Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing Group

Master Thesis

Peer-to-Peer Live Streaming

Remo Meier
remmeier@student.ethz.ch

Dept. of Computer Science
Swiss Federal Institute of Technology (ETH) Zurich

Summer, 2006

Prof. Dr. Roger Wattenhofer
Distributed Computing Group

Advisors: Thomas Locher, Stefan Schmid

Abstract

Live streaming is used in today's Internet to broadcast TV channels and radio stations, usually by deploying streaming servers. Peer-to-peer applications have become popular in recent years to overcome the limitations of centralized servers. This thesis proposes *StreamTorrent*, a peer-to-peer-based live streaming protocol. In contrast to most protocols, *StreamTorrent* provides robustness, efficiency, and scalability by combining different strategies. *StreamTorrent* has the efficiency of typical tree-based and the robustness of more random protocols. An overlay with small diameter and locality-awareness reduces latency and network load. And because peer-to-peer computing is about collaboration among peers, incentives are given to peers to share their resources to ensure good playback quality.

The *StreamTorrent* protocol has been implemented and can be used both to perform simulations and to stream in the real world. The simulations allow to evaluate the protocol and to perform automated tests for a large number of peers. The *StreamTorrent Player* is a real world application supporting live audio and video streaming.

Contents

Chapter 1

Introduction

Several peer-to-peer-based applications have emerged in recent years to overcome the limitations and costs of centralized servers. Especially file-sharing has become very popular and already accounts for most traffic on the Internet. Its main purpose is the exchange of audio and video files. Unlike file sharing, the adoption of peer-to-peer technologies for audio and video streaming is so far progressing very slowly, both for on-demand and live streaming. On-demand streaming would spare users from waiting until downloads have completed, and live streaming could be used, for example, to broadcast TV channels, radio stations, and sporting events. Unfortunately, streaming is inherently more complex than file sharing. Besides providing robustness and scalability, applications have to deliver data blocks in time to ensure a good playback quality.

In this master thesis peer-to-peer live streaming is studied. Native IP multicast is the typical and most efficient solution within local networks, but it lacks a large-scale deployment over the boundaries of local networks to the entire Internet. This has triggered studies of application-layer peer-to-peer-based protocols like the one presented in this thesis. Many different aspects have to be taken into account when designing a live streaming protocol, such as malicious and selfish peers, network delays and locality, packet loss, network diameter, scalability, efficiency, overhead, and churn. There are already numerous proposals such as [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]. Many of these proposals focus on giving a solution to some aspects, frequently leading to problems with other aspects. For example, tree-like overlays are efficient but fragile, while more random overlays are robust but inefficient. Or some of the proposed overlays are better suited than others to provide incentives for peers to share their upload bandwidth. Aspects such as packet loss, heterogeneous bandwidths, or network delays have frequently been omitted, but clearly have a profound impact in the real world. What still seems to be missing are more detailed studies about the entire problem. By introducing *StreamTorrent*, we aim at combining and improving some of the proposed solutions to overcome their respective limitations and thereby providing a more robust and efficient protocol.

StreamTorrent uses a combination of *pull-based* and *push-based* strategies to provide, unlike most other protocols, both efficiency and robustness. Peers maintain a set of neighbours to exchange notifications about data blocks and to request missing data blocks. A XOR-based metric topology, frequently adopted by distributed hash tables, is used to guarantee a logarithmic network diameter, to add locality-awareness, and to increase efficiency by pushing data blocks (sending data blocks without a request). Upload and download slots are allocated for neighbours according to the used bandwidth and observed packet loss. To give peers incentives for sharing upload bandwidth with neighbours, neighbour ranking, tit-for-tat, recommendations, and certificates are adopted.

The *StreamTorrent* protocol has been implemented and can be used both to perform simulations and to stream in the real world. Various simulations are used in this thesis to evaluate *StreamTorrent* with thousands of peers to verify its efficiency, scalability and robustness. The *StreamTorrent Player*, a real world application, supports live audio and video streaming over the Internet and is similar in its use to popular *BitTorrent* clients. The *StreamTorrent Player* is based on a plugin architecture to allow future extensions. The user interface provides an integrated browser to find streams on web sites, an integrated media player to playback streams, and many smaller features such as favorites, a history, time-shifted viewing, and recording support.

Chapter 2 outlines work related to this thesis. Chapter 3 presents the *StreamTorrent* protocol in detail and Chapter 4 its implementation. The simulator and an evaluation of the results are given in Chapters 5 and 6. The *StreamTorrent Player* is presented in Chapter 7. And finally, Chapters 8 and 9 give a conclusion and outline future work.

Chapter 2

Related Work

2.1 Streaming

Several protocols have already been proposed for peer-to-peer streaming [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]. Most of these protocols fall into two classes:

- Protocols based on a tree-like overlay. Trees are used to avoid duplicates when packets are sent without being requested, known as *pushing*. In its simplest form, the source is the root of a single tree and pushes new packets to its children, the children in turn push packets to their children, etc. Unfortunately, the bandwidth available to peers is limited by all their parents. And if peers leave the overlay, trees have to be repaired or the peers' children receive no data. Consequently, trees are fragile under churn, i.e. when peers frequently join and leave, or when some peers have not enough upload bandwidth.

- Protocols with an overlay of more random nature. Peers maintain a set of neighbours to periodically exchange notifications about available packets and to request missing packets. Protocols of this type are usually far more resilient against churn, but incur a higher overhead and longer delays due to the notify-request-response cycles. To some degree the protocols are similar to *BitTorrent* [11] and other file sharing tools.

Overcast [2] and *FreeCast* [9] are typical protocols using a single tree. Both are simple and work well within a stable network, but are fragile under churn. While inner nodes have to forward the stream to all their children, leaf nodes do not have to share anything. This is a major disadvantage since the upload bandwidth is often limited, e.g. ADSL broadband connections, and many peers might attempt to join as leaf. This problem is partly overcome by protocols like *Zebra* [10]. *Zebra* splits streams into two stripes. Each stripe is distributed among the peers using a separate tree. A peer has to be an inner node in one tree and a leaf in the other. If every inner node has two children, then every peer has to contribute the same upload bandwidth. Besides the disadvantages of any tree-like protocol, parts of *Zebra* use centralized sub protocols that limit its use to merely a few dozen, maybe hundred peers. The concept of multiple trees is further generalized by *SplitStream*, using k stripes and *multiple description coding* [12]. *Multiple description coding* allows to split a stream into stripes so that any subset of stripes allows to reconstruct the original stream (with reduced quality if not all stripes are available). Unfortunately, *multiple description coding* is still an active research effort and not used in practice. A similar, more centralized approach is taken by *CoopNet* [13]. The concept has been further improved by protocols such as *ChunkySpread* [5], for example, by introducing a "weak" tit-for-tat model, adding locality-awareness, and providing a more efficient protocol to build and repair trees.

Bullet [4] builds a mesh on top of an arbitrary tree overlay to increase robustness. Additional links are added to reduce the dependency of peers to their parents. The stream is split into disjoint blocks and distributed within the tree. Only as much packets are sent to children as bandwidth is available. Missing packets are then localized and downloaded using the mesh. Its robustness depends on the underlying tree, but it is improved by the additional links provided by the mesh. Naturally, the mesh imposes additional overhead which is substantial in total:

- 30 Kbps per node to maintain the mesh and locate missing blocks.
- 5% overhead if *erasure codes* are used ([14, 15, 16]).
- Less than 10% duplicate packets under churn.
- The overhead of the underlying tree.

Chainsaw [1], *CoolStreaming/DONet* [17], and *GridMedia* [18] belong to the second class of protocols, i.e. they are more random in nature. Simulations of *Chainsaw* [1] have shown that peers recover from 50% of all peers crashing simultaneously with minor packet loss. While such a robustness might seem unnecessary, it is what happens when, for example, TV channels start showing commercials or a movie ends. None of the tree-based overlays recover from such events without severe packet loss. The overhead of such random protocols increases with the number of neighbours, the notification frequency, and the needed buffer size. While in *Chainsaw* peers have between 20 and 40 neighbours, it is between merely two and six for *CoolStreaming/DONet.* However, the parameters also influence each other. More neighbours allows a lower notification frequency. Smaller buffers result in smaller notifications. The overhead of *CoolStreaming/DONet* is less than 2% in a stable network with 6 neighbours. The drawback of few neighbours are less robustness and larger buffer sizes (5 seconds with *Chainsaw* compared to 60 seconds with *CollStreaming/DONet*). *GridMedia* additionally pushes data packets to speed up packet distribution.

DagStream [7] and *Dagster* [6] use direct acyclic graphs (or *DAG*). *Dagster* maintains the current overlay topology at the source and does therefore not scale. *DagStream* is a pull-based, decentralized, and locality-aware protocol. It ensures connectivity by using properties of the underlying DAG. Similar to *CoolStreaming/DONet* peers maintain a small set of neighbours.

In contrast to these attempts, we strive for combining the advantages of trees with the robustness of random structures and having an overlay that provides connectivity with small diameter, locality-awareness, and incentive-compatibility.

2.2 Distributed Hash Tables

In contrast to peer-to-peer streaming, connectivity, neighbour selection, and expected diameter have been well-studied for distributed hash tables. Many protocols guarantee a logarithmic diameter ([19, 20, 21, 22]). *Pastry* [21], for instance, assigns a unique identifier to each peer and uses prefix routing. Each peer maintains a logarithmic number of neighbours. In its simplest form, a peer chooses a neighbour whose identifier is the same at the i most significant bits and differs in the subsequent bit $i + 1$, for $i = \{0, 1, 2, 3, ...\}$. Objects are also addressed by identifiers and stored on the peers with the most similar identifier. To speed up prefix routing, identifiers can be partitioned into disjoint blocks of size b. For each block i, peers choose $2^b - 1$ neighbours that have the blocks $1, \ldots, i - 1$ in common and differ in the current block in one of the $2^b - 1$ ways. This way, prefix routing fixes b bits at each hop.

Protocols like *Pastry* [21], *eQuus* [22], and *Vivaldi* [23] additionally provide locality awareness. A simple heuristic for neighbour selection, also adopted by *Pastry,* adds locality-awareness to *Kademlia.* For short common prefixes, many peers can be used as neighbours (about $n2^{-i}$ where n denotes the number of peers). By choosing the nearest one known and by asking nearest neighbours for more neighbours, peers gradually improve locality while maintaining connectivity with a logarithmic diameter. This concept is extended in *Vivaldi* by introducing virtual coordinates. Each peer obtains a virtual coordinate that is constantly updated based on other peers' coordinates. This has the advantage that peers can guess in advance which peers might be nearer than others without actually having to contact these peers to measure round trip times.

Some of these concepts will be applied in *StreamTorrent* to also gain these properties.

2.3 Incentives

One of the hardest and most essential challenges in peer-to-peer computing is to give peers incentives to share their resources since peer-to-peer is based on collaboration. For example, Huberman and Adar [24] have shown that in *Gnutella* nearly 98% of the responses were returned by 25% of the sharing hosts and almost 50% of the responses where returned by merely 1% of the sharing hosts. Clearly, all hosts would greatly benefit if there is a mechanism preventing hosts from downloading without uploading. Updating the client software to throttle the download bandwidth for free-riders is not a viable solution as software can be modified by a third party. Most prominent example is *Kazaa* [25] and its modified version *KazaaLite*.

What makes the problem hard within large overlays is that peers only interact with a small fraction of the peers. Maintaining private histories of good and bad peers does not scale [26], especially considering the zero-cost nature of online identities in most systems, exploited by *white-washing* and *Sybil* [27] attacks. Better systems leverage the opinion of other peers, for example, by sharing a common history. Such solutions usually face two challenges. First, storage and communication is needed to store, update, and query ranking information. Implementations either use centralized servers or incur a high communication overhead by building a distributed alternative. Second, such systems are vulnerable to collusions of malicious peers. Malicious peers might improve their ranking by recommending each other. Collusions can be overcome by adopting *subjective* shared histories [28, 29] where peers "favor" other peers with a "similar opinion". A related mechanism is the use of micro-payments, e.g. in Karma [30].

A popular and simple alternative are *tit-for-tat* mechanisms [31]. Simply speaking, a peer A only shares packets with peer B if peer B shares its packets with peer A. Clearly, one of the two parties has to start sending packets to prevent deadlocks and starvation and thereby risks supporting a free-rider. This problem can be partly overcome by only sharing certain packets for free. Free packets are determined based on the receivers identity. Simple approaches choose two hash functions, one for IP addresses and one for sequence numbers, and a packet is free if the hash values match. The set of free packet is known as *allowed fast set* [32] in *BitTorrent*. Naturally, tit-for-tat is only feasible for protocols with bidirectional packet exchange, immediately excluding DAG-based and most tree-based protocols. Attempts to overcome this limitation for trees usually split streams in stripes and periodically rebuild the trees to gain more symmetry, i.e. that the parent-child roles are periodically reversed and parents can then punish their children if they did not share in the past (e.g. [33]). Unfortunately, like all private history approaches, such attempts do not scale.

Chapter 3

Protocol Design

At its core, the *StreamTorrent* protocol is a pull-based protocol similar to *Chainsaw* [1] and file sharing protocols like *BitTorrent* [11]. A logarithmic number of neighbours are maintained to exchange notifications and data packets. The pull-based nature of *StreamTorrent* has the advantage of high robustness against churn. However, the latency is typically bad for pull-only protocols. To overcome this problem, neighbours are chosen based on the topology proposed by *Pastry* [21]. This allows the source to push new packets to speed up packet distribution. The chosen topology thereby ensures a negligible number of duplicates. Moreover, the flexibility in choosing neighbours is used to add locality-awareness while maintaining a logarithmic network diameter. In contrast to trees and directed acyclic graphs, neighbourhood is symmetric. This results in a more incentive-compatible overlay. Simple techniques like tit-for-tat and its *allowed fast set* extension are adopted to give peers incentives to share upload bandwidth. Together with the fact that buffers are small and there is not much time to obtain packets, peers are forced to share packets in order to sustain a continuous delivery with neglibile packet loss. The subsequent sections give an overview of the protocol.

3.1 Network Model

The protocol assumes that peers are connected among each other by unreliable, unordered channels and that each peer has unique identifier, a maximum upload bandwidth, and a maximum download bandwidth.[1] Real world implementations consequently use UDP instead of TCP for several reasons:

1. Packets have to be delivered in time. Chances are increased by sending retransmissions to different peers.
2. Peers usually have between 20 and 40 neighbours and exchange only few packets with each. Acknowledgements would significantly increase the number of packets. Some packets do not need to be acknowledged anyway.
3. More sophisticated protocols than the ones adopted by TCP are known to avoid congestion and packet loss and maintain a high bandwidth (e.g. [34]).

3.2 Neighbour Selection

Chainsaw [1], CoolStreaming/DONet [17], and GridMedia [18] adopt a random neighbour selection, guaranteeing neither connectivity nor locality-awareness. For a high node

[1] Assuming connectivity among each other is not entirely correct since there might NAT devices and firewalls. Techniques used to by-pass such devices and consequences for the protocol are outlined in Chapter 7.3.3.

degree, one might can assume good connectivity and a small diameter. However, extra precautions have to be taken to maintain these properties if the protocols are extended by, for example, locality-awareness and incentive mechanisms. For instance, choosing the nearest known peers as neighbours could result in an overlay more similar to a linked list, causing long delays and buffer underflows. Moreover, the number of neighbours has to be adapted according to the number of peers to ensure scalability.

A better overlay might use a combination of both local and distant neighbours, similar to the *small world phenomena* [35]. *StreamTorrent* uses the XOR-based metric topology proposed by *Kademlia*. It guarantees a logarithmic diameter with a logarithmic number of neighbours. As stated in Chapter 2.2, more neighbour candidates are available the shorter the common identifier prefix is. *StreamTorrent* selects candidates based on the observed round trip times and provided bandwidth. Peers attempt to minimize round trip times if enough bandwidth is available and the buffers are full, and attempt to maximize the received bandwidth otherwise. Moreover, the next section describes how the chosen topology can be used to speed up packet distribution.

3.3 Pushing Packets

While pull-based protocols are robust, they incur longer delays and a higher overhead compared to tree-based solutions. Waiting for a notification of a new packet, then sending a request, and waiting for an answer (or timeout) takes time that increases with each hop. Frequently sending notifications is essential for pull-based protocols to reduce delays. However, notifying neighbours causes communication overhead, limiting the notification frequency.

In an attempt to reduce both the delays and the associated notification overhead, it is essential to observe that pull-based protocols perform well once packets are already distributed among a fraction of the peers. Most delay is incurred in the interval between the source has made a new packet available and it is distributed to a few peers. One could devise more sophisticated notification strategies, taking the age of packets into account, leading, unfortunately, only to minor improvements. This suggest that a new approach is needed to distribute new packets.

