Improving Peer-to-Peer Video Streaming

Improving Peer-to-Peer Video Streaming

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. ir. K. C. A. M. Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op vrijdag 8 april 2016 om 15:00 uur

door

Riccardo Petrocco

ingenieur in de technische informatica
geboren te Rome, Italy.

The work described in this thesis was supported by the Future and Emerging Technologies program FP7-COSI-ICT of the European Commission through the P2P-Next project (grant no.:216217)

Printed by: CPI-Koninklijke Wöhrmann – Zutphen

Front & Back: Alessandro Petrocco

ISBN 978-94-6328-031-0

An electronic version of this dissertation is available at
`http://repository.tudelft.nl/`.

To my mother, for her love and encouragement.
To my father, and his everlasting support.

Riccardo

Acknowledgements

The work presented in this thesis would not be possible without the help, support,collaboration, and friendship of all the people that have been part of my PhD track.

Dick, thank you for all your help, your support, your guidance, and first of all for believing in me. I admire your infinite patience when reviewing my work, your calm personality, and your humor. I'm very grateful for the freedom you gave me. You never imposed your ideas, or tried to push me in another direction, even when I kept running in circles and it seemed like I would never achieve my goal. Thank you for teaching me how to write scientific papers, and the clarity and quality of the work in this thesis is only thanks to your guidance.

Johan, thank you for your great energy, enthusiasm, and passion. Thank you for the long discussions and for always keeping an open mind. I always admired how, even when we wouldn't agree, you kept a smile and never took anything personal. Also, thank you for putting together such a nice group of people, the best colleagues I ever had. All the best for you future and in keeping the research at PDF active. I'm sure you are already looking into the upcoming hot topics.

A special thank goes to Arno. Thank you for your support and for being such a good friend over the years. I truly enjoyed working with you, travelling to conferences and working as a team on projects such as P2P-Next. I wish you all the best for your new family, and I really hope that our working lives will meet again.

To my colleagues, Adele, Boxun, Boudewijn, Dimitra, Elric, Mihai, Niels, Nitin, Lucia, Rahim, and Rameez, thank you for being such great friends. It has been a real pleasure to share this long journey with all of you. I will always keep incredible memories of working at TUDelft, the movie and gaming nights, all the lunches and dinners, the travels, and generally the nice atmosphere that made coming to work such a pleasure. I wanted to thank each and every one of you separately, but then this section would become longer than the introduction! I know that you are all facing great new adventures, and I'm sure our paths will meet again.

Dave, Naza, and Tamás, thank you for all the long talks, and for giving me a different view on what it means to do research.

Alex, Andreas, Bogdan, Henk, and Otto, even through we never worked together, I really valued all of our chats and it has been great to share the office with you.

To all of my friends outside the PDS group, a huge thanks for everything. You are too many to be included in this acknowledgement, and you know that you are the ones that made it extremely difficult to leave The Netherlands, you made my life richer.

Finally, my very best thanks go to my family and to partner, Victoria. You made all of this possible, supporting me in my decisions, and being the stable point in a chaotic journey.

Contents

Acknowledgements **vii**

1 Introduction **1**
- 1.1 Background . . . 2
- 1.2 Research Context . . . 8
- 1.3 Problem Statement . . . 11
- 1.4 Contributions and Thesis Outline . . . 12
- 1.5 Design and Research Methodology . . . 14

2 A Framework for Distributing Scalable Content over Peer-to-Peer Networks **17**
- 2.1 Related Work . . . 18
- 2.2 NextShare Integration . . . 19
- 2.3 Producer-Side Architecture . . . 21
- 2.4 Consumer-side Architecture . . . 24
- 2.5 Evaluation. . . . 29
- 2.6 Conclusions and Future Work . . . 35

3 Deftpack: A Robust Piece-Picking Algorithm for Scalable Video Coding in P2P Systems **37**
- 3.1 Problem Statement . . . 38
- 3.2 Related Work . . . 39
- 3.3 Deftpack. . . . 41
- 3.4 Experimental Setup. . . . 45
- 3.5 Experimental Results . . . 47
- 3.6 Conclusion . . . 51

4 The Peer-to-Peer Streaming Peer Protocol (PPSPP) **53**
- 4.1 Overall Operation . . . 55
- 4.2 Messages . . . 56
- 4.3 Chunk Addressing Schemes . . . 64
- 4.4 Content Integrity Protection . . . 67
- 4.5 Live Streaming . . . 73
- 4.6 Extensibility . . . 78

5 Libswift: the PPSPP Reference Implementation **79**
- 5.1 Bins and Binmaps for Data Availability . . . 80
- 5.2 Libswift's State Machine . . . 84
- 5.3 Pull Mechanism. . . . 87
- 5.4 Experiments. . . . 90
- 5.5 Conclusions and Future Work . . . 95

6 Performance Analysis of the Libswift P2P Streaming Protocol 97
6.1 Libswift 98
6.2 Download Scheduling 101
6.3 Experimental set-up 105
6.4 Experimental results 109
6.5 Conclusions and Future Work 114

7 Hiding User Content Interest while Preserving P2P Performance 117
7.1 Related Work 118
7.2 System Design 119
7.3 Security Analysis 124
7.4 Experiments. 127
7.5 Conclusion 131

8 Conclusion 133
8.1 Conclusions 133
8.2 Reflection on the State of P2P Streaming 134
8.3 Suggestions for Future Work 136

Bibliography 139

Summary 151

Samenvatting 153

Curriculum Vitæ 155

Publications of the author 157

1 Introduction

Since its debut in the late nineteenth century, the motion picture has generated enormous interest. Around the same period, the first mechanical devices able to scan and reproduce images appeared [21], but it was not until 1925 that the modern interpretation of television was invented [111]. From that moment onwards, television underwent everlasting improvement, both in quality and in delivery mechanisms, making it accessible to a continuously growing audience. In 1936, the BBC started broadcasting the first public television as we know it today, introducing ”the magic of television” [1] to hundreds of thousands of viewers. Television was the only means of delivering video content to peoples' homes until the arrival of the Internet in the late 20th century. While the first patent for streaming over electrical lines was already granted at the beginning of the century [77], it was not until the late 1990s that the technological advances allowed its deployment. The commercialization of the Internet, and the arrival of the first personal computers, enabled the transmission of video content via this novel medium for the first time. This encouraged the standardization of protocols and procedures, and the coining of new terms such as Video on Demand [94].

Nowadays, the majority of traffic over the Internet represents video content. Following a recent trend analysis [47], all forms of IP video together will constitute around the 80 to 90 percent of total IP traffic by 2018. The content delivery infrastructure that dominated Internet video distribution over the last two decades has been the *client-server* architecture [63]. In a client-server architecture, the viewer requests and retrieves the video directly from the content provider, which has direct control over the service it provides. While this has worked well in the past, as the content providers were the only ones that could afford the prohibitive costs of producing and distributing video content, this trend is slowly changing. The current generation is moving from being passive viewers, or content consumers, to becoming active content producers. This transition needs to be coupled with

[1]Mitchell's introductory words to the first public broadcast performed by the BBC [10].

a change to a content delivery infrastructure that allows users to distribute their content directly from their homes, without having to rely on a centralised authority. Peer-to-Peer (P2P) systems are fully distributed networks where its users are both content providers and content consumers. Their widespread adoption [47] has shown how they provide a valuable solution for distributing files to a wide audience, but they require several enhancements, from the original design, in order to qualify for the distribution of time-critical content such as video.

In this thesis, we improve P2P streaming systems, presenting the design of architectures and algorithms that address the distribution of time-critical content in a fully distributed infrastructure. We focus on three major challenges that affect P2P streaming systems. First, we address the diversity of users that access the same content with different devices and connections. We present the architectural description and evaluation of the first open-source P2P system capable of delivering scalable video content, where the quality of the video changes based on the availability of resources. Furthermore, we enhance this system by designing, implementing, and evaluating a new algorithm for retrieving such content. Secondly, we present a novel P2P streaming protocol and describe its behaviour and implementation. We analyse its reference implementation in challenging environments and compare it with existing solutions. Finally, we provide a solution for increasing the privacy of P2P users without affecting their performance. All the solutions presented in this thesis are accompanied by a thorough evaluation, using both simulations as well as real-world evaluations.

1.1. Background

Streaming is defined as sending data, usually multimedia content, allowing the receiver to start viewing it before it has been completely retrieved. Multimedia content can be any form of audio and video content. It can be distributed, or streamed, in both an on-demand and real-time fashion. On-demand streaming means that the content is already present at the source, and that the user can request it at any time. Real-time streaming, usually referred to as live streaming, refers to content that is created on the fly, and transmitted to the user with a small and consistent delay. The content is usually provided in several qualities in order to address different types of devices and distribution channels. It can also be adaptive, in which case the content quality changes depending on the available resources.

1.1.1. Video Streaming

Video streaming is the act of delivering video content from the source to the destination. The distribution of video streams involves three steps:

1. The transcoding of the original video content into a quality suitable to the distribution infrastructure and user device.

2. The distribution of the content from the provider to the end user.

3. The retrieval, decoding, and playback of the content on the user's device.

The video is usually provided at its source in RAW format. The RAW format provides the highest quality but has a very high bitrate (hence file size), and needs to be encoded into a different video format before it is transmitted to the viewer. The transcoding process is affected by both the content provider and the viewer, or content consumer. On the one hand, the content provider aims to reduce its distribution costs by compressing the RAW video as much as possible. On the other hand, the more the video is compressed, the more resources the user needs to invest in order to decode the stream upon reception. Furthermore, the high variety of devices that currently populate the Internet also highly affects the transcoding process. Currently, multimedia content is consumed by a wide variety of devices, from mobile devices to PCs and TV sets. Each of those devices has different capabilities in terms of resources, processing power and storage space, as well as playback capabilities and requirements, from tiny mobile phones to huge TV sets. All of those parameters have to be considered when choosing how to transcode the original content. After the content has been transcoded to a suitable quality, the major challenge lies in its distribution.

The second step is the distribution of the content from the provider to the consumer. The approach has not changed much over the last two decades, and the usual way of delivering content to the final user follows the Client-Server (CS) approach. In the CS approach, the content distribution can be seen as a one-to-one communication, where the content consumer directly requests and retrieves the content from the content provider. This traditional approach is hard to replace as it provides several advantages, such as direct control over which users are allowed to access the content, and guarantees for a higher predictability of the quality of service, defined as Quality-of-Experience (QoE) for the end-user. A variety of protocols have been designed that specifically address video distribution in a CS scenario. The most popular streaming protocols fall into the Real Time Transport Protocol (RTP) family [145], and are differentiated based on the layer of the protocol stack in which they lie. RPT lies in the transmission layer and usually uses UDP, the Real Time Control Protocol (RTCP) [145] lies at the session layer, while the Real Time Streaming Protocol (RTSP) [146] operates at the application layer. Recently, given the popularity of multimedia content distribution in the browser, HTTP has also been considered, and solutions have emerged such as Dynamic Streaming over HTTP (DASH) [106].

Finally, the content consumer retrieves the video stream and starts to reproduce it in a multimedia player at the same time. The QoE of the user, which is what the content provider aims to maximise, depends on several factors. First of all, the user needs to be able to reproduce the video stream given his resources. His network link capacity needs to be higher than the bitrate at which the video has been encoded, otherwise he will experience continuous stalls and a long start up time, defined as the time-till-playback. At the same time, he needs to have enough hardware resources to be able to retrieve the stream, decode it, and reproduce it, all at the same time. The second factor that influences the QoE is the quality of the video stream. It has been shown that a higher video quality, defined as a combination of stream bitrate and resolution, leads to users watching the video stream for a longer period

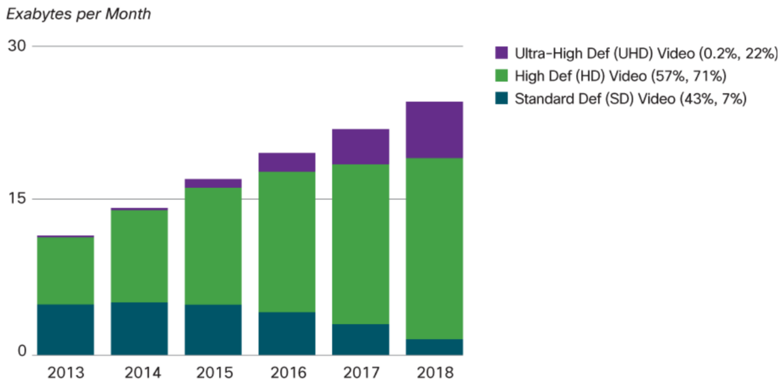


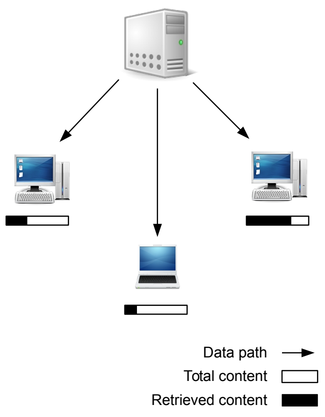
Figure 1.1: Cisco trend overview for the video traffic share of different qualities. The percentages next to the legend denote the traffic shares for the years 2013 and 2018, respectively. (Figure taken from Cisco VNI, 2014 [47]).

of time [47]. After the first two requirements are met, the QoE mostly depends on the start up time (the time required to start the playback), the playback continuity (the user should experience no stalls), and added features such as the possibility of seeking throughout the stream.

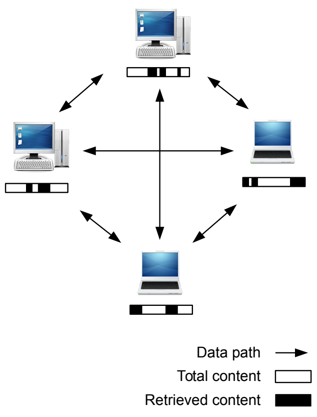
While it can be argued that the problems related to video streaming are temporary as the network capacities are constantly increasing, the video bitrates are increasing at the same rate. Figure 1.1 shows the trend predicted by Cisco [47], which clearly shows how video content is consumed at increasingly higher qualities, and how the consumption of Ultra-High Definition is expected to exceed the consumption of Standard Definition by 2017. This illustrates how encoding and delivery strategies need to constantly evolve in order to provide suitable solutions for the constantly increasing demand for higher quality multimedia content.

1.1.2. P2P streaming

P2P networks offer several advantages over traditional client-server architectures. In a P2P network, users collaborate in distributing the content by exchanging small parts of what they already retrieved with each other. In a P2P streaming network each user is a potential content provider, relying on the resources provided by the other participants in order to distribute its content. While it sounds as an ideal solution for content providers, it carries several challenges that are inherent to the technology, and a proper QoE, comparable to that of the CS architecture, cannot always be guaranteed. A solution is to use P2P-assisted infrastructures, in which the content consumers contribute to the content distribution, while relying on a CS infrastructure as a fallback. Many content providers have adopted P2P-assisted solutions in order to reduce their distribution costs while increasing the scalability of their systems [28–30, 32]. Before analysing the challenges of the P2P infrastructure applied to multimedia streaming, we first briefly describe P2P networks.



(a) The client-server infrastructure.



(b) The P2P infrastructure.

Figure 1.2: The differences in distribution patterns between a client-server infrastructure and a P2P infrastructure.

P2P networks

In P2P networks, users retrieve the content of interest by downloading from others and sharing their resources. The most popular P2P protocol, BitTorrent [5], has been specifically designed to efficiently distribute static content between a set of interested users, called *peers*. In order to do so, the content is divided into a number of pieces, called *chunks*, allowing the peers to download the content out of order. In the BitTorrent protocol, the group of peers interested in a specific content is called a *swarm*. Peers that have not retrieved all the chunks yet are called *leechers*, while peers that have retrieved the entire content and stay in the swarm to redistribute it to the leechers are called *seeders*.

