UNIVERSITY OF OSLO
Department of Informatics

Peer Selection in Peer-to-Peer Streaming Systems

Tore Langedal
Endestad

February 1, 2008

Peer Selection in Peer-to-Peer Streaming Systems

Tore Langedal Endestad

February 1, 2008

Abstract

One important task of any peer-to-peer streaming system (p2p-ss) is how to choose which peers should connect to which peers. How well a p2p-ss perform this task greatly influences its performance. This thesis explores how different peer selection algorithms affect the performance of such systems.

A framework for doing the comparisons of peer selection algorithms is built on top of the network simulator ns2, making it possible to later extend the simulations with new peer selection algorithms, congestion control algorithms, wireless networks, cross traffic and other. However, ns2 is a low-level simulator, hence limiting the number of peers in the simulations, because CPU-resources are limited. The simulations are limited to single-layered streams.

We find that a centralized selection method, which utilizes knowledge of bandwidth capacities and routing in the network, greatly outperforms both simple random selection of peers, and selection of close peers. Even though centralized selection does not scale well, and is therefore only applicable for a limited number of peers, this shows there is much room for improvement over basic strategies.

Acknowledgments

Thanks to Paul Vigmostad and Haakon Riiser at Netview Technology, for giving me the chance to write this thesis there. Thanks to my supervisors Haakon Riiser (again), Pål Halvorsen and Carsten Griwodz for valuable feedback and guidance. Thanks to my dear girlfriend, Elin Horsdal, for correcting countless misspellings, and for being who she is.

Oslo, January 2008

Tore Langedal Endestad

CONTENTS

List of Figures

CHAPTER 1

INTRODUCTION

Quite a few systems for Internet based streaming are in use today. The most well-known is perhaps YouTube which allows its users to upload their own video clips to a set of servers, which then distributes the video clips, making them available to other users. Some systems aim to replace the traditional television distributors by distributing the same television channels over the Internet, for instance Zattoo. Many of these systems seem to use a traditional client-server model, or a content distribution network such as the Akamai content distribution network.

Using peers to help distribute media streams reduces the content distributor's cost of distributing content as compared to using the traditional client-server model, because users of the system contribute resources by means of their normally under-utilized upload links. Several peer-to-peer streaming systems(p2p-ss)[1] for doing this exist: CoolStreaming [43] and PPLive [4] are currently the two most popular. CoolStreaming has been used by more than 1 million users world wide, and PPLive is in commercial use.

Performance of peer-to-peer streaming systems depend upon several of its components. One of the more important components is the peer selection algorithm of the system. In general, connecting to a physically close peer means low latency, which allows quick congestion response and congestion control. Also, local connections will take some load off the backbones. However, if one connects to peers that are from the same location, the connections from the peers to the server will be likely to use links that are common to all of the peers. If one of the common links then get congested, the peers would struggle to sustain the streaming rate. See

[1] Some places we use peer-to-peer *assisted* streaming to indicate the situation where a server contributes a large portion of the bandwidth for streaming, some places we use *pure* peer-to-peer streaming to indicate absence of a server. Peer-to-peer streaming includes both situations, and any in between.

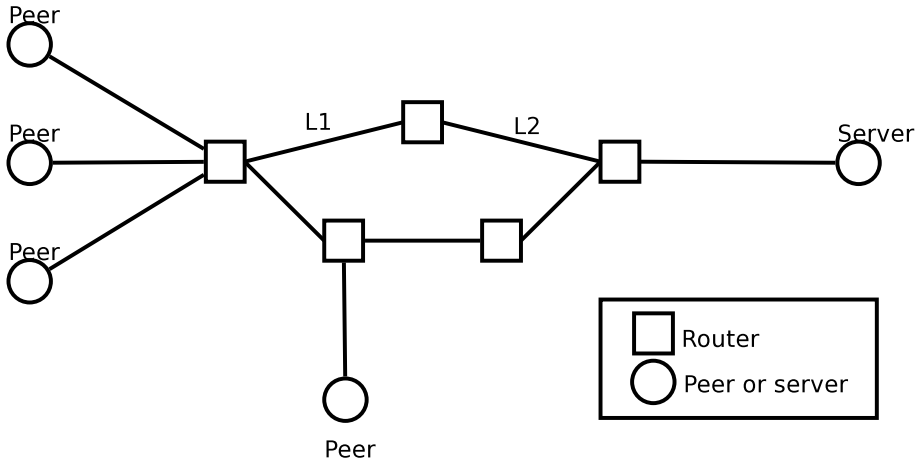


Figure 1.1: Common links

figure 1.1: Assume a shorest-path routing is used. If the three peers at the left only connect to each other and to the server, there will be very short latency between the peers, and only one copy of the stream needs to traverse the path (L1, L2). However, if the network conditions change, so that the peers get lower capacity over L1 or L2; they will have to select other peers to cooperate with. Selecting new peers take some time, it might therefore be useful for the left peers to select the lower peer even before any problems occur, despite being farther away.

Current p2p streaming systems tend to be receiver driven, meaning it is the responsibility of each receiver to find out which other peers it should connect to in order to maximize the qualityof the received media. The peers of receiver driven peer-to-peer streaming systems often use greedy heuristics for selecting the sender peers, peers are therefore often non-cooperatively competing for bandwidth. Although such an approach might be the most appropriate for selecting peers in a p2p streaming system, there might be room for quite large improvements, especially if the heuristic for selecting peers is not well-suited for the problem. Results on traditional p2p content distribution indicate that this is the case: The Julia content distribution network can reduce the network load by up to 33% when compared to BitTorrent, without any additional infrastructure. See section 4.2. This suggests that clever use of network resources can be used to support more users or higher quality in the same network.

1.1 Problem definition and scope

The focus of this thesis will be to find out how to do peer selection in the best possible way for the purpose of peer-to-peer streaming in the Internet. The selection of which peers should send content to which peers — peer selection — in such a system, is a big factor in the the performance of the system. A discussion of important characteristics of other peer-to-peer streaming systems will be given. It will be discussed how some existing peer-to-peer streaming systems work; including how they select peers.

One thing to note is that there is at least a couple of different definitions of what "best" means in the case of a p2p streaming system: high percentage of satisfied users, lowest possible network load for a given user set, high number of users which receive good video quality, maximum usage of users' upload capacities; and so on. In the case where the content provider provides single-layered content at one rate only, highest number of concurrent users can be a fair measure of what is good; average quality or satisfaction can also be good measures if different rates are provided. In any case, the network load should be kept low if possible, because this would both allow many simultaneous users and not reduce performance of other users of the net.

Peer selection problem For the sake of clarity: the primary target of the thesis is to test how well different algorithms solve what we call the peer selection problem. In a p2p-ss, there are peers and servers. Each peer has two primary goals: minimize start up delay of the chosen stream, and maximize audiovisual quality of the stream. The peer may also have a secondary goal: minimize resource usage. The variables considered in the peer selection problem are the bandwidth between each pair of nodes. A node is either a server or a peer. The **peer selection problem** is an optimization problem: maximize the goals of the peers, only by altering the variables above. The problem is obviously not well-defined, both since the primary goals contradict each other, and because it is not defined whether *maximize the goals of the peers* points towards an average, or towards a maximization of the minimum. How well an algorithm solves this problem is subjectively evaluated.

Criteria: There are many important characteristics of a p2p-ss. The most important to a user's experience is probably the quality delivered, the startup delay, and that the system actually works. This thesis will focus on the bandwidth delivered to the peers. Delivered bandwidth is perhaps the single most important technical characteristic of a streaming system since

higher delivered bandwidth means a user can request content coded at higher quality. Normally, higher delivered bandwidth also means shorter start up delay, because the buffer of the media player can be filled quicker. Also, if bandwidth is delivered to users, this is a great indication on that the system works. We are also concerned about how the system behaves if a flash crowd appears.

Other characteristics which are of great importance, but not considered in this thesis is: *Start up delay* If the start up delay gets too long, the users of the system would probably not be very satisfied. *Server load*. If the server load does not decrease, a pure client-server solution would be better, because of lower startup delay, trust issues etc. *Tolerate cross traffic.* The system should not break due to changes in cross-traffic, neither should it block the cross-traffic. *Tolerate massive user failures.* Large user failures could happen in the real world because of power-outages, however probably very infrequent; but the unaffected users should still be able to use the system.

There are still other important aspects of p2p-ss which are perhaps even more important; such as many security concerns. Security concerns will not be throughly discussed in this thesis. Other characteristics such as the price for a user to use a system, and moral issues is not discussed here at all.

1.2 Method

We develop a framework for doing simulations on p2p-ss on top of the network simulator ns-2 [3]. Ns-2 is a low-level discrete-event simulator. Simulations are done in Internet-like networks, obtained from the GT-ITM-tool [2, 7]. Three different peer selection algorithms are compared in the framework.

1.3 Main contributions

First, the thesis describes a framework for doing fair comparisons between peer selection algorithms so alternative algorithms can be implemented and tested under the same conditions as the algorithms that this thesis explores. The framework is also modular, making it possible to change any of peer discovery, peer selection, transport protocol, congestion control or media player abstraction without having to alter all of the others.

The main contribution from this thesis is a comparison among different

peer selection algorithms. Although many peer selection algorithms have been proposed and tested, it seems that little effort have been put into fair comparisons between the different algorithms; this thesis aims to do so.

And last, we also explore different strategies for selecting the ordering the pieces of a stream should be sent in.

1.4 Outline

A description of some important aspects of p2p streaming systems is given in chapter 2. Chapter 3 describes some existing p2p streaming systems, with emphasis on how they do peer selection. Chapter 4 summarizes other relevant work. We then describe some peer selection algorithms in chapter 5. Details on the framework and the simulations are given in chapter 6. Then results follow in chapter 7. Finally, the conclusions are in chapter 8.

CHAPTER 2

COMPONENTS AND ASPECTS

There are many different things to notice when designing a p2p-ss. We try to summarize the most important in the following. IP multicast has not been included in the discussion here because it is normally not deployed or activated in networks, even though it would be a great way of distributing live streams.

2.1 Peer-to-peer streaming components

A peer-to-peer streaming system contains the following parts:

Media player: The media player is the user interface of the system. The users watch or listen to the streams via the media player.

Post-play buffer management: This part handles how and which part of a stream that should be stored in order to be uploaded to other peers at a later time.

Pre-play buffer management The pre-play buffer bufferers data that is not yet sent to the media player.

Peer discovery: Should find candidate peers.

Peer selector: Responsible to select which of the candidate peers to use as active senders and/or receivers. Also responsible for dividing load between the peers it selects.

Piece manager: The stream is divided into pieces, which again consists of packets. The responsibility of the piece manager is to select which piece to select from which of the active peers.

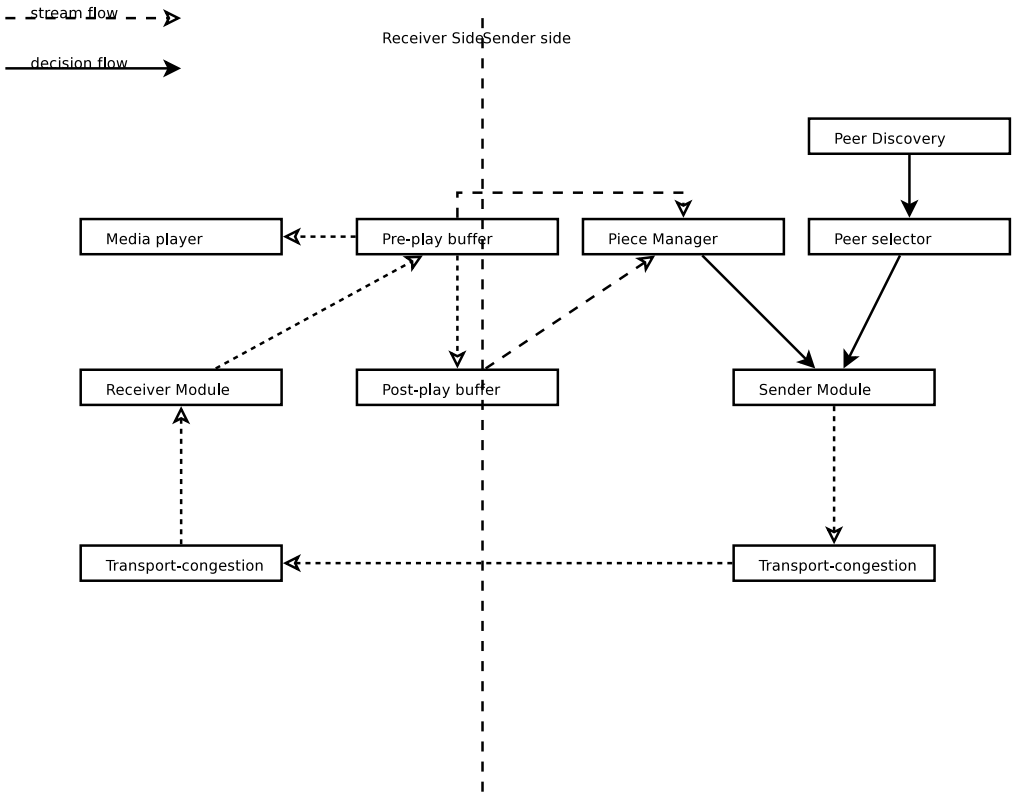


Figure 2.1: The components of a p2p streaming system, and their dependence on each other.

Sender and receiver modules: The sender module should do the actual scheduling of the pieces to send. If the peer selector has decided how the upload rate should be used, the sender module must follow the decision.

The receiver module sends 'stream request messages' to peers selected by the peer selector, and fills the messages with a requested rate as decided by the peer selector. The receiver module will adjust the requested rates if it is required to do so by the congestion control scheme selected.

Transport protocol and congestion control: Responsible for transporting the data across the network. Also responsible for performing congestion control in order to avoid collapse of the streaming session (and in the worst case: collapse of the network).

