

The following configuration options are available at the NEP:

1. **Number of initial neighbours:** The amount of contacts a joining peer will be given. This value should be between the minimum and maximum number of required neighbours.
2. **Ping timeout:** The time in milliseconds that is allowed to elapse between a ping request sent by the NEP and the corresponding pong message from the peer; it should not be smaller than 1000.
3. **Minimum number of neighbours required:** This defines the lower limit of neighbours each peer must have. It must be at at least 1.
4. **Maximum number of neighbours required:** The upper bound on neighbours each peer can have. This value must be larger than the lower bound.

Figure B.1 shows an unconfigured NEP. After having chosen appropriate values, you need to store them using the *Set* button. Thereafter, the *Start* button in the menu *Go!* will be enabled.



3 number of initial neighbours

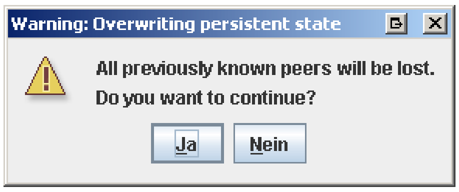
3000 ping timeout [ms]

2 minimum number of neighbours required

6 maximum number of neighbours allowed

Figure B.1: Screenshot of the NEP configuration tab.

Upon pushing the *Set* button, the NEP creates a file called `nepState.p2p` in which he stores the current configuration as well as the peers that have logged in. This file is updated whenever peers are joining. It is basically the persistent storage of the NEP. If such a file already exists when pushing the *Set* button, a dialog box (as in Figure B.2) will ask for permission to overwrite the current persistent storage. You can read back an existing configuration file using the *Recover* button from the *Go!* menu.



All previously known peers will be lost.
Do you want to continue?

Figure B.2: Screenshot of the dialog to confirm overwriting the current persistent state.

After having pushed the *Start* button, the NEP is running and listening for incoming connections on port 65000 until it is stopped using the *Stop* button from the *Go!* menu.

The *Status* tab is updated with current information and notes from the system (e.g. a new peer has joined). The currently available peers are listed in the *Nodes* tab.

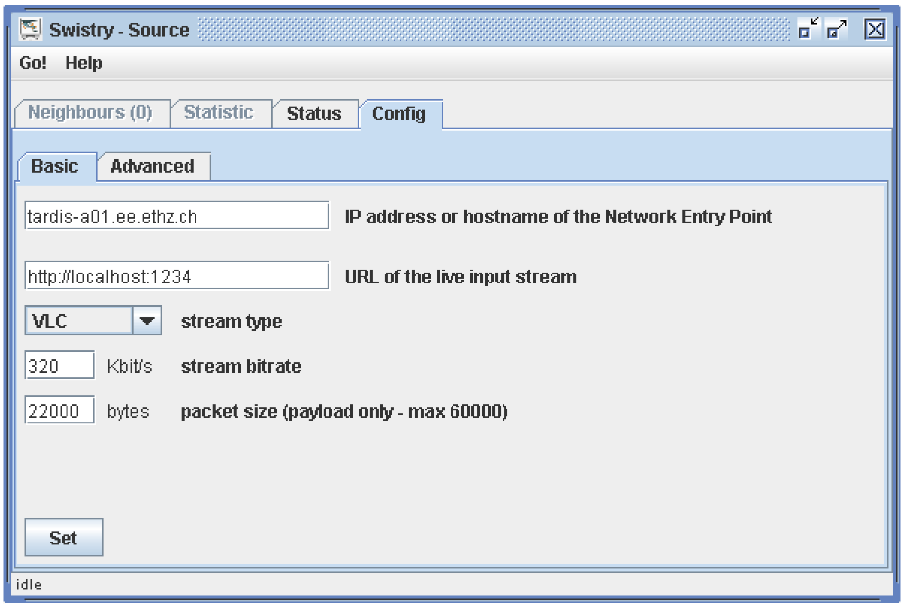
B.3 Starting a Source

A source can only be started successfully when the target NEP is already running. Otherwise it will complain that the network entry point is not available and exit.

The important parameters are:

1. **IP address or hostname of the NEP:** The URL of the network entry point this source will join. It can be specified as an IP address or by using the hostname.
2. **URL of the live input stream:** The address of the media stream source (e.g. a SHOUTcast server or a VLC player).
3. **Stream type:** VLC or SHOUTcast depending on the chosen media source. The latter one can be used to connect to any MP3 stream that is transmitted using the HTTP protocol.
4. **Stream bitrate:** The bitrate of the chosen stream in Kbits per second. This value should be chosen as accurately as possible.
5. **Packet size:** The size of an individual P2PPacket. All packets will have the same size. The upper limit is 60Kbytes because of the UDP limitation. It should not be lower than 1000 bytes.