A solution to this problem is to push new packets, i.e. sending packets without having received corresponding requests. But a mechanism is needed to avoid duplicates. That is where the chosen topology perfectly fits in. For each packet, a tree is dynamically created among a subset of the peers to push new packets. The tree might differ for each packet to balance the load and favor good neighbours. The tree does not reach all peers, but this is not needed anyway.

Let b again denote the number of bits that the prefix routing algorithm of *Kademlia* fixes at each hop (see Chapter 2.2), *StreamTorrent* typically uses $b = 2$. Given a new packet, the source selects a peer among its neighbours to be the root of the packet's distribution tree and sends the packet to this peer. The peer selects 2^b peers among its neighbours based on their identifiers to forward the packet. A neighbour is chosen for each of the 2^b bit strings of length b so that the neighbour's identifier starts with this particular prefix. These neighbours then proceed recursively. They select neighbours for each of the 2^b bit strings of length $2b$ that have the first b bits in common with themselves. This is repeated until no further neighbours are found to forward the packet.

Using this mechanism, there are still two ways how duplicates can occur:

- A notification for a pushed packet might arrive sooner than the packet itself and triggers a request. This happens, for example, if a peer near the root knows a peer further down the tree.

- It is possible that a child sends the packet back to one of the parent's parents. At each hop, $2^b - 1$ of the selected neighbours are known to have identifiers that differ from all the parent identifiers. This is not the case for the remaining one as one can easily verify.

The first type of duplicates is negligible except for a small number of peers. The impact of the later type is more noticeable, but there are techniques to overcome the problem. The simplest solution is to add the identifiers of the parents to the pushed packet and to exclude them in the neighbour selection. Unfortunately, the strategy reveals a path to the source, rendering it more vulnerable to denial of service attacks and malicious peers. Another solution is to only forward packets to $2^b - 1$ neighbours, avoiding the problem at cost of efficiency. This approach is well suited for up to few hundred peers. With a growing number of peers, the second type also becomes negligible. In between, the strategies can be combined, i.e. sending to all 2^b neighbours near the source and to $2^b - 1$ neighbours further down the tree.

Push packets are acknowledged near the root to get better resilience against packet loss and churn. If a timeout occurs, the packet is retransmitted to another neighbour of the same prefix. By acknowledging, for example, up to a depth half the estimated tree height, the number of acknowlegments is bounded by the square root of the number of peers within the tree. The resulting overhead is negligible.

3.4 Notification Strategy

Having a push mechanism to distribute new packets allows to reduce notifications. *StreamTorrent* sends a list of all available packets that have not been delivered so far. If the system is working properly, buffers are almost full and notifications have a low entropy. Therefore, a simple coding scheme is adopted, i.e. a notification contains:

- The sequence number of the next packet to be delivered.
- The number of packets that can be delivered in-order without packet loss.
- A bitmap of the remaining buffer showing whether packets are available or not.

In most cases the chosen coding is more efficient than sending a list of newly received packets because sequence numbers are 16-bit values. This way, the protocol is alsomore resilient against packet loss. Packets that have been delivered are dropped and are not longer available to the neighbours. This is no problem since peers are roughly synchronized among each other and deliver the same packet more or less at the same time (see Chapter 3.7).

Notifications are not sent to all neighbours at once, but alternately to disjoint subsets to also spread subsequent requests equally over time.[2] Peers start sending more notifications if they see that they do not share enough packets with neighbours (given that enough bandwidth is available). This is helpful when, for example, peers join the stream or a large number of peers crash. To avoid packet header overhead, notifications are bundled with other packets.

3.5 Pull-based Request and Response Strategies

Peers allocate download slots for neighbours to manage their own upload bandwidth, especially to not exceed the maximum bandwidth. The number of slots is periodically adapted according to the used and total upload speed, the observed packet loss, and the number of neighbours. Slot information is send to neighbours together with the notifications.

A round-robin scheme is used to issue requests. One neighbour after another is checked whether a new packet and a free slot are available. If several packets are

[2] Round trip times are usually significantly smaller than notification intervals

available from a neighbour, one is selected at random and depending on in its age. The selection works as follows. Let R_N denote the set of missing packets that are available from neighbour N. A sequence number s is selected uniformly at random within the interval between the oldest and newest sequence number of R_N. Sequence number s may or may no be in the set R_N. If it is the case, s is selected itself, otherwise the next older sequence number in R_N. A request is then issued for the selected sequence number and the next neighbour is checked for packets.

Typically, the same amount of slots is allocated for each neighbour.[3] Both the round-robin scheme and this uniform slot allocation are used to boost symmetry. The goal of *StreamTorrent* is that peers uniformly exchange data with all neighbours to speed up packet distribution and to avoid peers depending on a small set of neighbours. Naturally, the symmetry can only be sustained as long as all peers provide enough bandwidth. Otherwise, slow peers provide the bandwidth they have.

Because some slots may not be used, peers allocate more slots than available. It is therefore still possible for peers to receive more requests than bandwidth is available. In this case, peers might choose to send *overload* messages instead of the actual responses. The adopted strategy is expected to keep *overloaded* messages at a minimum to avoid more communication and delays. In contrast, selecting neighbours and packets at random results in a logarithmic overhead for some the neighbours. [4]

3.6 Incentives

For peer-to-peer streaming it is essential to give peers incentives to share enough upload bandwidth to quickly distribute packets. Incentive mechanisms influence virtually all parts of the system. The implementation is still in an early stage, but already works reasonably well in many cases. The following subsections give a brief overview.

3.6.1 Comparison to File Sharing

There are several differences between file sharing and live streaming. Live streaming uses small buffers and packets have to be delivered in time. This gives both advantages and disadvantages. Free-riding file-sharers might have longer download times, but there is no further penalty. By contrast, free-riding within a streaming applications may quickly results in packet loss and reduced playback quality. This suggests that given a reasonably good incentives mechanism, rational free-riders have no incentive to continue free-riding. On the negative side, free-riders and slow peers still consume some bandwidth. Given the small buffers, this could also lead to packet loss at good peers if the total bandwidth is scarce.

A further advantage is the notification overhead. All peers are forced to periodically notify their neighbours or they get dropped. This limits the number of neighbours a peer can maintain concurrently. At least it would be more reasonable to have less neighbours and to sent a few packets instead.

For live streaming, the source represents a single point-of-failure. If it leaves the overlay, all peers are affected. It is therefore crucial to protect the source from denial-of-service attacks and malicious peers, for example, by disguising its role as the source from the others.

[3] Note that the subsequent *incentive* section describes how cope to with free-riders and slow peers in general.

[4] The problem is related to the famous buckets and bins problem. It is well-known that by putting bins randomly into buckets, some buckets are expected to a have logarithmic overhead (in the number of buckets).

3.6.2 Hiding the Source

The source maintains a second buffer and requests packets like any other peer. By doing so, some precautions have to be taken when sending notifications and requests. When a notification is sent to a neighbour, packets where the neighbour has been selected as root of the packet's push distribution tree have to be added to the notification. Moreover, the neighbour should not receive request for those packets. Older packets that the source was unable to obtain from the neighbourhood are added from the original buffer (and maybe could be pushed a second time).

StreamTorrent does not adopt the mechanism of *Chainsaw* that allows the source to reply to requests with an arbitrary packet not sent so far. Such a strategy immediately reveals its role as the source and is unnecessary because all packets are pushed. Unfortunately, pushing packets as described in Chapter 3.3 also reveals its role to the selected roots within its neighbourhood. A simple trick avoids this problem. Packets are first forwarded a random number of hops before the distribution starts. At each hop, forwarding is stopped with constant probability. By observing pushed packets and notifications, peers might still be able to guess whether a source is located in the neighbourhood, but it becomes harder.

To further increase security, the source can be replicated. Three problems have to be considered when replicating the source:

- Replicas have to deliver the same content with the same sequence numbers.
- Packet timestamping has to be identical.
- The task of pushing packets has to be partitioned among the replicas to avoid duplicates.

3.6.3 Tit-for-Tat Response Strategy

Tit-for-tat with the *allowed fast set* extension of *BitTorrent* is adopted to limit free-riding. The IP address rather than a peer's identifier is used to determine freely available packets, ensuring that free-riders cannot obtain all packets by running multiple protocol instances on different UDP ports. For non-free packets, the *payback ratio*, specifying how many bits a peer is expecting in return for sending one byte, has to be fulfilled. The ratio is periodically adapted to the current load.

3.6.4 Neighbour Ranking

Neighbours receive a ranking, for example, to decide whether to drop neighbours or to select neighbours to forward push packets. Ranks are assigned by computing a score and sorting neighbours by score. The score of a neighbour is determined by the number of received packets. Other parameters, like the number of *overload* messages, have been added to the equation with only minor improvements. Fortunately, the equation is already working well. More information about the quality of the ranking is given in the *Evaluation* chapter. Note that up to now, round trip times are not included in the ranking, but handled separately.

3.6.5 Dropping Neighbours

There is no point in supporting peers that share only few packets or are far away if there are better peers available. To be able to drop neighbours, the system first ensures that connectivity is still fulfilled afterwards. If it is the case, peers decide based on the experienced delay whether to improve bandwidth (based on ranking) or locality. To prevent fast peers being dropped instead of much slower ones merely because the faster ones are slightly farther away, or the other way around, such alternative candidates are checked first before a neighbour is dropped.

3.6.6 Neighbour Recommendations and Certificates

While tit-for-tat strategies limit the amount of data sent to free-riding peers, another mechanism is needed to prevent bad neighbours in the first place. Determining the rank of peers before exchanging packets suggests some sort of history or recommendation mechanism. Shared histories are considered to be too expensive regarding communication cost. Private histories do not scale with the number of peers. Not having a history implies that peers first have to prove themselves upon joining a network. Otherwise a registration mechanism would be required to prevent *white-washing* and *Sybil* attacks (see Chapter 2.3 for more information). Within *StreamTorrent*, peers ask their neighbours for more neighbours.[5] Neighbours in turn respond with a subset of their own neighbourhood according to the requester's identifier and rank. While this strategy increases the probability of non-free-riders getting other non-free-riders as neighbours, it can not provide any guarantees. The main limitation is that neighbours receiving subsequent join requests do not have any information about the sender requesting a join. This manifests itself on several occasions:

- A peer may has enough neighbours and therefore rejects the join request. He has to compile list of alternative candidates without having any ranking information about the receiver.
- Entrypoints currently do not have any ranking information and return random candidates. This candidates will not know that an entrypoint provided the address.
- Free-riders could collude and exchange addresses of good peers.

A partial solution to this problem is given by adopting certificates and chains-of-trust. A certificate is issued for each recommendation. If peer P_A and peer P_B have a common neighbour P_C, P_A can issue a certificate to P_B for P_C. A certificate contains the identity of the issuer and the issuer's ranking of the recommended peer. However, certificates have to be used carefully to not cause deadlocks and starvation. New neighbours do not have any ranking and should nevertheless be able to join a stream. Free-riders typically have a bad ranking but they could pose as new neighbours. The mechanism is consequently used to reward highly ranked peers, for example by:

- Lowering the payback ratio enforced by the tit-for-tat strategy.
- Removing bad neighbours instead of rejecting joins.
- Allocating spare capacity in the neighbour buckets.

Certificates do not need to be based on public key cryptography. Recommended peers receiving join requests are the only ones that need to verify certificates. Symmetric cryptography is sufficient for this purpose. Peers in *StreamTorrent* classify their neighbours into four groups depending on their ranks. For each new neighbour, a random key is generated for each of the four groups and sent to the new neighbour. To issue a certificate, the neighbour computes the receiver's group based on his rank and hashes the corresponding random key with the receiver's identifier. The certificate is neither transferable to other peers nor extractable by the receiver to determine its ranking.

The mechanism can be extended in two ways. A trust value might be assigned to neighbours depending on how good the certificates match with the own experience. And peers rejecting other peers with good certificates might issue a new certificate for its own neighbours when providing alternative candidates, given that the peers trust the original certificate issuers, leading to chains-of-trust.

[5] Access to entrypoints is limited to reduce their load and avoid misuse by free-riders.

3.6.7 Pushing to Good Neighbours

Peers forward *push* packets to highly ranked neighbours to improve the overall distribution. By assuming that the source itself has a high ranking among its neighbours and good neighbours, packets first get distributed among good peers and then gradually to lower ranked ones. The good peers ensure that packets get further distributed using pull-based requests.

3.6.8 Slot Allocation

The uniform slot allocation strategy presented in Chapter 3.5 can be adapted to favor highly ranked peers. As every incentive strategy, it has to be used carefully. It is difficult to distinguish newly joined peers from free-riders and consequently, both get fewer slots.

3.7 Synchronization

The delivery of packets is weakly synchronized among the peers by exchanging *synchronization* messages. A *synchronization* message contains the sequence number and the timestamp of the next to be delivered packet and the time left to its delivery. Local delivery times are computed based on the packet timestamps, the *synchronization* messages, and the measured round trip times. Future versions might improve the synchronization by considering round trip time variances and by introducing an estimate for the synchronization accuracy at peers to weight *synchronization* messages (similar to the virtual coordinates of *Vivaldi* [23]).

3.8 Source and Network Coding

Peers rely on their neighbours to sent notifications for needed packets. Chances that peers benefit from each other are improved by adopting *source* coding. In its simplest form the source adopts *optimal erasure* code to create n encoded packets out of a block of $k < n$ original packets and distributes the n encoded packets. A peer having received any of the k encoded packets can reconstruct both the original k packets and the missing $n - k$ encoded packets. Given that peers randomly choose the packets to be requested, *source* coding increases the probability that two incomplete sets of packets for the same block do not overlap. *Optimal erasure* codes do not have any overhead, but are inefficient for large block sizes. They are nevertheless a good possibility for *StreamTorrent* because of its small buffers and even smaller block sizes. Examples for *optimal erasure* codes are the trivial polynomial code or *Reed Solomon* codes [36]. For larger block sizes, *near optimal erasure* codes provide better efficiency at the cost of an overhead of a few percent (e.g. [14, 15, 16]).

Earlier versions of *StreamTorrent* greatly benefited from source coding, especially under packet loss, churn, and when the upload bandwidth was close to the streaming bitrate. However, the benefits gradually decreased with newer versions because of the push mechanism and the improved slot allocation. More work has to done in the future to determine whether there might be a benefit when it is used with tit-for-tat. Alternatively, *network* coding [37] might help to further increase the chances that two peers can exchange packets. It would also help eliminating most duplicates we have in our system. To implement *network* coding, the notification strategy and the *allowed fast set* extension of the tit-for-tat strategy have to be adapted. The computational costs of *network* coding would be negligible if the block size is small.

Chapter 4

Protocol Implementation

Most parts of the protocol proposed in the previous Chapter have been implemented in Java. This implementation is used both by a simulator for testing and evaluation and by the *StreamTorrent Player*, a media player supporting live audio and video streaming. This chapter gives an overview over the protocol implementation itself, the simulator and the player are described in more detail in the Chapters 5 and 7.

4.1 Abstraction of the Protocol Environment

Two abstractions have been introduced to enable the implementation to run both within simulators and in the real world.

4.1.1 Task Scheduling

The protocol has to run tasks, either once at a given time or periodically, implying some concept of time. Naturally, the player runs in real-time using the clock provided by the operating system. On the other side, simulations are designed in an event-driven manner, i.e. events occur at given instants and might trigger future events in the progress. In the case of *StreamTorrent*, an event is either a task to be executed or an arriving packet. The simulator repeatedly executes the earliest event until none are left, implicitly yielding a simulated clock by using the timestamp of the current event.

To bring the two models together, an interface *Scheduler* has been defined and both the player and the simulator provide an implementation. It has methods to register tasks, obtain the current time, and dispatch incoming packets. To gain optimal performance within simulations, the protocol implementation is single-threaded to avoid synchronization overhead, i.e. at most one thread accesses protocol objects of a particular peer at any time. The player's scheduler implementation is straightforward using a single thread. The simulator's scheduler is more complex as it still exploits multi-threading by assigning simulated peers to threads. Each thread manages a given set of peers and has its own clock. The clocks are weakly synchronized among each other so that they do no differ by more than a few milliseconds. This is sufficient for round trip times of up to several hundred milliseconds. For a large number of peers, the load for every thread is almost identical and only periodical synchronization is needed, resulting in almost linear scalability with the number of available processing cores. The drawback of this threading model is that real world applications do not benefit from multi-threading, at least not within the protocol. But because the implementation has been heavily optimized to perform simulations, a single thread is sufficient.

The *blocking* scheduler is a third type of scheduler that suspends other threads before executing tasks and resumes them afterwards. It is used by the statistic components to maintain the invariant that at most one thread is accessing protocol objects of a given peer at any time.