A graphical representation of the differences between the client-server infrastructure and the P2P infrastructure is provided in Figure 1.2. In a client-server scenario, see Figure 1.2(a), users retrieve the content sequentially from a central server and are unaware of each other. In the P2P scenario, see Figure 1.2(b), users are connected to each other and exchange the chunks of content. Peers need to retrieve a metadata file, called *.torrent*, that contains a hash for each chunk, either from a centralised server, called *tracker*, or in a distributed fashion. The metadata file allows peers to verify the content they receive from others, avoiding the pollution of the system by malicious peers. P2P networks only work as long as their peers donate their resources by uploading what they have already retrieved to the other peers of the swarm. Hence, an incentive mechanism is required to avoid selfish behaviour. BitTorrent applies the tit-for-tat incentive mechanism (T4T) [75], by which peers are incentivised to upload to others, as they will have a higher chance of receiving the same treatment. Therefore, in BitTorrent, the best way of achieving a high download rate, and retrieve the entire content as soon as possible, is to upload as much as possible. In order to increase the chance of being requested for content, a peer needs to retrieve the chunks that his neighbours have not yet retrieved. This is achieved by retrieving the rarest chunks of content following a *rarest-first* piece-picking algorithm, which has been the key to BitTorrent's success over the last decade. While the rarest-first approach has proven to be a suitable algorithm for file sharing, it is definitely not suited for streaming. Several solutions have been presented that allow streaming in P2P networks [100, 109, 125], but some limitations are difficult to overcome.

Existing P2P streaming systems

Over the last years the P2P paradigm has seen widespread adoption by content providers. In this section we provide an overview of the most popular systems. P2P systems can be categorised based on the typology of their network connections as *mesh-based*, *tree-based*, or *hybrid*.

In a mesh-based P2P system, peers self-organise in a randomly structured network overlay. Peers randomly connect to other peers participating in the same swarm in order to retrieve their content of interest. Peers retrieve segments of the content by explicitly requesting it from their neighbours. This mechanism of delivering the content is called *pull-based*. The pull-based mechanism introduces some overhead in the network, as peers need to explicitly send requests and exchange information on the content that has already been retrieved and can be redistributed. The biggest advantage of this type of systems is that peers retrieve the content from multiple sources at the same time. This approach shapes the behaviour of the system. Mesh-based systems are highly dynamic, as peers frequently establish and close connections in order to find the best sources of content. The most relevant examples of such systems are Octoshape [22], PPLive [28], PPStream [29], and Tribler [39]. Octoshape is a very popular P2P plugin given its adoption by CNN [6] for distributing live content. Octoshape helps reducing the load on CNN's distribution infrastructure during flashcrowds. It is a valuable solution for P2P-assisted networks, and offers several features such as support for adaptive streaming content [55]. PPlive and PPStream both provide a similar solution for P2P-assisted streaming systems. They are very popular streaming systems in China, serving their content to millions of users, e.g., PPLive served more than 200 million users during the 2008 Olympics [115]. While the majority of P2P streaming systems are proprietary, there are also open-source solutions such as Tribler, which is a fully distributed BitTorrent-based P2P client developed by the Parallel and Distributed Systems group of TU Delft [39]. It is compatible with existing BitTorrent-based networks, while providing features such as video streaming of both live and on-demand content.

In a tree-based P2P system, peers are organised in a structured network, where the original content provider only serves a few selected peers, usually the peers with the highest availability of resources. Each peer receives the content only from a single peer, that in the network overlay represents its parent node in the tree. Since peers do not explicitly request the content, they rely on the parent node sending it on time for playback. This delivery mechanism is called *push-based.* In contrast to pull-based, the push-based mechanism introduces less overhead in the network, as peers do not need to send requests or signal their content availability. On the other hand, tree-based systems are affected by peer churn, defined as the peer turnover, as each peer relies on a single source for receiving the content. An example of such a system is End System Multicast [74], the first live streaming P2P system. Peers in End System Multicast connect to each other in a mesh-based network, but the data is propagated from the source to the users in a tree fashion.

Hybrid systems try to combine the low-delay features of tree-based systems and the robustness of mesh-based systems combining the advantages of both. On the one hand, the tree data delivery structure guarantees a low delay from the content producer to the content consumer, given the low overhead of this approach. On the other hand, mesh-based systems provide robustness to sudden peer departures. As a direct consequence, a hybrid system is characterised by a mixed push- and pull-based mechanism. An example of such a system is New Coolstreaming [114], where peers are organised in a mesh overlay, but instead of directly requesting chunks of data, they subscribe to a set of sub-streams that is delivered without explicit requests. This approach is both pull-based, as peers explicitly request some sub-streams, and push-based, as the users subscribing to a sub-stream form a structured network on top of the existing network. Another interesting project that seems to exploit a hybrid scheme is BitTorrent Live [4], from the same authors of BitTorrent [5], the most popular P2P protocol. Little is known about the architecture of this system, and its deployment is ongoing since half a decade. Nevertheless, BitTorrent Inc. recently published a patent [76] that reveals the hybrid nature of their P2P system.

P2P-assisted streaming systems

The design of P2P streaming systems has several challenges compared to CS streaming systems. Some are related to P2P systems in general, and others are specific to the time-critical streaming scenario. All P2P systems need to design solutions for peer discovery and connectability, content discovery, incentive and reputation mechanisms, which are required for the system to work. In a time-critical scenario, as for the streaming context, P2P systems need to provide a solution for an additional set of challenges such as algorithms for content retrieval and peer selection, required to achieve a good QoE.

Most of the solutions previously presented apply a P2P-assisted design where they rely on a Centralised Authority (CA) to simplify their design and guarantee a good QoE. CAs, such as trackers and servers controlled by the content provider, solve the problem of content and peer discovery that mostly affect the video playback start-up time. As previously described, in the BitTorrent protocol each peer needs to retrieve a metadata file, the .torrent, before being able to start the download. The biggest downside that initially affects the QoE of the users regards the metadata retrieval. Retrieving the metadata file increases the time till playback, thus decreasing the user's QoE in a streaming scenario. The fastest way of overcoming a slow start-up time, is to use a CA to provide the metadata file and the identity of the other peers of the swarm. The CA usually monitors the download and playback progress of each peer in the swarm. Hence it can provide the list of peers that can serve the content, either because they already retrieved it, in the Video-on-Demand (VoD) scenario, or because they are further on in the playback, in a live streaming scenario. While P2P-assisted streaming systems that rely on a CA might seem like a good solution, they comprise a big privacy drawback. In order to join the system, each peer needs to report its status to the CA, and just by joining the system each peer knows the identity of the other peers interested in the same content.

P2P-assisted streaming systems also facilitate the integration of adaptive streaming, which is required in order to compete with CS streaming systems such as YouTube [45] and NetFlix [20]. Adaptive streaming tries to provide the best possible video quality given the display capabilities and network connection of the user's device. Nowadays, many content providers are moving towards adaptive streaming in order to provide a better QoE, e.g., Adobe's Dynamic Streaming for Flash [8], Apple's HTTP Adaptive Streaming [130], and Microsoft's Smooth Streaming [17]. The content provider knows what device the viewer is using, and monitors the quality of the connection in order to decide when to increase or decrease the quality of the stream. P2P-assisted streaming systems use the CA in order to monitor the peer's network status and display capabilities, and decide when to switch to a different quality stream. Thanks to the monitoring of the peers' progress, the CA can provide a list of peers that have retrieved the desired quality stream, facilitating the quality switch. In fully distributed P2P streaming systems, the integration of adaptive streaming is significantly more complex than in P2P-assisted CS systems. This requires new approaches for fully distributed P2P systems that want to provide state-of-the-art streaming technology. First, because they need to balance the available resources among the different streams in order to provide a good QoE to all the users. Secondly, because they need to provide self-adapting algorithms, allowing users to decide when to switch to a different stream.

1.2. Research Context

The research presented in this thesis has been conducted in the context of the European Framework Programme 7 project *P2P-Next* [25]. The goal of the P2P-Next project was to design, develop and deploy an open-source platform to distribute multimedia content to millions of users. Its main means of delivering multimedia content is the P2P network infrastructure developed by the Parallel and Distributed Systems (PDS) group of TU Delft. The P2P-Next project aimed to provide a future-proof system for the delivery of high-quality video streams, overcoming the limitations of the standard CS approaches.

1.2.1. The NextShare Platform

The system developed by the P2P-Next project is called *NextShare*. The NextShare platform has been designed to provide an efficient, trusted, user-centric multimedia platform. The goal of the NextShare platform is to promote the collaboration of its users, offering a solution for users who want to distribute their own content

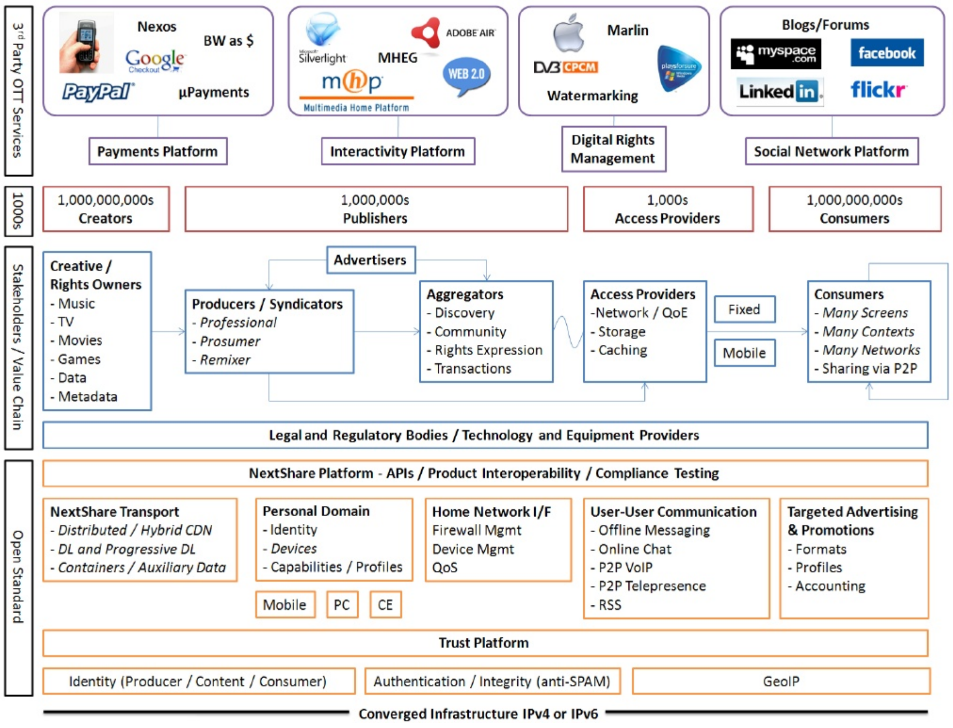


Figure 1.3: Overview of the P2P-Next project.

without incurring prohibitive costs. Moreover, the platform was also designed to offer an adequate solution for broadcasters. Broadcasters require monitoring, access control, enrichment with social networking, and the possibility to generate revenue in the form of advertisements, all of which has been integrated into the NextShare platform.

Solutions for all those requirements coexist in the platform developed by the P2P-Next project, which is modular enough to correctly address all its goals. An overview of the system and the modules developed by the P2P-Next project is presented in Figure 1.3. The entire system relies, and is based on, the NextShare platform, which is open source and provides several high-level APIs to interface with the different components. The NextShare platform provides all the features required by professional content providers and regular users that want to easily distribute content to a wide audience. Professional content providers have the ability to generate revenue by including advertisements, provide aggregators for content discovery and building communities around the content, integrate access control mechanisms and include payment platforms. Content consumers have the ability to become producers by freely distributing their content and taking full advantage of the P2P architecture. The system is not limited to the streaming context, as the platform is also suited to distribute static content such as games, software updates, and in general any type of file, might it be a collection of pictures or documents. Furthermore, third parties can easily interface with the system in order to provide additional services, such as payment solutions and social network interactions.

The role of TU Delft in this project has been creating the NextShare content delivery platform based on the P2P paradigm and deployed in the BitTorrent-based P2P client Tribler [39], developed by the PDS group. The focus of the NextShare system has been on a variety of subjects, such as efficient multimedia content delivery, reputation mechanisms, distributed search, incentive mechanisms, and overcoming the limitations of P2P systems such as reduced user connectability due to NATs and firewalls. The PDS group of TU Delft has been the P2P-Next partner that focused on addressing and solving those issues. The research conducted within the P2P-Next project by the PDS group of TU Delft has resulted in several PhD theses. In particular, L. D'Acunto [78] addressed the problem of peer connectability, R. Rahman [137] focused on designing incentive mechanisms, N. Chiluka [72] addressed the problem of integrating a social overlay in the platform, and B. Zhang [163] analysed the user behaviour in P2P networks.

1.2.2. The Swarmplayer

The modularity of the NextShare platform allowed the creation of a version of the core platform integrated in the browser, called the *Swarmplayer* [35]. The author of this thesis has been involved in the design and development of the Swarmplayer, which led to a collaboration with the Wikipedia [59] and Vodo [43] foundations. The Swarmplayer is a browser plugin that allows to retrieve and reproduce multimedia content from a P2P network directly in the browser, hiding the complexity of the underlying software. In no-profit organisations such as Wikipedia and Vodo, user collaboration is the only way of providing a good service without incurring prohibitive infrastructure costs, and users are generally inclined to participate as long as they are provided with a simple solution. Our browser plugin can be installed with few trivial steps, and has a default set of settings that hide the underlying process. The Swarmplayer architecture, presented in Figure 1.4, allows a transparent integration into the most popular browsers, and hides the complexity of retrieving content from a distributed infrastructure. The Swarmplayer is content agnostic, and it just provides a transport mechanism for the video player. This allows to stream content to a video player, such as VLC [42], or directly to the video tag of HTML5. Figure 1.4 presents some of the architecture details. A small browser plugin resides within the browser, and interfaces with the actual content retrieval engine called the *Swarm Engine*. The Swarm Engine retrieves the content mainly from a P2P network in a distributed fashion, but relies on a centralised HTTP server as fallback might the download rate not be sufficient to guarantee a continuous video playback. Since the plugin utilises both a centralised and a distributed infrastructure, in Figure 1.4 we present the centralised components in green, and the distributed components in brown. The plugin is activated whenever the custom *tribe* URL is specified as the source of the video content, in the form of *src=tribe://*, which then requests the video content from the Swarm Engine that is running in the background. The development of the Swamplayer was mainly motivated by our collaboration with the

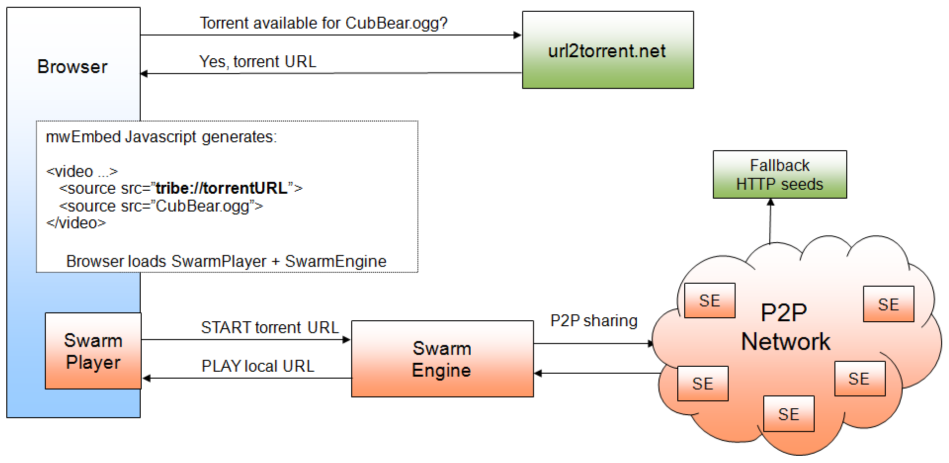

Browser
Torrent available for CubBear.ogg?
Yes, torrent URL
url2torrent.net
mwEmbed Javascript generates:
<video ...>
<source src="tribe://torrentURL">
<source src="CubBear.ogg">
</video>
Browser loads SwarmPlayer + SwarmEngine
Fallback HTTP seeds
Swarm Player
START torrent URL
PLAY local URL
Swarm Engine
P2P sharing
P2P Network
SE

Figure 1.4: The architecture of the Swarmplayer.