A figure of the components of the p2p streaming system and how they work together, is given in figure 2.1. The stream flow describes how the

packets of a stream traverses the system. Decision flow tells the dependence between the components. For instance, the peer selector needs information from the peer discoverer in order to make any decision. Other flows are possible, the ones shown in the figure are "typical".

2.2 Video-on-demand and live streams

There are two main stream types considered in p2p streaming; live streams and video-on-demand (VoD). Live streams are a bit like television programs: The program progresses continually, and when users tune in they start to watch at the current global point of progress. A video-on-demand stream is more like watching a rented movie: The user selects when to start the movie. Depending on the system, the user can also pause and skip to any point in the movie. However, some features common to VCR and DVD players are not well handled in most streaming systems, for instance fast forward. A suggestion on how to support DVD-like features in streaming systems, including fast forward, is given in [40].

There are some differences between supporting video-on-demand services and supporting live television. In the case of video on demand; users will tend to always start streaming from the the start of the content, and different users will start streaming at different times. In live streams, users will tune in to a stream in such a way that everyone is watching the same frame at approximately the same time. This means that in VoD, the content will often be distributed in the network when a user tunes in; and not in the case of live broadcast. However, in many cases of live streams, it will be possible to distribute the stream in the network before allowing users to tune in, and therefore help handle flash crowds. This can be done if a streaming system is used for distributing a given television channel. If a popular program series is to start at 6 pm, it can be distributed to some selected peers at 5pm. The key to decrypt and watch the series would then not be distributed before 6pm. This approach can of course not be used in great extent for live sports events, because it would impose a lot of delay, compared to other distribution methods.

The different usage patterns may also call for different distribution strategies. In the case of live streams, all users may be downloading from the exact same point of the stream. No caching of content for future distribution beyond some seconds or minutes would be required. Multicast techniques for distributing content could be appropriate as well. In the case of video-on-demand, probably very few users would stream at the same point of the stream. This would require much more caching if

peers should be able to participate in the distribution. Because no-one or very few users want the same data at the same time, pure multicast techniques would be unusable. Instead of considering both live streams and VoD, there is also one third stream-type, near-video-on-demand: Instead of supporting real video-on-demand, where a user can join at any time, one rather periodically broadcasts the video as a live stream and let users join any of the broadcasts. The advantage of near-video-on-demand over video-on-demand is that network multicast could be used if deployed. It would also reduce the caching requirements on the users' equipment.

2.3 Pull based and push based streaming systems

Any peer-to-peer streaming system is either pull based, push based or a combination.

In a pure pull based system, the receiver peer is responsible for sending request packets to its senders, in detail telling each sender which parts of the stream to send. The details are periodically updated to fit the needs of the receiver. Pull based systems use retransmission in case of packet loss. Many of the p2p streaming systems are pull based: one example is CoolStreaming (section 3.2 on page 32).

In a pure push based system, the stream is often divided into several sub-streams. For example: the stream is divided into 5 multiple description (see below) coded streams. A receiver requests one or more sub-streams from a node; the node will then forward the sub-stream until the receiver sends a stop message. An example of a push-based system is SplitStream (section 3.5 on page 34). Since push based systems do not allow retransmittance of lost packets, they tend to use erasure codes to protect their data streams. An erasure code will protect the data of the streams by adding redundant data, and sometimes also by spreading the data over time so that a burst of packet losses is not fatal. The use of erasure codes does not guarantee lossless playback, but some erasure codes allows adjustment to the amount of redundant data — and therefore also the probability of lossless playback. Erasure codes are also known as Forward Error Correction, or simply FEC. There are many different erasure codes, some of the best known are Turbo codes [5] and Raptor codes [37].

If push and pull approaches are combined, one could order the whole stream at start-up, and request retransmittance of fragments that get lost in transmission. In that case, the control overhead of pull-based systems are reduced since the requests only appear when the receiver needs to ask for lost packets, and the overhead of the use of erasure codes in push-based

systems can be reduced or removed entirely.

The decision of making a system push or pull based, will affect which peer selection algorithm is the best for the system. In a pull-based system it seems reasonable to try to pair with close peers, since requests for lost segments will then be quicker. In a push-based system, typically segments are protected by erasure codes and rescheduling of segments is not done, so short round trip times (RTT) are not that important.

The selection between push or pull based will also affect the requirement of the pre-play buffer. Pull based systems will have a pre-play buffer to allow retransmittance of lost packets. Push based systems will also have such a pre-play buffer, which is needed to tolerate the difference in delays from the different senders.

2.4 How to provide different rate-coded content

Users of a peer-to-peer streaming system want to get the best possible quality of the media they stream. Because users have different bandwidth capacities, a way of providing different quality encodings of the same media should be considered. The different ways of providing different quality editions are the same as for non-p2p streaming, but since peers should be able to redistribute their content later, coding efficiency is not the only thing in question.

It is interesting to consider layered coding and multiple description coding. Using these techniques can enable a larger selection of sending peers for a given receiver peer, when compared to coding the original stream into streams of different rate. This is because every peer that has received some of the content will have at least one layer that is interesting to any other peer requesting the same content. This is not the case with coding content into multiple representations with different bit rate. Though, actively researched, layered codecs are not used much in today's streaming systems.

2.4.1 Different rates

Media encoded at different rates is perhaps the most obvious way of providing streams that match each user's downlink. For instance, one could have a video, and encode it at 100 kbps, 500 kbps and 1500 kbps. The users will select the rate which suits they the best. Coding content into streams of different rates achieves high coding efficiency because single-layered coding is more efficient than other layering techniques: Encoding

at a quality measure Q will never require more bits when using single-layer coding as compared to multiple-layered coding. Normally it will require less bits.

However, having the same content coded differently around the network can be a drawback since a peer can only cooperate with peers that have the same representation of the content. Therefore, having multiple representations can dramatically reduce the stream group sizes[1].

Also, if network conditions change for the worse, it is not possible to handle it without changing to a different stream. If the network conditions then return to normal, the system would probably change back to the original stream. During stream change, more control packets would be needed, and some of the stream data might also be discarded.

2.4.2 Layered coding

Layered coding proposes a different way of providing differentiated quality streams. In layered coding, the media is encoded into different layers. There is always one base layer, and there can be a number of enhancement layers. The base layer will have fair quality and users with low bandwidth will choose to stream only this layer. If a user has excess bandwidth he or she can stream enhancement layers as well, and the enhancement layers can then be combined with the base layer to yield better play back quality. Enhancement layers can be dropped or added as network conditions changes. However, the layers are hierarchical, so one must have the base layer in order to use any enhancement layers, and as well enhancement layer K in order to use enhancement layer $K + 1$.

One nice feature offered by MPEG-4 FGS layered coding [32] is that it produces one large enhancement layer that can be truncated at any point, meaning that the full download capacity of a downloading user can be matched exactly.

2.4.3 Multiple description coding

Multiple description coding is also based on layers, but there is no hierarchical system. If one receives N layers, every layer can be used, no matter what layer number they have. However, since any layer must contain enough information to sustain playback, this imposes coding overhead; hence the coding efficiency of multiple description coding is not very high.

[1] Stream group size is the number of peers currently downloading the given stream

2.5 Commercial and non-commercial systems

In a commercial system, it would probably be inappropriate to lower the quality if the user is not contributing enough upload bandwidth. It could be that a low contribution is not a planned malicious activity, it could be that multiple processes are competing for the upload capacity. If applications are competing for downlink capacity, it might become a problem because there is no way that the system can deliver more than the free capacity of the user's downlink.

One of the advantages of having a commercial system is the possibility to support economic incentives: For instance by offering lower prices to users who contribute much of their upload bandwidth.

On the other hand, a non-commercial system can allow degradation of quality. It should also provide strong incentives for users to contribute their upload bandwidths, otherwise people will cheat: [22] finds that 85% of Gnutella users contribute nothing to the network. Many ways of building such incentives has been proposed. In [10] a linear taxation system is proposed; the idea is that how large a download rate one receives, is decided by how much upload rate one gives. Different parameters of the scheme can be set by the content provider, giving some flexibility. The parameters can be adjusted so that users with low upload capacity will get a fair amount of download rate, while encouraging resourceful peers to contribute more bandwidth.

2.6 Peer discovery

If one cannot discover other peers, it is obviously not possible to participate in a p2p system. There are several possible ways of discovering peers in a p2p streaming system. It is possible to let a server handle the discovery process (like the trackers in the BitTorrent system), and it is possible to have a distributed discovery process. In the latter case, peers could be discovered by the use of gossip protocols or by DHT lookup.

The advantage of a server based discovery service is that it can make discovery quickly, but requires server capacity. When all discovery traffic travel through the server, it can be utilized for different purposes: Peers can be clustered in different ways, for instance nation wise or ISP-wise. It is also easier to monitor the discovery process by administrators.

Distributed discovery services will not be as quick, but not necessarily much worse. An example is CoolStreaming: When a new peer joins, the

	Data rate (approximate)	Sample or frame size	Sample or frame frequency
Telephone speech	64 kbps	8 bits	8000/sec
CD-quality sound	1.4 Mbps	16 bits	44,000/sec
Standard TV video (uncompressed)	120 Mbps	up to 640 x 480 pixels x 16 bits	24/sec
Standard TV video (MPEG-1 compressed)	1.5 Mbps	variable	24/sec
HDTV video (uncompressed)	1000-3000 Mbps	up to 1920 x 1080 pixels x 24 bits	24-60/sec
HDTV video (MPEG-2 compressed)	10-30 Mbps	variable	24-60/sec

Figure 2.2: Characteristics of typical multimedia streams [12]

new peer contacts the server[2]. The server then selects a deputy peer at random from all the peers it knows. The joining peer then requests candidate peers from the deputy peer, and the joining peer start to stream from them. This does not stress the server much, and should be fairly quick. However, some applications might be very sensitive to startup delay, in those cases a distributed discovery service might not be quick enough.

2.7 The users

The users of any multimedia streaming system want to stream at the best quality possible. If the streamed media is audio, most users would be perfectly pleased if they receive audio coded at as little as 128kbps. If the streamed media is video, many users will not be satisfied before they receive DVD quality, which require about 5 Mbps[3]. Most likely, lots of people will soon have an HDTV at home and they will want to have media that takes advantage of their equipment. This would require much more bandwidth, see figure 2.2. Who knows which equipment and requirements will be present in 10 years? There is no reason to believe that the bandwidth requirements of users converge to a sensible limit. Today the limiting factor for streaming is the downlink capacity of the users, and

[2]More correct: the origin peer of the stream. CoolStreaming does not use the term server.

[3]This will of course greatly depend on the encoding and the format. The number is estimated from the capacity of a DVD and how long a typical movie is: 4.7 Gb (single side, single layer) and 2 hours.

it is reasonable to believe that it will stay this way, at least for a decade or so, for the above reason. Therefore, one should try to make the best possible use of the downlinks.

It is commonly believed that user access patterns follow Zipf-like distributions. This does not need to be entirely true for Internet media. [18] finds that Internet media access patterns instead follows the stretched exponential distribution, which have cumulative density function $1 - e^{-(x/x_0)^c}$ where c and x_0 are constants.

User experience is influenced by the startup delay of streaming. The most used p2p streaming system of today, PPLive, typically has startup delays around 20-30 seconds; sometimes delays can be as large as 2 minutes in less-popular channels [21].

A problem about users is that they cannot be fully trusted. A common problem in p2p systems is that many users tend to avoid to contribute resources to the network; they are freeloaders. An even more serious security problem is that some users can try to distribute degraded or totally altered content; so called poison attacks. Many p2p systems are without any protection against such malicious activities.

Another interesting point is that users will create flash crowds [39]. A flash crowd is a sudden increase in users streaming a specific content. During major news events, such as Olympic games or the 9/11 event, extreme large crowds of people are likely to visit the same web page at the same time; if the servers are not prepared, they might go down. Flash crowds might be overwhelming, but not necessarily. If a hypothetical Internet radio station which averages 3 listeners suddenly gets 10 listeners, this sudden increase in listeners will define the listeners to be a flash crowd. They would still be a flash crowd if the event occurs periodically, perhaps once a day at 6 pm. Even if the small radio station sends from an resourceful server with no problems during the sudden increase in listeners, it would still be a flash crowd.

2.8 The network

Today, the majority of Internet users use ADSL connections. ADSL users have much less upstream capacity than downstream capacity. If the downstream capacity of all users are to be fully utilized, then additional bandwidth must be provided. To some extent this can be compensated for by making use of idle users (users that stay on line without streaming), but it is hard to imagine that the ratio of idle users to active users will be anywhere near the upload/download rate of a typical ADSL user. This es-

sentially means that if all the downlink capacities of the users are to be saturated, one will need resourceful servers to contribute bandwidth as well.

One should note that routing in the Internet is not necessarily symmetric [31]. That is, the route a packet travels from A to B does not have to be the same as if the packet travels in the opposite direction. If one intends to build a streaming system which measures topology to optimize the routing, such measurements should therefore preferably be done by the sending peer. This makes measurements a more complex task, because sender peers which already serve other peers, suddenly need to generate packets towards a new peer.

Some other very important properties of the network is its capacity, its latency, and its loss rate. The capacity is finite, both at access links and in the backbone. The backbone has almost always much higher capacity than the access links of the end users. The latency is low between users that are close, with RTTs typically of a few milliseconds, between very distant users RTTs can be of 500ms to 1000 ms. The loss in a wired network is mainly because packets get dropped during congestion; however packets with bit errors occur. Paxson found the proportion of corrupted packets to be about 0.02% [31]. The rate of corrupted packets is normally much higher in wireless networks as compared to wired networks.

In addition, the Internet is a best-effort network without support for quality of service. This is a very important property to note, because it means that peers must be able to handle sudden changes in received rate. Also, if a streaming system is to use the Internet, the system should be TCP-friendly, competing with other streams in the Internet in a fair way.