4.1.2 Network

The network API provides methods to send and receive messages and an abstraction for network addresses. There are not two, but three different implementations, one for the real world and two to perform simulations:

- The real world UDP implementation is mostly simple to implement since *StreamTorrent* has been designed with UDP as transport protocol in mind (see Chapter 3.1). Providing support for firewalls and NAT devices is the most laborious part and outlined in Chapter 7.3.3.

- The main implementation to perform simulations is tailored towards scalability and efficiency. It does neither marshall packets nor simulates NAT and firewalls. It is used for testing with thousands of simulated peers. Models for packet loss and round trip times can be added by implementing a given interface.

- The second implementation for simulations implements the *Java Datagram Socket* API, allowing the real world UDP implementation to be used within simulations. It provides an API to design networks, i.e. to add peers, firewalls, and NAT devices and to enable and disable multicast. It is primarily used by *JUnit* [38] for automatic testing (see Chapter 5.3).

4.2 Architecture

As the previous chapters might have shown, many different aspects have to be considered when designing a peer-to-peer streaming protocol. For each aspect, various strategies might be feasible, each one typically having several parameters to further adapt its behavior. Consequently, there is room for many future improvements. This fact is also reflected within the architecture underlying the protocol implementation. There are various interfaces to abstract data structures and strategies, glued together by an implementation of *IStream* to provide the desired streaming functionality, for example:

- Identifiers:

  Most resources, such as streams and peers, are addressed by unique identifiers, typically 128-bit values.

- Stream descriptors:

  Stream descriptors contain information needed to connect to streams such as the bitrate, payload type, list of entrypoints, and their unique identifiers. Stream descriptors are created and distributed by the source.

- Data packets:

  The data broadcast by the source is transmitted in data packet. Each data packet has a sequence number and a timestamp, both 16-bit values. Timestamps are used to cope with varying bitrates and synchronization. The two main implementations are simulated packets and *RTP* packets. More sophisticated packets may include additional meta-data such as the identifier of the current audio track, movie, etc.

- Buffers:

  The buffer holds data packets received within the last few seconds and delivers them, if possible, in-order and in time according to the packet timestamps.

Strategies define how the system selects neighbours, requests packets, responds to requests, etc.

- Neighbour strategy:

  The neighbour strategy is responsible for selecting and managing neighbours among peers. There are two implementations, a random one and one based on hypercubes.

- Capacity Strategy:

  The capacity strategy allocates download slots for neighbours. The number of slots may depend, for example, on the neighbour's ranking, experienced packet loss, and total upload bandwidth.

- Notification Strategy:

  The notification strategy determines the frequency of notifications and bundles notifications with other packets.

- Request Strategy:

  The request strategy chooses packets among neighbours according to availability, age, etc.

- Pull Response Strategy:

  Upon data requests, the pull response strategy decides whether to answer with the requested data, to answer with an *overload* message, or to ignore the request.

- Push Strategy:

  The push strategy is used to forward push packets to neighbours based on their identifiers and ranking.

- Encoder and decoder:

  Encoders are applied at the source to modify data packets before they are distributed within the overlay. Similarly, decoders are applied at each peer before packets are delivered for playback. Encoders and decoders can be used to add, for example, encryption, meta-data, or fault-tolerance (e.g. source coding).

For each of these strategies, there are two interfaces. One interface is implemented by the strategy itself to provide the functionality, e.g. selecting a packet. The second interface, denoted the *strategy site*, specifies how the strategy can access the remaining system to send packets, read the buffer, obtain ranking information, etc. This way, improvements are possible in two ways, either by a replacing or extending an existing strategy and by modifying the *strategy site* to alter its view of the remaining system. For example, source coding has been added by providing an encoder and a decoder. In contrast, tit-for-tat has been implemented by modifying the request and response strategy sites. If, for example, the response strategy has determined that enough bandwidth is available and invokes its site to send the response, the new tit-for-tat site further checks whether the receiver has provided enough data itself and aborts if its not the case.

On top of all abstractions and strategies, there is a *facade* providing convenient methods to initialize strategies, create and join streams, etc. Overall, the chosen design should be generic to allow future updates and extensions.

4.3 Additional Protocols

Besides the streaming itself, other protocols are used to find peers and share information. The following sections give a brief overview over their implementations.

4.3.1 Entrypoints

Entrypoint servers are contacted by peers to find groups of peers, addressed by a group identifier. Upon request with a group identifier, entrypoint servers return a few members of the corresponding group. One example is a peer joining a stream after having obtained the corresponding stream descriptor. Stream descriptors provide a list of entrypoints and stream identifiers can be used as group identifiers.

4.3.2 Distributed Hash Tables

StreamTorrent adopted the concepts of the topology of *Kademlia*. An implementation for a distributed hash table comes therefore almost for free. Only the *put* and *get* operations have to be implemented.

Both *put* and *get* operations are performed by the peer that initiated the operation. To publish a new object o with identifier k, a peer compiles a list of x neighbours most similar to o regarding their identifiers. This list is gradually improved by concurrently asking the the most similar $y < x$ endpoints in the list for better peers. New peers are added to the list, replacing the worst existing ones, if their identifiers are more similar to k. This step is repeated until no more peers are found. Object o is then sent to the final x endpoints. Timeout handling and maintaining a list of x endpoints ensure good robustness against churn and packet loss.

Search operations using *get* are performed in a similar fashion. A list is maintained and peers are asked to either return the desired object, if possible, or to return better peers. The search is performed until a threshold of peers returned the same object or no peer can provide better peers. Waiting for a threshold of answers ensures that malicious peers can alter the result with only negligible probability. It is assumed that the application itself takes additional precautions, e.g. by choosing the content's hash value as key or by adopting digital signatures. If a peer does not receive the desired content, the number of concurrent candidate requests and the threshold of needed objects can be increased to gain more robustness.

An alternative, faster approach is to directly forward requests to the next best hop instead of sending list back. However, this approach is also more vulnerable to packet loss, churn, and malicious behavior. In the future, a better implementation might combine both approaches to allow faster searches while maintaining robustness.

To reduce the load on peers storing frequently accessed objects, a simple replication mechanism is adopted. Once a peer received the desired object, he replicates it to the best candidates in its list that not already have the object. With high probability the chosen candidates will be useful in the future.

Chapter 5

Simulator

The simulator is used to evaluate the protocol implementation presented in the previous chapter. It allows to run the protocol with tens of thousands of simulated peers. The simulator provides both automated tests to verify the basic functionality and various statistics to study its properties in detail. This way, the implementation can be tested before it is deployed in the real world, saving a lot of debugging time.

5.1 Simulations and Simulation Suites

There are various strategies to cope with problems like packet loss and churn, each one usually having various parameters. To facilitate setup and execution of simulations, they can be defined in simple XML files (see Figure 5.1). The document structure is similar to *ANT* [39], a popular Java build system. The files are usually small and easy to understand and modify. Virtually all parameters of the protocol and the network setup can be changed. For example, groups of peers can have their own set of strategies (free-riding, honest, etc.). A templating mechanism allows to reuse simulation settings within new simulations and to selectively override inherited settings with new values.

```
<simulation>

  <log showUnderflows=”true”/>
  <import file=”../default.xml”/>

  <network topology=”uniform”>
    <packetLoss type=”none”/>
  </network>

  <peers nPeers=”${nGood}” group=”1” upload=”150000” download=”800000”/>
  <peers nPeers=”${nNormal}” group=”2” upload=”75000” download=”800000”/>
  <peers nPeers=”${nFreeRiders}” group=”3” upload=”10000” download=”800000”/>

  <streamSource extends=”defaultStreamSource”/>

  <streamEndpoints nEndpoints=”${nGood}” extends=”honestPeer”/>
  <streamEndpoints nEndpoints=”${nNormal}” extends=”honestPeer”/>
  <streamEndpoints nEndpoints=”${nFreeRiders}” extends=”freeRider”/>

  <event at=”21000”>
    <stopStreaming group=”1” p=”${p}”/>
    <stopStreaming group=”2” p=”${p}”/>
    <stopStreaming group=”3” p=”${p}”/>
    <snapshots id=”availableWindow,bufferImage,degree” each=”250” for=”6000”/>
  </event>

  <import element=”run” file=”../default.xml”/>

</simulation>
```

Figure 5.1: Example XML file describing a simulation.

Simulation *suites* allow to run several simulations together and to compare results. An example suite that sets up crash simulations is given in Figure 5.2.

```
<suite name=”Crash”>

  <param name=”rate”>36</param>

  <param name=”nGood”>300</param>
  <param name=”nNormal”>1000</param>
  <param name=”nFreeRider”>50</param>

  <param name=”simulationLength”>31000</param>

  <simulation name=”crash 25%” file=”crash.xml”>
    <param name=”p”>0.25</param>
  </simulation>

  <simulation name=”crash 50%” file=”crash.xml”>
    <param name=”p”>0.5</param>
  </simulation>

</suite>
```

Figure 5.2: Example XML file describing a simulation suite.

5.2 Statistics

To analyze the simulations, the simulator compiles various statistics that give an overview of what is going on within the system, for example:

- Network throughput aggregated by packet type.
- Expected round trip time to neighbours.
- Average number of neighbours in total and by group.
- Data sent and received within the last seconds.
- Packet loss
- Buffer state
- Histogram of the number of hops required for packets to reach a peer.
- Packet exchange between different groups.
- Histogram of how fast packets are distributed.
- Histogram of when peers start delivering packets.

Data can either be collected at given instants or over a period of time. Peers are usually combined into groups to compare groups with each other. The collected data is also stored in *csv* files, a simple and popular file format that ca be read by every spreadsheet application.

5.3 JUnit

There are various tests to analyze *StreamTorrent* automatically without having to manually inspect the statistics. The unit testing framework *JUnit* [38] is used for this purpose. Example tests are:

- NAT detection tests with various network configurations.

- Connectivity within the overlay.
- Synchronization tests verifying that peers properly resynchronize after the source left and later rejoined.

5.4 Simulation Efficiency

Both the simulator and the *StreamTorrent* protocol implementation have been heavily optimized with Java profilers [40, 41]. Within a second, the simulator is able to simulate up to 2,000 peers for one simulated second on a decent computer. While the subsequent chapter mostly evaluates the protocol with 10,000 peers, upto 50,000 peers have been simulated. It is expected that this number increases to 100,000 and more in the near future.

Chapter 6

Evaluation

The following sections give a brief overview of the simulation results that have been performed with the simulator.

6.1 Network Model

The simulated network roughly correspond to UDP in today's Internet. The header consists of 28 bytes and the maximum transfer unit is 1500 bytes. The payload of data packets has a length of 1328 bytes, corresponding to the length of RTP packets generated by the VLC player.[1] The source usually sends 36 packets per seconds, resulting in a bitrate of roughly 50 KB per second. This corresponds to the bitrate needed to send a video stream with reasonably good quality.

Peers are uniformly distributed within a square and round trip times are computed using the euclidean distance. The diagonal of the square corresponds to 200 ms, leading to a maximum round trip time of 400 ms, a typical value for today's Internet. Each peer has a maximum upload and download bandwidth. We focus on the upload bandwidth in this evaluation since it is usually significantly smaller than the download bandwidth.

Currently, there are two packet loss models. One model drops packets with constant probability, the other drops packets if the used bandwidth exceeds the maximum. The second model has been implemented based on tests performed with a broadband connection of a larger Internet service provider. For example, small bursts exceeding the total available bandwidth are allowed, but only for a short period of time. Packet loss is either enabled or not depending on the focus of the simulations. It will be shown that the bandwidth used by peers rarely exceeds the available one.

6.2 Locality

Figure 6.1 compares the locality-aware neighbour selection strategy with a random one. The simulations have been performed with 10,000 honest peers. Clearly, the expected round trip time to neighbours is significantly smaller while connectivity is maintained.

Note that the trade-off between connectivity, locality, and incentives makes it more difficult to reduce round trip times. Ignoring either connectivity or incentives would naturally result in better locality, but a more fragile protocol in general. The subsequent section about incentives gives more information on this trade-off.

[1] More information about payload size, codecs, and players is given in Chapter 7

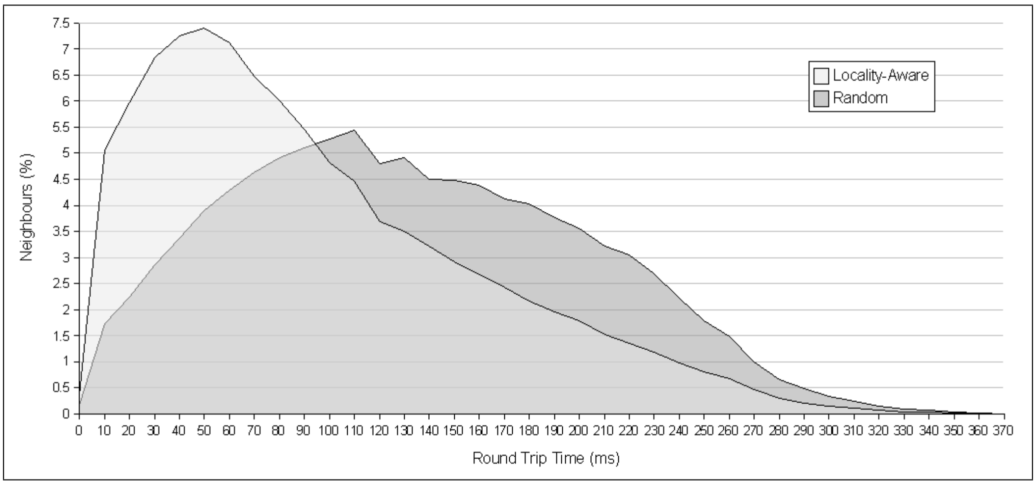


Figure 6.1: Effect of locality-awareness.

6.3 Radius

Arranging peers within a hypercube guarantees a logarithmic diameter. Packets should therefore arrive at peers after a logarithmic number of hops. Figure 6.2 shows a histogram of the number of hops taken by packets to reach a particular peer for 10, 100, 1,000, and 10,000 peers.

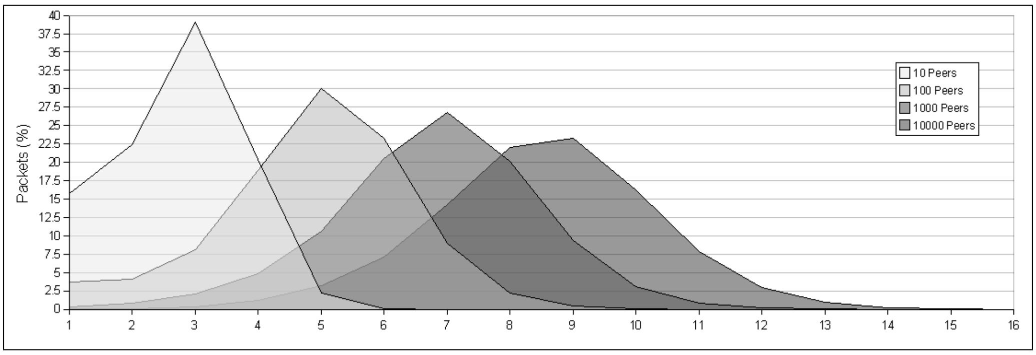


Figure 6.2: Histogram of the number of hops taken by packets.

By computing the averages and plotting them together with the number of peers, the linear increase of hops with exponentially more peers becomes more apparent as depicted in Figure 6.3.

6.4 Pushing

Figure 6.4 shows how fast packets get distributed among the peers if pushing is used. While it takes time for the pull-only strategy, e.g. *Chainsaw*, to reach the 10%-threshold, it is clearly sped up by the push-based strategy. Once a given threshold of peers is reached, pushing is not needed anymore to quickly distribute packets to the remaining peers. The figure also reveals that the adopted strategy scales well with the number of peers. As for the radius, delays grow logarithmically with the number of peers. The difference between 1,000 and 10,000 peers is between merely 100 and 175 milliseconds.

Note that the pull-only strategy performs slightly worse than the one adopted by *Chainsaw*. This has several reasons:

- *StreamTorrent* sends less notifications, stretching the initial distribution interval.