Wikipedia foundation [44], which was looking for an easy and efficient way to gain resource donations, in terms of bandwidth and storage capacity, from its users. This approach is the key characteristic of Wikipedia, as users contribute their knowledge so that others can benefit. Following the same approach, Wikipedia was looking for a way of distributing multimedia content without incurring prohibitive expenses. While the first version of the Swarmplayer is based on the BitTorrent protocol, the second version integrates a novel protocol, Libswift, that we will analyse in great detail in this thesis, see Chapter 4. Furthermore, given the low resource requirements of the second version of the Swarmplayer, we also ported it to the mobile environment by creating an Android application for the retrieval and playback of multimedia content, see Chapter 5.

1.3. Problem Statement

The research questions we address in this thesis involve different aspects of video content delivery over P2P networks. In particular, we address the following research questions:

[RQ1] What is the appropriate architecture for delivering video content to the heterogeneity of devices and connections in the current Internet? One of the major challenges of distributing multimedia content is that different users often require the content in different qualities. On the one hand, this is due to the differences in the users' network connections, which can differ depending on their locations during the content consumption. On the other hand, the users consume the content on various terminal types like TV sets or mobile devices, which may have different capabilities in terms of resolution, processing power, and power supply.

[RQ2] What is the best approach to efficiently retrieve layered video content? Videos are nowadays provided in different qualities to ensure that various types of end-user terminals can be supported. Layered coding schemes allow the encoding of content into several qualities within a single stream, providing the viewer with the best quality depending on his needs and capabilities, a solution that fits the P2P scenario very well. With layered video coding, the content is divided into several layers that can be combined to achieve different qualities and bit-rates. Scalable Video Coding (SVC) is the best candidate for retrieving such content, but it lacks proper retrieval algorithms.

[RQ3] How can we design a novel P2P streaming protocol that overcomes the limitations of the existing protocols? The original BitTorrent+TCP stack, the most popular P2P protocol, has limited the applicability of P2P infrastructure to video streaming. Several inherent characteristics, such as TCP's in-order retrieval, are not required in a P2P environment. Peers often have connectivity problems due to NATs and firewalls, which require complicated manual configurations, and they need to retrieve metadata files before being able to start downloading the content of interest. All of those aspects affect the user's QoE.

[RQ4] What is the performance of this novel P2P streaming protocol, and how does it compare with existing protocols? Each new protocol needs to offer advantages over previous solutions, or it will never see widespread adoption. It should hold up against existing systems, while excelling in the specific area for which is has been designed. Furthermore, the algorithms that determine its behaviour should be specifically crafted to exploit the features of this new protocol.

[RQ5] How can we improve the privacy of P2P users without sacrificing performance? The lack of privacy in P2P systems is an inherent characteristic of their design, as users have to expose their content interests. In order to improve the privacy of P2P systems a variety of solutions have been proposed to reduce the exposure of the users. Such solutions present systems that increase the users' privacy level, even providing complete anonymity, but always at the cost of performance. However, most P2P users are reluctant to trade performance for privacy, and therefore to accept the exposure that comes with P2P systems.

1.4. Contributions and Thesis Outline

The contributions of this thesis towards solving the research questions stated in Section 1.3, are as follows:

[RQ1] A framework for distributing scalable content over P2P networks (Chapter 2) The problem of providing multimedia content to the diversity of devices and connections is addressed by layered streaming systems, which provide the content in different qualities within a single bitstream. The distribution of layered content enables its consumption in a quality suited for the available bandwidth and the capabilities of the end-user devices. In this chapter we present the architecture of the NextShare P2P streaming system (see Section 1.2.1) designed to support the distribution and consumption of scalable content in a fully distributed P2P network. The architectural description includes how the scalable layers of the content are mapped to the pieces distributed in the P2P system and detailed descriptions of the producer- and consumer-side architecture of the system. Additionally, we provide an evaluation of the system's performance in different scenarios. Our evaluation assesses the performance of our layered piece-picking algorithm, which is the core feature of the system, and provides a comparison of the performance of our system's multi-layer and single-layer implementations. The presented system is to our knowledge the first open-source P2P network with full Scalable Video Coding support. This chapter is largely based on our work published in the *International Conference on Advances in Multimedia 2010* [88], in the *International Journal on Advances in Internet Technology 2011* [69], and in the *ACM Multimedia Systems Conference 2012* [89].

[RQ2] Deftpack: a robust piece-picking algorithm for scalable video coding in P2P systems (Chapter 3) In this chapter we propose a new self-adapting piece-picking algorithm for downloading layered video streams, called Deftpack. Our algorithm significantly reduces the number of stalls, minimises the frequency of quality changes during playback, and maximizes the effective usage of the available bandwidth. Deftpack is the first algorithm that is specifically created to take all these three quality dimensions into account simultaneously, thus increasing the overall quality of experience. Deftpack has been integrated into the NextShare system, although it can also be integrated into any BitTorrent-based P2P system and so has the chance of large-scale deployment. Our results from realistic swarm simulations show that Deftpack significantly outperforms previously proposed algorithms for retrieving layered content when all three quality dimensions are taken into account. This chapter is largely based on our work published in the *IEEE International Symposium on Multimedia 2011* [134].

[RQ3] The Peer-to-Peer Streaming Peer Protocol and its reference implementation, Libswift (Chapters 4 and 5) The Peer-to-Peer Streaming Peer Protocol (PPSPP) is designed to distribute multimedia content in a P2P fashion. PPSPP has been proposed for standardisation at the IETF (Internet Engineering Task Force) by TUDelft, and has been recently published in the standard track as RFC7574 [60]. PPSPP can provide static, on-demand, and dynamically generated content. It has been designed to provide short playback start up times, and to prevent disruption of the streams by malicious peers. It provides a high modularity and can be easily extended with several mechanisms and algorithms for peer selection and content retrieval patterns. Furthermore, it provides NAT/Firewall traversal and a lightweight solution for distributing content. It can run over several existing transport protocols, but UDP has been chosen as the default protocol carrier. While Chapter 4 presents the core of the official RFC, Chapter 5 presents Libswift, its reference implementation, discussing several characteristics of the protocol which are out of the scope of the official document, and showing the protocol's behaviour in challenging network environments. Chapter 4 is largely based on the official PPSPP standard, RFC7574 [60], while Chapter 5 is largely based on our work published as a TUDelft PDS Technical Report 2014 [132].

[RQ4] Performance analysis of the Libswift P2P streaming protocol (Chapter 6) In this chapter we present design features of the Libswift protocol, and a piece-picking algorithm that uses the transport features of Libswift in an essential way. We present an algorithm for selecting peers from which to request time-critical data, an algorithm for ordering data requests that guarantees upload fairness to all requesting peers that applies a form of Weighted Fair Queuing, and a downloading algorithm that takes peer locality and robustness into account. We investigate the performance of Libswift on both high-end and power-constrained low-end devices, comparing it to the state-of-the-art in P2P protocols. This chapter is largely based on our work published in the *IEEE International Conference on Peer-to-Peer Computing 2012* [135].

[RQ5] Hiding user content interest while preserving P2P performance (Chapter 7) We present a novel solution for P2P systems that hides user content interest without affecting performance. Our solution uses cover traffic in order to hide user interests while improving the performance of the system as a whole. The cover traffic and performance benefits are provided through several techniques, such as caching, subannouncing, relaying requests, and creating private swarms. We show that our system hides the real interests of a user from third parties, providing plausible deniability. We describe its design and implement it as an enhancement of PPSPP, IETF's Internet standard protocol for P2P Streaming systems. Our solution offers backwards compatibility with BitTorrent-based systems, and can also be integrated into other similar P2P protocols. Analysis of possible attacks shows that only an adversary who controls a very high percentage of the peers in the system can infer the content interest of the user, but even then without complete certainty. Furthermore, using actual P2P client software, we show that our privacy enhancements do not lead to a performance loss. This chapter is largely based on our work published in the *ACM Symposium on Applied Computing 2014* [133], which earned the best-paper award in the Distributed Systems track.

Conclusions (Chapter 8) In this chapter we summarize the key contributions of the work presented in this thesis, present some reflections on the current state of P2P streaming, and provide suggestions for future work.

1.5. Design and Research Methodology

In this section we present the design and research methods used throughout this thesis, the rationale behind its structure, and the differences in character and the relations among the chapters. At a higher level, in this thesis we always follow the standard research paradigm of identifying a problem in a system, designing and implementing a software solution to address this problem, and evaluating this solution and comparing it with alternatives by means of simulations or emulations. This is the common approach in research in distributed systems in general, and in P2P systems in particular [78, 126].

It is hard to characterise this work uniquely as a design thesis or as a research thesis, as it includes aspects of both. The chapters of this thesis are differentiated by having either a stronger design and engineering flavour, or a stronger research flavour. In this thesis, we always first present a design of (parts of) a system or a protocol that aims to address and satisfy the requirements of its application, and then we present an analysis of the entire design or of one of its specific algorithms.

The thesis is composed of design chapters that do the first, research chapters that do the second, and one chapter that does both.

More precisely, Chapters 2 and 4 describe the design of an entire system and a protocol, respectively. Chapter 5 can also be categorized as a design (or engineering) chapter, but it has a different character as it presents a reference implementation and evaluation of the protocol of Chapter 4. Because Chapter 4 has a significantly different style from Chapter 5, and as a matter of fact, from any of the other chapters of this thesis, as it comprises the most essential sections directly extracted from the official IETF standard [60], we have kept these two chapters separate. Chapters 3, 6, and 7 are the research chapters of the thesis; the former two present an evaluation of the different download schedulers of the systems of the preceding chapters, while the latter presents both the design and the evaluation of a privacy-aware P2P system. In summary, Chapters 2 and 3 are closely linked, as are Chapters 4, 5, and 6, whereas Chapter 7 is a stand-alone chapter. The author has made the research Chapters 3 and 6 independent from the preceding design chapters (in fact, they were published in the present form as conference papers), even if this implied a small amount of repetition in their introductions and in the explanation of their basic data structures.

In Chapters 2 and 4, we present the design of a P2P framework for distributing scalable content, and the design a P2P protocol respectively. For both chapters, the design of the system is driven by the requirement of its application. For Chapters 2, the requirements have been finalised within the context of the European proposal for the P2P-Next project [25]. The requirements have been defined by the members of the consortium, including Delft University of technology, and the objective has been to provide an open-source framework with a high focus on modularity, allowing third parties to replace its components and easily interface with the system to provide additional services such as payment solutions. Each of the components has been developed independently and in an iterative fashion, while keeping a simple interface. Furthermore, an additional requirement has been the possibility of integrating our solution with existing P2P systems, such as Libswift [13] or BitTorrent [5]. For Chapter 4, the major requirement has been overcoming the limitation of existing P2P protocols when used in a streaming environment. Other requirements, such as its modularity and security features, have been driven by the IETF PPSP working group [26]. The reference implementation of PPSPP, Libswift [13], has a common core that reflects the official description of the standard, and it has followed an iterative development with several design and implementation phases. Moreover, it allows the integration of different policies and mechanisms, e.g. download and reputation mechanisms. An example of the modularity of the current implementation, Libswift [14], can be seen by the three different download mechanism implemented.

In Chapter 3, we evaluate the performance of the Deftpack download algorithm by means of simulations. The use of simulations allows us to easily scale the size of swarms up to several hundreds of peers on a single machine. Also, as simulations allow us to speed up time, they greatly reduce the time required for experiments, hence providing a quicker feedback on the performance and correctness of the investigated algorithms. While simulations provide clear advantages, they do not always reflect reality well enough. Given the complexity of large distributed systems such as P2P systems, it is very hard to predict the behaviour of a specific algorithm or the feasibility of a design choice once deployed in the real world, with all of its unpredictability and variations. Therefore, in Chapters 5-7, we evaluate our solutions by mean of emulations on real hardware, implementing them on top of existing P2P clients, either BitTorrent [5] or Libswift [13]. The modified P2P clients are then run within Linux containers, which are emulated operating systems that provide a completely isolated environment and allow us to tune specific resources, such as the throughput and latency of the network connections. The P2P clients are then executed on the nodes of the DAS-4 multi-cluster [61], representing as closely as possible their behaviour in the real world.

Finding the optimal values of the parameters of the algorithms we investigate is based on the empirical evaluation presented in the research chapters. In order to do so, first we determine the feasible ranges of the values of the parameters from the design space. Then we run multiple experiments in different scenarios and select the optimal parameter values for a specific application. For every experiment we present and discuss throughout this thesis, we always show the optimal, and the closest sub-optimal values of the parameters. In all the chapters that present experimental evaluations, we always compare our solutions to similar approaches. Thus, when analyzing download algorithms in Chapters 3 and 6, we compare them to mainstream approaches and to the-state-of-the-art. When analysing the P2P Streaming Peer Protocol in Chapters 5 and 6, we compare it to the hugely popular P2P protocol BitTorrent [5], both to their flagship client [40] as well as to the most performing implementation, LibTorrent [15]. In Chapter 6, we also present a comparison of the power consumption between two video streaming Android applications, YouTube, and our port of the Libswift reference implementation. Finally, in Chapter 7 we model different attack scenarios, and present a security analysis of our privacy-aware P2P solution, as well as a comparison with the popular OneSwarm [105].

2

A Framework for Distributing Scalable Content over Peer-to-Peer Networks

The streaming of content over Peer-to-Peer (P2P) networks becomes more important as the popularity of Internet multimedia services is increasing and the corresponding server costs are rising. One of the major challenges of distributing multimedia content is that different users often require the content in different quality. On the one hand, this is due to the differences in the users' network connections, which can differ depending on the users' location during the content consumption. On the other hand, the users consume the content on various terminal types like TV sets or mobile devices, which have different capabilities in terms of resolution, processing power, and power supply.

These problems are addressed by layered streaming systems that provide the content in different qualities within a single bitstream. In this chapter we describe the architecture of our P2P streaming system supporting scalable content and provide an evaluation of the system's performance. We describe our entire framework for the distribution of scalable content in a fully distributed P2P network. The P2P system targeted for the integration is the NextShare system, which is developed within the P2P-Next project [25]. The main goal of P2P-Next is the development of the NextShare system, an open-source next generation P2P content delivery platform presented in Section 1.2.1.

The NextShare system has been developed based on the Bittorrent protocol [5] and thus provides an implementation of a fully distributed P2P system. To support Video on Demand (VoD), live streaming and the distribution of scalable content in the NextShare system, a number of modifications to the original Bittorrent protocol have been performed [125], as the original Bittorrent protocol does not support streaming. The scalable codecs used within NextShare are based on the Scalable Video Coding (SVC) extension of the Advanced Video Coding (AVC) standard [147].

One of the main reasons for implementing SVC support for the NextShare system is the lack of availability of systems with similar features. Currently, to our knowledge, no open-source P2P system supporting SVC is available and can be downloaded and tested by interested users. The advantages of distributing scalable content compared to simulcast approaches have been evaluated in a number of surveys (see, e.g., [148]). Additionally, we provide a comparison to our implementation for single layer content to illustrate the advantages of using scalable content. The author of this thesis designed, implemented, deployed and tested the core of the platform, called the *NextShare Core*, which provides the download engine and its algorithms.