2.9 Congestion control

Which congestion algorithm one uses in a p2p-ss will have an influence on what kind of peer selection algorithm will be the best. Having a large system with none or bad congestion control is potentially harmful for the Internet [14, 15]. Since much of the Internet traffic is TCP-based, the congestion control algorithm of a p2p-ss should be TCP-friendly. That is, it should compete fairly with other traffic.

There are many ways of being TCP-friendly, the most obvious is to simply use TCP. One can also use other additive increase multiplicative decrease(AIMD) algorithms[4], such as RAP [33]. However, in-order de-

[4]Multiplicate the rate by a constant less than 1 when loss occures, increase the rate by a constant otherwise.

livery and reliability of TCP if often unnecessary for streaming, and can actually harm performance because the application has to wait while TCP waits for a lost packet. The AIMD methods (including TCP) often fail do deliver a steady rate to the application, which in some cases is not optimal. TCP-friendly rate control(TFRC) [16,19] tries to deliver a steady rate, while still be TCP-friendly.

2.10 Privacy concerns

In p2p streaming system the peers will only download content the user requests, or at least: most of the content it downloads will be due to user request. The system obviously needs a way of informing its requesting peers which other peers have the requested content. Lets say that the system is used to distribute sensitive content, for instance political propaganda, religious channels or adult content.

A straight-forward exploit can be done by anybody who can join a streaming session, and starts by requesting sensible content. The system will then hand over addresses to peers which has the given content. If one is able to link the address to a person, one has obtained information which can be abused. Even if the system encrypts the content-to-address information, the attacker can simply wait and monitor the packets received during streaming.

2.11 Summary

There are certainly much to think of while working on peer-to-peer streaming systems. It will not be possible to simulate all the details of each of the aspects above. Instead we will focus our attention towards single-layered coded content, and towards the pull based systems. We believe that results for single-layered coded content to some extent could be a guide-line for multiple-layered content. Also, we will only focus on wired networks.

Amongst the components described above, this thesis is primarily interested in the peer selector. Much attention is also given to the piece management How cached content is stored and retrieved is ignored, simply assuming that all previously received pieces of a stream can be stored and retrieved instantly[5]. The only part of the media player this thesis consid-

[5]Although probably a fair assumption for live streams because the cached content is small enough to fit RAM, this might not hold for video-on-demand.

ers is the play out point, which moves at constant speed when the required parts of the stream are present, or pauses otherwise.

CHAPTER 3

EXISTING PEER-TO-PEER STREAMING SYSTEMS

We will now present a few streaming systems that make use of peer-to-peer technology. The systems are in general pretty different. The differences come both from different goals, and from design choices. E.g. Cooperative Networking [30] is designed to help sites handle flash crowds. CoolStreaming is designed to be a complete, maximum scalable, peer-to-peer media streaming system. BitTorrent-Assisted Streaming System [13] explores the possibility of using BitTorrent to move some of the load off a streaming server, and onto the users.

Some of the systems stipulate some properties of the stream as well. PALS [28, 34] uses layered coding, Cooperative Networking uses multiple description coded streams. CoolStreaming and BitTorrent Assisted Streaming System use single-layered coded streams.

The deployment of PPLive [4] demonstrated that peer-to-peer live television can be done commercially in huge networks. Most of the users of PPLive reside in China, but it is possible for any Internet user to use the system. Unfortunately; the inner workings of PPLive are not available to the public. Some of the properties is given in [21].

3.1 BitTorrent Assisted Streaming System

BitTorrent Assisted Streaming System (BASS) [13] is a hybrid system for hosting video-on-demand services. In addition to a media stream server, BASS uses BitTorrent. See section 4.1. The idea is that the use of BitTorrent will reduce the load on the server.

When a user starts to stream a video file, the media server will stream at full rate to the user. At the same time, the user will start to download

pieces of the stream from BitTorrent clients. When the streaming enters a point where a data block already was downloaded by the use of BitTorrent, the streaming from the server is paused until new data is needed. Some of the stress on the server will therefore be removed as the streaming progresses, because larger parts of the file will be present later in the streaming session.

The advantage of such a system is that it is easy to deploy. BitTorrent is a highly established protocol, and can be used in the system with only minor changes.

There are some disadvantages: First, if the user decides not to watch the entire stream, the future bits downloaded by BitTorrent will be of little use to the user. Second, the user needs to be able to cache large portions of the media. And third, the user's downlink needs to both support the full bit rate from the media, and to receive from BitTorrent clients at the same time if the system is to work. This means that a user can only stream media with a bit rate well below the downlink capacity, and hence at a lower quality than optimal.

One possible flaw of BASS is that its users aren't rewarded for contributing their upload bandwidths. The tit-for-tat part of BitTorrent reward contributing peers, but if a user does not contribute to the BitTorrent part of the system, he or she will still be able to stream directly from the server in full quality. This could make users cheat, and if that happens BASS is equal to a pure client-server system. The system is also vulnerable to flash-crowds because most of the stream is sent from the server during start-up phase.

Peer-to-peer connections are of course made as in BitTorrent. While the

3.2 CoolStreaming

CoolStreaming [43], also known as DONet, is a data-driven overlay network for peer-to-peer streaming. There is no distinction between sender and receiver peers; directions of data flow are changed dynamically to what suits the system the best. For each stream in the system, there is one originating peer. The originating peer is the peer that initially served the stream to the network.

When a user initially wants to watch the content, it sends a request to the originating peer of the stream. The originating peer then selects at random one of the peers it knows to be the joining node's deputy. The joining peer will then contact its deputy to get a set of candidate peers, with which the joining peer will cooperate.

Each peer has a membership cache, which contains identifiers of nodes that participate in the session. Peers generate heartbeats, which are transported via a gossip protocol. When a peer receives a heartbeat, it updates its membership cache. Peers that are in a peer's membership cache, are candidates for partnership. Partners exchange buffer maps, data and membership caches. Members, on the other hand, exchange heartbeats only. To find better partners and to cope with partner departures, randomly selected peers from the membership cache are promoted to partners at regular intervals. The peer track a score for each of its partners, which is the maximum of average received segments and of average sent segments. If the peer has more partners than it needs, it can remove the one with the lowest score.

Each peer keeps a buffer map, which describes which segments of the stream the peer is in possession of. The buffer map is limited to a fixed number of segments, set to 120 in the original prototype [42]. Each peer also keeps buffer maps from each of its partners, which are updated regularly. The peer uses these maps to determine which segments it should request, and from which partner. High bandwidth and available time of a partner will also attract requests.

CoolStreaming has been deployed in the Internet and over 1 million unique IP addresses have used the system. More than 50000 users have used the system at the same time [42].

3.3 Peer-to-peer adaptive layered streaming system

Peer-to-peer adaptive layered streaming system [28, 34] (PALS) is concerned with how streaming from multiple senders to one receiver is to be done. PALS is not much concerned about how the peer-to-peer connections are formed. However, PALS seem to be very good at doing the actual moving of streamed data, which could be extendable to general p2p streaming systems.

The peer selection in PALS is as following (we quote [34]):"The receiver starts with a randomly selected peer from the list of available peers. Then it periodically adds another random peer from the list of available peers to the subset of active senders while monitoring variations of both overall throughput and throughput of individual senders. If the overall throughput increases, the new sender is kept. Otherwise, the receiver drops the new sender and tries another random peer after a period."

3.4 PROMISE

PROMISE [20] is a peer-to-peer streaming system that uses some topology information of the underlying network when creating connections in order to increase performance. When a user joins the system, it gets a set of 10 to 20 candidate sender peers by contacting a p2p content management system. The user then prompts all its candidates to measure both topology and available bandwidth between the candidate and the user. When that information is established, the user selects the best sender peers. If the performance of a sender peer suddenly drops, the user is allowed to switch to its standby candidates. During streaming, the receiver is passively monitoring the network, and constantly updating its topology map. PROMISE also supports random selection and non-topology aware selection of peers; this will come with less start up delay, but also with less performance.

In order to cope with packet loss, PROMISE divides the content into equal-sized segments and Tornado codes (a FEC code) the segments. The amount of protection is dynamically altered such that the coding overhead is kept reasonably low.

PROMISE has been evaluated by simulations of 1000 peers, and by Internet experiments. Three different peer selection algorithms were tested during simulation: random selection of sender peers, selection based on measurement of bandwidth, and a topology aware selection algorithm. The simulations indicate that the topology aware selection algorithm manages to let receivers receive at higher streaming rate than the other selection schemes evaluated with differences of 5% to 10% [20].

3.5 SplitStream

SplitStream [8] uses multiple application-level multicast trees to distribute a stream. The stream is divided into a set of stripes, and each stripe has its own multicast tree. SplitStream aims to create the trees so that any peer is an interior node in exactly one tree. Consequently, no peer looses more than one stripe if any of the peers should fail. The generation of the trees is distributed, the exact way of generating trees being implementation-specific. If SplitStream is implemented by the use of Pastry [35] and Scribe [9], tree construction can take advantage of some of the properties of Pastry. Pastry chooses routing table entries which has low delay whenever it can. Because the trees in SplitStream are constructed by looking at the Pastry overlay, it means that nodes that are neighbors in the

trees often will be physically close. The trees are extended as nodes join, and repaired when nodes fail or leave the network.

The content distributed by SplitStream should be coded so that peers can tolerate loss of one (or a few) stripes. This could for instance be done by use of multiple description coding. Peer failures would then only result in temporary degradations of quality, and only until the failed tree is rebuilt. If the distributed content is single-layered, it would be necessary for all downloading peers to receive the entire stream. This could be ensured either by using forward error correction codes at the cost of more network packets and CPU usage; or by allowing retransmittance of lost packets.

The main advantages of SplitStream are: its nodes are self-organizing, which is good for scalability, and it exploits locality (to the extent done in Pastry) which should result in relatively little load on the network.

SplitStream multicasts its content. This is suitable for live broadcasts, but not for video-on-demand.

3.6 Discussion

When the number of users of a streaming system is moderate, the backbone should normally not be a bottleneck because the aggregated requirements from all users would be moderate as well. However, if millions of television users should switch to Internet as a delivery service of high bandwidth demanding streaming, the backbone could become a bottleneck. This would be especially noticeable in presence of today's low upload/download bandwidth ratio of ADSL users; the server will have to provide lots of bandwidth per user, and all of this has to be transported on the backbone. If the upload/download bandwidth ratio was to improve, a larger part of the streamed data could be interchanged locally and hence not stress the backbone to the same extent.

For the above reason, it is important for a peer-to-peer streaming system to use minimum capacity of the backbone if the system aims to be scalable. This can be achieved in different ways, for instance by use of geographical scattered servers, or by letting peers exchange data locally.

Most current peer-to-peer streaming systems, as well as p2p file distribution systems, do not consider the underlying topology of the network when deciding which peers should mate with each other, implicitly or explicitly assuming that the Internet backbones have infinite bandwidth. The one of the above systems that explicitly considers properties of the underlying network in order to increase its performance by making clever peer-to-peer connections is PROMISE. SplitStream builds itself upon a p2p

content management system, and can implicitly exploit properties of the network if the underlying p2p system does so.

CHAPTER 4

OTHER RELATED WORK

4.1 BitTorrent

BitTorrent [1] is a protocol for distributing files. A BitTorrent file distribution consists of a BitTorrent tracker, a meta info file and peers. It also makes use of a web server and web browsers to distribute the meta info file. A meta info file describes the file to be downloaded: it contains the URL a tracker server, the piece size, SHA1 sums of all pieces of the file, the length of the file, and its name. If more than one file is distributed at once, they are concatenated, and the meta info file carry filenames and lengths for all of them. The tracker keeps track of how much of the file each peer has downloaded.

In order to download a file, a peer must first obtain its meta info file. The meta info file is normally found on a web server. The meta info file contains the address of a tracker, and the peer will request a list of other peers from the tracker. The tracker will reply with a list of peers, typically randomly chosen. The peer will connect to some of the peers of the list, and obtains information regarding which pieces each peer have.

Connections are bi-ended and are initially *choked* and *not interested* in both ends. Transfer of data only happen when the connection is *interested* in one end, and *unchoked* in the other end. Whether the connection end is interested or not, mirrors if the other end has a piece of the file which the first end does not. Choking a bit more complicated, and is done for several reasons: the number of simultaneous uploads should be limited to give good TCP-performance, the peer should reciprocate to peers who let it download, and the peer should try out new connections.

The BitTorrent protocol does not specify how the tracker should select the peer lists it sends to peers, neither how the peers should use these lists.

Typically, the tracker forms a list of 50 peers it chooses at random from the peers which are active in the distribution of a given file, when a peer

joins. By default, the joining peer will initiate connections to up to 40 of these peers. Also, other peers will discover this peer through communication with the tracker; some of these peers will initiate connections to the joining peer as well. Whenever a the count of a peer's connections drops below 20, the peer requests a new list from the tracker. [26].

In [41], a comparison between BitTorrent and an optimal scheme is done; where optimality is minimal total elapsed time. The finding is that in very heterogeneous networks, BitTorrent is far from optimal with a difference ratio of up to 40%. In homogeneous network, this ratio is much smaller, about 6%. However, the paper says that BitTorrent is designed to be good at minimizing average finish times, not at minimizing the maximum finishing time.

4.2 The Julia content distribution network

The Julia content distribution network can reduce the network load of up to 33% in comparison with BitTorrent, with a penalty of only slightly later finishing times for the users [6]. The reason Julia can perform so well, is because its design minimizes the maximum download time amongst peers, while BitTorrent rather minimize average download time. Hence, with the use of Julia, resourceful peers take longer time to download a file than with BitTorrent; and resource-weak peers take shorter time as compared to using BitTorrent.

4.3 Other related work

Most systems aim to do single-stream p2p streaming, silently ignoring the fact that there will be more than one stream at a time in the Internet. [24] proposes a priority oriented scheme for sharing bandwidth in a fair way when multiple users streams the same layered-coded content. The base layer gets high priority, enhancement layers get less priority. Layers then get dropped when congestion occurs, and low priority layers get dropped quickly, leaving room for the higher priority layers of other users.