There is a *Basic* (Figure B.3) and an *Advanced* (Figure B.4) configuration tab in the source GUI. The first one contains stream specific settings and the latter one is used to modify the communication parameters as needed.



tardis-a01.ee.ethz.ch IP address or hostname of the Network Entry Point

http://localhost:1234 URL of the live input stream

VLC stream type

320 Kbit/s stream bitrate

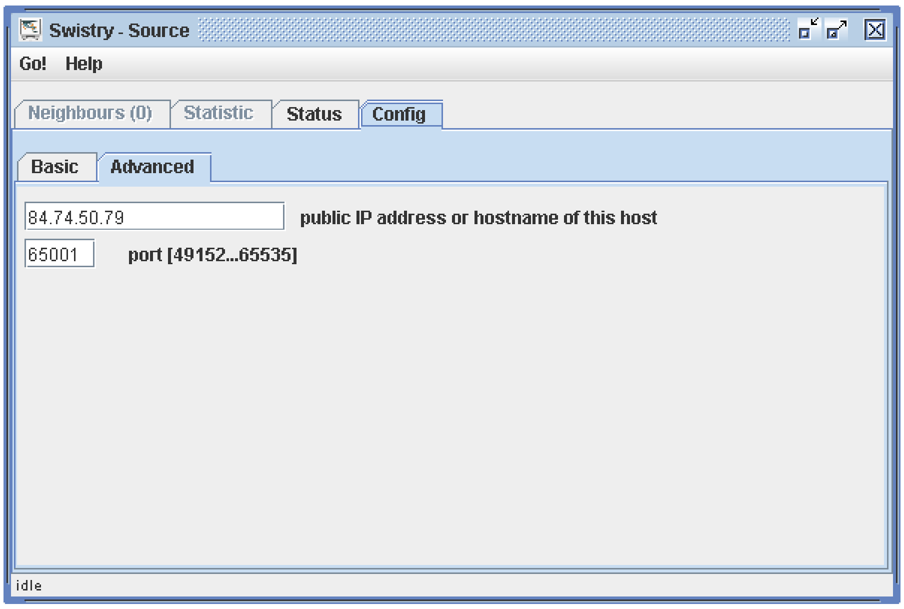
22000 bytes packet size (payload only - max 60000)

Figure B.3: Screenshot of the basic configuration tab of the source application.

The current configuration will be stored by pushing the *Set* button. It is copied into a file for persistency. If such a file exists when starting the GUI, the values from the last session will be restored.

The next step is to join the network by selecting *Go!– >Join Network*. Now it is time to start forwarding packets into the Swistry network. This done by choosing *Go!– >Start Stream*. The stream can be stopped and restarted without loosing the connections to the current neighbours with *Go!– >Stop Stream* followed by *Go!– >Start Stream*.

If *Go!– >Leave Network* is selected, all connections will be closed and the level of this source will be destroyed. The stream is stopped as well and the current neighbours will have to re-join at a lower level.



84.74.50.79 public IP address or hostname of this host

65001 port [49152...65535]

idle

Figure B.4: Screenshot of the advanced configuration tab of the source application.

B.3.1 Configuring VLC

We used the VideoLAN client to grab the TV signal and connected it to the Swistry source over the HTTP protocol. If you decide to use VLC, you need to set it up like this:

1. Choose the desired media. For TV select `File−>Open Capture Device...` (Figure B.5)

2. Select the audio and video devices. (Figure B.6)

3. Set the stream to be forwarded using HTTP and choose an encoding and a multiplexer. (Figure B.7)
   For a list of possible combinations see http://www.videolan.org/streaming/features.html.

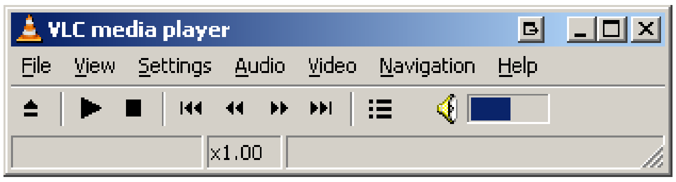
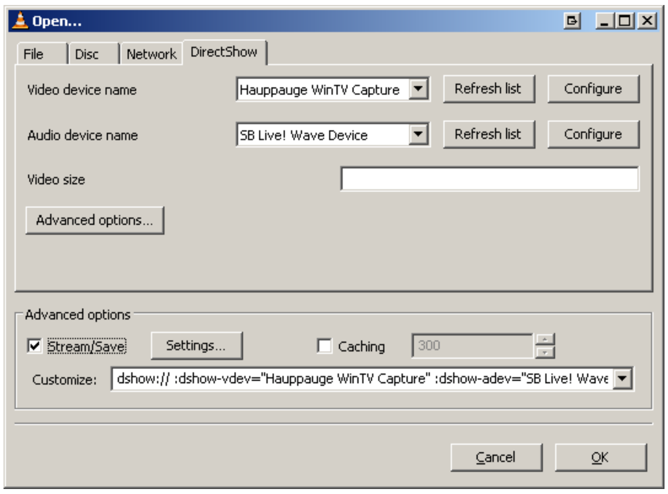