The remainder of this chapter is organized as follows. Section 2.1 provides an overview of the related work. In Section 2.2, the approach for the integration of the scalable content into the NextShare system is described. In the following two sections, Sections 2.3 and 2.4 the producer- and consumer-side of this architecture are described in detail. Section 2.5 provides an evaluation of our implementation in terms of piece download efficiency as well as a comparison to the traditional single layer approach. Finally, conclusions are presented in Section 2.6.

2.1. Related Work

The distribution of multimedia content over P2P networks has been a popular research topic in recent years. Due to the increasing popularity of streaming high-quality multimedia content over the Internet, P2P provides a cost-efficient alternative to reduce server costs.

The distribution of layered content over P2P systems has also been addressed in the literature before. LayerP2P [117] provides a well defined solution for distribution SVC content over P2P, but does not utilize real SVC codecs for the prototype implementation and relies on the usage of H.264/AVC-compatible codecs that can only be used to test one of SVC's scalability dimensions, the temporal scalability. Thus, one of the goals of the NextShare implementation was to design, implement, and distribute an open-source system with full SVC support. Other systems supporting the distribution of SVC content over P2P are described in [139] and [57]. PALS [139] provides a receiver-driven solution for receiving layered content over P2P. [57] describes how SVC can be integrated into a tree-based P2P system. However, both approaches do not allow an easy integration into existing P2P systems, as the implementations have been based on proprietary systems and protocols. Regarding compatibility an advantage of our implementation is that it has been based on the wide-spread Bittorrent protocol and all architectural choices have been performed while ensuring backwards compatibility to existing Bittorrent clients. This allows an easy integration of the new scalable video technology into existing P2P communities. Furthermore, backwards-compatibility of the base layer for existing Bittorrent clients is provided.

Table 2.1: The four scalability layers used for encoding and decoding content.

Bit Rate	Resolution	Quality	frame/sec
512 kbps	320x240	low	25
1024 kbps	320x240	high	25
1536 kbps	640x480	low	25
3072 kbps	640x480	high	25

2.2. NextShare Integration

To fully integrate scalable content into the NextShare system, a number of problems had to be addressed. Two main problems, the selection of suitable scalability layers and the mapping of the layers to Bittorrent pieces, are described in detail within this section. While the selection of the scalability layers tries to consider all popular qualities and to support a number of different network connections, the mapping of the scalability layers to the Bittorrent pieces tries to ensure that the best trade-off between flexibility for possible quality switches and overhead in terms of piece management is found.

It should be noted that even though we are using SVC within our NextShare system, all design decisions have been made with the intention to make the architecture codec-agnostic. Thus, if another scalable video codec is utilized within the NextShare system, only the coding and packaging tools need to be replaced, while the integration into the NextShare core will remain suitable for every other layered codec.

2.2.1. Scalability Layers

The first step for the integration of scalable content into the NextShare system was the selection of the desired scalability layers. The selected layers are described in Table 2.1, which presents the four layers used for the integration. The main reasons for selecting this layer structure were to maintain a good coding efficiency and to provide all popular qualities. The possibility to add further layers to support HD content is also fully supported by our framework, but has been omitted for the current version due to constraints in the upload bandwidth of our system's users. From the coding-efficiency point of view, the difference between the layers in terms of bit rate should be not too low, as the coding efficiency decreases drastically in such cases [147], while the selected bit rates represent the most popular qualities that are provided nowadays by multimedia portals. Furthermore, it should be noted that the audio bitstream is provided together with the video bitstream of the base layer. Thus, the 512 Kbps for the base layer includes the bit rate for the 128 Kbps audio bitstream. This is necessary to ensure that the audio is always received in time for playback, which can start as soon as the base layer is received.

To ingest the different layers into the P2P system the layers need to be provided as separated files. The base layer is multiplexed with the audio content and provided in a proper container format. The enhancement layers are provided as separate optional files. By using this file structure, Bittorrent clients without SVC support

Table 2.2: The mapping of the four scalability layers to BitTorrent pieces.

Layer	Kb/time slot	KB/time slot	pieces/time slot
BL	$512Kbps * 2.56 \approx 1.310Mbits$	$/8 \approx 164KByte$	3 pieces @ 55 KByte/time slot
EL1	$1024Kbps * 2.56 \approx 2.621Mbits$	$/8 \approx 328KByte$	6 pieces @ 55 KByte/time slot (3 pieces in previous layers, 3 new pieces)
EL2	$1536Kbps * 2.56 \approx 3.932Mbits$	$/8 \approx 492KByte$	9 pieces @ 55 KByte/time slot (6 pieces in previous layers, 3 new pieces)
EL3	$3072Kbps * 2.56 \approx 7.864Mbits$	$/8 \approx 983KByte$	18 pieces @ 55 KByte/time slot (9 pieces in previous layers, 9 new pieces)

can still download the H.264/AVC-compatible base layer and decide not to download the optional enhancement layers without wasting any bandwidth.

2.2.2. Mapping to BitTorrent Pieces

The second step of the integration process is the mapping of the scalability layers to Bittorrent pieces. Firstly, the unit shall represent a synchronization point for dynamic switches between different quality layers. To achieve this goal each unit starts with an Instantaneous Decoding Refresh (IDR) reference frame. Secondly, it should be noted that we do not perform a direct mapping to pieces but to a unit. This unit represents a fixed number of frames for a specific layer and can be mapped to a fixed number of pieces. The reason for this approach is that the piece size might be changed in the P2P system for various reasons, and by basing the mapping on units rather than on pieces only the unit/piece-mapping needs to be updated when the piece size is modified.

The mapping to the units has been performed based on several criteria. First, the units need to be selected large enough to allow for a good coding efficiency. As it should be possible to decode each unit independently, since they provide synchronization points for quality changes, the number of frames within one unit should be high enough to allow for good coding efficiency. Additionally, the number of frames within one unit should be low enough to provide the flexibility to conveniently switch between qualities when the network conditions change.

Based on these considerations, a mapping of 64 frames, which represent 2.56 seconds of content at a frame rate of 25 frames/sec, has been selected. Such a unit is subsequently mapped to three pieces; however, as noted previously, the piece mapping can always be changed based on the requirements from the P2P system. The piece mapping is illustrated in Table 2.2.

The mapping to the 55 KByte pieces results in a small overhead of available bits per piece. However, this overhead is utilized to compensate the small drifts of the constant bit rate (CBR) algorithm utilized during the SVC encoding process (see Section 2.3.1). Based on the calculations in Table 2.2, a mapping of the layers to

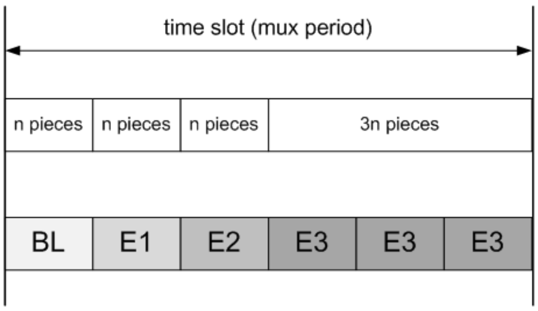


Figure 2.1: A graphical representation of the piece mapping within a time-slot.

Bittorrent pieces could be performed as illustrated in Figure 2.1. The figure shows that the unit for each layer can be mapped to a specific number of actual pieces. In our example, we choose a piece size of 55 KByte, but different implementations might use different sizes based on their needs. The only constrain, that comes form the BitTorrent protocol, is that the pieces size needs to be constant.

2.3. Producer-Side Architecture

The producer-side architecture describes all steps from encoding the SVC bitstream to the ingestion into the core of the P2P system. The topics addressed in this section include the encoding process, the splitting of the bitstream, creating metadata based on the bitstream's supplemental enhancement information (SEI), packetizing the bitstream, and ingesting the bitstream into the core of the P2P system. An illustration of this architecture is provided in Figure 2.2, more details on each of the processing steps are provided in the following sections.

2.3.1. Bitstream Preparation

As the first step of the bitstream preparation process, the raw video (i.e., the YUV video frames) is encoded by an optimized encoder JSVM 9.15 [46], which uses a CBR algorithm to ensure that the pieces created from the video content have a constant size. The CBR algorithm works at GOP (Group of pictures) level and maintains the bit rate at GOP level throughout the encoded bitstream. However, the CBR algorithm still produces a small offset compared to the desired bit rate. As a constant piece size has to be maintained, a positive offset could results in frame dropping while a negative offset can be easily addressed by using padding bits during the splitting process. To ensure that no frames are dropped in case the small drifts of the CBR algorithm result in a positive offset, the target bit rate for the CBR algorithm is chosen slightly lower (approx. 1-2 % below the target bit rate). Thus, the CBR algorithm produces only negative offset compared to the real target bit rate, which can be easily handled.

The encoded SVC bitstream is subsequently split into the H.264/AVC-compatible base layer (BL) and the enhancement layers (EL) by the Network Abstraction Layer Unit (NALU) demuxer. The demuxer analyzes the NALU headers and splits the

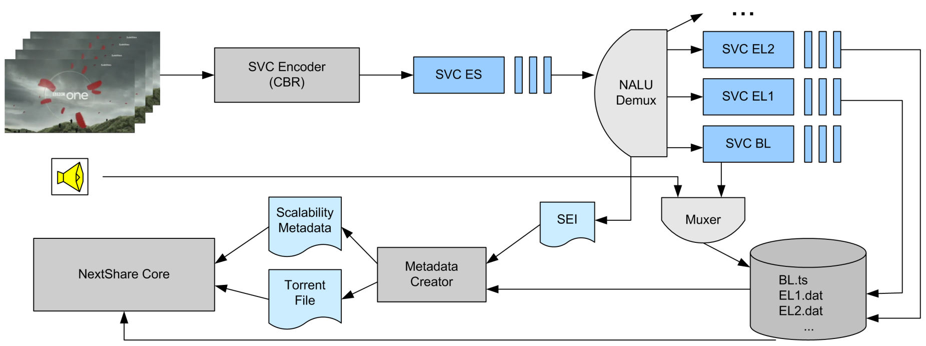


Figure 2.2: The Producer-Side Architecture of the NextShare system with SVC support.

access units into separate bitstreams for each layer. Each of these layer bitstreams consists of several pieces of constant size. If within one bitstream the GOP size exceeds the piece size, subsequent NALUs (frames) would be dropped. However, as mentioned in the previous paragraph, such a situation is avoided by setting a slightly lower target bit rate for the CBR algorithm. If the GOP size is less than the piece size, the remaining size bits are filled with padding bits. Additionally, the SEI information at the beginning of the bitstream (i.e., the scalability info message) and the Sequence Parameter Set (SPS) and Picture Parameter Set (PPS) are provided to the metadata creator (see Section 2.3.3).

The audio data can be provided already encoded, e.g., as an MP3 or AAC audio file. If a raw PCM audio file is provided, the audio content is encoded to the desired audio coding format.

2.3.2. Bitstream Packetizing

In the bitstream packetizing step, the base layer of the SVC bitstream is muxed with the audio into a proper container format. The main reason for this step is that the base layer should be provided in a backwards-compatible way, so that also end user P2P clients or terminals that only support H.264/AVC can successfully process the base layer. For such a purpose two different container formats were investigated for our system, the MPEG Transport Stream (MPEG-TS) [50] and the MPEG-4 file format (MP4) [51].

MPEG-TS is a standard able to encapsulate audio and video Packetized Elementary Streams (PESs) and other data and is supported by a majority of systems and applications. The main disadvantage in using MPEG-TS is that it usually has a rather high overhead in terms of bit rate(10-20% on average). An alternative muxing scheme is provided by the MP4 format, which provides functionalities similar to the ones of MPEG-TS while having a clearly lower overhead ($\sim$1%). Thus, MP4 is the preferred container format used in our system, while MPEG-TS support is provided for compatibility to older systems.

The overall architecture is codec-independent: the system is able to recognize the container format and apply the corresponding processing. A general problem during the muxing phase is that the output should have a certain fixed size to ensure that a full GOP of video content and the corresponding audio content can be mapped to one unit. Considering that muxing schemes can have variable overheads, it is in principle not possible to a priori know if the output of the muxer for a certain audio and video input will respect the size limits. In case the output size is smaller than expected it will be possible to add padding bits and solve the issue (muxing codecs usually provide routines for that). The real problem is when the muxing output is larger than the allowed one: in such a case the muxer tries to change its parameters to lower the overhead to the minimum. However, if adjusting the muxer's parameters is not sufficient, it would usually not be possible to meet the size constraints. Thus, as previously mentioned in Section 2.3.1, the target bit rate is set lower than desired to ensure that only the first case (lower output size) occurs. To avoid possibly wasting too much bit rate on padding bits, the architecture optionally provides support for a feedback mechanism between the muxer and the encoders to solve this. Thus, in case the output size would be higher than the target size, the muxer asks the SVC and the audio encoders to re-encode both audio and video using a lower target bit rate. For the enhancement layers, the padding mechanism described in the previous section is applied.

2.3.3. Scalability Metadata Support

Although the pieces of the video stream are transmitted over the network in a layered way, the de-packetizer at the consumer-side needs to know the properties of the layers for the decoding process and the decoder needs access to the parameters from the beginning of the bitstream. Those parameters are contained in the Sequence Parameter Set, SPS, and the Picture Parameter Set, PPS, which are both types of NAL units. Thus, the properties of the layers, which are usually provided by the Supplemental Enhancement Information (SEI) at the beginning of the bitstream, and the parameter sets need to be forwarded to the consumer-side.

To store these metadata and transmit them to the de-packetizer when needed, the SEI message and the parameters are forwarded from the NALU demuxer to the metadata creator. The metadata creator subsequently parses the SEI data and stores the properties of the layers in an XML metadata document. Additionally, the SPS and PPS elements are encoded in base64 to allow their storage in XML and are added to the metadata document. The resulting metadata document contains all the layer information and parameter sets required by the de-packetizer and decoder modules (see Section 2.4 for details).

2.3.4. Ingest into the Core

The NextShare core represents the P2P engine responsible for creating and injecting the content into the network. The main metadata file required for the ingestion of the content into the P2P system is the torrent file. The torrent file provides the information required for the download of the previously encoded base and enhancement layers, as well as metadata related to the content including the previously created scalability metadata. The created torrent file is compatible with the Bittorrent protocol [5] and can therefore be processed by every peer running a Bittorrent-compatible client. This backwards-compatibility increases the reliability of the torrent swarm and the scalability of the distribution costs, as the torrent file can not only be processed by NextShare-compatible clients, but by all Bittorrent-compatible clients.

The fact that the H.264/AVC-compatible base layer and the audio stream are provided commonly packetized into a proper standardized muxing format enables also clients without SVC support to join the swarm as they are able to consume the stream in base layer quality. Therefore, every peer has an incentive to download at least the base layer, which increases its availability in the swarm. As the base layer is the most important layer (it is sufficient to start the playback and is always required) this really helps to ensure real-time playback for all clients in the swarm.

After the creation of the torrent file, the content is ingested into the NextShare core (i.e., the torrent file is distributed to other peers and the files containing the base- and enhancement layers are seeded). During the ingestion process, the base and enhancement layer files are split into pieces as illustrated in Figure 2.1. During the mapping process some problems have to be taken into consideration. Firstly, the integrated CBR algorithm does not provide an exactly constant bit rate, but allows for minor drifts. Thus, the piece mapping is always performed with a small overhead. Additionally, the 188 byte size utilized for the MPEG-TS packets cannot be mapped to a power of two, while the pieces ingested into the NextShare core should have a multiple of two as their size. Thus, the possibility of choosing a piece size that differs from the standard power of two has been successfully investigated.