One paper proposes a way of optimizing layer size of layered video in order to maximize satisfaction of the receivers, when receivers form groups of somewhat equal bandwidth [27]. Satisfaction is then modeled as an application-aware fairness index, which is defined as the utility divided by the maximal utility. Utility is a non-linear function of the received bandwidth, and is based on earlier established work. Maximal utility is

the utility that would appear if the user could use its full download capacity. The result of using optimized layer sizes is an up to 10% increase of overall satisfaction when compared to fixed layer sizes.

One problem of peer-to-peer systems is that peers often are behind firewalls or network address translation devices (NATs) which can reduce the peers abilities to participate resources. However, NAT traversal can be done on virtually any NAT service deployed [17].

CHAPTER 5

PEER SELECTION

There are two different places to put the management of the peer-to-peer connections that seems reasonable. The first place is a central server. The server can gather lots of information about the network since it is present for a long time. If that information can be utilized to its full extent, great performance could be achieved. The other place is at every receiving peer. Most p2p-ss do at the latter.

5.1 Control at the receiver

It is not strange that many p2p streaming systems and p2p file distribution systems let the receiver make most decisions. The receiver is aware of its needs and it stays during the entire session. The first means less communication and faster response. The second means there is no need to keep redundant control nodes. Control at the receiver is quick and easy. Knowledge of the underlying network can be stored in a p2p content management system, and used by the receiver to generate good connections.

When designing a p2p streaming system, one obvious goal is that the perceived performance of the system is as high as possible. That is, a receiver should not be allowed to downgrade the performance of another receiver. (Unless its gain is larger than their loss.) One way of approaching that requirement when using layered coding is to apply Intersession Fairness [24], in which a client easily gives up bandwidth that is used for high layer numbers in favor of base layers of other clients, if congestion occur.

When describing the selection algorithms below, it seems like much of the responsibility is placed on the server rather than on the peers. However, it should be possible to replace the server with a distributed hash table.

5.1.1 Random peer selection

The server has a list of every peer that is streaming, and the peer's progress into the streams. A new node will request a list of candidate peers from the server, and select peers at random from that list. The new node will discard peers it receives little bandwidth from, and connect to new nodes on its candidate list. When the candidate list is emptied, the new node will request a new list from the server.

5.1.2 Closeness guided peer selection

Selecting peers that are close has some benefits over selecting peers far away. Closer peers tend to have shorter round trip times between them, making retransmissions and communication in general quicker. It will also have the benefit of not moving data further than necessary, perhaps avoiding the creation of bottlenecks.

There are many ways of defining distance in the Internet. One can use physical distance by using an IP-to-coordinate system [25], round trip times, and hop count. Here we will rather use an approximation to closeness by using the information gained by use of traceroute towards a fixed point, the server. There seems to be three different reasonable estimates one can use from this information, see figure 5.1. The first estimate we call the length-from-last-common-router, and is the number of common links from peer A to the last common router, and then to B. By consulting the figure, this gives 5. The max-length-from-last-common-router is the maximum distance from the last common router to either A or B. The length from the common router to A is 3, which is the maximum. The last estimate (which is the one we will use) is the common-path length; the number of common links from server to the peers. The common-path length of peers A and B is 2.

Is it possible to find the K peers closest effectively? Of the tree estimates, at least one is computable with a complexity independent of the number of peers, the common path length. If one store the paths towards the media sender in a trie[1], it is not difficult to find the K "closest" in time linear in K, and linear in the maximum path length.

[1] A trie is a special case of a suffix tree [38].

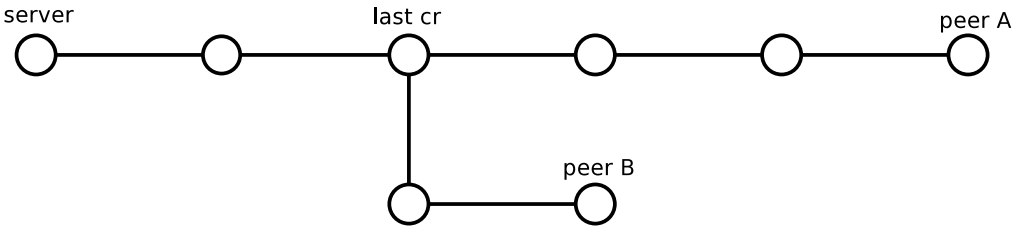


Figure 5.1: Topological distance heuristics

5.2 Centralized control

If one puts the control of peer selection in a centralized place, there will be advantages and disadvantages as compared to having the control at the receiver. A great advantage is that with centralized control, one will know who everyone is connected to. By using this information, some unfortunate connections can probably be avoided.

The downside of centralized control is that it does not scale well. Since a central unit would need to manage the peer connections of tens of thousands — perhaps millions of peers — the management would need to be incredible efficient.

It could be possible to optimize the network resources from a central node if one has knowledge of the underlying network, and a manageable number of peers. For a small ISP, serving its own customers, these assumptions might easily hold.

The centralized control must make sure a few constraints are not violated when selecting connections. First, links should not be oversaturated. Second, peers should receive their requested rates. And third, the central should make sure that the overhead of control messages is not huge. In other words: the central should try to minimize the deviation, E, from the acceptable situation, which we define to be:

$$E = C_l \sum_{\text{links}} a_l - C_b \sum_{\text{peers}} b_p + C_c n_{\text{control messages}} \tag{5.1}$$

Where a_l is the amount of overuse on the link l, and b_p is the received rate or satisfaction of user p. C_l, C_b, and C_c are parameters which should be adjusted to a given scenario. The parameters should always be non-negative numbers. For instance, if it is more important that users receive their requested rate than it is to not overuse a link, C_l should be reduced in favor of C_b. n is the rate of control messages.

Equation 5.1 cannot be solved by linear programming because the first term is non-linear. This comes from the non-linearity of each of the a_l.

Lets denote the use of a link l, u_l. When u_l is below a threshold, t_l, then $a_l = 0$. When l is above the threshold, then $a_l = u_l - t_l$. So unless, either all $t_l = 0$ or $C_l = 0$, linear programming is not possible to use. Whether the last term of equation 5.1 is linear or not, depends on the solution, making things even worse.

5.2.1 Simulated annealing

Simulated annealing [23] is a general heuristic method for optimization. The simulated annealing method got its name from its physical counterpart: annealing. It simulates the cooling process of a physical system: if a system of atoms is cooled slowly, it will obtain a low-energy state. Likewise, the simulated annealing process could also find a low-energy, or low-cost, state for a general optimization problem.

Assume that one has a system in a given state, and that one wants to find the minimum energy state of the system. A step in the simulated annealing process works by randomly altering the state a tiny bit, and then accepting or rejecting the new state based on if the energy difference between the new and old state. If the new state has lower energy, it is accepted immediately. If the new state has higher energy, the new state is accepted with the probability $e^{-dT/kT}$, where dT is the change in energy, T the temperature of the system, and k a constant which typically is adjusted to the problem. If the new state got accepted, the process continues with the new state. If not, the process continues with the old state.

The simulated annealing process starts with an initial temperature T_0, and an initial state. The temperature is gradually lowered, and at each temperature a number of steps are performed. A high temperature accepts many increments in energy, and a lower temperatures accepts almost only decrements in energy.

If the temperature is lowered slowly enough, the system should obtain a low-energy state, preferably the lowest.

5.2.2 Using simulated annealing

In order to minimize the deviation from the acceptable situation, we select to use the simulated annealing technique. An advantage of using simulated annealing, is that simulated annealing can be adapted to almost any need.

However, there are also problems with simulated annealing: if the simulated annealing doesn't converge quickly, it will probably be useless in the presence of large number of peers. And to converge fairly efficient,

it needs tuning the energy functions, temperature decrement function as well as the transforms. That is: almost everything. Even though everything is tuned, the simulated annealing might get stuck at a local maximum.

This thesis will not try to reduce the control overhead, so we set $C_c = 0$ in equation 5.1, and get the new energy function:

$$E_{c0} = C_l \sum_{\text{links}} a_l - C_b \sum_{\text{peers}} b_p \tag{5.2}$$

We will now try to minimize E_{c0} by applying simulated annealing.

Transforms

The most natural transforms would be the following:

- Add new connection (from, to, content, size)
- Split connection (original_from, additional_from, to, factor)
- Swap connection (original_from, new_from, to) - same as split connection with factor 1
- Scale connection (from, to, factor(0 to 1)) - factor 0 removes totally, only apply when this doesn't affect other receivers
- Recursive scale connection - scales receivers as well
- Adjust connection (from, to, delta_rate) - alters by delta_rate

Each of the transforms work over a number of links, and the execution time depends on the length of the connections. However, it should be somewhat beneficially to utilize locality, hence most connections should have few hops. The length of the connections can be affected easily by more often trying to establish few-hop connections (non-uniform selection of peers).

We only use one transformation, namely adjust connection. Also, we make sure that no peer ever receives more rate than it requested. Also, we needed to make sure that a peer cannot send a negative amount of rate, nor receive a negative amount.

As a cooling scheme, we starts with temperature $T = 1.0$, then multiplies the temperature with 0.97 to get the next temperature, we try 450 temperatures. Between each change of temperature we perform $3p$ steps, where p is the number of peers. We set k to equal the average stream rate

of the peers. Both C_l and C_b are 1 in our simulation. The parameters have been set like this after some trial and error, and seem to work fairly well. The implementation can bef found in appendix C

CHAPTER 6

SIMULATION FRAMEWORK AND SIMULATION

Through the use of simulations we hope to figure what makes a good peer selection algorithm for use in a p2p streaming system.

The simulations are built on top of network simulator ns-2 [3]. Most p2p simulators are built to simulate huge amounts of peers, and therefore (more often than not) does not consider the underlying network because of performance issues. This thesis goes in the opposite direction, by simulating atop of a detailed packet-level network simulator, the simulations could not support huge amounts of peers. During such a low-level approach, details that could have been missed by higher level approaches might show up. Discovery of such details could then be inserted into a high level approach to simulate the performance of a system when huge numbers of peers are involved. The reason for choosing ns-2 that it is the closest to a standard network simulator today; thereby making it easier for other persons to utilize or expand the framework.

According to [29] there are currently no p2p simulator good enough to become a standard p2p simulator; like ns-2 has become for other areas of network research. Typical short-comings of the simulators are poor or non-existing documentation, no support for underlying network, difficulty of gather statistics, and low scalability. However, most papers on p2p systems which use simulations, either use a custom simulator or an unspecified one. Ns-2 has been used to simulate streaming in p2p systems in at least two papers: [34] and [36].

Different peer selection algorithms will be implemented to complete the simulation. The different peer selection algorithms will then be evaluated based on the results from the simulation. Average peer contribution, startup delay of streams and jitter will be examined. A subjective measure of usability for VoD service and live television of the different selection algorithms will be given as well.

6.1 Framework

All simulations are based on our simulation framework. The framework is designed to be simple, intuitive and to match the division into parts as given in figure 2.1 on page 20. Note that the pre-buffer and the post-play management here fusion into a buffer module, and that the congestion control is performed in sender and receiver modules.

The framework modularizes a p2p-ss into the following modules: media player module, peer discovery module, peer selection module, sender module, receiver module, buffer module, segment piece manager. Transport protocols and congestion control is intended to be done by the sender and receiver modules. The framework also has a statistics module to simplify how collection and reporting of performance is done. The media player module reports details on startup delay and jitter (pauses in playback because the buffer is empty).

The framework includes some simple variants of all the modules.

Normally, interactions between modules are as drawn in figure 6.1 on the facing page. Before sending a data packet, the sender module will contact the piece manager (and in some cases) the buffer module to see if it actually has data to send. The peer discovery module will create packets to initialize the discovery process, and also read replies. It will notify then notify the peer selection module about the newly discovered peers. The peer selection module will then run its peer selection algorithm, and notify the receiver module. The sender module can also be notified (although not shown in the figure).

When a packet is received, most of the modules will read data from the packet.

The simulation considers distribution of data streams. The streams are partitioned into **segments** of equal size, each segment is in turn partitioned into a number of **segment pieces**. The size of a segment piece is equal to the net size of a network packet, which we assume to be limited by the Ethernet frame size of 1500 bytes. In the perspective of the media player, the segment is the fundamental type; play out of a segment cannot

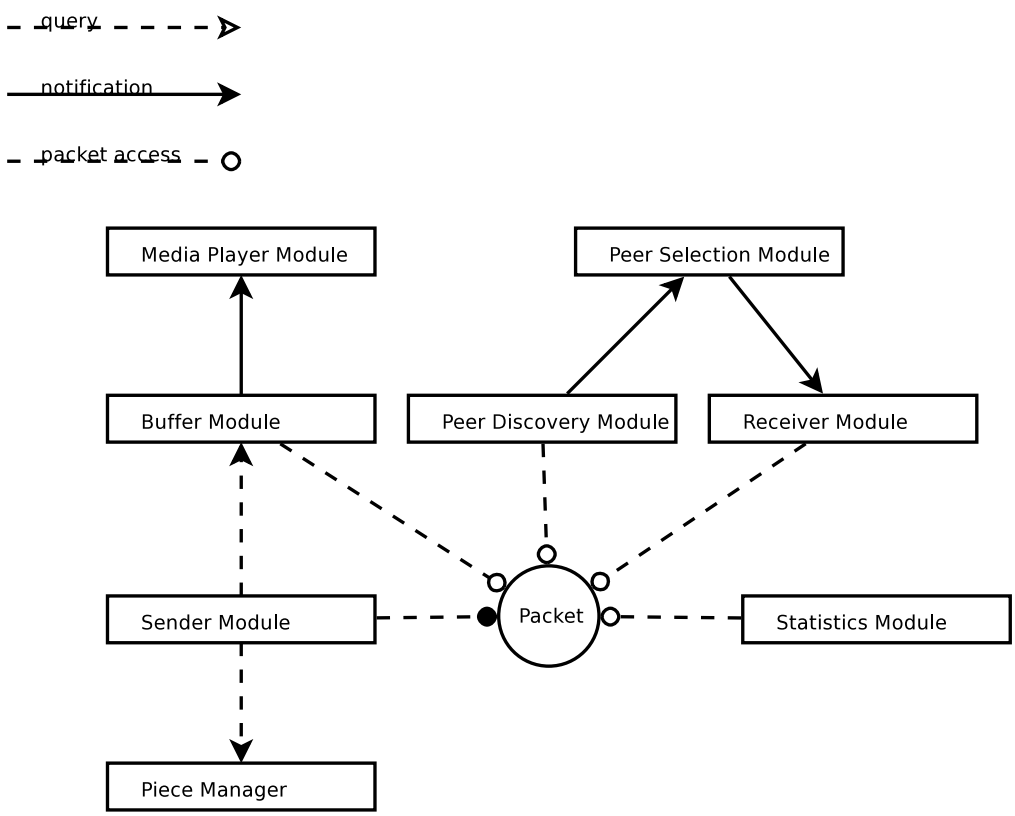


Figure 6.1: The modules of the framework, and their typical interactions.

start before the segment is completely downloaded.