Figure B.5: Screenshot of the VideoLAN client.



Open...

File | Disc | Network | DirectShow

Video device name: Hauppauge WinTV Capture — Refresh list — Configure

Audio device name: SB Live! Wave Device — Refresh list — Configure

Video size:

Advanced options...

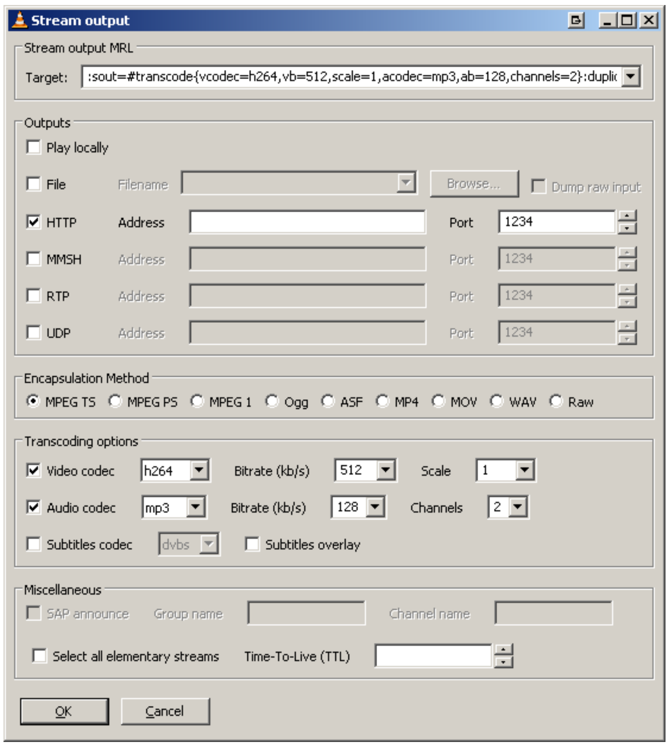
Advanced options

[✓ Stream/Save] Settings... — [ ] Caching 300

Customize: dshow:// :dshow-vdev="Hauppauge WinTV Capture" :dshow-adev="SB Live! Wave

Cancel — OK

Figure B.6: Set the desired audio and video devices.



Stream output

Stream output MRL

Target: :sout=#transcode{vcodec=h264,vb=512,scale=1,acodec=mp3,ab=128,channels=2}:duplic

Outputs

[ ] Play locally

[ ] File — Filename — Browse... — [ ] Dump raw input

[✓ HTTP] Address — Port 1234

[ ] MMSH — Address — Port 1234

[ ] RTP — Address — Port 1234

[ ] UDP — Address — Port 1234

Encapsulation Method

[✓ MPEG TS] MPEG PS, MPEG 1, Ogg, ASF, MP4, MOV, WAV, Raw

Transcoding options

[✓ Video codec] h264 — Bitrate (kb/s) 512 — Scale 1

[✓ Audio codec] mp3 — Bitrate (kb/s) 128 — Channels 2

[ ] Subtitles codec dvbs — [ ] Subtitles overlay

Miscellaneous

[ ] SAP announce — Group name — Channel name

[ ] Select all elementary streams — Time-To-Live (TTL)