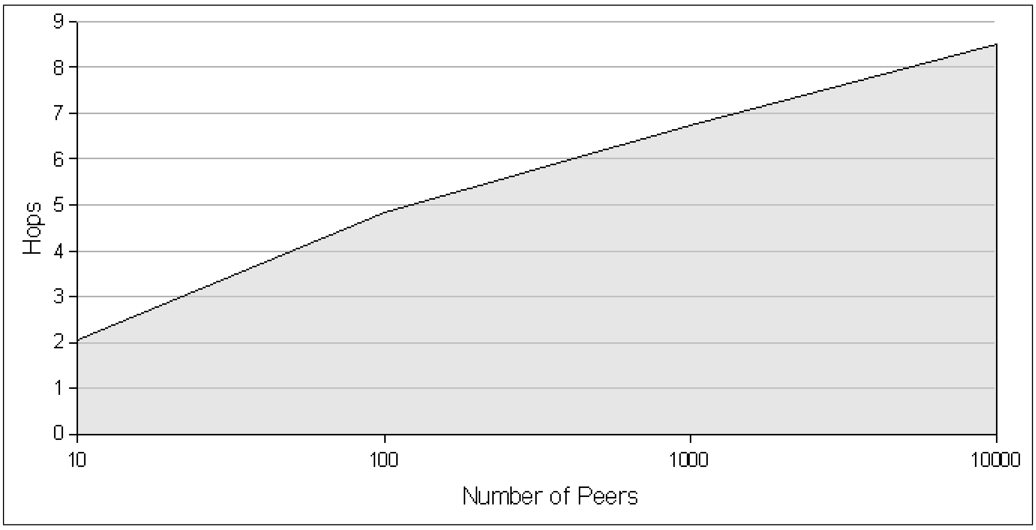


Figure 6.3: Average number of hops for a given number of peers.

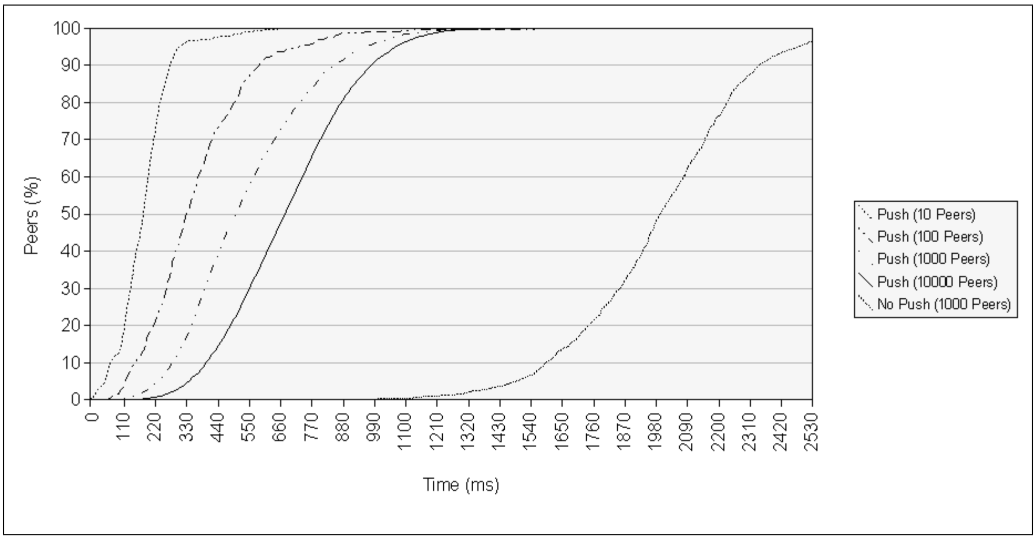


Figure 6.4: Packet distribution with and without pushing.

- Incentive mechanism such as *tit-for-tat* are used.
- The source behaves like a regular peer, i.e. it does not answer requests with packets that have never been sent before. Such a strategy immediately reveals its role as the source. On the other hand, it reduces the load on the source and speeds up packet distribution for pull-only strategies.
- The request strategy is not entirely random because peers try to avoid exceeding their total upload bandwidth.

On the other hand, the pull-only strategy benefits from locality-awareness.[2]

Figure 6.5 shows the ratio of packets received by pushing. The corresponding duplicate ratios are given in Figure 6.6. The *simple* strategy forwards a push packets to all 2^b neighbours, while the *duplicate avoidance* strategy only to $2^b - 1$ as described in Chapter 3.3. Both the ratio of push packets and duplicates decrease logarithmically with the number of peers. The *simple* strategy is about two magnitudes better in terms of efficiency, but yields more duplicates. Future versions of *StreamTorrent* will switch

[2] *Chainsaw* uses a constant delay of 50 ms.

between the two strategies based on estimations of the number of peers and the number of hops already taken by packets. For less than 100 peers it might be beneficial to further throttle pushing to reduce duplicates, a push ratio of more than 20% is not needed anyway. Interesting to note is that the *duplicate avoidance* strategy performs with 10,000 peers not much worse than the *simple* strategy, even though the push ratio is significantly smaller. This further confirms the observation from Figure 6.4 about scalability and suggests that 100,000 or even 1,000,000 peers does not cause any problems. The push strategy only has to provide a good initial distribution of new packets, the pull strategy ensures that packets are quickly distributed to the remaining peers.

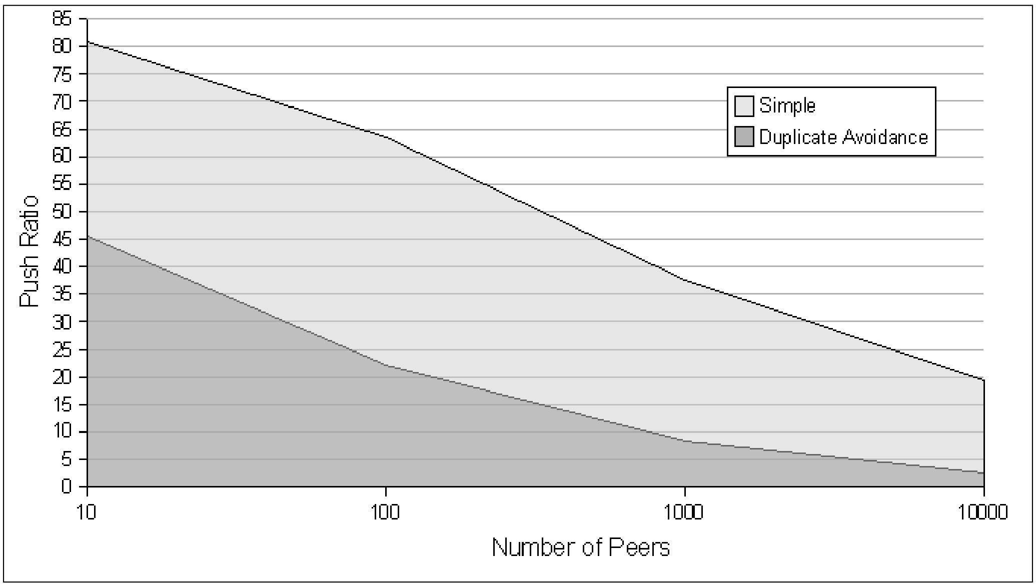


Figure 6.5: Push ratio for a given number of peers.

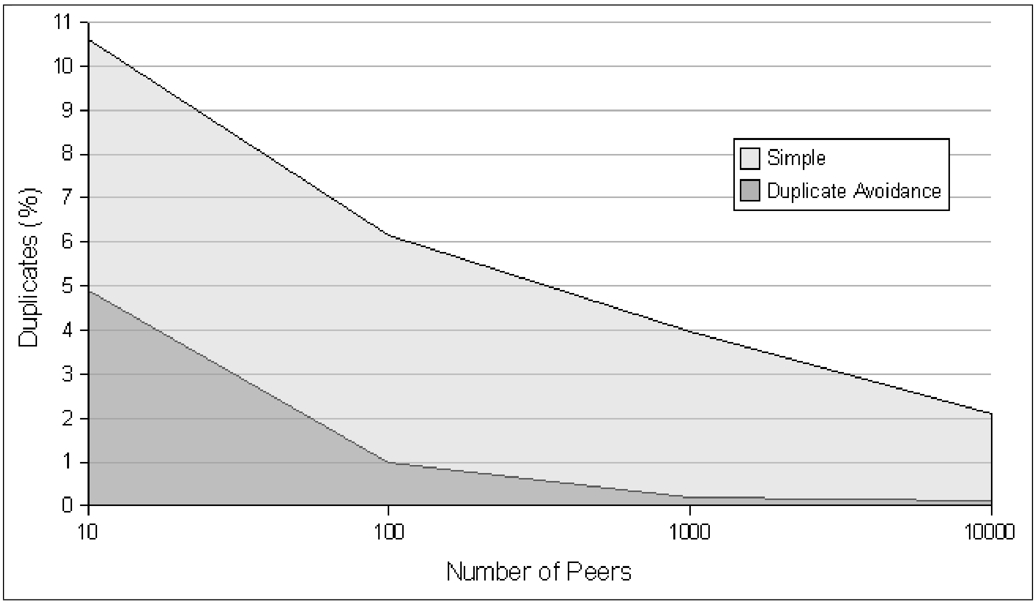


Figure 6.6: Duplicate ratio for a given number of peers.

6.5 Overhead

The overhead imposed by *StreamTorrent* depends on many parameters. More peers trigger a logarithmic increase of the number of neighbours and packet delays, leading to both larger and more notifications. Depending on the incentive mechanism, peers might search more aggressively for new neighbours and frequently drop neighbours. On the other hand, the absolute overhead increases only slightly with the streaming bitrate, i.e. *StreamTorrent* becomes more efficient if the source sends more packets. This holds because there is no need to search for more neighbours or to send more notifications at higher rates. The only difference is that notifications become larger. However, this increase is small for reasonable packet rates because the constant-size packet header already takes a lot of space.

Figure 6.7 shows the overhead of *StreamTorrent* compared to an optimal (offline), unicast-based solution, i.e. the ratio between the number of bytes sent in data packets (including the header) to the number of bytes sent in total. Duplicates are accounted for as overhead. The simulations have been performed with 5,000 peers. A combination of the *duplicate avoidance* and the *simple* strategy is used to push packets (the *simple* strategy is used near the root).

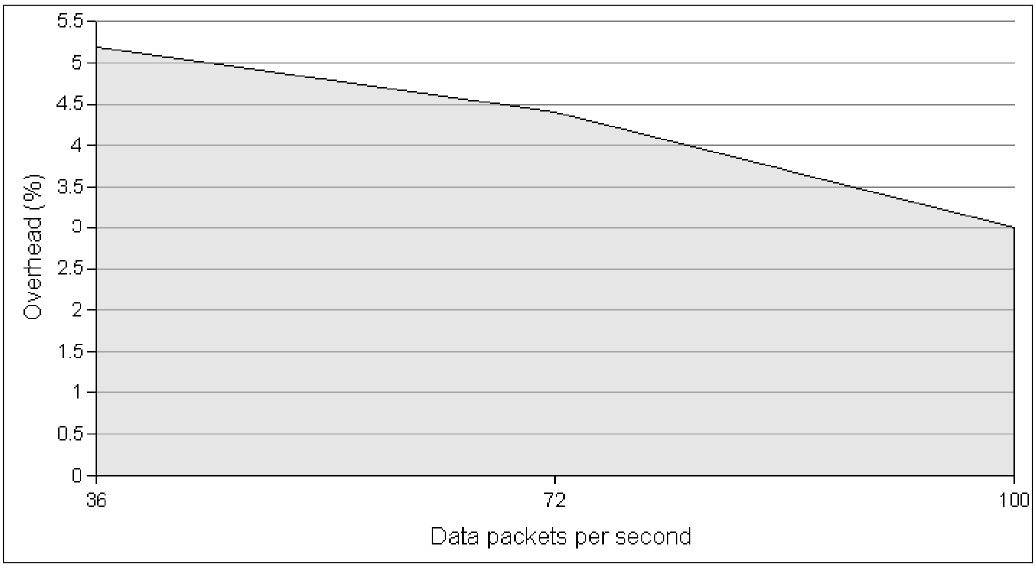


Figure 6.7: Protocol overhead with 5,000 peers.

By tuning the parameters to a particular situation, the overhead can be further reduced. For example, we were able to get 97.5% efficiency at a rate of 72 packets per seconds. We therefore expect further improvements in the future. Especially at higher packet rates one might better exploits packet bundling.

Overall, an overhead of 5% or less is already remarkably good. Considering that many tree-based solutions apply some sort of *rateless* erasure coding, they immediately incur the same overhead only by the coding itself. Additional overhead is created by duplicate packets and to maintain the tree.[3]

To show that the protocol is efficient, it is essential to also study how close the total upload bandwidth and the stream bitrate can get. It can not be expected that peers send with the maximum possible bandwidth. To test this boundary, two peer groups are used. The first group consists of 100 peers with 125 KB/s upload bandwidth. The second group consists of 1,000 peers. All peers managed to sustain a continuous stream with negligible packet loss if the peers in the second group have an upload bandwidth of 60 KB/s. Figure 6.8 shows that peers rearely excceed their bandwidth limits and

[3] An example using *Bullet* is given in the *Related Work* chapter.

Figure 6.9 shows that buffers are almost full. Note that the second figure only shows in-order packets that are ready to be delivered, usually there are more packets with some missing in between. A second, more powerful group has been used since it is unlikely in the real world that all peers have the same capabilities. This groups is prefered by the incentive mechnaism to push new packets, resulting in a better packet distribution. Without having this group of better peers, the upload bandwidth has to be between 65 KB/s and 70 KB/s.

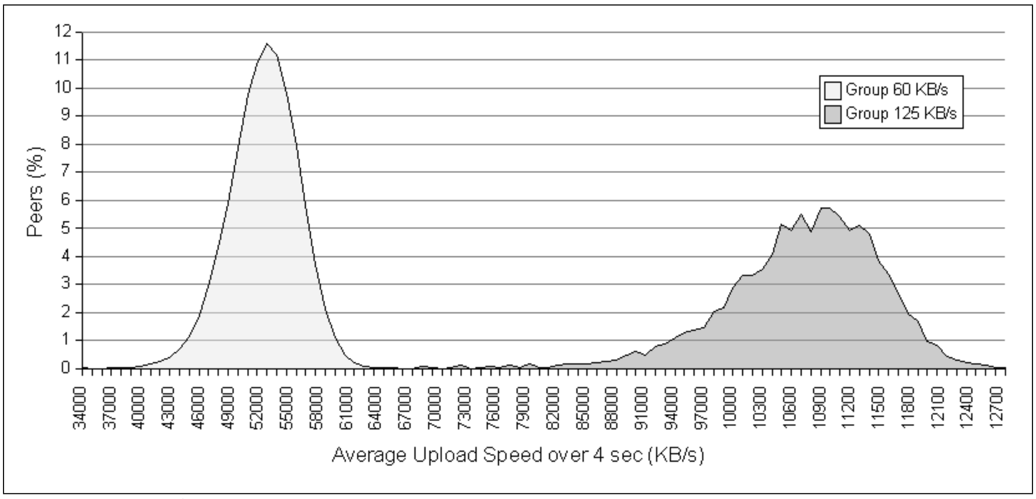


Figure 6.8: Average upload bandwidth over 4 seconds if upload bandwidth is scarce.

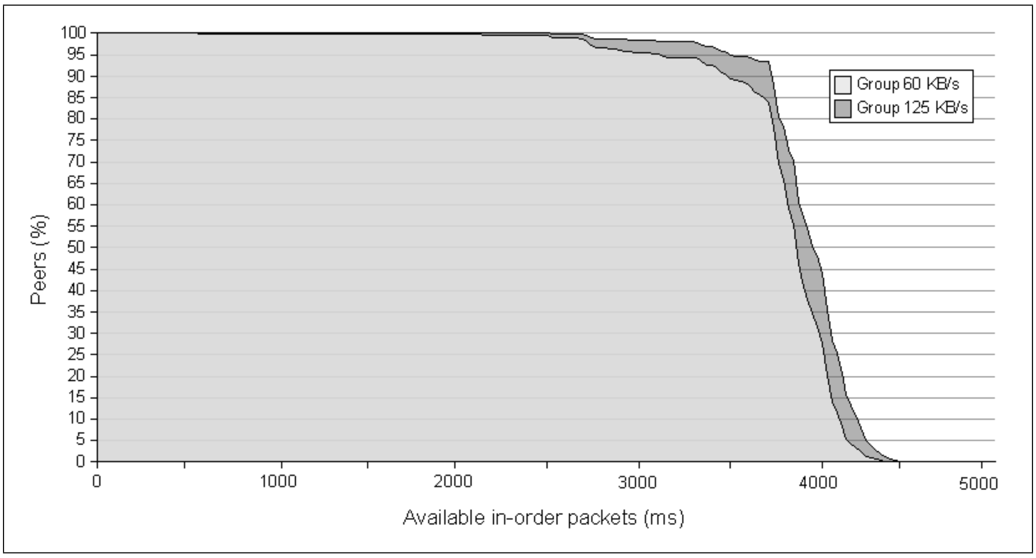


Figure 6.9: Buffer if upload bandwidth is scarce.

6.6 Crash & Churn

Results about robustness like the ones of *Chainsaw* [1] can be confirmed. Tests have been performed with 10,000 peers, 1,000 peers having an upload bandwidth of 125 KB/s and 9,000 peers with 70 KB/s. Peers recover extremely fast if 25%, 50%, or even 75% leave the overlay simultaneously. The remaining peers deliver all packets in time, no underflows occur. By leaving is meant that peers send a final *leave* message to their neighbours. An example with 75% leaving is given in Figure 6.10.