A **sender peer** is a peer that is currently sending parts of a stream to a **receiver peer**. Any peer is capable of both acting as a sender peer and as a receiver peer. A peer is not allowed, however, to send data to itself. A **node** is either a peer or a server, used when suitable. The phrase **sender node** is used to denote a sender peer or a server.

During the simulations, the default data packet size is 1500 byte from which 40 byte are reserved for IP, UDP, and RTP headers. A maximum transmission unit of 1500 byte might not be supported in some real networks, but is typically the maximum size possible, because of limitations of Ethernet. Using a larger packet size would mean fragmentation, and lower performance. It is reasonable to use IP, UDP and RTP headers in a streaming system. In some cases it would probably be possible to swap the RTP header for a custom header in order to save a few byte.

The framework consists of different modules, and each module can be retrofitted individually. It is also possible to provide different module implementations for peers and servers. The modules are: media player

module, buffer module, receiver module, sender module, peer-selection module, peer discovery module, segment piece manager, and statistics module.

When looking at different peer selection algorithms, it is natural to make multiple implementations of the peer-selection module and the peer discovery module. Then the different implementations of the two modules can be simulated and compared without altering other modules of the system.

If one is interested in how a system behaves with different transport protocols or congestion control algorithms, then it is natural to implement different sender and receiver modules.

The segment piece manager is somewhat special. It is the only module not residing in every peer or server. Instead there is one segment piece manager per stream, and the sender module contacts the manager in order to figure out from which segment it should send content, if at all.

6.2 Standard modules

The framework also has a few basic modules. We now describe basic implementations of all modules, except for the segment piece manager which is explained in section 6.3. The standard modules are intended to be as simple as possible, not to have great performance.

6.2.1 Media player module

The standard media player module is very simple. It starts playing the stream when the amount of buffered content is above the "low buffer limit" of the active stream. When the player finishes playing of a segment, it moves on to the next. Should the next segment be missing, the player will pause until buffered content is again above minimum threshold.

Every event, such as pauses, restarts or changes of streams; are logged.

6.2.2 Buffer module

The standard buffer module is simple as well. It stores every piece it gets, and when it observes that a new segment is completed, it notifies the media player module.

6.2.3 Sender module

The sender module is not very complicated either. It receives "stream request messages" from other peers, and serves pieces of streams to its receiver peers. The request message includes information on which stream the requesting peer needs, and at which rate the requester wishes to receive from this sender. The requesters are put into a list.

The sender module has a timer which expires with intervals which correspond to the maximum upload rate of the sender. At each of the timer ticks, the sender searches for a peer to serve. The first peer it finds which it can send a piece to (never send faster than the peer requested, and ask the segment piece manager if there is a piece to send) is served. The sender uses a round-robin scheduler, making sure its rate is distributed approximately fairly amongst receivers.

The standard sender module sends each piece using an UDP agent, and performs no congestion control. Note that the rate can be adjusted by the receiver, thus making it possible to perform congestion control without using a different sender module.

6.2.4 Receiver module

The standard receiver module is simple. It receives notifications from the peer selection module on which peers to request information from, and also at which rate content should ideally be received from each of the sender peers. When the sender receives a notification from the peer selection module, it will send request messages to each of the sender peers the peer selection module selected, containing the rate decided by the peer selection module.

As one can see, no congestion control is performed in the standard receiver module.

6.2.5 Discovery module

The standard discovery module performs discovery by having peers request candidate senders from a server.

The discovery module is therefore divided into two pieces. The first piece is the server version, and the other is the peer version. When a user changes stream, the p2p streaming system needs to get a new set of sender peers, since it is not likely that all peers change to the same stream simultaneously. When the change occurs, the peer sends a 'candidate request'

message to the server[1].

The server will reply by sending a message containing peers, selected at random from all peers which download the actual stream.

When the peer receives a peer list, it notifies the selection module if it received new peers.

6.2.6 Selection module

The standard selection module select the first K peers it receives from the discovery module. K is a number describing how many senders a peer is allowed to request content from at a given time. It distributes its required bit rate evenly between the K selected peers.

The selection module also request estimates from the receiver module on how much rate is received from each of the selected peers. If the cumulative rate do not match the need of this peer, the worst performing peer is replaced with a new peer. Then the difference is requested directly from the server.

6.2.7 Statistics module

Every second, the statistics module writes statistics on play progress of the media player, buffer progress, rate received from peers, rate received from servers, which stream is active, etc.

6.3 Segment piece management

In this simulation framework, peers do not know which pieces they are in possession of. A peer only knows how many pieces of each segment it has. When a sending node wants to send a piece to a receiver peer, it contacts what we call an oracle, and asks the oracle if it can send a piece to this receiver peer. The oracle will then determine if the node can send a piece. The sender node is not told which piece was selected, only the oracle is allowed to know.

The oracles allows each receiver peers to download multiple segments simultaneous, but will always select to send a piece of the lowerest numbered segment, if possible. That is, if the oracle's rules indicate that the sender can send both a piece from segment 1, and from segment 2; the the

[1]How the peer gets to know the address of the server, is not a part of the system. A way of obtaining information about streams or the server address, is to use a web site. There are of course a lot of other possibilities as well.

first piece will be selected. This is done because segments close to the play out point is more critical than later segments.

By taking this approach, a peer does not need to decide at send time which piece is to be sent. Instead, an oracle chooses which piece to send, and only the oracle knows which piece was sent. Such an oracle can have different properties, some of the more important are:

Validity The oracle should not allow a peer to send a piece it does not have. In the real world, a peer obviously cannot send data it does not have. Violating the validity property could make data appear out of nowhere. We require data to first appear at the server. Since the peer only know how many pieces it has, not which, it must be the responsibility of the oracle to ensure validity.

Duplicate pieces avoidance It is a waste of bandwidth to send a piece to a peer which already have the piece. Also, if the oracle allows duplicate pieces in transit to a peer, it must make sure that the duplicate pieces only count as one piece towards the peers' piece counters.

Bidirectionality If two peers have at least one piece the other misses from some segment, the oracle should allow both peers to send the missing pieces.

Response strategy The sender nodes will request permission to send a piece from the oracle to a given receiver the oracle can have different response strategies: a greedy oracle will grant a permit if it can see a piece which is possible to send. More sophisticated oracles could for instance use statistical analysis as well. All oracles described here will use the greedy response strategy.

Under the assumption that the oracle is valid, a peer knows a segment is complete when it has received the correct number of pieces of the segment. It is naturally impossible to do segment piece management with oracles in the real world, but possible in simulation. The advantage of doing segment piece management like this, is a possibility to experiment with different ideal strategies for selecting the pieces. E.g. in order to figure out if it is beneficial to arrange the pieces such that a peer can send any piece as early as possible, we implement an oracle which arranges pieces such that a send-request is granted by the oracle if there was any ordering of the sent pieces which would allow the sending. If the peers should make the decisions at send time, it would be very difficult to guarantee such an ordering. Exploring such ideal strategies should give directions on how to design real-world segment piece management.

6.3.1 Oracle with incremental strategy

A simple oracle would do it decision like this: Peer A is allowed to send a piece from segment S to peer B, if peer A has more pieces of the segment than do peer B. Also, the difference must be larger than the number of pieces in transit to peer B.

It is easy to see that this is a valid oracle. Also, duplicate pieces are avoided. However, this oracle is not bidirectional, because only the peer with the highest number of pieces are allowed to send.

6.3.2 Oracle with rarest-first strategy

We concluded that the incremental strategy for an oracle suffered because of lack of bidirectionality; not allowing two peers to cooperatively download a segment.

The oracle keeps track of which pieces each peer have, and which pieces that are in transit to each peer. Now, peer A wants to send a piece of segment S to peer B. The oracle examines the pieces of peer A and peer B. If A actually has a piece that B do not have, the oracle will further examine all peers that recently has sent a piece to peer B. The piece which is least common among these other peers, and not already in transit to B, is then sent to B. Although selecting the least common piece is not guaranteed to give an optimal use of bandwidth, it seems like a good idea.

Again, this is obviously a valid oracle, and the oracle also avoids all duplicates. The oracle also allows two peers which is downloading the same segment to cooperatively download the segment. In fact, this oracle looks a bit like the segment piece management of CoolStreaming [43].

6.3.3 Oracle with early-send strategy

We shall now describe an oracle that is bidirectional, has duplicate piece avoidance, but is not completely valid. Given how many pieces each peer has received from each segment, and a request to send from a given peer to another; this oracle aims at ordering the pieces such that the request can be permitted. Note that this is not the same as maximizing the probability to accept requests. It is possible that the choice to accept the given request actually will lower the probability of acceptance of later requests.

Consider figure 6.2 on the next page. The server S has sent 3 different pieces to peer A. A again has sent 2 different pieces to peer B, and 2 different pieces to C. Is B in possession of a piece which C does not have? This is not possible to decide. If A had to decide which piece to send at send

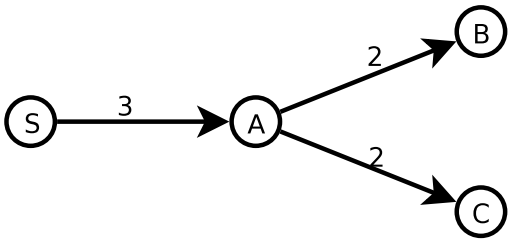
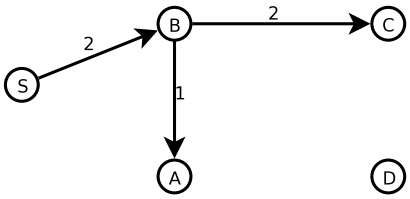


Figure 6.2: Idea behind the early-send oracle

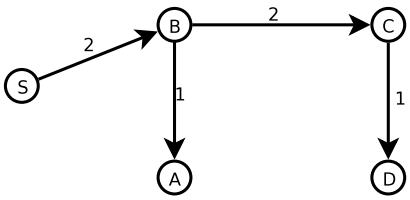
time, it might end up sending pieces 1 and 2 to both B and C. A better choice would be to send for instance 1 and 2 to B, and 2 and 3 to C. Then B can send a piece to C (and C to B as well). By just counting how many different pieces each peer has, instead of explicitly naming the pieces the peers posses, and assuming that the oracle always make a clever choice, we should be able to select the latter scenario, hence B can send to C. Unfortunately, this conclusion is not correct. In the case peer A had sent the 4 pieces before it received its third piece, the only possible scenario is that B and C have the same two pieces, and B cannot send to C! If, and only if, the conclusion had been correct, the oracle would be valid. The implication of using a not valid oracle in the simulations, is that the oracle will allow more pieces to be sent than a valid oracle would (if they otherwise are identical), hence the oracle which is not valid should make the p2p-ss have a greater performance in terms of delivered bandwidth. It can be seen from the simulation results in chapter 7, that the invalid early-send oracle has less performance than the rarest-first oracle. A valid early-send oracle would have even less performance! Therefore we do not bother adding much more complexity in order to make a valid early-send oracle; we simply conclude that the rarest-first strategy is better than the early-send strategy.

In order to efficiently decide if peer B can send a piece to C, we let us inspire by the Ford-Fulkerson method of solving the maximum flow problem (see appendix A). There will be one capacity graph which is common to all peers, like figure 6.2 and 6.3(a) on the next page, and the capacities on an edge from u to v is the number of pieces received at v from u. There is also a flow graph and a residual graph for each of the peers. In short, peer B can send a piece to C if there is an augmenting path from any server to B in the residual graph of C.

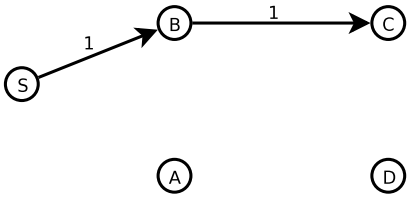
Example: Lets focus on segment 1 of a stream. Server S has sent 2 pieces to peer B. Peer B has sent two pieces to peer C, and one piece to peer



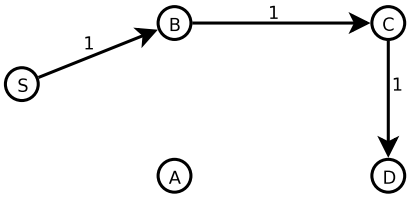
(a) Initial capacities



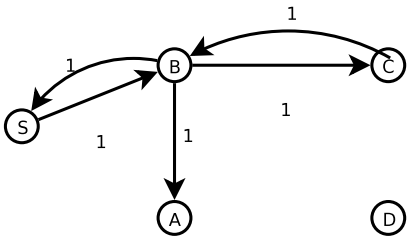
(b) Capacities after the piece is received by peer D



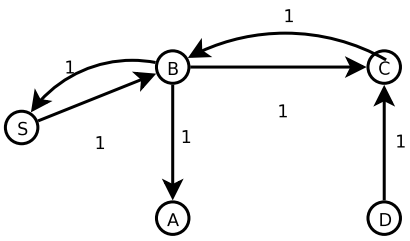
(c) The flow, as seen by peer D, after adding augmenting path S → B → C



(d) The flow, as seen by peer D, after the piece is received by peer D



(e) The residuals, as seen by peer D, after adding augmenting path S → B → C



(f) The residuals, as seen by peer D, after the piece is received by peer D

Figure 6.3: Snapshots of the maximum flow approach to manage segment pieces. Note that each of the peers will have its own flow graph, while the capacity graph is shared between peers. The residual graphs are calculated by "subtracting" the flow graphs from the capacity graph, and then adding to the opposite edges.