2.4. Consumer-side Architecture

The consumer-side architecture describes all steps from receiving the bitstream through the P2P system to decoding the bitstream for displaying it in the media player. The steps described include the local streaming of the received content, the de-packetizing of the content, the signaling of the layer properties using suitable metadata, and the merging and decoding of the received layers. An overview of this architecture is provided in Figure 2.3 and described in detail in the subsequent sections.

2.4.1. Provision of the Received Content

When the content is accessed by the user the layers from the NextShare swarm are downloaded in an intelligent way in order to maximize the Quality of Experience (QoE) for the current available download bandwidth. A great advantage of changing quality by displaying more or fewer enhancement layers regards the fall-back scenario: in a P2P system peers are considered to have an unreliable and selfish nature, leaving the swarm and decreasing the total available bandwidth as soon as they have received the desired content.

In such a case the fall-back scenario will occur, where the user will experience a slow decrease of the QoE, as the download engine avoids downloading higher layers

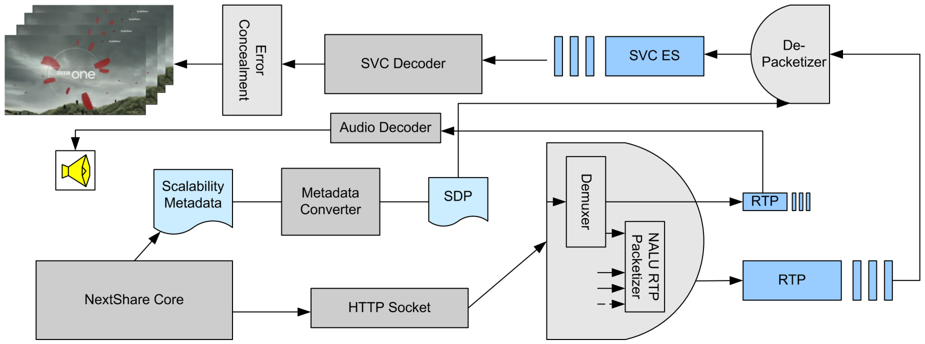


Figure 2.3: The Consumer-Side Architecture of the NextShare system with SVC support.

for upcoming time slots, if enough pieces of the previous layers are not already available. Retrieving the content from the network is based on a modified approach of the Give-to-Get (G2G) algorithm [125]. The main reason for using G2G is that Bittorrent's Tit-for-Tat algorithm is not suitable for streaming multimedia content, as packets need to be retrieved in a certain order.

The original G2G algorithm divides the part of the piece buffer close to the current playback position into high-, mid-, and low-priority sets with regard to the current playback position. The high-priority set is the part following immediately after the playback position. The G2G algorithm selects the pieces in the high-priority set based on their deadline, i.e., the piece with the nearest deadline in the high-priority set is downloaded first to ensure a continuous playback of the content. In the mid- and low-priority set, the pieces are selected using Bittorrent's rarest first strategy. Using this piece-picking policy the G2G algorithm tries to ensure that the pieces are downloaded in time for playback while still ensuring that piece's that are desired by neighbour peers are downloaded as well.

For the layered application of the G2G algorithm, the priority sets are applied to all the active available layers in a proportional way. Thus, for every layer a high-priority set is created where the pieces are selected according to their deadline. Obviously the dependency between the layers has to be taken into account, and pieces from the higher layers are downloaded only if the corresponding pieces from the lower layer have already been retrieved.

As discussed in the previous section, by providing backwards compatibility with other clients, the availability of the base layer at all peers in the same swarm can be assumed. Therefore, if a download bandwidth of at least the base layer's bit rate is provided by the peer's connection, it can also be assumed that the playback will never stall (if the neighbour peer seeds the content or is ahead in its playback position). Note that once a peer finishes watching the content, the download engine will start retrieving the remaining pieces of all the layers for two major reasons: firstly, to increase the layer's availability in the swarm, and secondly, to enable watching the content at the highest quality once all the layers have been downloaded.

However, if this behaviour is not desired due to bandwidth restrictions it can be disabled in the configuration.

After enough content has been downloaded to guarantee a continuous playback of at least the base layer, the download engine of the NextShare core will initialize the demuxer module. The multimedia data is forwarded to the demuxer utilizing an HTTP socket, which was selected to ensure interoperability between the NextShare core and third-party de-packetizing/decoding solutions. A persistent connection will be established between the demuxer and the NextShare core, allowing the demuxer module to sequentially ask for the available content for the following time slot, depending on the requests it receives from the decoder module. It is important to notice that the available pieces of the following time slot will be sent to the demuxer as late as possible, allowing the NextShare core to manage the major buffer, to increase the quality until the last moment and to try to increase the quality if the user pauses the playback.

2.4.2. Bitstream Demuxing and Packetizing

As described before, the demuxer works online: it receives from the swarm at least the stream of the base layer, which is processed by another suitable demuxer. The demuxer firstly removes any possible padding from the production phase and splits the container format stream into the audio content and the H.264/AVC-compatible base layer.

The elementary audio stream is directly encapsulated into a Real-Time Transport Protocol (RTP) packet stream (e.g., according to [92] for an MP3 audio stream) and is sent to the successive module. The elementary SVC base layer video stream is forwarded to the NALU RTP packetizer, which puts the base layer and the received enhancements layers into an RTP packet stream, reordering the packets and forwarding the RTP stream containing all layers to the next module. Furthermore, the demuxer establishes a RTCP channel with the player. This is useful in order to maintain essential synchronization information among the video and the audio layers and also to support playback control commands.

Please note that all the modules represented in Figure 2.3 are typically running on the same host, i.e., the peer that receives the content. However, the main reason for selecting the RTP protocol to convey the audio and video data was to provide flexibility and to enable a possible integration of the P2P network with a more traditional server-based network. It is worth noting that the overhead in using the RTP protocol does not really affect performance. In such a server-based network the NextShare consumer-peer could act additionally as a server, receiving the content from the P2P network and redistributing the scalable content within an RTP streaming network.

2.4.3. Scalability Metadata Support

To decode the layers received from the NextShare system, the de-packetizer needs to be aware of the scalability properties of these layers. To provide a generic signaling mechanism for these properties, which could also be used by third-party solutions, we have decided to provide this information as a Session Description Protocol (SDP) document to the de-packetizer. The SDP document is formatted according to [143], which provides the capabilities to signal the properties and dependencies of the scalable layers.

As the metadata is provided by the NextShare core in a NextShare-conforming XML format, the scalability metadata document is firstly converted to the targeted SDP format. Subsequently, the SDP message containing the layer properties and the parameters sets is forwarded to the de-packetizer.

2.4.4. Bitstream Consumption

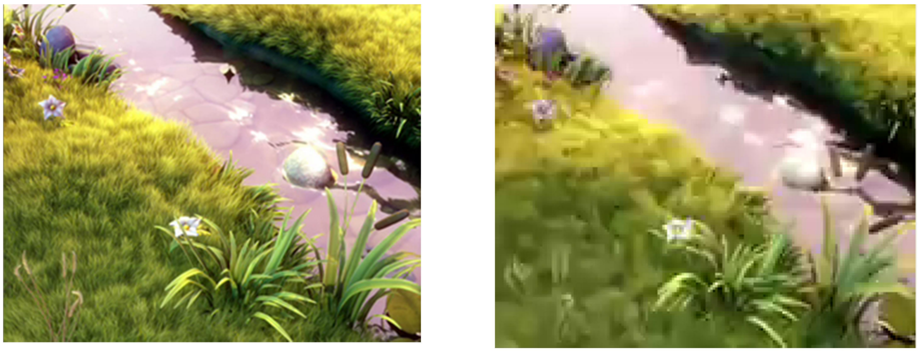
After the demuxing and RTP packetizing of the audio and video content, the RTP streams are forwarded. While the audio stream is forwarded to a standard de-packetizer and decoder, the SVC RTP stream is processed by our customized tools.

Firstly, our SVC de-packetizer parses the RTP packets and provides the payload and the time stamps to the SVC decoder. To perform this extraction process, the de-packetizer needs to have SVC layers properties along with the audio and video RTP port information in advance. This information is provided by the SDP file. Additionally, the de-packetizer parses the parameter sets from the SDP and provides them to the decoder, as they are required for the decoding process. The SDP file contains the SVC layer information for each layer and the de-packetizer extracts the suitable information for the desired layer playback. Finally, our highly optimized version of the JSVM 9.15 reference decoder performs the real-time decoding of the SVC content utilizing the error concealment algorithm embedded in the decoder (described in the next section).

Error robustness in the video decoder is important since transmission errors are very common in current (especially wireless) video streaming and transmission systems[150]. The transmission errors can lead to very poor quality of experience and in worst case scenario, they can lead to decoder crashes. Usually the error concealment is performed by monitoring the order of the NAL slices and their macroblocks to see if all the NALUs are received. If NALUs are missing suitable concealment operations for the missing macroblocks are performed, e.g., by using a frame copy from the previously correctly received frame[152].

The difference in the error concealment implementation in the NextShare system to traditional error concealment approaches is that random frames or burst of frames cannot occur, as each piece contains a full GOP for a specific layer. Thus, a GOP can either be received or not received, but single frames cannot be lost during the transmission. As the whole GOP between the IDR pictures can either be present or missing, this can lead to varying resolution in the player if spatial scalability layers exist (as suggested in Table 2.1). However, a major advantage of the layered content provisioning in NextShare is the awareness of the layer dependency. The higher enhancement GOPs are not sent to the decoder if not all the pieces from the lower layer GOPs have been previously sent for the current time slot. Additionally, the base layer is always retrieved, which makes the error concealment easier and more effective.

The error concealment is integrated into the optimized JSVM 9.15, which was integrated into the VLC plug-in. To cope with the missing spatial enhancement



A) Correctly received EL3 picture: PSNR 42,4 dB

B) Upscaled picture from the base layer: PSNR 31,7 dB

Figure 2.4: The result of upscaling the video content.

layers, an upscaling functionality was integrated into the decoder based on normative integer-based 4-tap filters. The upscaling algorithm is provided by the JSVM reference software; please refer to [46] for more details.

2.4.5. Error Handling

The target resolution for the sequence is defined in the SPS NAL unit, which is compared with the received resolution (the resolution of the IDR picture) when starting the decoding for a new time slot. As mentioned earlier, the SPS information reaches the decoder within an SDP description from the RTP de-packetizer. If the resolutions do not match, frames of the new time slot are up-scaled from the lower resolution to the target resolution to maintain the preferred resolution. Since the layer with lower spatial resolution usually provides lower quality, the upscaled picture is blurred, as illustrated in Figure 2.4. However, compared to changing the window size during video playback, the blurring is usually the better solution.

The selected structure of layers presented in Table 2.1 supports two different resolutions for the video. In some cases the layers for high resolution video cannot be received within the defined time slot, even though the consumer prefers to watch the video in higher resolution. As an example, this could happen while streaming on a heavily congested access point. Figure 2.5 shows an example situation with a four layer SVC sequence, where the second enhancement layer provides the spatial enhancements whereas the first and third enhancement layers provide quality enhancements. In Figure 2.5, the number of received layers decreases during the streaming from best quality to the base layer level. In this case, an upscaling of the base layer quality to the higher resolution is performed, to avoid changing the playback window size, which is very disturbing for the consumer. The upscaling algorithm is performed when only the base layer or the base and first enhancement layer are received. As soon as the second enhancement layer is received, no upscaling is necessary as the desired resolution is already provided.

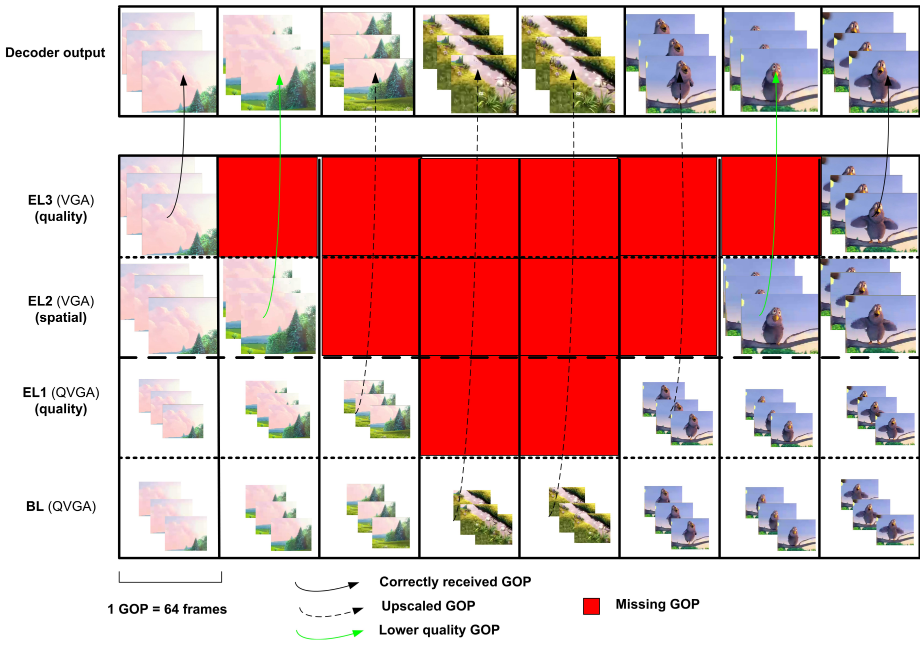


Figure 2.5: The principle behind upscaling the content.

2.5. Evaluation

To investigate the performances of our solution we performed a series of experiments with the implementation of our architecture. The experiments were performed in a lab-only environment composed of several peers connected with each other using heterogeneous connections. We monitored the performance of a peer acting as leecher in the swarm, retrieving the content from at least one seeder. Each of the seeders provides a limited upload bandwidth capacity to the leeching peer, which limits the download bit rate the leecher can achieve.

For our tests we encoded a two minute video sequence using four quality layers. The properties of these layers are described in Table 2.3. The properties of the sequence used for the evaluation differ slightly from our reference layer structure described in Table 2.1. The main reason for using a different layer structure for the evaluation were to keep a simple uniform chunk size for all layers. Additionally, only quality layers were used for this evaluation to enable an easy comparison of the received quality layers in terms of peak signal-to-noise ratio (PSNR).

The evaluation consists of two test series. First, the behaviour of the layered piece download algorithm is illustrated in different scenarios to demonstrate the efficiency of our layered piece-picking implementation (Section 2.5.1). Second, the received quality of the layered implementation is compared to the single layer implementation of our P2P system to show how improvements in terms of received quality can be achieved (Section 2.5.2).

Table 2.3: Evaluation Layer Structure

Bit Rate	Resolution	Quality	frame/sec
512 kbps	640x480	basic	25
1024 kbps	640x480	low	25
1536 kbps	640x480	medium	25
2048 kbps	640x480	high	25

For the evaluation process five scenarios for the two minute test video sequence were defined. Four of those scenarios were investigated for both test series, while the first scenario is only interesting for the layered piece-picking series. In the first three scenarios all peers remain in the swarm for the whole time. In the other two scenarios, seeders leave the swarm at specific time points to test the robustness of the system against churn.

- Scenario 1: The leecher peer connects to a single seeder, which provides sufficient bandwidth to download only the base layer. This scenario is only investigated for the first test series.
- Scenario 2: The leecher peer connects to three seeding peers. Together the seeders provide enough bandwidth to download all layers for the test video.
- Scenario 3: The leecher peer connects to two seeders, which provide more than sufficient bandwidth for the download of three layers, but not sufficient to constantly download all four layers.
- Scenario 4: At the beginning of this scenario two seeders provide sufficient bandwidth to download all layers. After 40 seconds one of the seeders leaves the swarm and the available bandwidth is decreased to half the original value.
- Scenario 5: At the beginning of this scenario three seeding peers are in the swarm and sufficient bandwidth for all layers is provided. After 30 seconds, one seeder leaves the swarm and decreases the available bandwidth to half the original value. After 70 seconds, a new seeder joins the swarm and increases the bandwidth back to the full capacity.