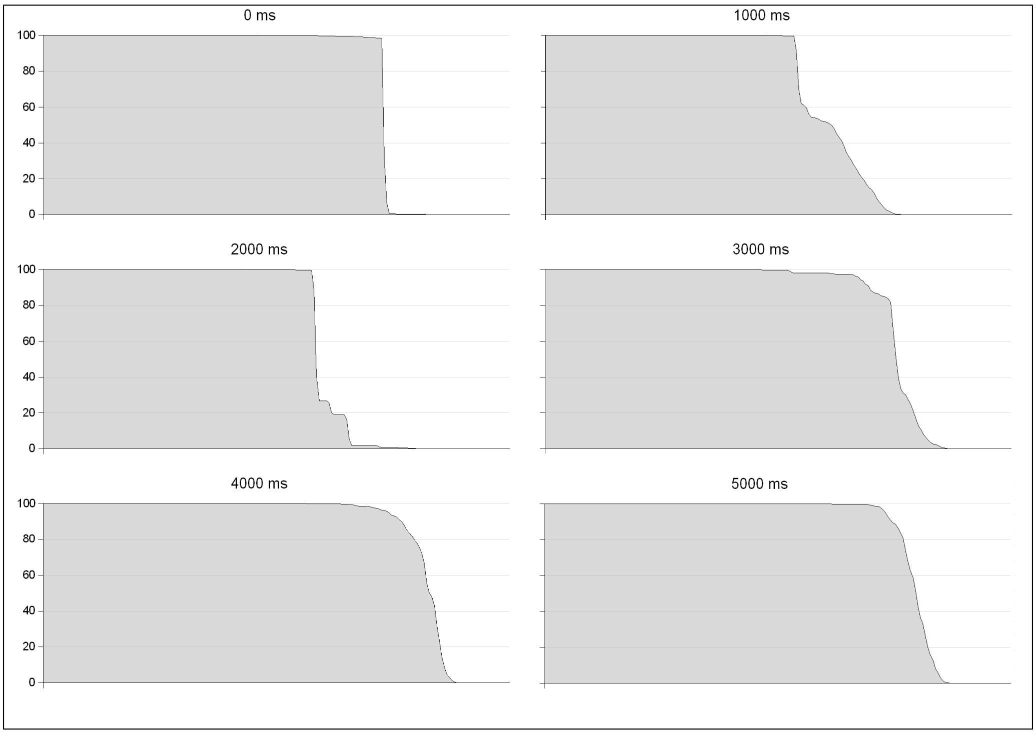


Figure 6.10: Buffer snapshots when 75% of the peers leave simultaneously.

The overlay of *StreamTorrent* does not yield any new problems. In contrast to distributed hash tables, no expensive replication among the remaining peers has to be performed and it does not matter if the overlay is partly broken for a short while. It gets slightly worse if peers crash and do not send a *leave* message. It takes longer until crashed neighbours are removed and replaced by others. Acknowledgments for *push* packets near the root of the tree help to maintain efficiency. 50% of the peers can fail without any packet loss. If 75% fail, 1,700 underflows (0.54%) occurred in an interval between 4,300 ms and 7,800 ms after the peers failed. After 8,000 ms the overlay has stabilized and delays are smaller than before due to the smaller number of peers.

The protocol can be made even more robust, for example, by adopting *source* coding and increasing the buffer size. Naturally, robustness depends on the available bandwidth. If peers have only 60 KB/s upload bandwidth, recovery takes longer and underflows occur more often.

Given the connectivity guaranteed by the overlay and the robustness shown in this section, smaller churn rates are no problems either, typically not even noticeable since peers periodically drop some of their neighbours anyway. A comparision with tree-based protocols is given by *Chainsaw* in [1].

6.7 Packet Loss

Figure 6.11 shows that the protocol is also robust against packet loss. The tests have been performed with a constant probability of packet loss, 150 peers having 125 KB/s upload bandwidth, and 850 peers having 80 KB/s upload bandwidth. The packet loss probability is between 1% and 20%. Note that with 20% packet loss, the usable upload bandwidth drops to 64 KB/s.

The packet distribution speed with 10% packet loss is depicted in Figure 6.12. What can be seen is that the push strategy is still working well. However, it takes longer until packets are delivered to the last 20% of the peers. This is not surprising because the probability of a request failing is 0.19 and the timeout length is about three times

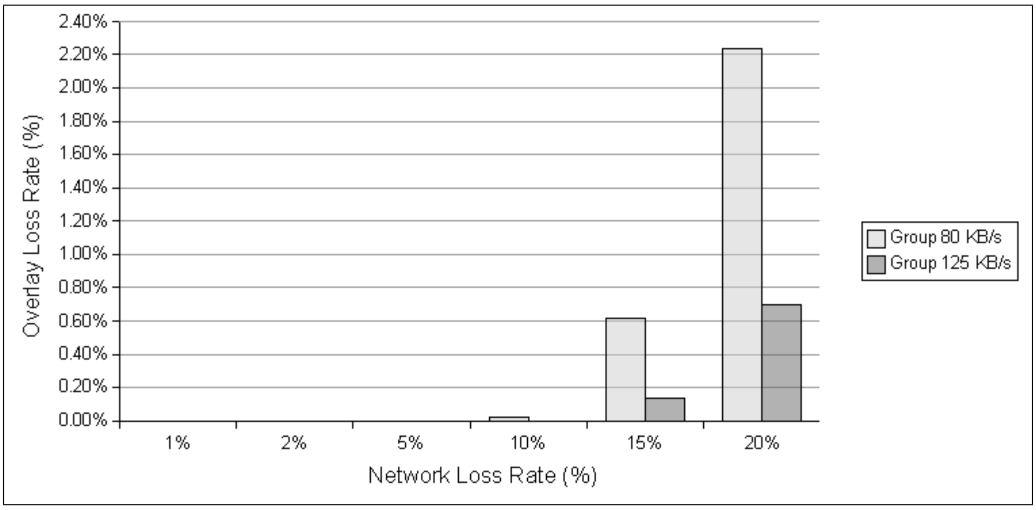


Figure 6.11: Packet loss: Ratio of packets not delivered in time.

the round trip time. If there are longer delays, e.g. more packet loss, the notification strategy will start increasing its notification frequency. Note that such scenarios are unlikely in the real world as packets are distributed among good peers first, which generally experience only minor packet loss.

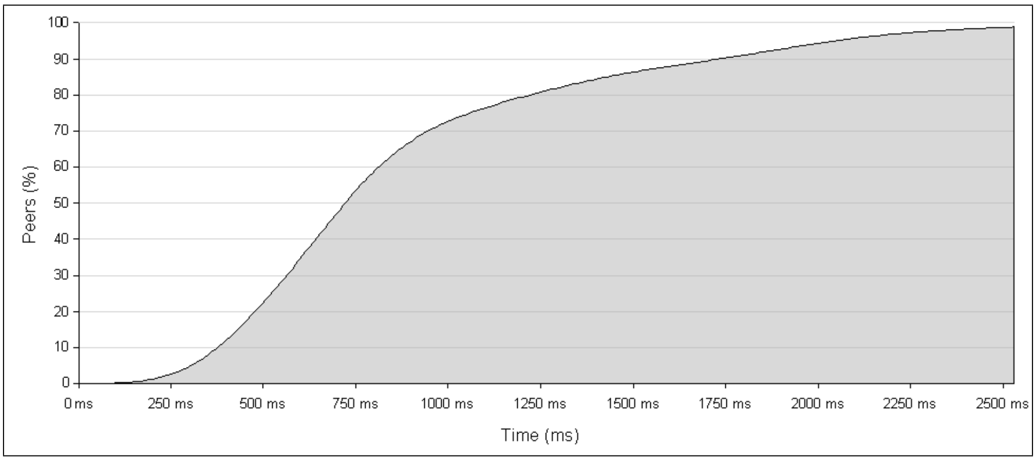


Figure 6.12: Packet loss: Distribution speed with 10% packet loss.

6.8 Incentives

As mentioned in Chapter 3.6, the incentive mechanism is still work in progress. This sections gives three examples and outlines a few problems and work that still has to be done.

When designing an incentive mechanism, there is a trade-off between efficiency and robustness against free riders. By making the incentive mechanism more aggressive, weaker peers are dropped more quickly, but it also becomes harder to join a network and to find a neighbour to request a packet, especially if a peer's upload bandwidth is close to the stream bit rate. Consider an example where 75% are almost free-riding, i.e. provide an upload bandwidth of 15 KB/s. Naturally, these peers consume some bandwidth from the remaining peers since they share a few packets and the remaining peers first have to determine that they do not share enough. The remaining peers mostly managed to sustain a continuous stream if they have an upload bandwidth of 120 KB/s and by making the incentive mechanism very aggressive. Figure 6.13 shows that the honest peers have 40% free riders as neighbours (instead of the expected 75%). This value is low considering that connectivity is still fulfilled and that free-riders continuously search for new neighbours.

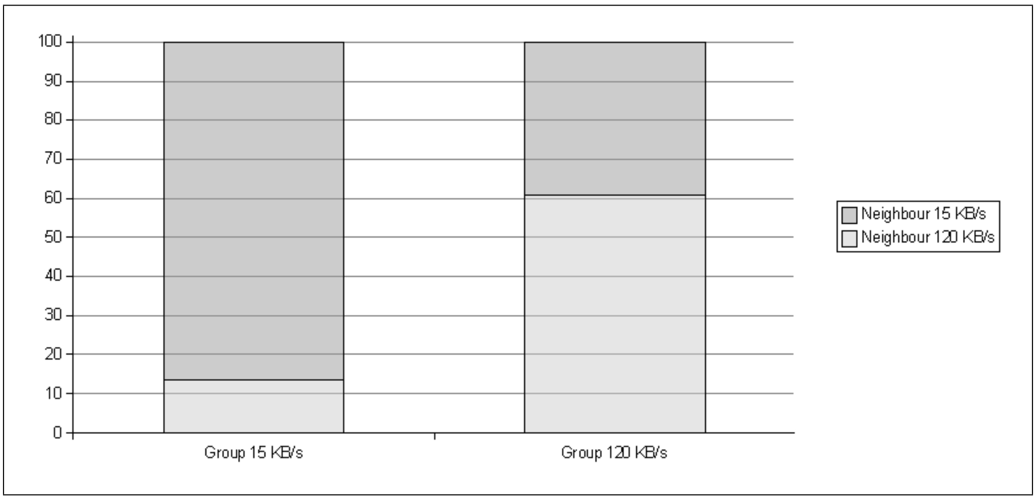


Figure 6.13: Neighbourhood when 75% are free riding.

Figure 6.14 shows the number of in-order packets within the buffers of each group. Free-riders receive only a fraction of the packets and none is able to sustain a continuous delivery. In contrast, buffers are almost full in the honest group.

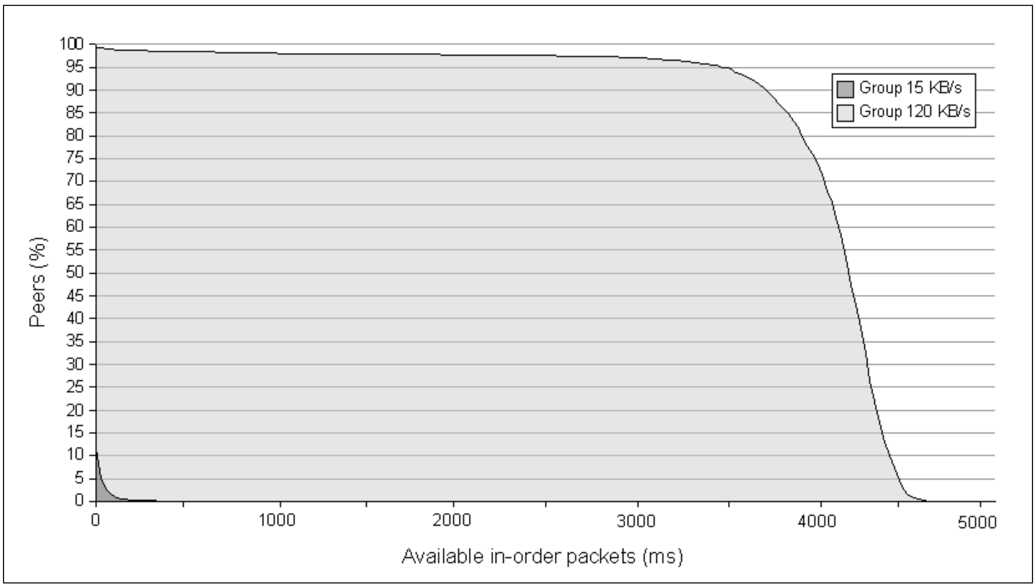


Figure 6.14: In-order packets within the buffers when 75% are free riding.

The given example is possible since the ranking of neighbours is working well. Figure 6.15 shows a histogram of the ranks of honest peers computed by honest peers. The ranking is solely based on the provided bandwidth. Attempts to optimize the ranking function resulted only in minor improvements.

The average upload bandwidth of honest peers is 85 KB/s, although 120 KB/s are needed. Better slot allocation and tit-for-tat strategies might improve this bound in the future.[4]

[4] Note that the slot allocation strategy has already been adapted for this example to favor good neighbours.

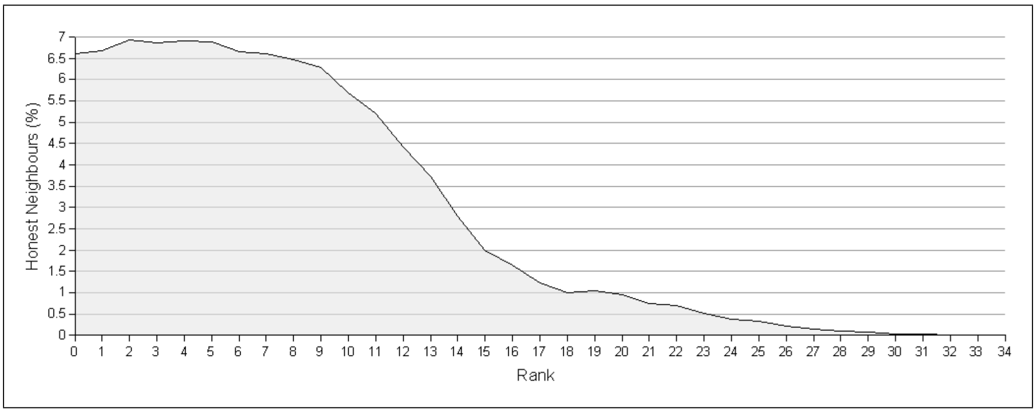


Figure 6.15: Ranking of the honest neighbours when 75% are free riding (computed by honest peers).

Given that none of the free-riders is able to deliver the stream, there is no reason to continue free-riding, at least for rational free-riders. It might therefore be more reasonable to fine-tune the incentive mechanism to a small number of free riders so that they will not be able to sustain stream delivery. The two subsequent examples are based on four groups:

1. 250 peers with 125 KB/s upload capacity
2. 500 peers with 68 KB/s upload capacity
3. 300 peers with 40 KB/s upload capacity
4. 200 peers with 15 KB/s upload capacity

Figure 6.16 shows the result for a weak incentive mechanism. The first two groups have no problem at all. The 40 KB/s group is also doing well, while the 15 KB/s group clearly has problems getting all packets. Figure 6.17 reveals that the 15 KB/s group more aggressively attempts to get high-bandwidth neighbour at the cost of locality-awareness.

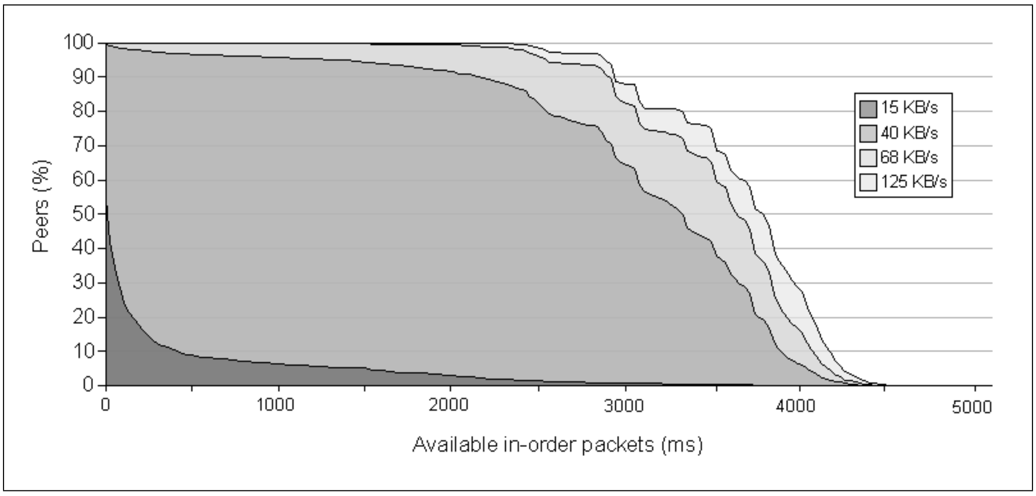


Figure 6.16: Buffers with a weak incentive mechanism.