A. Peer D has not received any pieces. We need to know whether peer C can send a piece to peer D. The scenario described above is treated as the maximum flow problem with capacities on the edges as described in figure 6.3(a) on the facing page. However, since we only need to send one piece, we are not interested in finding the maximum flow, it is enough to find an augmenting path. The augmenting path need to start at the server, and end at the potential sender, peer C. As we see, there is only one option: The path S $\rightarrow$ B $\rightarrow$ C. The augmenting path tells us that peer C has at least one piece that peer D does not have. Therefore peer C can send a piece to D.

When peer C sends a piece to peer D, this is registered by adding the augmented path, from S to C, to the flow-graph owned by peer D. See figure 6.3(c) on the preceding page. It is also registered that one piece is in transit towards peer D. The latter is done so that other sender peers do not send pieces towards peer D, which already are in transit. When the piece is received by peer D, the capacity from C to D is increased by one. See figure 6.3(b) on the facing page. As we can see, the oracle will allow peer C to send exactly one more piece to D if peer C requests it.

Discussion: Does this Ford-Fulkerson-inspired method with finding augmenting paths actually success in deciding whether peer B can send a piece to C or not? Under the false conclusion given above, it actually does. If we translate the question into Ford-Fulkerson terms, it becomes: Can the flow to C be increased by one if we increase the capacity from B to C by one, and limit the capacity from any other peer to C to what is already sent? It is not difficult to see that this is equal to finding an augmenting path from any given server to B.

6.3.4 Multiple segments

The example of the last section looked at a simplified version, where all peers downloads the same segment. Normally, all peers do not download the same segment at the same time. Also, a peer does not want to wait for the whole of segment 1 to finish before starting to download segment 2. Imagine what happens if a far-away sender peer sends the last piece of segment 1: should the peer wait to start on segment 2 until the last piece arrives?

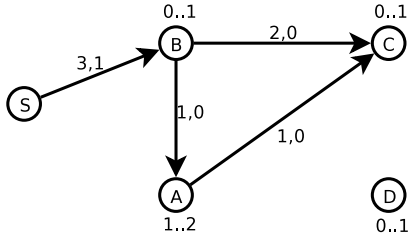
In our management we let each peer download up to K ($= 2$ in the implementation) segments simultaneously. The K segments cannot be separated: K is 2, a peer is for instance allowed to download segment 2 and 3

simultaneously, but not 2 and 4. The management will always try to fill the lowered numbered segment first. A nice observation is that the capacities (the number of different segments sent from a sender peer) towards a peer are implicitly given both when the peer has downloaded a segment completely, and when the peer has not started the segment. Therefore, it is only necessary to consider the K segments which the peer is currently downloading. Different peers will also have different progress, adding even more complexity.

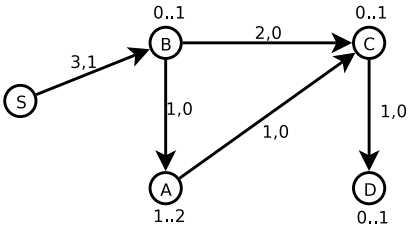
Example See figure 6.4(a): Peers B, C and D are downloading segments 0 and 1 (numbers above the peers). Peer B has received 2 pieces of segment 0 and 1 piece of segment 1 from the server. The first number on an edge correspond to how many pieces the receiver has received the lowest segment, and the second number corresponds to the highest segment number. Peer C has received 2 pieces of segment 0 from B. Peer A, which is downloading segments 1 and 2, has received one piece of segment 1 from peer B. It is also implicitly given that peer A has finished downloading segment 0. Peer D has not received anything.

Now, peer C wants to know if it can send anything to peer D. It tries segment 0 first, since it is the lowered-numbered segment of D need. D then probes for an augmenting path in the residual graph of D. Since D has yet to receive anything, the residual graph is equal to the capacity graph. Peer C can find two paths. The two paths are $S \rightarrow B \rightarrow C$ and $A \rightarrow C$. It does not matter which is chosen. Note that peer A acts as a server because it has the whole segment 0.

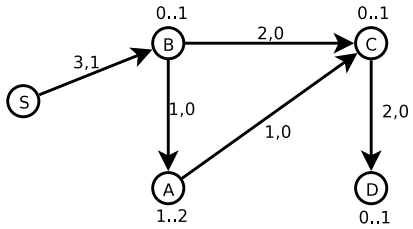
Next, peer B receives another piece of segment 0 from the server. Assume there are 4 pieces per segment. Now peer B has a complete lower segment, so it changes its active downloading segments to be 1 and 2. See figure 6.4(d).



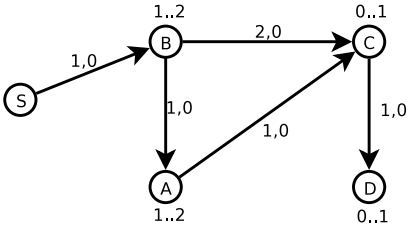
(a) Initial capacities.



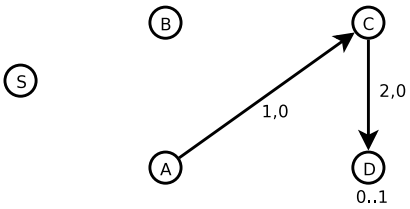
(b) Capacities after peer D has received a piece from peer C.



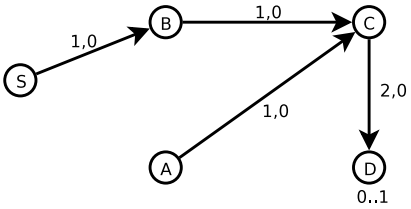
(c) Capacities after peer D has received two pieces from peer C.



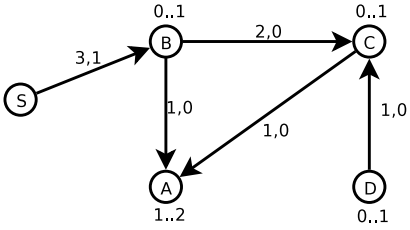
(d) Capacities after peer B has received a piece from server S.



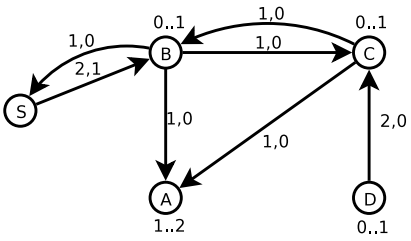
(e) Flow after D has received a piece from C



(f) Flow after D has received two pieces from C



(g) Residual graph after D has received a piece from C



(h) Residual graph after D has received two pieces from C

Figure 6.4: When multiple segments are downloaded simultaneous, the complexity rises.

6.4 Simulating the different peer selection algorithms

We now describe in more detail how the different peer selection algorithms are simulated.

6.4.1 Random Peer Selection

When a peer starts to stream, or changes to another stream, the peer will ask the server for candidates to download from. The server replies by sending a list of randomly selected candidates to the peer. The list consist only of peers which the server knows has some of the stream in question. The peer will request updates to the list at constant intervals as well.

When the peer gets the list, its selection module will select the first candidate peers on the list, and request downloads from them. The selected candidate peers will be called active senders from now on. The maximum number of active senders can be changed at simulation start, and is 5 in the simulations done later.

A downloading peer will at intervals calculate its need, which is how large a rate the peer should receive in order to feed its media player continuously. When the buffer level of the peer gets below a lower threshold, the need is adjusted to be 1.2 times the rate of the streamed media. If the buffer level for some reason should get larger than a upper threshold, the need is reduced to 0.9 times the rate of the media. If neither of the thresholds are broken, the need is simply the rate of the stream.

In order to reduce congestion, a receiver associates a limit to all its sender peers. The receiver does not request rate at above the limit. If a receiver receives less bitrate from a sender than it requested, it assumes this is because of congestion and sets the limit to the received rate. Every 2 seconds, the following is performed: First, limits are updated as described above. Then ehe request rate of all peers are first set to the lesser of: the limits of the peer, the last requested rate of a peer plus an increase. If now the sum of the request rates is more than the need of the peer, all request rates are multiplied with the ratio of the need over the sum. These request rates are then sent to the sender peers. If the received rate was less than the need, the worst (lowest receive rate) sender peer is swaped for a new peer, and blacklisted. Blacklisting is done so the peer does not immediately reselects the thrown-away peer. The server is treated just like the peers.

6.4.2 Closeness Selection

When a peer want to start to stream, it first performs a traceroute towards the server. If the stream already has performed an earlier traceroute to the server, the traceroute process is skipped. When the traceroute process is finished, the peer requests candidates from the server by sending a packet containing the stream identificator of the stream it wants and the list of routers between it self and the server. When the server receives the packet, it elects as candidates the "closest" peers to the new peer. Closeness is determined by how many routers the new peer have in common with the candidate; many routers means they are close. The candidates are put into a list, with the closest candidate first.

When the peer gets the list, its selection module will select the first candidate peers on the list, and request downloads from them. Since the "closest" peer is in the first entry, the peer will select its closest peers first. The maximum number of active senders can be changed at simulation start, and is 5 in the simulations.

Need calculation is identical to the need calculation when using random peer selection(see over). If the received rate is less than the need, the difference is requested from the server. If the difference is greater than a threshold, the peer will swap its worst sender with a new peer, and blacklist the worst sender. The server is never blacklisted, since it is used to request the difference.

6.4.3 Central server and simulated annealing

When a peer wants to start a streaming session, it tells the server. (In the current implementation, there are no packets sent, the peer calls a function in the server — hence startup delays are shorter than they should be. This will be corrected). The server then tries to optimize bandwidth usage over the whole network in order to, hopefully, serve the need of every peer in the network. The server will notify the sender module of all peers to whom and how much to send. The sender modules then do their best. The details of the algorithm can be found in section 5.2.

CHAPTER 7

SIMULATION RESULTS

The test runs included here were performed on sample network topologies from the GT-ITM tool [7], obtained from [2]. There are 100 routers, distributed between stub domains and transit domains. Bandwidths between transit domain routers are 15Mbps. Bandwidths between stub routers, and between stub routers and transit domain routers is uniformly selected from 6Mbps to 9Mbps. 100 user nodes are connected to randomly selected stub routers. The user nodes are divided into three groups. Group 1 have uplink bandwidth 1.3 - 2.0 Mbps. Group 2 have 1.8 - 3.0 Mbps. Group 3 has 0.3 - 0.7 Mbps. The users' download links also varies, but are always sufficient for downloading at least 1.5 Mbps. The server is connected to a random stub router, and its bandwidth is 6Mbps. The bandwidths are selected to challenge the peer selection algorithms, not to be an accurate model of the Internet.

All runs are performed on 8 different networks: 4 different sample graphs from the GT-ITM tool, and each sample graph are assigned random bandwidths twice.

Between time 0 and time 3.5 minutes, one new peer requests stream 1 each 2.1 seconds. At time 3.5 minutes, all peers have requested stream 1. At 4 minutes 40 seconds, all peers switch to stream 2. The first 4 minutes 40 seconds tests the capability of the system to handle VoD-like workloads. Although it is unlikely for the users to arrive with constant time difference, this will show how the system performs if the stream is is distributed. All users which tune in to stream 1, starts playback from segment 0 (this is a VoD session). When all users suddenly change to the next stream, the users are a flash crowd. So the latter part will tell how the system performs under such events. The rate of the streamed media is 1.5Mbps in all cases.

The peer selection algorithms tested here, are the ones described in section 6.4.

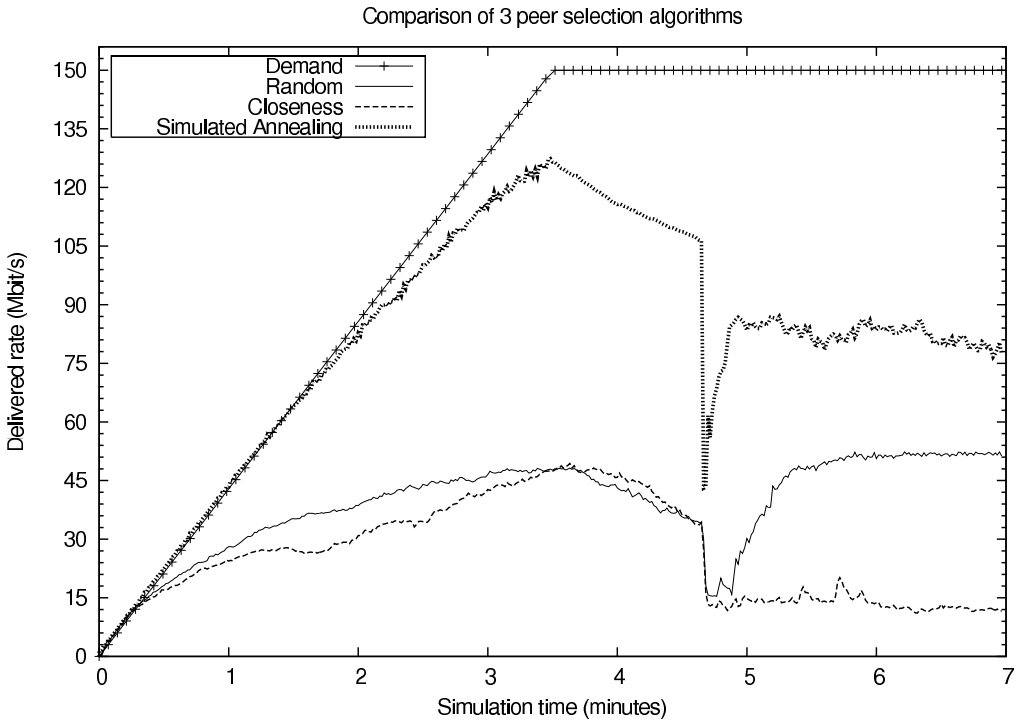


Figure 7.1: Comparison of 3 peer selection algorithms.