As the player of our system can switch between qualities without flickering (see Section 2.4.5) for this evaluation the playback quality is increased as soon as possible, even if only for one time slot. This particular behaviour can be changed to a more conservative approach that might be needed in case the decoder/player module is replaced or a constant quality playback is preferred. To influence the download strategy, the piece-picking algorithm can be configured to only perform switches to higher layers if a specific number of higher layer pieces have been downloaded (this number is one for the presented evaluation, i.e., immediate quality switches are performed).

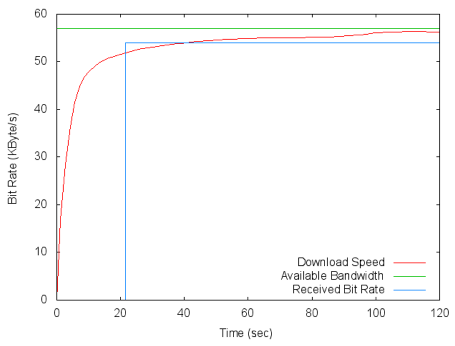


Figure 2.6: Layered Piece-Picking Series: Scenario 1

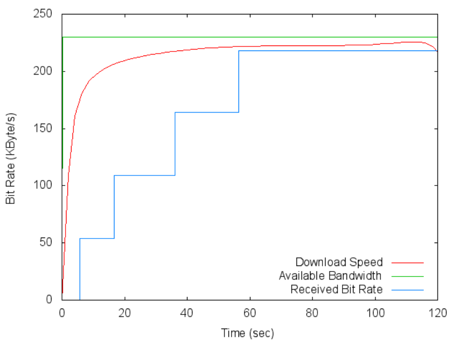


Figure 2.7: Layered Piece-Picking Series: Scenario 2

2.5.1. Layered Piece-Picking Test Series

In the following graphs, all five scenarios are presented to analyze the behaviour of the piece selection when layered content is streamed. In every graph the available upload capacity, the download rate, and the received bit rate are presented. The upload capacity illustrates the download bandwidth that is provided from the seeders to the leecher peer. The download bit rate describes at which rate the leecher peer downloads pieces from the seeders. Please note that the download speed is never calculated instantly, instead it is based on the piece arrivals and therefore averaged over a small period of time. This explains the smoothness and delay of the download curve and avoids spikes in the results. Finally, the received bit rate represents the number of pieces that were received in time for playback. It should be noted that the received bit rate differs slightly from the actual playback bit rate of the video. The reason for this deviation is that the CBR algorithm of our system's SVC encoder does not provide an exactly constant bit rate but allows for small drifts. Thus, padding is used to achieve the constant piece size, as described in Section 2.3.1.

In Figure 2.6 the results for the first scenario are presented. As the available download bandwidth allows to download only the base layer, the behaviour of the piece-picking algorithm is very simple. Only base layer pieces are downloaded and the playback of the base layer is started after the initialization phase.

Figure 2.7 illustrates the behaviour in case of the second scenario. As mentioned before, a switch to a higher layer is performed as soon as the first piece of the layer is downloaded. Thus, the leecher peer starts with the playback of the base layer and gradually increases the quality until the highest layer is reached. The playback remains at best quality until the end of the scenario. An interesting behaviour emerging from the configuration of the piece-picking algorithm for this test series can be seen in Figure 2.8. Having a download rate higher than the first three layers combined, the piece-picking algorithm will try to download the highest layer whenever possible. As the available bandwidth is not sufficient to constantly download all layers, the playback quality switches frequently. If a constant playback quality is preferred, the piece-picking algorithm can be configured accordingly.

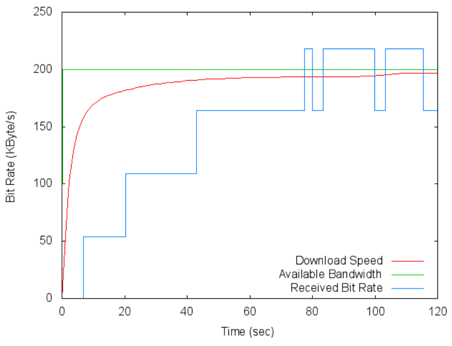


Figure 2.8: Layered Piece-Picking Series: Scenario 3

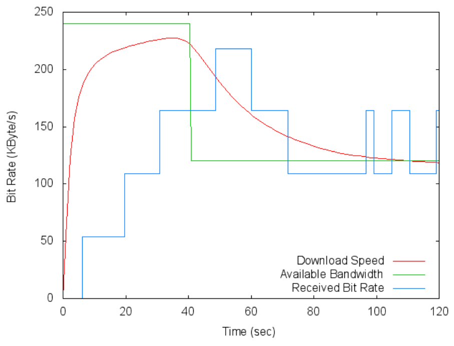


Figure 2.9: Layered Piece-Picking Series: Scenario 4

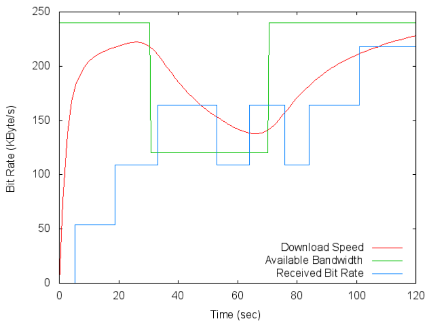


Figure 2.10: Layered Piece-Picking Series: Scenario 5

In Figure 2.9 the fourth scenario is presented. At the beginning of the scenario the playback quality is gradually increased, similar to the behaviour in the second scenario. However, as soon as the highest playback quality is achieved, one seeder leaves the swarm and causes a decrease of the available bandwidth. Thus, the leecher peer cannot download the higher layer pieces any more and decreases the playback quality. Nevertheless, the playback never stops and the piece-picking algorithm stabilizes after some time and downloads the best possible quality for the given bandwidth until the end of the scenario (between two and three layers, similar to the behaviour in scenario 3 but with less available bandwidth). The results of this test scenario show that our implementation is robust to departing peers and can keep the video playback going, as long as sufficient seeding peers to download at least the base layer quality remain in the swarm.

Figure 2.10 illustrates the results for the final scenario. At the beginning the playback is again gradually increased. However, one of the seeding peers leaves the swarm after 30 seconds and the piece-picking algorithm adjusts and downloads the best possible quality for the new bandwidth conditions (between two and three layers). After the joining of the new seeder, the playback quality is increased to the best possible quality, which can now be downloaded due to the improved bandwidth conditions. The results of this scenario show that our system is robust to churn and reacts suitably to leaving and joining seeders.

Another interesting observation of this test series is the initial playback delay when streaming layered content using our NextShare system. The test results of all scenarios show that the playback will start no more than ∼25 seconds after the leeching peer joins the swarm, assuming a download rate at least as high as the base layer's bit rate (otherwise no real-time playback is possible). Even is the playback could start sooner, the client will not start reproducing the video until the download speed is consistently higher than the video playback rate. As previously explained, the network speed is averaged over a small period of time, which explains the delay. Also, the playback of the base layer can start as fast as after five seconds, if the bandwidth conditions are good. This reduces the initial playback delay greatly compared to the single layer implementation of our system.

2.5.2. Quality Comparison Test Series

In this test series a comparison of the single layer and multi layer implementations of our system is performed. The goal of this test series is to compare the received quality for both approaches. For the evaluation process the previously described scenarios 2-5 are investigated.

The following graphs represent the results for this test series. Each graph illustrates the received video quality in PSNR for the single and multi layer implementations. The PSNR for a received piece is calculated by averaging the PSNR of all 64 frames contained within a piece for the multi layer implementation (the piece structure is described in Section 2.2). For the single layer approach, one piece contains in average 16 frames providing similar quality as all four layers of the multi layer approach (the same piece size is used for both implementations). The received PSNR is calculated by summating the PSNR of the frames in the received pieces and dividing it by the total number of frames for the time slot.

In Figure 2.11 the results for the second scenario are displayed. During the initial phase where the quality is gradually increased, the received quality is clearly better for the multi layer implementation, as the base layer already provides a decent quality, while the single layer implementation only receives a part of the frames. However, when the highest quality playback is achieved (download of all pieces), the single layer approach shows slightly better results. This is due to the fact that a better PSNR can be achieved with single layer encoding at the same bit rate, as the multi layer encoding has a certain overhead in terms of coding efficiency (8.6 % for the example test sequence).

Figure 2.12 shows the results for the third scenario. As the available bandwidth is not sufficient to constantly download all pieces, the single layer implementation has always a lower average PSNR than the multi layer implementation.

The results of scenario four in Figure 2.13 are similar. As one of the seeders leaves the swarm after 40 seconds, the average PSNR for the single layer approach remains rather low, while the multi layer approach can fall back to the lower layers and in comparison does not lose much in terms of average PSNR.

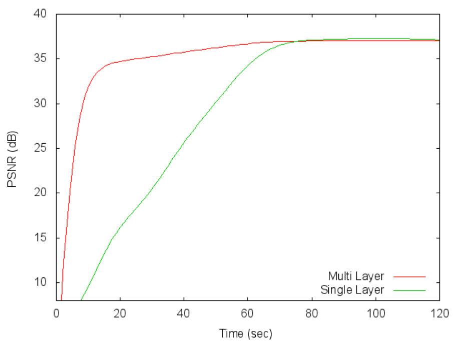


Figure 2.11: Quality Comparison Series: Scenario 2

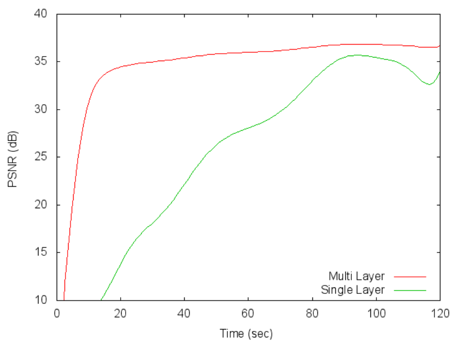


Figure 2.12: Quality Comparison Series: Scenario 3

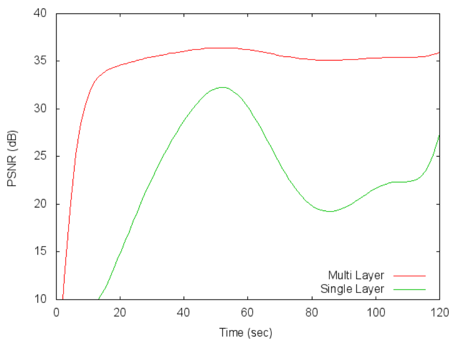


Figure 2.13: Quality Comparison Series: Scenario 4

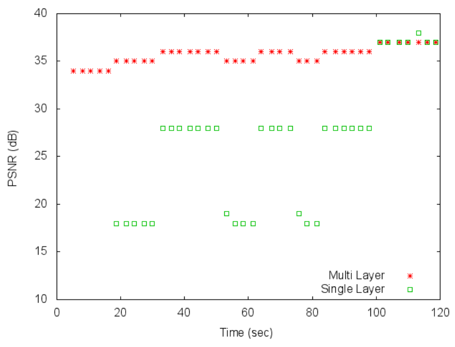


Figure 2.14: Quality Comparison Series: Scenario 5

In Figure 2.14 the results for the fifth scenario are illustrated. Due to the leaving of one of the seeders after 30 seconds the average received quality of the single layer approach stays rather low. However, after the joining of the new seeder the quality for both implementations is increased and at the end of the scenario the quality for the single layer approach is slightly better, as all pieces are received in time.

Overall, the multi layer implementation has shown clearly better performance in terms of received quality during this test series. The single layer approach can only provide better quality if there is always more than sufficient bandwidth to download all pieces in time and no fluctuations occur. Additionally, for this evaluation we have only considered rather high bandwidth scenarios where most of the pieces can be downloaded in time. In low bandwidth scenarios the performance of the single layer approach gets even worse compared to the multi layer approach (only a smaller part of the frames is received in time).

2.6. Conclusions and Future Work

In this chapter we presented a framework for the provision of scalable content in a fully distributed P2P system. We explained the selection of the scalable layers and the mapping of the layers to units and subsequently Bittorrent pieces. We described both the producer- and consumer-side architecture. The producer-side architecture includes the encoding of the SVC bitstream and splitting into layers, the packetizing of the base layer and the audio stream into a suitable container format, the creation of the scalability metadata, and the ingest of the content into the P2P system. On the consumer-side, we presented the modules for retrieving and consuming the content. The retrieved content is provided to the demuxer utilizing an HTTP socket and the demuxed audio and video streams are forwarded to the media consumption solution using RTP. The de-packetizing and decoding of the SVC content is subsequently performed by our customized SVC tools. We evaluated the performance of our SVC-enabled P2P system in order to assess the performance of our piece-picking algorithm and to compare the new multi layer implementation to the already existing single layer implementation. The results show that, after the initialization phase, the layered piece-picking algorithm can efficiently utilize the available bandwidth. Additionally, the layered implementation greatly reduces the start-up delay and is robust to churn. Furthermore, we showed that the multi layer implementation outperforms the single layer implementation when the bandwidth is restricted or bandwidth fluctuations occur.

3

Deftpack: A Robust Piece-Picking Algorithm for Scalable Video Coding in P2P Systems

Videos are nowadays provided in different qualities to ensure that various types of end-user terminals can be supported. Layered coding schemes allow the encoding of content into several qualities within a single stream, providing the viewer with the best quality depending on his needs and capabilities. In the previous chapter, Chapter 2, we have presented the design of the Next-Share BitTorrent-based P2P system. We have discussed the design of the producer- and consumer-side architecture, and presented an evaluation of a download algorithm based on a modified version of the G2G algorithm [125]. After several deployment and experimentation of the system, mainly in the Living Lab [24], we have determined that an improvement of the download algorithm was required. In this chapter we propose a novel algorithm for downloading such layered content over P2P networks called Deftpack. which we have integrated into the Next-Share Bittorrent-based P2P system.

Due to the diverse capacities of end-user terminals for multimedia content like HDTV sets, notebooks, and mobile phones, and the heterogeneity of their network connections, the provisioning of content in a single quality is not sufficient anymore. Although the content can be provided multiple times in different qualities, such an approach is inefficient and leads to a waste of bandwidth. An alternative is provided by *layered coding schemes*, which provide the content in multiple qualities within a single bitstream, see Chapter 2. With layered content, the problem of selecting the best sustainable quality is introduced, increasing the complexity of the *piece-picking* algorithm responsible for determining the best order to download the pieces of the required layers before their playback deadlines. As the download bandwidth available to a peer is limited, the piece-picking algorithm has to decide whether to focus on downloading pieces needed to ensure continuous playback, or on downloading pieces of higher layers to increase the playback quality. Deciding when to increase quality is not trivial as it involves the risk of not being able to maintain the new playback quality, thus reducing the overall quality of experience (QoE) [165].

Solutions for retrieving layered content over P2P networks have been extensively investigated in the last years [113, 116, 117, 123, 124, 139], but they do not simultaneously address the quality dimensions of reducing the number of stalls, minimising the frequency of quality changes, and maximizing the effective usage of bandwidth. In addition, our algorithm has been integrated into the open source P2P system Next-Share [69], which provides a BitTorrent-based solution for distributing layered content taking advantage of existing BitTorrent communities. Although the architecture and algorithms for supporting layered content within Next-Share are codec-agnostic, the system has been implemented using Scalable Video Coding (SVC).

The remainder of this chapter is organized as follows. Section 3.1 provides an overview of the problems arising when selecting pieces of layered content for download. In Section 3.2 related work is discussed. In Section 3.3 the design ideas and details of our new layered piece-picking algorithm are described. Section 3.4 describes the simulation settings used to evaluate our new piece-picking algorithm, while Section 3.5 compares its performance to that of previous algorithms. Finally, Section 3.6 concludes the chapter.