By adjusting the parameters, it is possible to further decrease the number of packets free-riders receive at the cost of the 40 KB/s group (see Figure 6.18). Future versions will have to better adjust parameters to the current situation. Moreover, real world tests will be needed to get more information about peers and their upload bandwidth to determine both the stream bit rate and a suitable incentive strategy.

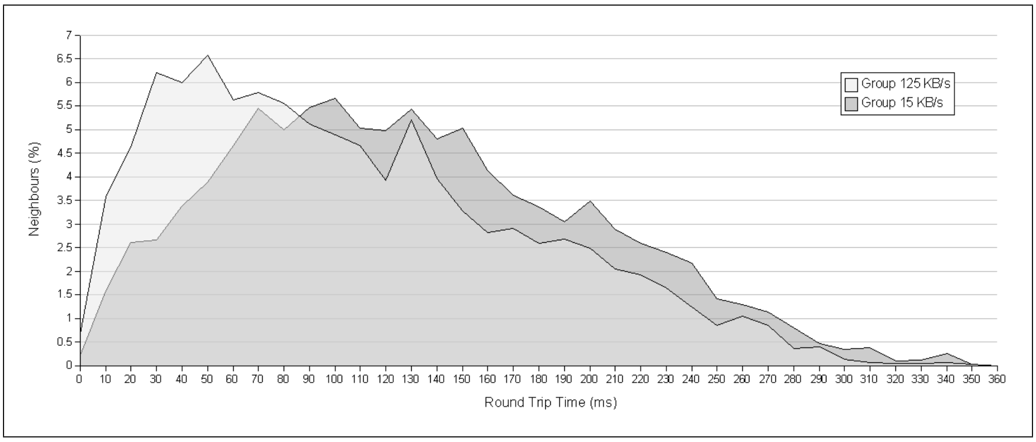


Figure 6.17: Locality-awareness with a weak incentive mechanism.

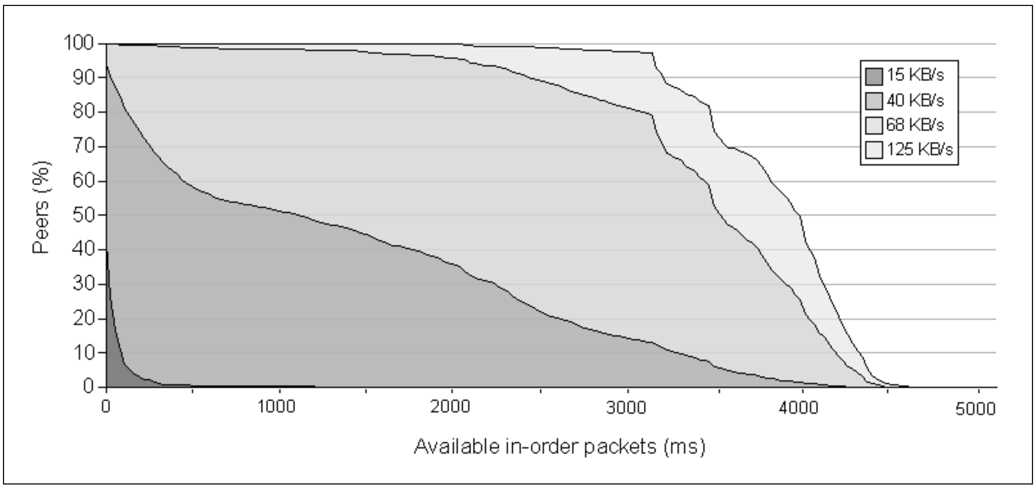


Figure 6.18: Buffers with a moderately aggressive incentive mechanism.

Chapter 7

StreamTorrent Player

The *StreamTorrent Player* is a media player supporting live audio and video streaming based on protocol proposed in chapter 3. Similar to *BitTorrent* and its *.torrent* files, streams are described by XML files and can be hosted on any HTTP server. An integrated third-party player is used for playback.

7.1 Architecture

The design of the *StreamTorrent Player* is composed of several layers shown in Figure 7.1. The chosen architecture should provide a clean, modular design and facilitate future extensions. The subsequent sections describe the layers in more detail.

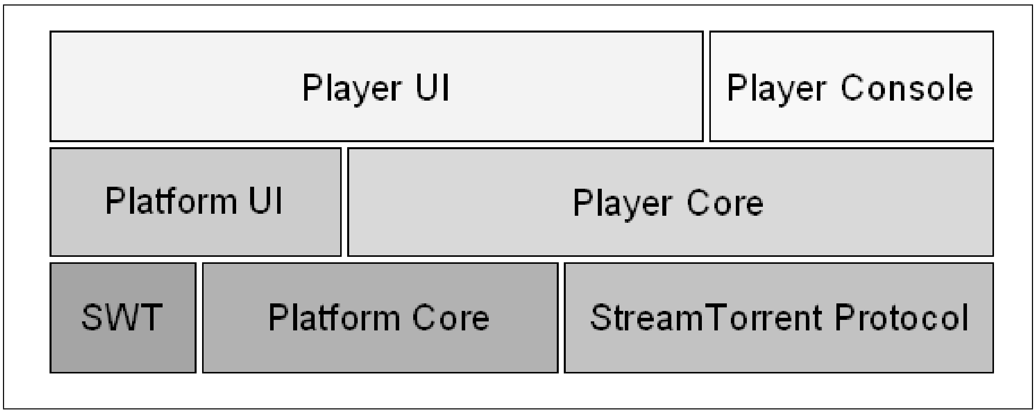


Figure 7.1: Software Layers of the *StreamTorrent Player*.

7.2 Platform Core

The *Platform Core* layer provides a runtime environment for the player. This includes a plugin architecture, a unified event model, an adapter repository, among others. It is independent from the player and can be used in other projects.

7.2.1 Plugin Architecture

A plugin framework is used to support future extensions like new protocols, integrated players, and media formats. The plugin framework of *Spamato* [42], a collaborative spam filter, has been adopted for this purpose. It is extremely small but still provides all important features.

Plugins can be installed, updated, and removed at runtime without restarting the system. A plugin consists of a *descriptor*, libraries, and other files such as images. Similar to streams and stream descriptors, plugins are described by plugin descriptors. Plugins might define dependencies to access other plugins. Dependencies are either strictly required or optional. The plugin framework ensures that the plugins are initialized and disposed in a correct order. Libraries are written in Java like the player itself, although additional support for scripting languages like JavaScript [43] is in preparation.

One of most important features are *extensions* and *extension points*, a concept borrowed from *Eclipse* [44]. By defining an *extension point*, a plugin can enable other plugins to add their *extensions*. For example, the user interface offers a menu *extension point* in order that other plugins can place their menu items by writing a corresponding *extension*. The plugin framework itself provides *extension points* so that it can be extended itself.

7.2.2 Adapter Repository

Adapters are a simple, yet powerful approach to extend the functionality of objects. For instance, the user interface provides a *copy* action within the menu bar and naturally, there are various kinds of objects that can be copied, such as stream descriptors, favorite items, and plugins. The simplest solution would be to let each object implement an interface. However, this leads to several problems:

- Most of the objects usually not belong the user interface. Introducing such a dependency is not wise.
- Maybe some of the objects cannot be changed, e.g. objects from third-party libraries.
- Actions might be added and removed in the future. All objects would have to be updated in the progress.

Adapters are a more convenient alternative. An adapter, a compositional adapter to be more precise, is an object implementing type B for an object of type A. In order to find such adapters, or a factory to create them, an *adapter repository* is provided. It has just two methods, one to register new adapters and one two find an adapter given an object and the desired interface. This way, new features can be added to an object by contributing a new adapter and without having to change the original object. The implementation is rather simple and based on the *Java Reflection API*. The *adapter repository* is frequently used within all layers of the *StreamTorrent Player*.

7.2.3 Event Handling

A unified event handling mechanism is provided and is used within all upper layers. Every object interested in receiving events implements the interface *IListener* and every object creating events implements *IEventSource*. *IEventSource* provides methods to register listeners and fire events. *IListener* has a single method called upon new events. An event has three properties, its source, its type, and the affected object, e.g. a stream having changed its state to *buffering*. Unifying event handing has proven extremely useful. For example, most of the user interface items can automatically listen to their underlying objects and update their appearance accordingly.

7.3 Player Core

The *Player Core* layer is based on *Platform Core* and the *StreamTorrent* protocol. It implements the main functionality of the *StreamTorrent Player* independent from any user interface. This includes initialization and management of the *StreamTorrent* protocol, providing access to resources, feeding an overlay with audio and video streams, delivering streams to players, and recording streams. Both the *Player UI* and the *Player Console* simply provide an interface for this functionality.

7.3.1 Stream Input

The player supports both audio and video streaming based on the *Real-time Transport Protocol* (or *RTP*) defined in RFC 3550 [45]. *RTP* is a standardized packet format developed by the Audio-Video Transport Working Group of the IETF [46]. It provides, for example, sequence numbering, time stamping, and payload-type identification. RTP is supported by many media players and streaming servers, and hence, there is a lot of support and no dependencies to a single vendor. And since it was designed as a multicast protocol, it fits in perfectly.

RTP supports a wide range of payload types. *MPEG* is one of these payload types and supports various audio and video codecs, whereas some are better suited for streaming than others. Popular video codes are *H.264* [47], *XviD* [48], and *WMV* [49]. *H.264* provides the best quality among these three codecs, especially at low bitrates, but also incurs the highest computational overhead. *MP4* and *AC-3* are popular audio codecs providing good quality at similar bitrates.

The *Player Core* layer provides the required implementations that *RTP* can be used within the *StreamTorrent* protocol (accessing the *RTP* source, packet marshalling, and delivery).

7.3.2 Resource Repositories

The player uses various kinds of resources like stream descriptors, plugins, video snapshots, and stream program information. Typically, there are several sources to obtain resources. In order to facilitate resource management, access to these so-called *resource repositories* has been standardized. Each kind of repository implements the same interface to hide the underlying implementation and to extend and replace repositories more easily. There are currently three implementations:

1. Local repository

   The *local repository* is a local, file system-based repository implementation. It is usually used as a cache for other kinds of repositories.

2. Traditional servers

   There is a web application running on any Java-enabled HTTP server. It hosts its own *local repository* and provides access to it using remote procedure calls. The communication is either XML-based or binary. Automatically generated stubs are used on the client side to hide implementation details.

   The server also provides a web-based interface. It can be used by users to browse for and download resources. The most prominent example is *streamtorrent.org.* For this purpose, a browser has been integrated into the user interface and is shown directly after startup.

3. Distributed hash tables

   A peer-to-peer-based implementation is given based on the distributed hash table presented in Chapter 4.3.2. Using the described replication and locality techniques, it is a scalable and efficient alternative to centralized web-based repositories. However, for performance reasons, resources can only be addressed by their identifiers, meaning that no full-text search is provided. Its main use is to update already obtained resources to reduce the load on the centralized servers.

Depending on type of resource, repositories provide additional services. Stream descriptors are usually indexed and can be queried using a simple query language (see Figure 7.2). The query language supports both boolean and full-text expressions. Repositories translate queries into corresponding native queries when they are executed.

- `mediaType == 'video' ORDER BY name`
- `mediaType == 'audio' && keywords CONTAINS 'soccer' ORDER BY name`
- `name like 'Soccer Finals' and mediaType == 'video' LIMIT 10`

Figure 7.2: Examples queries to search for stream descriptors.

7.3.3 NAT Support

NAT devices and firewalls are one of the main obstacles when deploying peer-to-peer applications. Several strategies are adopted to both detect and (partly) by-pass NAT devices and firewalls.

To detect the presence of a NAT device, one of the freely available IP checker services is queried to obtain the external IP address. If a NAT device is present, then the external address does not match the local address. If this is the case, ping messages are exchanged with non-local entrypoints or peers to determine the NAT type. An overview over the different NAT types is given in the appendix. Entrypoint and peer addresses are obtained either from a previous session, i.e. a list of the latest peers is saved to disk on shutdown, or from predefined http servers. Users might also provide some addresses by themselves using the preference dialog. Once some addresses are available, a ping message is sent to one of the peers. The peer, given that it is still alive, responds with a pong containing the observed sender address and port. Moreover, it also forwards the ping to another, third peer that also responds to the original sender. This is repeated several times. If no pongs have been received, either a firewall is present or the peer list is invalid. If no majority of peers reported the same address, it is assumed that the NAT device is symmetric. Otherwise, if there is a majority, it is either *full cone* or *restricted cone* depending on whether a forwarded ping response has been received. Note that we do not distinguish between *restricted cone* and *port restricted cone* since only one port and multiplexing is used. Besides the majority vote, each ping message contains a unique identifier that has to be included in the answer of the regular and the forward ping. Naturally, only one answer is accepted per ping identifier and answer type.

If either none or a *full cone* NAT device has been detected, everything is ok. Otherwise, *Universal Plug and Play* [50] (or *UPnP*) is used to attempt to open a port on the NAT device matching the external address. If the attempt succeeds, the local NAT type becomes *full cone*, otherwise a warning message is shown.

Symmetric NAT is currently not supported since peer identifiers are computed by computing the hash value of the external address and port. In the future, another approach has to be taken to create an identifier for such peers. We also strive to support automatic relaying over peers in the local network if they do have a better connection, e.g. manually configured within the router. The implementation is however not finished at the time of writing.

The steps are repeated until the NAT type has been determined. If all strategies fail, the user can still manually open a port within the NAT device's configuration and it will be detected by *StreamTorrent*. Once the NAT type has been determined, ingoing and outgoing packets are observed and, if necessary, some of the steps repeated to ensure that the network configuration has not been changed. The local NAT type is added to the packet header to inform other peers. They will then take extra precautions, e.g. when providing recommendations.

Moreover, NAT devices also complicate the communication inside a local network. It is usually not possible to address a peer by its external address. To discover each other within a local network, peers join the same multicast group and periodically send notifications, e.g. containing the external address. The network layer of *StreamTorrent* uses this notifications to transparently map local and external address. The external address is also added to local packets (in case that multicast is disabled).

7.3.4 Build System

StreamTorrent uses its own, small build system. There are builds for both plugins and the player itself. Builds are stored within *build repositories*. The repository allows to more easily publish and to find new builds. Concepts of *Mozilla* and *Eclipse* have been adopted to manage versioning. There are four types of builds, i.e. *continuous*, *integration*, *milestone*, and *release* builds. While *release* builds should be the most stable ones, *continuous* builds are created on a daily basis. Users can specify within the preferences which kind of builds they are interested in. Moreover, it is possible to create builds for each platform and language independently, if needed.

7.3.5 Recordings

There is an initial implementation to record streams. It is used in two ways, for regular recordings and for *time-shifted viewing*. Regular recordings are stored on the local hard disk and can be watched later at any time. *Time-shifted viewing* allows to pause playback and to later resume playback at the same position, packets are stored temporarily until they get delivered.

Recordings are based on a simple, proprietary file format that cannot be read by other players. Future extensions might adopt the standardized *MPEG* file format. We also strive to support programmed recordings in the future.

7.4 UI Framework

Implementing a user interface is usually time-consuming and requires a large amount of code. Consistency within the entire application and similar design and behavior as popular applications are essential to guarantee a good user experience. This includes many smaller details like tooltips, drag & drop, alerts for important messages, default actions like copy and paste, and drop-down menus.

To simplify its design, the player user interface is composed of two layers. a framework[1] providing the most basic features and the player-specific components. Looking at web application, a field where Java has become popular in recent years, many frameworks emerged simplifying the development. By contrast, Java was considered to be unsuited for desktop applications by many developers for years (for example [51]) due to its memory footprint, lack of responsiveness, problems with native look & feel, among others. There has been a lot of progress, but it still lacks the diversity of available frameworks that simplify development.