7.1 Comparison of peer selection algorithms

We see in figure 7.1, the three peer selection algorithms differ quite a bit in performance. The centralized simulated annealing approach outperforms the two other algorithms, achieving more than twice the throughput during when streaming stream 1. During stream 2, the simulated annealing approach is still the best, with 50% more throughput than the second best. The clear drop in delivered bandwidth at time 4 minutes 40 seconds is due to stream change.

This clearly indicates that proper use of information of the underlying network indeed can increase the streaming rate significantly. Our simulated annealing approach cheats by directly (instantly) notifying peers of its decisions, instead of sending notifications through the network. This is done because we think it is possible to tweak the simulated annealing approach to send few notifications, without getting much worse throughputs. Since our heuristics do not minimize notifications, we chose to send no packets instead of a lot.

A surprising finding is that random selection of peers seems to perform better than closeness guided selection, especially during the flash crowd.

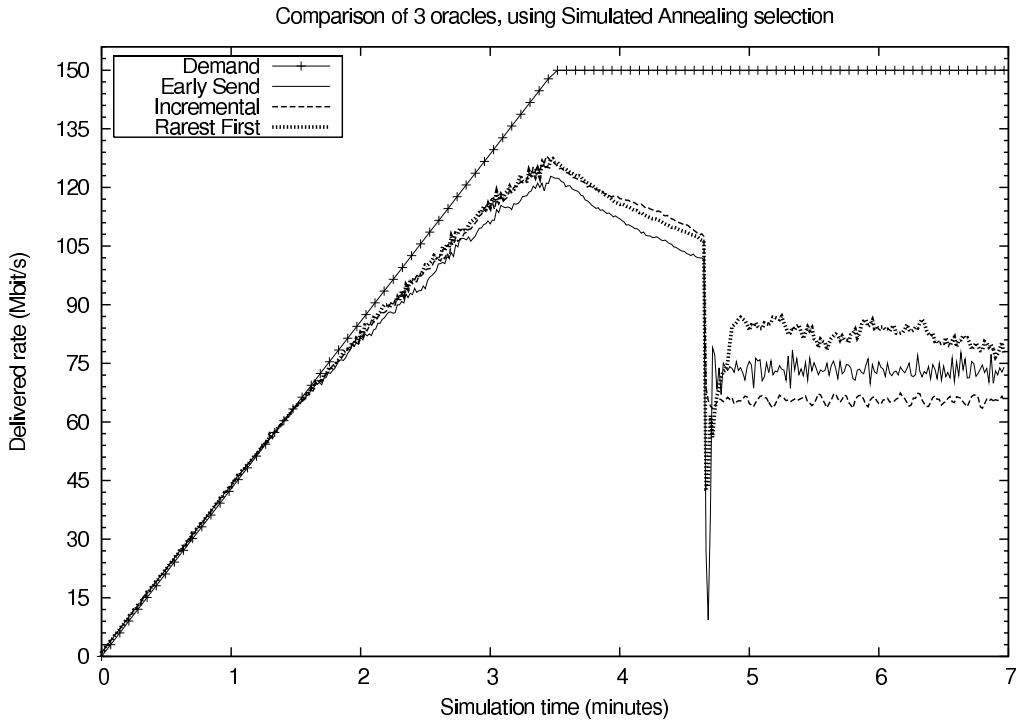


Figure 7.2: Comparison of 3 segment piece oracles.

7.2 Comparison of segment piece oracles

Not surprisingly, there is some impact on the received rate from the choice of segment piece management. In figure 7.2, we have plotted the rate peers receive when using different segment piece oracles. We see that the highest throughput is associated with the rarest-first oracle. This happen even though the early-send oracle is not completely valid, and therefore sometimes 'spawns' segment pieces. The incremental oracle achieves the worst performance, this is of course because it does not allow peers which downloads the same segment to cooperate. The clear drop in delivered bandwidth at time 4 minutes 40 seconds is (as in last section) due to stream change.

During stream 1, the oracles performance of the oracles are quite similar. During stream 2, there is more difference. During the first stream, the peers tend to have different progresses: the first peer to start have progress greater than the second, and so on. A sender peer will often have completed the whole segment, which the receiver requests. Therefore, the impact of segment piece management is little. During the second stream, peers tend to have the same progress, so the segment piece management

is much more important.

However, the throughput seems to be more steady when using the incremental oracle, and less steady when using the rarest-first oracle.

7.3 Discussion

Given the topology here, the centralized simulated annealing easily outperform by far both random peer selection and closeness selection. We see that rarest-first piece management achieves the highest throughput amongst the piece manager oracles tested here.

The results here further indicates that peer selection algorithms have greater impact on the performance of a p2p streaming system than segment piece management does.

CHAPTER 8

CONCLUSION

Although the connectiveness amongst peers in a peer-to-peer streaming system is only a small part of the system, it is perhaps the single factor that affects the performance the most. Most systems do not explicitly utilize knowledge of the underlying network when they make connections, which does not yield optimal performance.

This thesis compared three different peer selection algorithms: Random peer selection, selection of close peers, and a centralized peer selection algoritm. We found that the choice of peer selection algorithms greatly affects the performance of a p2p-ss: Our centralized peer selection algorithm, which uses knowledge of the underlying network, greatly outperformed the other algorithms in the simulated network. Doing the peer selection decision in a sentral node, does not scale well, and are therefore only usable when a limited number of peers are active.

In addition, we compared three different piece manager oracles. We found that the choice of oracle also affect the performance, but not to the same extent as the choice of peer selection algorithm. The best of the three tested oracles, was the rarest-first oracle. This indicates that a piece manager should try to send the rarest pieces first.

Future work It would be interesting to see how random peer selection and selection of close peers perform when using different congestion control algorithms. We expect that better performance would be achieved.

It would also be very interesting to be able to simulate with much greater numbers of peers, and for a longer time, to use multiple-layered streams, to incorporate control overhead into the centralized algorithm, and to have realistic cross-traffic.

BIBLIOGRAPHY

[1] BitTorrent protocol. `http://www.bittorrent.org/protocol.html`, last visited January 26, 2008.

[2] Gt-itm site. `http://www.cc.gatech.edu/projects/gtitm/`, last visited January 20, 2008.

[3] The network simulator ns-2. `http://www.isi.edu/nsnam/ns`, last visited February 1, 2008.

[4] PPLive. `http://www.pplive.com`, last visited February 1, 2008.

[5] Claude Berrou, Alain Glavieux, and Punya Thitimajshima. Near shannon limit error-correcting coding and decoding: Turbo-codes (1). In *IEEE International Conference on Communications (ICC 93)*, volume 2, pages 1064–1070, may 1993.

[6] Danny Bickson and Dahlia Malkhi. The Julia content distribution network. In *WORLDS'05 Second Workshop on Real, Large Distributed Systems*, 2005.

[7] Kenneth L. Calvert, Matthew B. Doar, and Ellen W. Zegura. Modeling Internet topology. *IEEE Communications Magazine*, 35(6):160–163, June 1997.

[8] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. Splitstream: High-bandwidth multicast in cooperative environments. In *ACM Symposium on Operating Systems Prinsiples (SOSP)*, October 2003.

[9] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony I. T. Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications*, 20(8):1489–1499, oct 2002.

[10] Yang-Hua Chu, John Chuang, and Hui Zhang. A case for taxation in peer-to-peer streaming broadcast. In *ACM SIGCOMM'04*, 2004.

[11] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[12] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design*. Addison Wesley, 4th edition, 2005.

[13] Chris Dana, Danjue Li, David Harrison, and Chen-Nee Chuah. BASS: BitTorrent assisted streaming system for video-on-demand. In *IEEE 7th workshop on multimedia signal processing*, 2005.

[14] Sally Floyd. RFC2914 - congestion control principles. (Best Current Practice), September 2000.

[15] Sally Floyd and Kevin Fall. Promoting the use of end-to-end congestion control in the Internet. *IEEE/ACM Transactions on Networking*, 7(4):458–472, August 1999.

[16] Sally Floyd, Mark Handley, Jitendra Padhye, and Joerg Widmer. Equation-based congestion control for unicast applications. In *ACM SIGCOMM 2000*, August 2000.

[17] Saikat Guha and Paul Francis. Characterization and measurement of TCP traversal through NATs and firewalls. In *Internet Measurement Conference (IMC'05)*, pages 199–211, 2005.

[18] Lei Guo, Enhua Tan, Songqing Chen, Zhen Xiao, and Xiaodong Zhang. Does Internet media traffic really follow Zipf-like distribution? In *ACM SIGMETRICS'07*, June 2007.

[19] M. Handley, S. Floyd, J. Padhye, and J. Widmer. RFC3448 - TCP friendly rate control (TFRC): Protocol specification. January 2003.

[20] Mohamed Hefeeda, Ahsan Habib, Boyan Botev, Dongyan Xu, and Bharat Bhargava. PROMISE: Peer-to-peer media streaming using CollectCast. In *ACM MM'03*, 2003.

[21] Xiaojun Hei, Chao Liang, Jian Liang, Yong Liu, and Keith W. Ross. Insights into PPLive: A measurement study of a large-scale p2p IPTV system. In *IPTV Worksop, International World Wide Web Conference*, May 2006.

[22] Daniel Hughes, Geoff Coulson, and James Walkerdine. Free riding on Gnutella revisited: The bell tolls? *IEEE Distributed Systems Online*, 6(6), June 2005.

[23] S. Kirkpatrick, jr C. D. Gelatt, and M. P. Vecch. Optimization by simulated annealing. *Science*, 220(4598):671–680, may 1983.

[24] Wei Kuang Lai and Chieh Ying Pan. Achieving inter-session fairness for layered video multicast. *IEEE Transactions on Broadcasting*, 48(3), September 2002.

[25] Jonathan Ledlie, Paul Gardner, and Margo Seltzer. Network coordinates in the wild. In *USENIX Second Symposium on Networked Systems Design & Implementation*, April 2007.

[26] Arnaud Legout, Guillaume Urvoy-Keller, and Pietro Michiardi. Understanding BitTorrent: An experimental perspective. Technical report, I.N.R.I.A., November 2005.

[27] Jiangchuan Liu, Kin-Man Cheung, Bo Li, and Ya-Qin Zhang. On the optimal rate allocation for layered video multicast. In *Tenth International Conference on Computer Communications and Networks*, 2001.

[28] Nazanin Magharei and Reza Rejaie. Adaptive receiver-driven streaming from multiple senders. *Multimedia Systems*, 11(6):550–567, June 2006.

[29] S. Naicken, B. Livingston, A. Basu, S. Rodhetbhai, I. Wakeman, and D. Chalmers. The state of peer-to-peer simulators and simulations. *ACM SIGCOMM Computer Communication Review*, 37(2), apr 2007.

[30] Venkata N. Padmanabhan, Helen J. Wang, Philip A. Chou, and Kunwadee Sripanidkulchai. Distributing streaming media content using cooperative networking. Technical report, Microsoft Research, apr 2002.

[31] Vern Paxson. End-to-end Internet packet dynamics. *IEEE/ACM Transactions on Networking*, 7(3), June 1999.

[32] Fernando Pereira and Touradj Ebrahimi, editors. *The MPEG-4 Book*. Prentice Hall PTR, Upper Saddle River, New Jersey, 2002.

[33] Reza Rejaie, Mark Handley, and Deborah Estrin. RAP: An end-to-end rate-based congestion control mechanism for realtime streams in the Internet. In *IEEE INFOCOM'99*, March 1999.

[34] Reza Rejaie and Antonio Ortega. Pals: Peer-to-peer adaptive layered streaming. In *International Workshop on Network and Operationg Systems Support for Digital Audio and Video*, June 2003.

[35] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM Symposium on Principles of Distributed Computing (PODC 2002)*, nov 2001.

[36] Eric Setton, Jeonghun Noh, and Bernd Girod. Rate-disortion optimized video peer-to-peer multicast streaming. In *ACM workshop on Advances in peer-to-peer multimedia streaming (P2PMMS'05)*, nov 2005.

[37] Amin Shokrollahi. Raptor codes. *IEEE Transactions on Information Theory*, 52(6):2551–2567, June 2006.

[38] Steven S. Skiena. *The Algorithm Design Manual*. Springer-Verlag, 1998.

[39] Kunwadee Sripanidkulchai, Bruce Maggs, and Hui Zhang. An analysis of live streaming workloads on the Internet. In *2004 Internet Measurement Conference (IMC'04)*, October 2004.

[40] Nevena Vratonjic, Priya Gupta, Nikola Knezevic, Dejan Kostic, and Antony Rowstron. Enabling DVD-like features in p2p video-on-demand systems. In *ACM SIGCOMM 2007 Workshop on Peer-to-Peer Streaming and IP-TV*, August 2007.

[41] Gang Wu and Tzi-Cher Chiue. How efficient is BitTorrent? In *13th Annual Multimedia Computing and Networking (MMCN'06)*, January 2006.

[42] Xinyan Zhang, Jiangchuan Liu, and Bo Li. On large scale peer-to-peer video streaming: experiments and empirical studies. In *IEEE 7th Workshop on Multimedia Signal Processing*, pages 1–4, 2005.

[43] Xinyan Zhang, Jiangchuan Liu, Bo Li, and Tak-Shing Peter Yum. Coolstreaming/DONet: A data-driven overlay network for peer-to-peer live media streaming. In *IEEE INFOCOM'05*, 2005.