3.1. Problem Statement

In this chapter we assume the video streams to be distributed by a P2P system to be encoded using the Scalable Video Coding (SVC) scheme. In SVC, the video bitstream consists of the H.264/AVC-compatible *base layer* and spatial or quality *enhancement layers*. The scalability of the video codec allows changes of the resolution and the visual quality by simply adding or discarding enhancement layers for the decoding process. While the base layer provides the minimum quality and can be decoded on its own, every enhancement layer requires all lower layers to be available for decoding. An example of a four-layer encoded stream is presented in Figure 3.1, where BL stands for base layer, and EL1-3 for the enhancement layers. The time slots on top of the figure represent the time intervals at the end of which the playback quality can be changed.

The main goal when designing a layered piece-picking algorithm is to provide the best possible quality for the following time interval and still ensure that all pieces are received in time for playback. Therefore, the P2P client needs to receive the pieces for every time slot from the lowest layer up to the desired playback quality to ensure that the content can be processed by the decoder. Thus, the order in which layers are received is crucial, as high-layer pieces cannot be processed if the corresponding pieces of the lower layers have not been received as well.

When layered content is transmitted through P2P systems, four major problems may arise. First of all, *stalling* may occur, that is, the decoder may not have data available for continuous decoding of the video stream. Stalling occurs when data from the base layer is not received in time for playback. This can happen if the peer's download capacity is not sufficient, or if the available download bandwidth is not smartly used to download the more important pieces first in highly dynamic environments such as P2P systems. This behaviour is particularly evident if the base layer is only prioritized over the enhancement layers of the same time slot, but not over the enhancement layers from previous time slots. An example of this situation is presented in Figure 3.1, where, given the playback position at time instant *t*, the playback will stall at the end of time slot *t* because the base layer in time slot *t+1* is not available (but the enhancement layers are).

Second, layered piece-picking algorithms may not make optimal use of the available bandwidth, introducing a *bandwidth usage* problem. These algorithms usually focus on the download of pieces within a sliding window that contains pieces for a limited number of time slots starting from the current playback position. However, if only the pieces for the near future are considered, the bandwidth may not be fully utilized if the network conditions are good and the peer has a download link capacity higher than what is needed for the playback of the highest available quality. As the network conditions in P2P systems are often very dynamic, the bandwidth should be utilized as much as possible, possibly for pieces further in the future.

Third, *bandwidth waste* may occur if pieces from higher layers are downloaded but are never used for playback. Piece-picking algorithms usually try to increase the quality as soon as possible, trying to fully utilize the available download bandwidth. Therefore it can happen that pieces are often selected for download even though their playback deadlines are already close. In such cases, the pieces may be received after their playback deadlines and thus be discarded because they are useless for the current playback. In Figure 3.1, if the playback would reach time slot *t+5* and the corresponding data of the enhancement layer EL1 for that slot would still be missing, (e.g., because it is requested from a slow peer), the player will only be able to display the base layer, wasting the bandwidth that has been invested in the download of the second enhancement layer (EL2).

The last problem regards the frequency of *quality changes* that occur during playback. Enhancement layer pieces are usually downloaded as soon as there is sufficient bandwidth, and as a consequence, the viewing quality may frequently be changed. Such frequent quality changes should be avoided, as they lead to a worse viewing experience than viewing the content at a lower, but constant, quality [165]. As an example, in Figure 3.1 it is better to avoid displaying the EL2 layer in time slot *t+8*, thus having a continuous playback of EL1 between *t+7* and *t+9*, rather than switching quality every time slot.

To address all of these problems, we have developed a new layered piece-picking algorithm that is described in Section 3.3.

3.2. Related Work

Over the last few years, P2P video on demand and live streaming have been widely investigated. The most recent work converges on the opinion that the usage of layered coding solutions such as SVC can be beneficial for P2P networks. Several

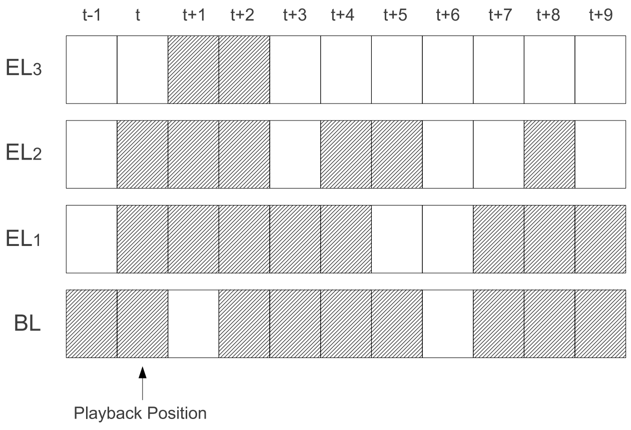


Figure 3.1: An overview of the problems arising with SVC

solutions have been presented on how to design and implement layered coding support in P2P networks. In [123] a new approach that takes advantage of SVC as well as network coding is presented. Our approach differs since we only focus on the download algorithm relying on the existing Bittorrent [5] overlay and communities. In [82] an algorithm using a ”zigzag scheduling” approach is presented. The zigzag-like piece-picking algorithm defines a zigzag priority order on the pieces within a sliding window that contains the pieces with deadlines in the near future, prioritizing lower-layer pieces over higher-layer pieces. Using this zigzag order to sort the pieces according to their priority, the pieces are subsequently selected for download as long as there is sufficient download bandwidth available. Although the results show the advantage of using layered coding technology in P2P, their approach consists of a stream divided into 50 layers with a 8 kbit/s average bit-rate reaching a total of 400 kbit/s, causing many quality changes. Changing viewing quality has a negative effect on the user-perceived QoE [165], hence one of the main goals of this chapter is to show how to reach HD quality using as few layers as possible to minimize the number of quality changes during playback. In this chapter we compare our algorithm with the ”zigzag scheduling” when applied to four layers.

In [117] a complete new system has been designed and implemented focusing on robustness rather than on offering highly different bit-rates for different devices/connections. The paper focuses on the arriving requests of the peers from the network and presents a smart algorithm for scheduling. This design relies on the adoption of the algorithm by all the peers in the system and on the good nature of the seeders. Furthermore, there is no proposed algorithm for retrieving pieces in an efficient manner.

The deadline-based algorithm used in the Next-Share system [69], which is very simple and has a low complexity, downloads pieces according to their playback deadline, i.e., pieces with an earlier deadline are downloaded before pieces with a later deadline. This kind of algorithm is usually applied for single-layer streaming and ensures that the pieces are received in time for playback. When applied to layered content, it first downloads all of the base layer pieces according to their deadline, then continues downloading the pieces for the first enhancement layer, and so on, as long as there is bandwidth available for higher-layer pieces.

The KP-based piece-picking algorithm [90] is based on algorithms for solving the knapsack problem (KP) [118], which is a problem in combinatorial optimization. As the piece-picking of layered content and the knapsack problem are very similar, the KP-based algorithm reuses approaches to solve the KP for addressing the layered piece-picking problem. The algorithm first calculates the priority for all enhancement layer pieces based on a utility formula (see Equation 3.1), which takes the layer weights (lower enhancement layers are required for the decoding of higher enhancement layers), the remaining time until the playback deadline, and the download probability into account. After calculating the utility for all enhancement layer pieces within the sliding window that are not yet selected for download, these pieces are sorted according to their utility. From this sorted list the pieces are selected for download as long as there is download bandwidth available.

These last two algorithms provide good solutions for different scenarios, protocols and network conditions, and are considered for the design of our new piece-picking algorithm in the following section.

3.3. Deftpack

The layered piece-picking algorithm presented in this chapter, Deftpack, is based on a single-layer VoD algorithm to retrieve the base layer, and on the knapsack problem-based algorithm [118] for retrieving the enhancement layers. Deftpack divides the pieces to be downloaded into five sets, and decides from which one to select them, following the order of importance of the sets, which is indicated by the numbers in Figure 3.2. As the sizes of the sets are not relevant to their importance, Figure 3.2 does not show their real proportionality. Every set has a horizontal size, a range of pieces that represents a specific time window in the stream, and a vertical size, covering a number of layers.

The horizontal boundaries of the high-priority set (set 1 in Figure 3.2), which is also called the sliding window, are determined by the current playback position in the stream and a certain size that might vary depending on the content duration. The right end of the high-priority set determines the beginning of the mid-priority set (set 2 in Figure 3.2), the size of which is set to four times the size of the sliding window. The remaining pieces, from the end of the mid-priority set until the end of the stream, define the low-priority set (set 3 in Figure 3.2). The *active layers* are defined as the layers that are currently used for playback, and they define the vertical boundary of the mid- and low-priority sets. In the example of Figure 3.2, the number of active layers is three.

Once all the pieces from the high-, mid-, and low-priority sets have been downloaded, Deftpack will select pieces from the lowest-priority set (set 4 in Figure 3.2). This gives a chance of improving the playback quality towards the end of the stream, increasing the QoE [165]. The past set (set 5 in Figure 3.2), represents all the pieces, for all the layers, from the beginning of the stream until the current playback po-

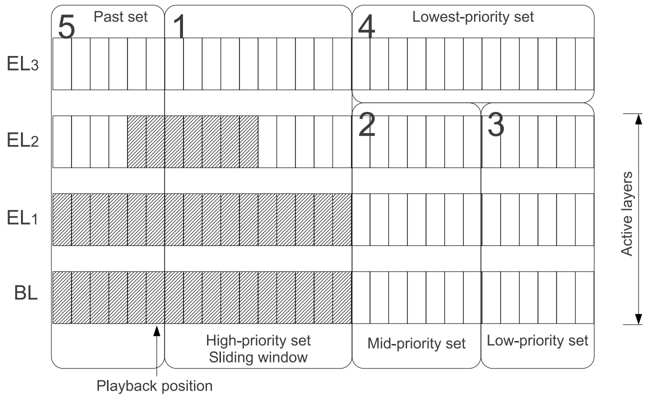


Figure 3.2: The five priority sets distinguished by Deftpack. The grey pieces represent downloaded pieces.

sition. Downloading pieces in the past set increases their availability in the swarm, and allows the user to watch the stream at its best quality once the download is finished.

Except for the high-priority set, pieces within all the sets are selected following the rarest-first approach. For the mid-, the low-, and the lowest-priority sets, this behaviour guarantees that the spare bandwidth, that is not being used for pieces from the sliding window, is invested in retrieving any pieces that will probably be needed in the future, has it has been proven with rarest-first based algorithms [75].

In the high-priority set, the pieces of the base layer are always downloaded in order, based on their playback deadlines, before pieces of the enhancement layers. Pieces of the enhancement layers are downloaded in decreasing order of their utility. The *utility* at time slot t_k of piece j (corresponding to time slot t_j) of layer i is defined as:

$$u_{ijk} = \frac{lw_i \times dp_i}{(t_j - t_k)^{\alpha}}, \tag{3.1}$$

where lw_i is the weight of layer i, dp_i is the probability that a piece can be downloaded under the current network conditions in time for playback [118], and α is a positive constant.

The low computational complexity of Deftpack when downloading only the base layer greatly reduces the computational requirements for hardware-limited devices, as these devices will never be able to display higher qualities due to their network connection or the resolution of their display. For devices that can also process the enhancement layers, the more complex KP-based algorithm is applied to display the best possible quality. The complexity of the algorithm is give by:

$$O(Deftpack) = \begin{cases} O(n) & \text{BL} \\ O(m \cdot n \cdot log(\max(m,n))), & \text{ELs} \end{cases} \tag{3.2}$$

where n and m are the horizontal and vertical size of the sliding window, respectively.

3.3.1. Dynamic sliding window

Previously proposed piece-picking algorithms for layered content use a sliding window to identify urgent pieces that need to be retrieved sooner than others for playback [82] [117]. This sliding window usually has a fixed horizontal size, related to a time range, and a fixed vertical size that includes all the available layers. Deftpack, on the other hand, uses a dynamic approach in order to optimize piece retrieval depending on the available resources.

To guarantee a near-zero-stall algorithm, and therefore continuous playback, the size of the sliding window is adjusted to ensure a stable download rate for the active layers. In contrast to other adaptive window approaches [67], where the window grows for fast peers and shrinks for slow peers to increase piece availability in the swarm, our algorithm will increase the window sizes for slow peers and shrink them for fast peers. As a consequence, slow peers avoid switching to a higher quality when their download speed is barely sufficient to guarantee the playback of the active layers, and fast peers sooner start downloading pieces of the enhancement layers.

We adjust the window size by monitoring the speed of data arrival of the pieces of the currently active layers as follows:

$$w_t = \left\lfloor \frac{k \times d_{al}}{b_{al}} \times w_{t-1} \right\rfloor, \quad b \leq w_t \leq w_{max}, \tag{3.3}$$

where w_t is the window size at time t, d_{al} is the average download speed of data of the active layers, b_{al} is the cumulative bitrate of the active layers, and k is a positive value that, starting from 1, increases by 0.1 for every 10 stalls. The window will never be smaller that a pre-defined size b (i.e., the player's buffer size in pieces), and never larger than the maximum preferred window size, w_{max}, to avoid retrieving all the remaining pieces in-order.

While the right end point of the sliding window is the same for all layers, the starting point is the same only for the active layers, as it might vary when Deftpack decides to increase quality. When increasing quality, it is unlikely to perform a smooth quality switch for the time slot immediately following the current playback position. Therefore, we move forward the window's starting point for the *target layer*, defined as the layer we are moving to, reducing bandwidth waste and increasing the probability of a successful quality switch. Deftpack will attempt to increase playback quality if the current download rate is higher than the *target bitrate*, defined as the playback bitrate that is achieved if the quality switch occurs. If the current download rate is 20% higher than the target bitrate, the window of the target layer will start one time slot after the current playback position, if it is between 10% and 20% higher, it starts two time slots later, and otherwise, it starts three time slots later. If the download speed suddenly increases, offering enough capacity to support an increase in quality of more than one layer, the same policy is applied to all the desired layers.

Figure 3.3 presents an example of Deftpack increasing the quality from EL1 to EL2, the target layer. Here, the download rate is between 10% and 20% higher than the playback rate of BL+EL1+EL2, and therefore a gap of two time slots is set. As

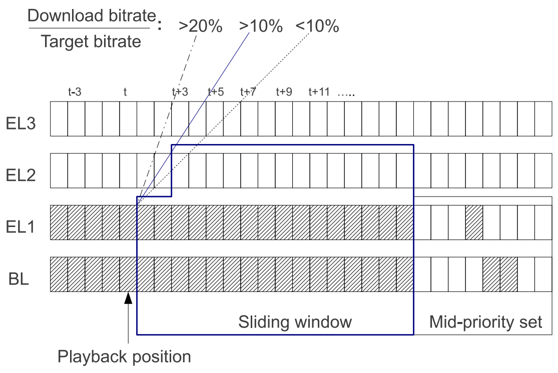


Figure 3.3: An example of the dynamic window used by Deftpack when increasing quality.

a second example, if in Figure 3.3 at instant *t*, the download rate would support the playback of all four layers, the window for EL2 will start at *t+3*, while the window for EL3 will start at *t+5*. This behaviour ensures a minimum amount of wasted bandwidth when increasing quality.