The most popular framework is the *Eclipse Rich Client Platform* [52] that evolved from the *Eclipse* development environment. It incorporates many useful features, especially its flexible plugin architecture, but it imposes a high overhead for smaller application, such as the *StreamTorrent Player* itself, and frequent tasks are sometimes still to time-consuming. Eclipse uses the *Standard Widget Library*[2] to implement its user

[1] Usually referred to as a *rich client platform* within Java community

[2] In contrast to its name, it is a non-standard solution (or *SWT*). *Swing* is the standard library distributed with every Java runtime environment.

interface. *StreamTorrent Player* also uses *SWT* because of its small memory footprint, native look-and-feel, responsiveness, and integrated browser,

The *NetBeans Platform* [53] and *Spring Rich Client Project* are two other, Swing-based frameworks slowly emerging as alternatives for *Eclipse RCP*. Another approach is taken by the *XML user interface language* [54] (or *XUL*). *XUL* is basically a combination of XML to describe a user interface and a scripting language to implement the behavior. It has been introduced by the *Mozilla Foundation* and is used among others by the popular browser *Firefox*. It has also been ported to Java. Unfortuneately, scripting languages tend to become unreadable and hard to debug when a project grows.

To keep the *StreamTorrent Player* small and to avoid any overhead, it uses an own, very small framework. It is about 200 KB in size and focuses on providing the most important features with the least amount of code. It is based on the *Platform Core* layer and makes heavy use of the *adapter registry* and the unified event handling. The following subsections give a brief overview.

7.4.1 Selection Handling

Selection is an important aspect within any user interface. Determining the currently selected object is vital for many parts of the user interface such as menu items, tooltips, and drag & drop. The concept itself is important, the implementation is straightforward. Objects providing selection implement *ISelectionProvider*. The event framework of *Platform Core* has been adopted to fire events upon changed selection.

7.4.2 Actions

Actions represent commands that are associated with menu items, toolbar items, drop down menus, etc. User trigger actions by clicking on these user interface elements. Each action obtains its *action site* during initialization to enable the action to modify its behavior and appearance within the user interface, for example, by changing its text and image or by disabling itself. Development is simplified by providing various convenient methods, e.g. regular, disabled, and hover images are found automatically by specifying a common key and following some simple naming conventions. Actions are managed by an *action repository*. Actions can be associated with *objects* to automatically create drop down menus.

7.4.3 Drag & Drop

Providing drag & drop is usually time-consuming to implement. The implementation has to consider both drag & drop within the application and with the operation system and other applications. In the second case, dragged objects have to be transformed into an appropriate format. Within *StreamTorrent*, drag & drop is handled automatically once actions like cut, copy, and paste are implemented.

7.4.4 Viewers

Viewers simplify the use of widgets like tables, tree, combo boxes, and lists by providing a flexible default implementations hiding most of the details. A *content provider* specifies which objects to display and a *label provider* specifies their appearance (font, text, images, etc.). Instantiating a viewer with a content and a label providers yields the corresponding view widget, for example, a table for the *table* viewer or a list for the *list* viewer. Selection, drag & drop, tooltips, and drop down menus are managed automatically once corresponding adapters are registered. A default content provider exploits the event framework to detect updates, which reduces the task to merely implementing a label provider.

There are a few more advanced content and label providers. The *filter* content provider wraps another content providers and only displays objects that satisfy a given predicate. The *compositional* content provider takes several content providers and builds trees. Each content provider represents a root and its items are the children. There is also a corresponding *compositional* label providers. They are used, for example, by the overview sidebar of the *StreamTorrent Player* to show different kinds of objects within a single sidebar to give an overview over the system. The *aggregating* content provider takes a content provider and an aggregation function as input, evaluates the function for each item, and puts items with the same function value under a common root. The *aggregating* content provider is used by the history sidebar to group history items by date. *Comparators* can be used to sort items.

7.4.5 Tooltips

Unfortunately, native tooltips provided by *SWT* are rather limited. For example, neither text formatting nor custom tooltips for table items are supported. *StreamTorrent* uses an alternative, non-native approach based on the *Browser* widget. Any object can contribute a tooltip by either implementing or providing an adapter for *IToolTipSupport*. *IToolTipSupport* has methods to generate the required *HTML* code.

7.4.6 Updates

A simple user interface is provided to search for updates of both the main application itself and installed plugins. The integrated browser has been extended to enable the user to just click on plugins on a web-site to trigger their installation, giving a convenient way to distribute and install new plugins.

7.5 Player UI

The *Player UI* layer provides a user interface to the features of the *Player Core* layer based on *Platform UI* . In its use it is similar to popular *BitTorrent* clients and browsers. There is an integrated browser to access web-based repositories and an integrated player to playback streams.

7.5.1 Browser

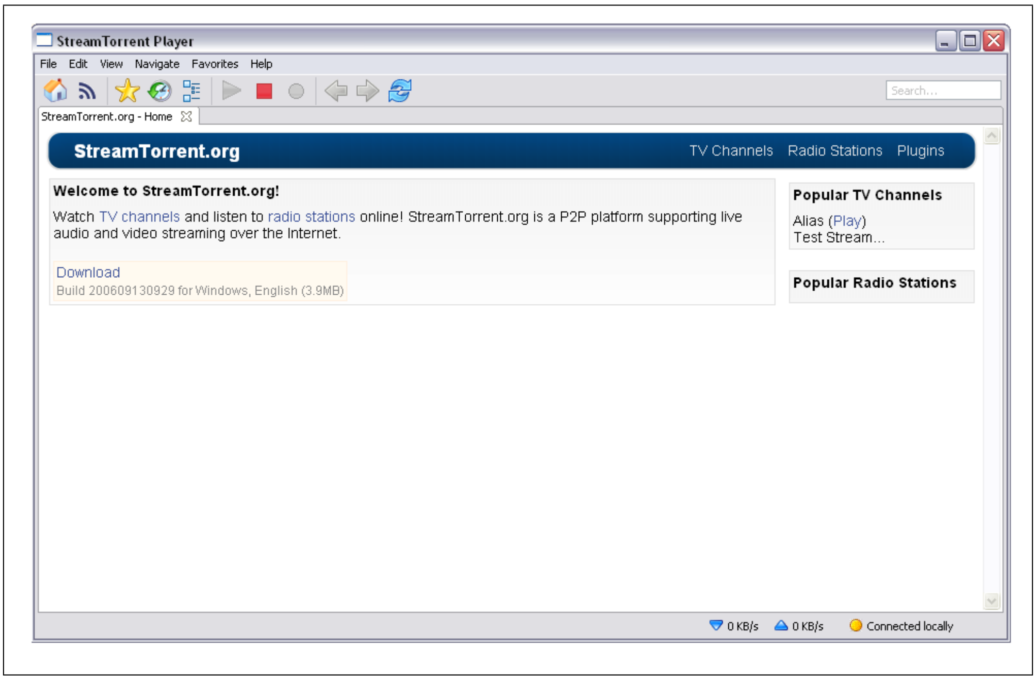
The integrated browser is shown right after the start and uses the main repository as the welcome page. The browser of the underlying operating system is used for this purpose. The behavior is slightly modified to enable users to directly click on stream descriptors and plugins to trigger playback, respectively the installation. A screenshot is given in Figure 7.3.

7.5.2 Player Integration

Third-party media players can be integrated into the *StreamTorrent Player* to playback streams. Unfortunately, media players are inherently platform-dependent and differ in the number of supported features. This is further complicated by the fact that the integration into Java differs for each player and operation system.

Two popular open source video players are the *VLC player* [55] and the *MPlayer* [56]. Both *MPlayer* and *VLC* have the same licensing terms, i.e. GPL[57]. This prohibits the distribution with our own player, unless it is also GPL licensed. It can, however, be included as an optional, GPL-licensed plug-in.

The Java integration of *MPlayer* is simple. *MPlayer* is started in its own process and the video output is redirected to the Java window, addressed by a *window handle.*



Welcome to StreamTorrent.org!

Watch TV channels and listen to radio stations online! StreamTorrent.org is a P2P platform supporting live audio and video streaming over the Internet.

Download
Build 200609130929 for Windows, English (3.9MB)

Popular TV Channels

Alias (Play)
Test Stream...

Popular Radio Stations

Figure 7.3: Screenshot: *StreamTorrent Player* showing a browser after startup.

SWT allows access to *window handles* by simply invoking a method.[3] *MPlayer* is then remotely controlled via its console using the process input and output streams. Unfortuneately, streaming support is not as mature as in other players.

VLC features excellent streaming support, but the Java integration is more complicated. On *Windows* platforms, *OLE automation* [58] can be used. [4] *JVLC* [59] provides an alternative approach based native libraries and the *Java Native Interface* (or *JNI*). By bundling the native library with *StreamTorrent*, no further *VLC* installation is required.[5] Regrettably, *JVLC* is still in an early beta-stage at the time of writing.

Similar to *VLC*, *windows media player* can be controlled using *OLE automation*. However, current versions lack the most codecs unless they are installed manually or only Microsoft's codecs are used. And finally, *QuickTime* is available for *Windows* and *Mac*, supports streaming and *H.264*, and features a Java library.

The Windows version of *StreamTorrent* currently uses *VLC* (see Figure 7.4. Implementations for other players and operation systems will follow in the future.

7.5.3 Broadcast Dialog

The broadcast dialog is used to create and publish new streams (see Figure 7.5). Properties like the bitrate and the content type are determined automatically by observing the input stream. Streams can be published to any kind of repository. Once a stream is stopped, it is removed from the repository as well.

[3] Integrating *MPlayer* into a *Swing* application seems considerably harder.

[4] A minor disadvantage is that the user has to explicitly enable *OLE* support during the *VLC* installation.

[5] Given that the licensing issues are resolved

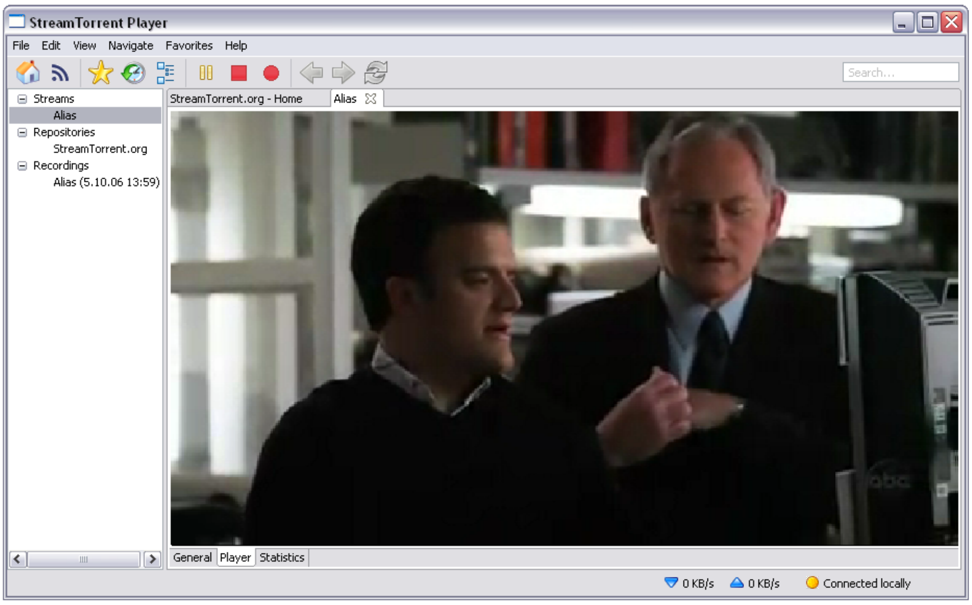
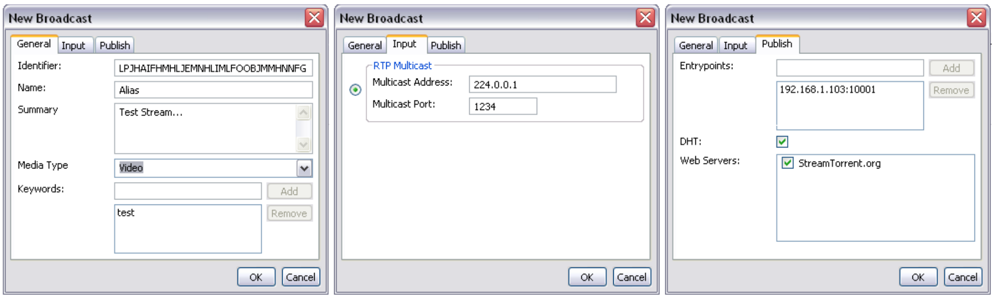


Figure 7.4: Screenshot: *StreamTorrent Player* with *VLC* for playback.



Identifier: LPJHAIFHMHLJEMNHLIMLFOOBJMMHNNFG
Name: Alias
Summary: Test Stream...
Media Type: Video
Keywords: test

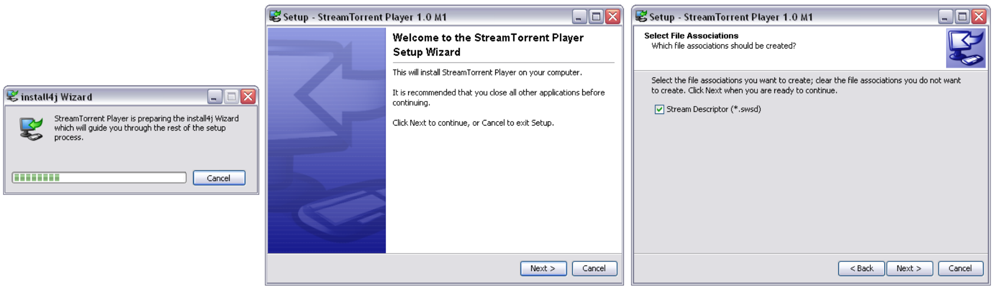
RTP Multicast
Multicast Address: 224.0.0.1
Multicast Port: 1234

Entrypoints: 192.168.1.103:10001
DHT: ☑
Web Servers: ☑ StreamTorrent.org

Figure 7.5: Screenshot: Dialog to broadcast new streams.

7.5.4 Installation

Installation files are generated using *Install4J* [60] and *Exe4J* [61]. They provide all the desired features like file associations, licence dialog, menu icons, and automatic installation of the Java Runtime Environment. Some screenshots are given in Figure 7.6.


install4j Wizard
Cancel
Setup - StreamTorrent Player 1.0 M1
Next >
Cancel
Setup - StreamTorrent Player 1.0 M1
< Back
Next >
Cancel

StreamTorrent Player is preparing the install4j Wizard which will guide you through the rest of the setup process.

Welcome to the StreamTorrent Player Setup Wizard

This will install StreamTorrent Player on your computer.

It is recommended that you close all other applications before continuing.

Click Next to continue, or Cancel to exit Setup.

Select File Associations
Which file associations should be created?

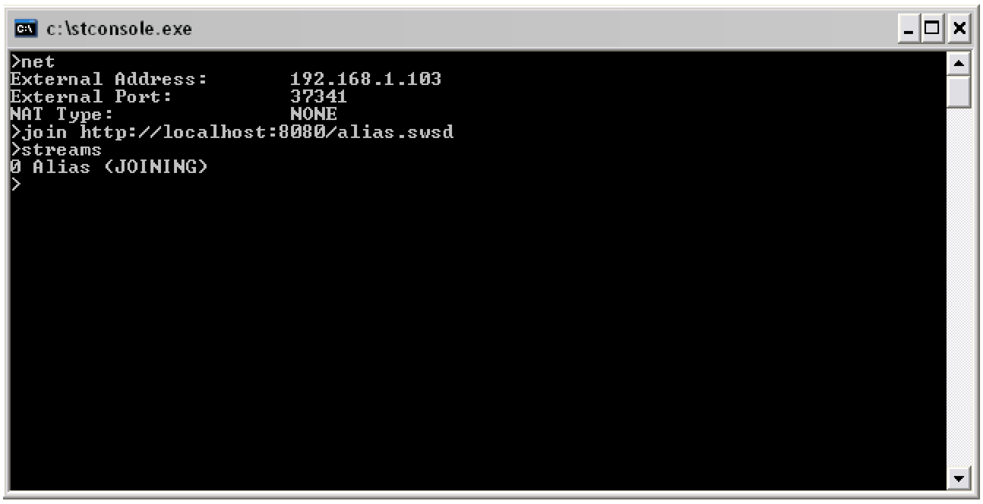
Select the file associations you want to create; clear the file associations you do not want to create. Click Next when you are ready to continue.

☑ Stream Descriptor (*.swsd)

Figure 7.6: Screenshot: Setup dialogs.

7.6 Player Console

Instead of using the *Player UI* layer, the player can also run within a console using the *Player Console*. The console provides almost the same functionality. It is possible to broadcast and join streams. Instead of local playback within an integrated player, streams can be broadcast into the local network. Presumably the console will be extended by a remote control feature using a UI-based player as guest. Currently. it is used to perform tests on the *PlanetLab* [62] testbed. A screenshot is given in Figure 7.7.



```
>net
External Address:       192.168.1.103
External Port:          37341
NAT Type:               NONE
>join http://localhost:8080/alias.swsd
>streams
0 Alias (JOINING)
>
```

Figure 7.7: Screenshot: *Player Console.*

7.7 Planet Lab

Initial tests have been performed in the local network for several weeks to determine whether *StreamTorrent* is working properly. . The tests worked flawlessly, mainly because of the number of tests that have been performed with the simulator. We also deployed *StreamTorrent* on 150 *PlanetLab* nodes. We chose a random set of nodes and it worked on most of them. The ones that did not work seem to be overloaded and experienced unusual high delays. More tests will be performed in the future to confirm the simulation results and to further optimize the protocol.