APPENDIX A

THE MAXIMUM FLOW PROBLEM AND THE FORD-FULKERSON METHOD

The ideal segment piece manager is inspired by the Ford-Fulkerson method, which solves the unweighted maximum flow problem by finding augmenting paths. See for instance [11] for information on both the method and the problem.

The unweighted maximum flow problem Let $G(V, E)$ be a graph consisting of vertices and directed edges, and let each edge e have a real number that is its capacity, $c(e)$. Select one vertex S to be the source, and one vertex T to be the sink. Now, the maximum flow is any weighting of the edges of the graph that satisfy the two constraints:

1. For any edge e, the weight of e, $w(e)$, is larger or equal to zero and less than or equal to its capacity.

2. For any vertex v except the sink and the source, the sum of the weight of the edges leaving v exactly equals the sum of the weight of the edges entering v.

The unweighted maximum flow problem is to find such a maximum flow given a graph G

The Ford-Fulkerson method In addition to the capacities c, and the final weights w; the algorithm uses the flow f as temporal variable that is equal to w when the algorithm halts, and a residual flow graph r to make it easier to find the augmenting paths.

The initialization step is to initialize the residual graph to be the same as the capacities, that is the original graph. Then for all pairs of vertices u and v: if the residual graph have an edge between vertices u and v, then an opposite edge is created if not already existing, with capacity 0.

The second step is to find an augmenting path. An augmenting path is any path from the source to the sink in the residual graph. If such a path is not found, then a maximum flow has been found. Otherwise, the augmenting graph is added[1] to the flow graph f and subtracted[2] from the residual graph. Then the second step is run again.

The running time of the Ford-Fulkerson method depend upon the path-finding algorithms it uses. If the path-finding algorithms always finds a shortest path in the residual graph, the complexity is independent of the capacities of the graph.

[1]For each edge e in the augmenting path: If the opposite edge of the graph have a flow greater than zero, the flow of the opposite edge is reduced, otherwise the flow of the corresponding edge is increased.

[2]For each edge e in the augmenting path: If the corresponding edge of the graph have a capacity greater than zero, the capacity of the corresponding edge is reduced, otherwise the capacity of the opposite edge is increased.

APPENDIX B

HACKING TRACEROUTE INTO NS-2

Unfortunately, ns-2 does not reply with an ICMP packet when a packet timed out in the simulated network, nor does it have support for it. Since some of the simulations done rely on traceroute, we were required to make support for it by ourselvs.

common/ip.h We inserted the following fields into the ip-header: `is_tracing`, `is_reply` and `addr_of_router`. Normally, `is_tracing` is set to false. When an agent wants to initiate a traceroute, it sets the `is_tracing` field to true and the ttl field to an appropriate value. WE also added the `time_sent` and `trace_uid` in order to make life easier afterwards.

common/ttl.cc The part of ns-2 that is handling timeout of ip-packets is inside the common/ttl.cc file. We changed it so that if the packet's `is_tracing` field is true, the TTLChecker will not drop the packet, but instead unset `is_tracing`, set `is_reply`, set ttl to 255, and change source and destination addresses of the packet. If `is_tracing` is not set it will drop the packet when timed out.

In order to find the address of the node/router asossiated with the far end of the link, we made the field `destination-id` in TTLChecer. When a packet times out, it inserts the value of that field into the `addr_of_router` field of the packet.

tcl/lib/ns-link.tcl In order to correctly set the `destination-id` field of TTLChecker, we added a line in `SimpleLink instproc init`. The line must go below the creation of `ttl_`. The added line was `$ttl_ destination-id [$dst id]`, which sets the destination-id field of TTLChecker to the id of the destination node of the link.

APPENDIX C

SOURCE CODE FOR SIMULATED ANNEALING METHOD

Header file

```
#ifndef CENTRALCORE_H
#define CENTRALCORE_H
#include <vector>
#include <map>
#include <fstream>
#include <cmath>
#include <limits>
#include <set>
#include "assert2.h"

#include "app.h" //to get access to tcl

using namespace std;

struct FromTo{
  int f,t;

  bool operator<(const FromTo & rhs) const{
    if ( f == rhs.f ) return t < rhs.t;
    return f < rhs.f;
  }

  FromTo(int f, int t) : f(f), t(t) {}
};

class CentralCore{

  map<FromTo, int> to_linkno; //translate [from][to] into a link number
  int stream_c;
```

```
  int server_c;
  int peer_c;
  int router_c;
  int node_c; //= peer_c + server_c
  int link_c;
    double * capacity;      //capacity[linkno] between devices on the net
  double * latency;
  double * used; //[linkno]
  int  *    streamID; // -1:'any stream', 0:'no stream', positive numbers: stream
  double ** progress; //progress[stream][node]
  double progress_delta;
  double ** send_rate; //[from_peer][to_peer]
  double * demand; //demand of download rate to each node
  double * offer;  //currently offered download rate
  vector<int> ** paths; //path[sender][receiver], enumerates links

  vector< pair<int,int> > peer_pairs; //enumerates possible peer selections

  double k;   //boltzmann constant. Selected to equal rate of streams
  double energy;
  double T;   //temperature
  double Co;  //cost of overuse of the capacity of a link
  double Cu;  //cost of too low rate down to a peer

  void apply_change(int from, int to, double delta_rate);
  double get_dEnergy(int from, int to, double delta_rate);
  void remove_path(int from, int to);
  void enumerate_peer_pairs();
  pair<int,int> select_peer_pair();
  double select_delta_rate();
  bool step();
  void read_topology_file();
  void update(double T0, int maxN); //called on time step, and on peer's stream ch
public:
  void update();
  void set_progress(int stream, int node, double progr){progress[stream][node] = p
  double * get_send_rates(int sender){return send_rate[sender];}
  int get_node_c() {return node_c;}
  void register_path(int sender, int receiver, vector<int> path);
  void alter_demand(int receiver, int new_stream, double new_rate);
  CentralCore();
  ~CentralCore();

};


#endif
```

Body

```
#include "central-core.h"


void CentralCore::update(double T0, int maxN){
  enumerate_peer_pairs();
  T = T0;
  int change_c;
  int steps_per_round = 3*node_c;
  for (int N = 0; N < maxN; N++){
    change_c = 0;
    for (int i = 0; i < steps_per_round; i++)
      if (step())
        change_c++;

    T *= 0.97;
    if (energy <= 0.0)
      break;
  }
  //printf("Energy after annealing: %f\n", energy / k);
}

void CentralCore::update(){
  update(1.0, 450);
}


void CentralCore::alter_demand(int from, int sID, double rate){
  if (rate != demand[from]){
    double e_now = Cu * max(0.0, demand[from] - offer[from]);
    double e_after = Cu * max(0.0, rate - offer[from]);
    energy += (e_after - e_now);
  }
  streamID[from] = sID;
  demand[from] = rate;
  update();
}

//remove the reserved bandwidth between from and to
void CentralCore::remove_path(int from, int to){
  double delta_rate = -send_rate[from][to];
  if ( abs(delta_rate) < 0.00001) return;
  double dEnergy = get_dEnergy(from, to, delta_rate);
  apply_change(from, to, delta_rate);
  energy += dEnergy;
}

//make a list of all valid <sender,receiver> pairs, invalid peer pairs
```

```
//will have their reserved bandwidth removed
void CentralCore::enumerate_peer_pairs(){
  peer_pairs.clear();
  for (int s = 0; s < node_c; s++){
    for (int r = 0; r < node_c; r++){
      int str = streamID[r]; //id of request stream
      if (
        (s == r) || //need diffent recver and sender
        (str == -1) || //server should not be receiver
        (str == 0 ) || //inactive receiver
        (streamID[s] == 0 ) || //inactive sender
        ( (streamID[s] != -1) && (progress[str][s] + progress_delta <
                  progress[str][r] ) ) //progress is less at sender
         ){
        remove_path(s,r);
         }
      else{
        peer_pairs.push_back( pair<int,int>(s,r) );
      }
    }
  }
}

//select a (sender,receiver) pair such that receivers have same streamID's,
//and such that sender.progress + progress_delta > receiver.progress
pair<int,int> CentralCore::select_peer_pair(){
  int idx = rand() % peer_pairs.size();
  return peer_pairs[idx];
}

//randomly select rate to be between (-k, k). This is equal to
//(-mediarate, mediarate)
double CentralCore::select_delta_rate(){
  return 2.0*rand()/(float)RAND_MAX * k - k;
}

double CentralCore::get_dEnergy(int from, int to, double delta_rate){
  vector<int> & path = paths[from][to];
  double dEnergy = 0;
  //sum energy change due to less/more overuse of links
  for (unsigned int i = 0; i < path.size(); i++){
    int linkno = path[i];
    double e_now = max(0.0, Co * ( used[linkno] - capacity[linkno] ));
    double e_after = max(0.0, Co * ( used[linkno] + delta_rate -
                                     capacity[linkno]));
    dEnergy += (e_after - e_now);
  }
  //add energy change due to low receive rate
  {
```

```
    double e_now = max(0.0, Cu * (demand[to] - offer[to]));
    double e_after = max(0.0, Cu * (demand[to] - (offer[to]+delta_rate)));
    dEnergy += (e_after - e_now);
  }
  return dEnergy;
}

void CentralCore::apply_change(int from, int to, double delta_rate ){
  vector<int> & path = paths[from][to];
  offer[to] += delta_rate;
  send_rate[from][to] += delta_rate;
  for (unsigned int i = 0; i < path.size(); i++){
    int linkno = path[i];
    used[linkno] += delta_rate;
  }
}

bool CentralCore::step(){
  pair<int,int> peer_pair;
  int from;
  int to;
  double delta_rate = 0.0;

  //loop until a non-zero delta rate is found,
  //try up to a maximum number of times (to avoid looping forever)
  for (int j = 0; abs(delta_rate)<0.001 && j < 100; j++){

    peer_pair = select_peer_pair();
    from = peer_pair.first;
    to = peer_pair.second;

    //select a delta rate and make sure offered rate stay in
    //interval [0, demand]
    delta_rate = select_delta_rate();
    delta_rate = min( delta_rate, demand[to] - offer[to] );
    delta_rate = max( delta_rate, -offer[to]);
    //noone can send a negative amount
    delta_rate = max( delta_rate, -send_rate[from][to] );
  }


  double dEnergy = get_dEnergy(from, to, delta_rate);

  bool change = false;
  // if dEnergy <= 0, then accept
  // else accept with probability exp(dEnergy/(kT))
  if ((dEnergy <= 0) || (exp(-dEnergy/(k*T)) > rand()/(float)RAND_MAX)){
    apply_change(from, to, delta_rate);
    change = true;
```

```
    energy += dEnergy; //needed only for statistical analysis
  }
  return change;
}

CentralCore::CentralCore(){
  //constants which should be tuned
  stream_c = 5;
  k = 187500.0; //rate of 1500kbps - in bytes
  progress_delta = 1;
  Co = 1.0;
  Cu = 1.0;

  read_topology_file(); //this also initialize some structures

  //init the other strucures, now we know how large they should be
  used = new double[link_c];
  paths = new vector<int>*[link_c];
  for (int i = 0; i < link_c; i++){
    used[i] = 0.0;
    paths[i] = new vector<int>[link_c];
  }
  send_rate = new double* [node_c];
  demand = new double[node_c];
  offer = new double[node_c];
  streamID = new int[node_c];
  progress = new double* [stream_c+1];
  for (int i = 0; i < stream_c+1; i++){
    progress[i] = new double[node_c];
    for (int j = 0; j < node_c; j++)
      progress[i][j] = 0.0;
  }
  for (int j = 0; j < node_c; j++)
    progress[0][j] = numeric_limits<double>::max();
  for (int i = 0; i < node_c; i++){
    send_rate[i] = new double[node_c];
    for (int j = 0; j < node_c; j++)
      send_rate[i][j] = 0.0;
    demand[i] = 0.0;
    offer[i] = 0.0;
    streamID[i] = 0; //meaning 'no stream'
  }
  energy = 0.0;


  for (int i = 0; i < server_c; i++){
    streamID[i] = -1; //meaning 'any stream'
  }
}
```

```
CentralCore::~CentralCore(){
  delete [] capacity;
  delete [] latency;
  delete [] used;
  for (int i = 0; i < link_c; i++)
    delete [] paths[i];
  delete [] paths;
  for (int i = 0; i < node_c; i++)
    delete [] send_rate[i];
  delete [] send_rate;
}

void CentralCore::register_path(int sender, int receiver, vector<int> path){
  //the path received is in opposite order:
  vector<int> line_path; //path with linenumbers instead
  assert( path[0] == receiver );
  assert( path[path.size()-1] == sender );
  for (unsigned int i = 1; i < path.size(); i++){
    int from = path[i];
    int to = path[i-1];
    line_path.push_back( to_linkno[FromTo(from,to)] );
  }

  //more than one zero means to_linkno is incomplete
  int zerocount = 0;
  for (unsigned int i = 0; i < line_path.size(); i++){
    if (line_path[i] == 0) zerocount++;
    assert2(line_path[i] >=0);
    assert2(line_path[i] < link_c);
  }
  assert2(zerocount < 2);

  paths[sender][receiver] = line_path;
}

void CentralCore::read_topology_file(){
  Tcl& tcl = Tcl::instance();
  tcl.evalc("set opt(edge-file)");
  ifstream efile( tcl.result() );
  assert2(efile.good());
  tcl.evalc("set opt(vertex-file)");
  ifstream vfile( tcl.result() );
  assert2(vfile.good());
  vfile >> server_c >> peer_c >> router_c;
  node_c = peer_c + server_c;
  vfile.close();
  efile >> link_c;
  capacity = new double[link_c];
```

```
  latency = new double[link_c];
  for (int i = 0; i < link_c; i++){
    int from, to;
    double bw, lat;
    efile >> from >> to >> bw >> lat;
    to_linkno[FromTo(from,to)] = i;
    bw *= (1000.0/8.0); //transform from kbps into Bps
    capacity[i] = bw;
    latency[i] = lat;
  }
  efile.close();
}
```