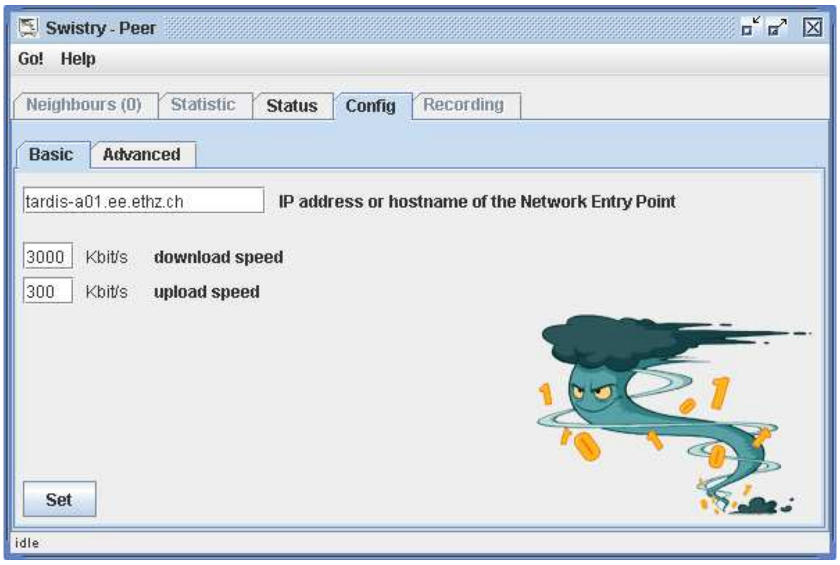
OK — Cancel

Figure B.7: Choose the encodings and the multiplexer.

B.4 Joining an Existing Network as a Peer

The first step to join a Swistry network is to find out the URL of the NEP that is responsible for the desired stream. It is published on the corresponding website. The URL of our NEP is published on http://dcg.ethz.ch/projects/swistry since there is no global list of streams yet.

Like the source, the peer has two different configuration tabs. The *Basic* tab is shown in Figure B.8 and the *Advanced* one in Figure B.9.

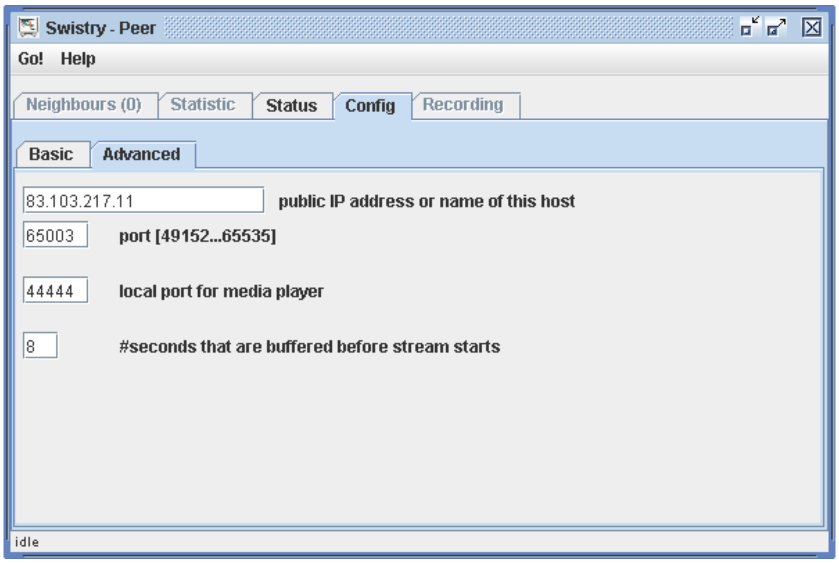


tardis-a01.ee.ethz.ch IP address or hostname of the Network Entry Point

3000 Kbit/s download speed

300 Kbit/s upload speed

Figure B.8: Screenshot of the basic configuration tab of the peer application.



83.103.217.11 public IP address or name of this host

65003 port [49152...65535]

44444 local port for media player

8 #seconds that are buffered before stream starts

Figure B.9: Screenshot of the advanced configuration tab of the peer application.

The most important setting is the URL of the network entry point. In addition to that, the upload- and download bandwidths (in Kbits per second) can be specified in the *Basic* configuration. If you do not know the exact values, use one of the following bandwidth estimation tools:

Europe: http://speedtest.cnlab.ch/test

America: http://www.speakeasy.net/speedtest

Others: http://www.testmyspeed.com/speedtests/international.htm

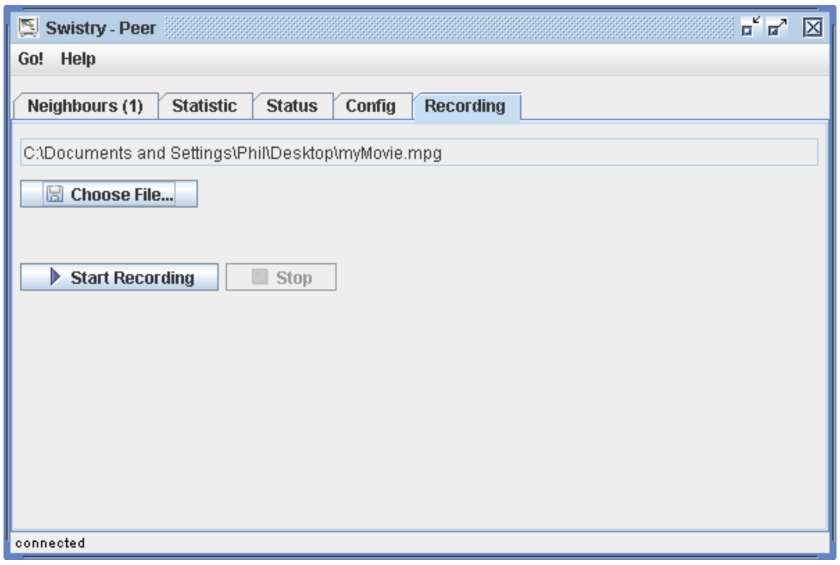
It is very important that the bandwidth values are as accurate as possible in order to get an optimal stream quality!

The *Advanced* configuration tab allows you to set the IP address and the port number that is used to communicate with the other peers. Furthermore, the port on which the media player will be connected and the buffer size (in seconds) can be specified here. The chosen settings are saved using the *Set* button.

In order to join the network, the *Go!*− >*Join Network* command is executed. The peer now registers with the NEP and thereafter with the given initial neighbours. The packet exchange will start automatically and is monitored in the *Statistic* tab. The *Neighbours* tab contains a list of currently connected neighbours and statistics about the current packet exchange.

You are now ready to start watching the stream. For that, you just need to start the VideoLAN client or an MP3 player depending on the stream type and connect it to the Swistry application using the HTTP protocol. The target URL to be entered in the player is *localhost:44444* where *44444* is the port number chosen in the *Advanced* config tab of the peer.

If you want to record the stream, got to the *Recording* tab of the Swistry peer GUI (this option is not available in console mode). It will be enabled as soon as you have joined the network. Choose a file to save the stream to and push *Start Recording*. Video files that are multiplexed using MPEG-TS should be called `<filename>.mpg` and MP3 stream recordings `<filename>.mp3`. It is then most likely that you have associated an appropriate player for the files.



C:\Documents and Settings\Phil\Desktop\myMovie.mpg

connected

Figure B.10: The recording functionality of the Swistry peer.

B.5 Using the Console Mode

Each component of the Swistry application is available for the console as well. They are all contained in the `SwistryConsole.jar` executable. The parameters are basically the same as when using the GUI.

In order to start a component from the console, the following command is used:

```
java -jar SwistryConsole.jar <Component>
```

`<Component>` can be one of the following: `nep`, `source` or `peer` followed by the respective arguments.

```
nep <nInitNeighbours> <minNeighbours> <maxNeighbours> <recover>

<nInitNeighbours>:  the number of neighbours that a joining peer will receive;
must be between <minNeighbours> and <maxNeighbours>
<minNeighbours>:  the minimum number of neighbours each peer must have
<maxNeighbours>:  the maximum number of neighbours each peer can have
<recover>:  whether or not to recover a previous state---either 'true' or
'false'
```

```
source <ip> <port> <bitRate> <packetSize> <nepAddress> <streamType>
<inputStreamURL>

<ip>:  public IP address or hostname of this source
<port>:  port number on which this source will listen for incoming connections
<bitRate>:  the bit rate of the stream in kbps
<packetSize>:  the size of the packets that will be sent over the network in
bytes (payload only)
<nepAddress>:  IP address of the Network Entry Point
<streamType>:  the type of the stream---either 'audio' or 'video'
<inputStreamURL>:  the URL from which the stream will be received
```

```
peer <ip> <port> <inSpeed> <outSpeed> <nepAddress> <dePacketizerThreshold>
<mediaPort>

<ip>:  public IP address or hostname of this peer
<port>:  port number on which this peer will listen for incoming connections
<inSpeed>:  inbound speed in kbps
<outSpeed>:  outbound speed in kbps
<nepAddress>:  IP address of the Network Entry Point
<dePacketizerThreshold>:  the number of packets that need to be available in
order before the stream starts to play
<mediaPort>:  local port number to which the media player will be connected
```

Appendix C

Contents of the CD ROM

Swistry Software v.0.2

- Swistry Java Source Code
- Swistry Binaries (JAR)
- Network Simulator
- Traffic Simulator

Report

- PDF, PS & DVI
- LaTeX Sources Including BibTex Files
- Image Files in EPS and Original Format

Project Website

- Mirror of the Swistry Website

Lab Experiments

- Scripts Used to Run Experiments on Lab Computers
- List of the Lab Computers and Their Respective Roles