Chapter 8

Conclusions

Given the growing number of radio stations and TV channels available online, peer-to-peer live streaming has the potential to overcome the limitations of traditional, centralized approaches towards streaming and enables content providers to both increase playback quality and to reduce costs. Especially for smaller organizations, not having the resources to afford traditional servers or to get a permission for the regular cable network, peer-to-peer streaming is a viable alternative.

StreamTorrent is a peer-to-peer live streaming protocol combining pull-based and push-based techniques to achieve both efficiency and robustness. The chosen overlay is locality-aware and incentive-compatible, has a guaranteed logarithmic diameter, and enables the source to push new packets to speed up packet distribution. Having a push mechanism allowed to reduce the notification frequency, which led, together with packet bundling, to a smaller overhead.

Simulations have shown that *StreamTorrent* scales well with the number of peers. For example, having 10,000 instead of 1,000 peers incurs an additional delay of less than 200 ms. The protocol is also robust against packet loss and churn. 75% of the peers can leave the network simultaneously without underflows at the remaining peers. The communication overhead is between 3% and 6%, a remarkably good value once duplicate ratios, coding overhead, and churn are accounted for in other protocols.

The *StreamTorrent Player*, a peer-to-peer media player supporting live audio and video streaming, shows that the *StreamTorrent* protocol can be used in the real world. Besides enabling users to broadcast own streams, the user interface provides many small features that improve the user experience, like an integrated player for playback, automatic update support, a plugin infrastructure, NAT detection, favorites, a history, recording support, time-shifted viewing, a browser to find streams, among many others.

There are still interesting directions for future work in peer-to-peer. streaming. The subsequent chapter gives a brief overview.

Chapter 9

Future Work

While most parts of the protocol have been implemented and the *StreamTorrent Player* is already working well, much more work can be done. The following sections outline possible future work.

9.1 Protocol

9.1.1 On-demand Streaming

StreamTorrent in its current form is a multicast protocol supporting live streaming. Some of the main application scenarios are broadcasting radio stations, TV channels, and live events like soccer matches. It would be interesting to further extend *StreamTorrent* to support *on-demand* streaming. This would enable users to select and immediately playback movies or audio tracks without having to download them first. There would be no need to store hundreds of gigabytes of movies on local hard disks, the system itself could ensure that files are sufficiently replicated.

9.1.2 Incentives

More work has to be done to provide incentives for peers to share their upload bandwidth. While the basic mechanisms have been implemented, their parameters have to be better adjusted to the current situation to achieve an optimal efficiency. *Source* and *network* coding could increase chances that neighbours are able to exchange data.

9.1.3 Overhead

The communication overhead in *StreamTorrent* has been reduced by the adoption of pushing, the reduced notification frequency, and the bundling of packets. For smaller packets the UDP and IP headers impose a significant overhead. Especially at higher bitrates, it might be possible to better exploit packet bundling to attach most additional information to the actual data packets.

9.1.4 Pushing

The adopted push mechanism reduces the communication overhead and speeds up packet distribution. The *Evaluation* chapter has shown that *StreamTorrent* scales well with the number of peers. However, the fraction of push packets decreases with the number of peers and more requests have to be sent. Maybe it could proof useful to have a second push mechanism to distribute packets within a peer's neighbourhood, not triggered by the source. *GridMedia* [18] synchronizes peers to partition time into intervals. Rules are then applied to specify which peers can push packets in which intervals. Another solution might partition the overlay into two or more layers. For example, the overlay of *eQuus* [22], a locality-aware distributed hash table, has two layers. The first layer arranges all peers into disjoint cliques of a few dozen peers. Nearby peer typically join the same clique. A second layer connects all cliques among each other by assigning an identifier to each clique and building a hypercube. The regular *StreamTorrent* protocol could be used to distribute and exchange data among cliques, and a more efficient, specialized one could be adopted within cliques. This could lead to a protocol scaling 10 to 100 times better with a smaller overhead while maintaining robustness and small delays.

9.1.5 Locality

Recommendations and certificates give a hint about the rank of a peer sending a *join* message. The protocol lacks a similar mechanism for locality. Peers can only assume that recommended peers are in the local neighbourhood because peers frequently drop distant neighbours. Maybe a protocol like *Vivaldi*, assigning virtual coordinates to peers, or mapping IP addresses to countries could further improve locality-awareness.

9.1.6 Bandwidth Management

The maximum upload and download bandwidths are manually configured. In real world applications it would be convenient if these bandwidths are adapted depending on the experienced packet loss. The current protocols already ensures that the maximum bandwidths are not exceeded, extending this functionality should not be difficult.

9.1.7 Source Replication

The source is a single point-of-failure within the overlay and should be replicated. Replicas have to agree on the payload, sequence numbering, and timestamping. Moreover, the task of pushing packets has to be partitioned among the replicas. If the source's input is an external stream, it might also be necessary to replicate the input.

9.1.8 Data Integrity

Before deploying *StreamTorrent* in the real world, a mechanism is needed to maintain the integrity of packets. Malicious peers should not be able to inject, drop, or modify packets. Public key cryptography, the simplest solution to this problem, is too inefficient in terms of both computational and communication complexity.

9.1.9 Different Quality Levels

It might be beneficial to offer streams at different bitrates to enable weaker peers to join streams as well. Peers can then automatically switch between the levels. An implementation would be mostly straightforward since multiplexing is already implemented. Peers simply need to join the new level to fill buffers and gradually leave the old one. However, it is essential to have a good incentive mechanism and to carefully select the number of levels and their respective bitrates.

9.2 StreamTorrent Player

9.2.1 NAT Support

There are several ways to improve the current NAT implementation to simplify the setup and improve connectivity. Hole-punching would allow connections between peers behind *restricted cone* NAT, the most typical device type. Relaying could help in cases where peers are unable to connect, e.g. because of firewalls, but another connected peer is available in the local network. *Symmetric* NAT is currently not supported at all since peers need a unique identifier, created by hashing the external address and port (which differ depending on the destination for peers behind *symmetric* NAT devices).

9.2.2 Recording

Program schedules of streams would allow, for example, programmed recordings of favorite shows and more detailed information about currently played streams. Moreover, it would be convenient if recordings are stored as standard *MPEG* files instead of *StreamTorrent*'s proprietary format.

9.2.3 Video Snapshots

Sources should periodically generate snapshots of video streams. These snapshots could be shown in several places, for example:

- On the associated web pages.
- As image for favorite video channels.
- When opening a stream descriptor.

9.2.4 Finalizing the API

The API and extension points should be finalized to allow contributions by others. Work in progress can be moved into internal packages.

9.2.5 User Interface

There are many smaller improvements for the user interface. It would be nice to have better icons. Tooltip and drag & drop support is not completed, for example, it is not possible to drag favorite menu items to the desktop. Alerts, shown at the bottom right on the screen, could show important messages, e.g. NAT errors. There should be periodical checks for updates. Moreover, it would be convenient to bundle a third-party player with the *StreamTorrent Player* or to support more players besides *VLC*.

9.2.6 Supported Platforms

At the moment, there is only a Windows version of the *StreamTorrent Player* because the player integration is platform-dependent. Other platforms should follow soon.

9.2.7 Supported Input Types

RTP streams are used as input to create *StreamTorrent* streams. This requires the user to not only setup *StreamTorrent*, but also an external streaming server. Especially setting up codecs and bitrates takes time and is error-prone due to codec incompatibilities. There should be a more convenient way to broadcast streams, for example, by just providing a set of files.

9.3 Outlook

More *PlanetLab* [62] tests have to be performed to verify the simulation results. We also aim at streaming the upcoming *International Workshop on Peer-to-Peer Systems* to test *StreamTorrent* in the real world.

Bibliography

[1] Vinay Pai, Karthik Tamilmani, Vinay Sambamurthy, Kapil Kumar, and Alexander Mohr. Chainsaw: Eliminating trees from overlay multicast. In *4th International Workshop on Peer-To-Peer Systems (IPTPS)*, Ithaca, New York, USA, February 2005.

[2] John Jannotti, David K. Gifford, Kirk L. Johnson, M. Frans Kaashoek, and James W. O'Toole, Jr. Overcast: Reliable multicasting with an overlay network.

[3] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. Splitstream: High-bandwidth content distribution in a cooperative environment. in proceedings of (iptps'03) (february 2003), 2003.

[4] Dejan Kosti, Adolfo Rodriguez, Jeannie Albrecht, and Amin Vahdat. Bullet: high bandwidth data dissemination using an overlay mesh. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 282–297, New York, NY, USA, 2003. ACM Press.

[5] Multicast Vidhyashankar Venkataraman. Chunkyspread: Multi-tree unstructured peer-to-peer.

[6] Wei Tsang Ooi. Dagster: contributor-aware end-host multicast for media streaming in heterogeneous environment. In S. Chandra and N. Venkatasubramanian, editors, *Multimedia Computing and Networking 2005. Edited by Chandra, Surendar; Venkatasubramanian, Nalini. Proceedings of the SPIE, Volume 5680, pp. 77-90 (2004).*, pages 77–90, December 2004.

[7] Jin Liang and Klara Nahrstedt. Dagstream: locality aware and failure resilient peer-to-peer streaming. volume 6071. SPIE, 2006.

[8] Xiaofei Liao, Hai Jin, Yunhao Liu, Lionel M. Ni, and Dafu Deng. Anysee: Peer-to-peer live streaming. In *Proc. of INFOCOM*, April 2006.

[9] Freecast. http://www.freecast.org.

[10] Maya Dobuzhskaya, Rose Liu, Jim Roewe, and Nidhi Sharma. Zebra: Peer to peer multicast for live streaming video, 2004.

[11] Bittorrent protocol specification v1.0, July 2006.

[12] Vivek K. Goyal. Multiple description coding: Compression meets the network. *IEEE Signal Processing Magazine*, 18(5):74–93, September 2001.

[13] Venkata N. Padmanabhan and Kunwadee Sripanidkulchai. The case for cooperative networking. In *Peer-to-Peer Systems: First International Workshop, IPTPS 2002*, pages 178–190, Cambridge, MA, USA, 2002.

[14] John W. Byers, Michael Luby, Michael Mitzenmacher, and Ashutosh Rege. A digital fountain approach to reliable distribution of bulk data. *SIGCOMM Comput. Commun. Rev.*, 28(4):56–67, October 1998.

[15] Michael G. Luby, Michael Mitzenmacher, M. Amin Shokrollahi, Daniel A. Spielman, and Volker Stemann. Practical loss-resilient codes. pages 150–159, 1997.

[16] Michael Luby. Lt codes. In *The 43rd Annual IEEE Symposium on Foundations of Computer Science*, pages 271–282, 2002.

[17] *CoolStreaming/DONet: a data-driven overlay network for peer-to-peer live media streaming*, volume 3, 2005.

[18] *Gridmedia: A Multi-Sender Based Peer-to-Peer Multicast System for Video Streaming*, 2005.

[19] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

[20] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 53–65, London, UK, 2002. Springer-Verlag.

[21] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329–??, 2001.

[22] Thomas Locher, Stefan Schmid, and Roger Wattenhofer. eQuus: A Provably Robust and Locality-Aware Peer-to-Peer System. In *6th IEEE International Conference on Peer-to-Peer Computing (P2P), Cambridge, United Kingdom*, September 2006.

[23] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi: a decentralized network coordinate system. *SIGCOMM Comput. Commun. Rev.*, 34(4):15–26, October 2004.

[24] Eytan Adar and Bernardo A. Huberman. Free riding on gnutella, 2000.

[25] Kazaa: Peer-to-peer file sharing. `http://www.kazaa.com`.

[26] Kevin Lai, Michal Feldman, Ion Stoica, and John. Chuang. Incentives for cooperation in peer-to-peer networks, 2003.

[27] John R. Douceur. The sybil attack. In *1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, February 2003.

[28] Michal Feldman, Kevin Lai, I. Stoica, and John Chuang. Robust incentive techniques for peer-to-peer networks, 2004.

[29] Sepandar D. Kamvar, Mario T. Schlosser, and Hector Garcia-Molina. The eigentrust algorithm for reputation management in p2p networks. in proceedings of the twelfth international world wide web conference, 2003.

[30] Vivek Vishnumurthy, Sangeeth Chandrakumar, and Emin G. Sirer. Karma : A secure economic framework for p2p resource sharing. In *1st Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, June.

[31] Karthik Tamilmani, Vinay Pai, and Alexander E. Mohr. Swift: A system with incentives for trading. In *2nd Workshop on Economics of Peer-to-Peer Systems*, 2004.

[32] Bittorrent: Fast extension. `http://bittorrent.org/fast_extensions.html`.

[33] Tsuen-Wan Ngan, Peter Druschel, and Dan S. Wallach. Incentives-compatible peer-to-peer multicast. In *2nd Workshop on Economics of Peer-to-Peer Systems*, June 2004.

[34] Richard Yang, Min Sik Kim, Xincheng Zhang, and Simon S. Lam. Two problems of tcp aimd congestion control. Technical Report TR-00-13, Department of Computer Sciences, University of Texas at Austin, June 2000.

[35] Jon. Kleinberg. Small-world phenomena and the dynamics of information, 2001.

[36] Irving S. Reed and Gustave Solomon. Polynomial codes over certain finite fields. 8(2):300–304, June 1960.

[37] Philip Chou, Yunnan Wu, and Kamal Jain. Practical network coding, 2003.

[38] JUnit, Testing Resources for Extreme Programming. `http://www.junit.org`.

[39] ANT - Java Build Tool. `http://ant.apache.org`.

[40] JProfiler: Java Profiler. `http://www.ej-technologies.com/products/jprofiler/overview.html`.

[41] YourKit Java Profiler. `http://www.yourkit.com`.

[42] Spamato Spam Filter System. `http://www.spamato.net`.

[43] JavaScript 1.5 Specification. `http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Reference`.

[44] Eclipse: Java Development Environment. `http://www.eclipse.org`.

[45] RTP: A Transport Protocol for Real-Time Applications (RFC 3550), 2003. `http://tools.ietf.org/html/rfc3550`.

[46] IETF: The Internet Engineering Task Force. `http://www.ietf.org`.

[47] H.264/MPEG-4 AVC. `http://en.wikipedia.org/wiki/H.264`.

[48] XviD Codec. `http://www.xvid.org`.

[49] WMV: Windows Media Video. `http://www.microsoft.com/windowsmedia`.

[50] UPnP: Universal Plug and Play. `http://www.upnp.org`.

[51] JavaScript 1.5 Specification. `http://sunsite.uakom.sk/sunworldonline/swol-08-1998/swol-08-torvalds.html`.

[52] Eclipse: Rich Client Platform. `http://www.eclipse.org/rcp/`.

[53] The NetBeans (Rich Client) Platform. `http://www.netbeans.org/products/platform/`.

[54] XML User Interface Language (XUL). `http://www.mozilla.org/projects/xul`.

[55] VLC media player. `http://www.videolan.org`.

[56] MPlayer - The Movie Player. `http://www.mplayerhq.hu`.

[57] GNU General Public License (GPL). `http://www.gnu.org/copyleft/gpl.html`.

[58] Object Linking and Embedding (OLE). `http://en.wikipedia.org/wiki/ActiveX`.

[59] JVLC - Java VLC Binding. `https://trac.videolan.org/jvlc`.

[60] install4j - Multi-Platform Java Installer Builder. `http://www.ej-technologies.com/products/install4j/overview.html`.

[61] exe4j: Java Exe Maker. `http://www.ej-technologies.com/products/exe4j/overview.html`.

[62] PlanetLab: An open platform for developing, deploying, and accessing planetary-scale services. `http://www.planet-lab.org`.

[63] RFC 3489: STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs). `http://www.faqs.org/rfcs/rfc3489.html`.

Appendix

NAT Types

There are four types of NAT devices as proposed by STUN [63]:

- *Full cone* NAT maps internal addresses and ports one-to-one to external addresses and ports. Every external host can sent packet to internal hosts using the external addresses and ports.
- *Restricted cone* NAT is similar to *full cone* NAT, but an external host can only send to an internal host if the internal host has previously sent a packet to the external host.
- *Port restricted cone* NAT is slightly more restrictive than *restricted cone* NAT, i.e. external hosts cannot reply with a local port that never received a packet from the internal destination.
- *Symmetric* NAT uses a different mapping for each destination. Different destinations observe different external address port pairs and only the destination can reply.