3.3.2. Meeting the Design Goals

This section describes how the problems described in Section 3.1 are addressed by Deftpack. The occurrence of *stalling* events has been greatly reduced by prioritizing the base layer, and by adjusting the size of the sliding window. This behaviour ensures that a continuous playback is guaranteed even when the available download bandwidth is low. Additionally, it increases the availability of pieces from the base layer in the swarm, reducing the risk of stalling caused by the high competition that can occur with rare pieces. Deftpack optimizes the *bandwidth usage* of a peer by avoiding investing available bandwidth beyond the active layers, which might risk downloading pieces that might never be used for playback. As the algorithm downloads not only the pieces within the high-priority set, but also uses the remaining bandwidth to download pieces from the sets presented at the beginning of this section, the download link's capacity is always fully used. This behaviour reduces the overall *bandwidth waste* of a peer. Furthermore, it is also reduced by the fact that our piece-picking algorithm only downloads pieces that can be realistically downloaded in time for playback. Frequent *quality changes* are avoided when possible as our piece-picking algorithm monitors the download bandwidth and only decides to switch to higher layers if the next enhancement layer can be sustainably downloaded. Utilizing this conservative approach, switches to higher qualities occur rarely and only when sufficient download bandwidth is available, while switches to lower layers only occur if the download bandwidth suddenly decreases.

Finally, it should be noted that Deftpack does not try to find the best possible solution considering each of the problems separately. Instead, the algorithm tries to find a feasible solution for each of the problems while ensuring that all of the other problems are still considered.

Table 3.1: Peers bandwidth distribution

Class	**Distribution** [%]	**Downlink** [kbit/s]	**Uplink** [kbit/s]
DSL1	21.4	768	128
DSL2	23.3	1,500	348
Cable1	18	3,000	768
Cable2	37.7	10,000	5,000

Table 3.2: Layer scheme used in simulations

Layer	**Layer bitrate** [kbit/s]	**Cumulative bitrate** [kbit/s]
BL	400	400
EL1	400	800
EL2	800	1,600
EL3	1,600	3,200

3.4. Experimental Setup

In this section we describe the experimental setup used to evaluate the performance of our Deftpack algorithm and for comparing it to three existing algorithms, ZigZag, KP-based, and NS-Core, described in Section 7.1.

3.4.1. Simulator Setup

To perform an evaluation of our algorithm and compare it with the existing solutions, we have modified the discrete event-based Microsoft Research BitTorrent simulator [64], implementing only the investigated piece-picking algorithms. For the results presented in this chapter we assume that all peers are connected to each other and that for all the investigated algorithms the playback will start as soon as an initial set of pieces has been downloaded. Throughout the rest of the chapter we will refer to this initial set of pieces as the buffer.

For the peers participating in the swarm we show in Table 3.1 the distribution of their network capacities, dividing them into classes. The bandwidth distribution in Table 3.1 is based on the results of measurements [84, 103] and traces [142]. Peers slower than DSL1 connections have been omitted, adding their ratio to DSL1 peers, since recent measurements from public and private communities [122] have shown that the average Internet connection speed has increased significantly in the last few years. In Table 3.2 we show the bitrates of the layers used in the simulations, which for simplicity are approximations of the bit-rates presented in Chapter 2.

3.4.2. Scenarios

In our simulations, we assume a peer leaves the swarm if either it has completely downloaded all the layers, if its playback has reached the end of the stream, or if its overall participation time in the swarm since its playback started exceeds the video

Table 3.3: Simulation settings

Parameter	Value
Video duration	60 minutes
Piece size	128 KB
Upload slots	5
Initial window size	20 pieces
Maximum window size	50 pieces
Buffer size	10 pieces
Seeders upload capacity	6 Mbit/s

duration by 50%. This occurs especially with slow peers when they experience too many stalls during the playback. If a peer experiences more than 50% of stalling time but its playback is currently progressing, it will leave the swarm at the next stall.

We focus on the behaviour of the investigated algorithms in scenarios with only one seeder, the initial content provider, forcing the peers to cooperate with each other. We also present results of running over-seeded scenarios, in which the initial seeders can support the swarm bandwidth demand. The scenarios we use to evaluate the performance of Deftpack can be grouped into three categories, depending on the peer arrival distribution and arrival rate.

In the *steady state* scenario, 500 peers join the system according to an exponential interarrival-time distribution during the first 30 minutes with an arrival rate of 0.11 peers per second. After that time, every peer that leaves is immediately replaced by a peer of the same class, thus maintaining a constant number of peers. In this scenario, the content provider is the only initial seeder, causing the need for peers to collaborate with each other. We simulate 10 hours of operation, which gives results that are sufficient to draw conclusions.

In the *flashcrowd* scenario, the content provider is the only initial seeder, and peers arrive at an exponentially decaying rate, starting at a very high arrival rate. This scenario simulates a critical situation for a P2P system with a large number of peers joining at roughly the same time, e.g., to consume the broadcast of a live event. The arrival rate is given by:

$$\lambda(t) = \lambda_0 e^{-\gamma t}. \tag{3.4}$$

In our simulations we set $\lambda_0 = 10$ and $\gamma = 1/150$. With these parameter values, 1500 peers join the swarm during the first 50 minutes.

In the *over-seeded* scenarios, hundreds of seeders never leave the system, reducing the need for collaboration between peers. The first two of these are equal to the steady-state scenario but with 150 and 300 seeders instead of 1. The third of these is equal to the flashcrowd scenario but with 150 initial seeders.

The common properties of all the scenarios are presented in Table 3.3. When simulating the same scenario with the four algorithms, we use the exact same arrival pattern of peers. Furthermore, the three algorithms against which we compare will download pieces in a rarest-first fashion from the entire stream, starting at the current playback position, if no piece can be selected from the sliding window.

3.4.3. Performance Metrics

In our performance evaluation, we use the following metrics. The *total playback bitrate* is defined as the cumulative playback bitrate of all the peers viewing the content at every second of the simulation run. The *total wasted bitrate* is defined as the bandwidth spent downloading data from the enhancement layers that is never displayed, which occurs when during playback a piece of an enhancement layer is needed but has only been partially downloaded. We only consider the wasted bandwidth for pieces from the sliding window, ignoring partial pieces that have been selected from the other sets.

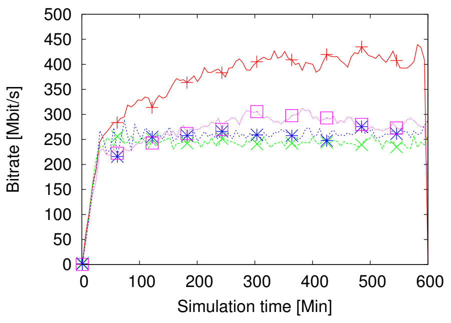
If a peer does not download a base layer piece before its deadline, the playback will stall. For every stall occurrence, the playback pauses and resumes only after the initial buffer has been restored. We also monitor the *stalling time*, defined as the time spent by a peer waiting for the playback to resume from a stall. We report the occurrence of *stalls over time* to show how it influences the playback bitrate and pieces availability in the swarm. As stall occurrences are closely related to the availability of pieces of the base layer to be processed by the decoder, we monitor the *piece availability* of these pieces within the sliding window. Finally, we define the *abort percentage* as the percentage of peers aborting the download because they stalled for 50% of the video duration. We consider this as the most important metric to determine the QoE.

3.5. Experimental Results

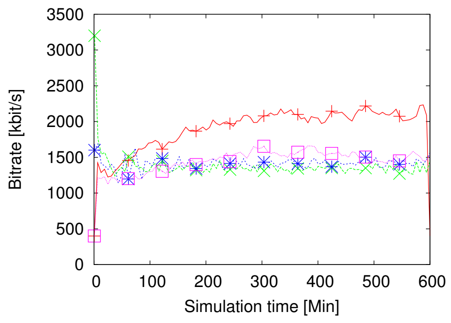
In this section we evaluate the performances of Deftpack by means of simulations. In Section 3.5.1 we present the results for the steady state scenario, in Section 3.5.2 for the flashcrowd scenario, and in Section 3.5.3 for the over-seeded scenarios.

3.5.1. Steady State Scenario

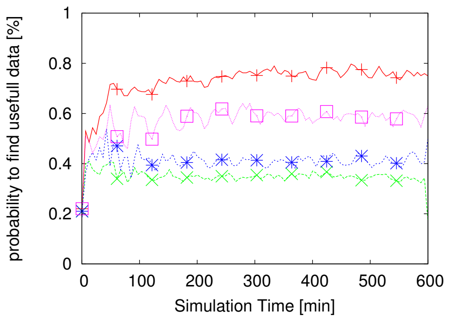
Figure 3.4 presents the behaviour of the algorithms in a steady state. Figure 3.4(a) shows the cumulative playback bitrate for all the peers participating in the swarm for every second of the simulation run. Without the condition for moving to a higher quality, Deftpack would drastically outperform the other algorithms, but this would come at the cost of more frequent quality changes, and therefore, a more unstable playback. Figure 3.4(b) shows the average playback bitrate of all the peers in the swarm. It is interesting to note how the ZigZag and the KP-based algorithms start at very high rates, 1.6 and 3.2 Mbit/s respectively, as all the layers are retrieved before the initial buffer is filled, and then quickly stabilize at lower rates. On the other hand, the NS-Core and Deftpack algorithms start at a lower rate, 400 Kbit/s, but reach a higher stable rate during the steady state. We do not show the wasted bandwidth, as it turns out to be very minimal (0.3% for the NS-Core and KP-based algorithms, 0.2% for Deftpack, and 1.1% for the ZigZag algorithm).



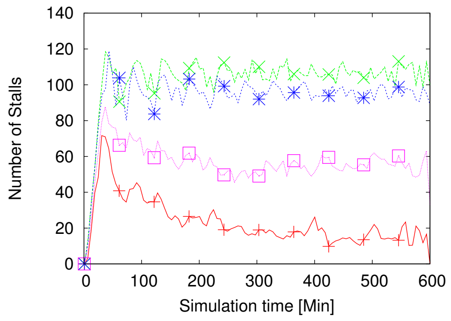
(a) The cumulative playback bitrate of all the peers



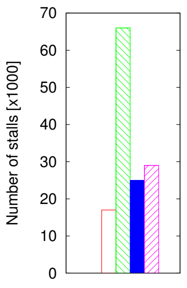
(b) The average playback bitrate of all the peers



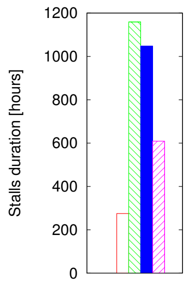
(c) The availability of pieces of the base layer in the high-priority set



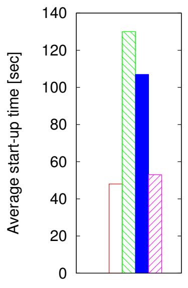
(d) The number of peers with a stalled playback



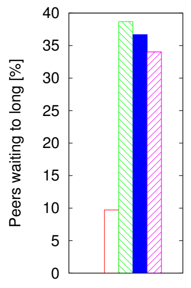
(e) Number of stalls



(f) Stall duration



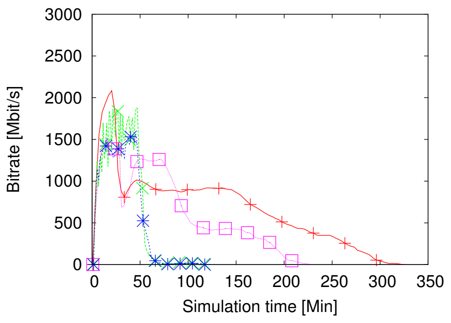
(g) Start-up delay



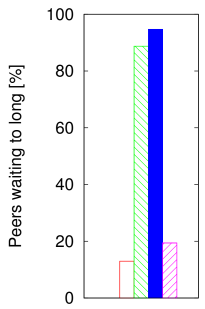
(h) Abort percentage

Figure 3.4: The performance of the four investigated algorithm in the steady state scenario. Deftpack (—+— □) ; KP-Based (- -×- - ▨); ZigZag (-✱- ■); NS-Core (··□·· ▨).

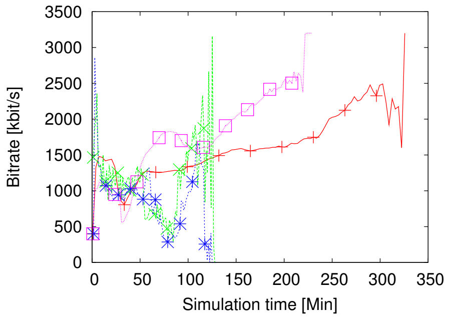
Figure 3.4 shows that with Deftpack, the availability of pieces of the base layer in the high-priority set is much higher than with the other algorithms, leading to a much lower number of stalled peers over time (Figure 3.4(d)), a (much) lower total number of stalls (Figure 3.4(e)), and much less time spent by the peers refilling



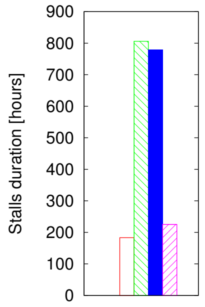
(a) The cumulative playback bitrate of all the peers



(b) Abort percentage



(c) The average playback quality from all peers



(d) Stall duration

Figure 3.5: The performance of the four investigated algorithm in the flashcrowd scenario. Deftpack (—+— □) ; KP-Based (- ✕ - ▧); ZigZag (- ✱ - ■); NS-Core (··□·· ▨)
.

their buffers after stall occurrences (Figure 3.4(f)). Figure 3.4(g) shows the average start-up delay, while Figure 3.4(h) shows the percentage of peers leaving the swarm after waiting for more than 50% of the video duration.

It has to be noted that the KP-based algorithm provides the best playback quality to fast peers, but the poorest quality to slow peers. The main idea behind Deftpack is to be able to serve a broader audience, even if fast peers consume the content at a lower quality, giving a greater chance of a decent playback to slow peers.

Clearly, Deftpack outperforms the compared algorithms, optimizing the distribution of the base layer (Figure 3.4(c)), reducing the time for resuming the playback (Figure 3.4(f)), and therefore having the largest number of peers reaching the end of the stream (Figure 3.4(h)), while still providing the highest QoE (Figures 3.4(a) and 3.4(b)).

	Deftpack	**KP-Based**	**ZigZag**	**NS-Core**
Steady State scenario (150/300 seeders)				
PlaybackBW [Mb/s]	325 / 348	251 / 257	198 / 233	280 / 303
WastedBW [%]	0.21 / 0.34	0.39 / 0.35	1.77 / 1.36	0.31 / 0.29
Stalls [x1000]	15 / 12	129 / 95	76 / 55	21 / 17
Stall time [x1000s]	622 / 485	4177 / 3940	4439 / 3747	1237 / 905
Abort [%]	7 / 5	44 / 60	62 / 63	22 / 17
Start-up [s]	36 / 32	105 / 142	119 / 129	36 / 32
Flashcrowd scenario (150/300 seeders)				
PlaybackBW [Mb/s]	369 / 622	572 / 763	654 / 631	605 / 504
WastedBW [%]	0.21 / 0.10	0.53 / 0.30	0.48 / 1.14	0.43 / 0.17
Stalls [x1000]	15 / 8	75 / 44	13 / 18	12 / 11
Stall time [x1000s]	657 / 358	3056 / 2301	805 / 2148	789 / 568
Abort [%]	10 / 4	93 / 62	19 / 61	38 / 14
Start-up [s]	68 / 43	211 / 188	54 / 167	46 / 43

Table 3.4: Simulation results of the four investigated algorithms in the over-seeded scenarios. (Best results are highlighted.)

3.5.2. Flashcrowd Scenario

Figure 3.5 shows how the investigated algorithms perform in the flashcrowd scenario. Figure 3.5(a) shows the cumulative playback rate of all the peers in the swarm. During the first 50 minutes, after which no more peers join the swarm as then the arrival rate is effectively equal to zero, the KP-based and the ZigZag algorithms reach a high playback rate. This behaviour is caused by the way the buffer is filled before the playback can resume after a stall, as those two algorithms will download pieces from enhancement layers before the buffer has been successfully filled. As a consequence, during the same time period, they present 70% more stalls than Deftpack and the NS-Core algorithm.

While the cumulative playback rate decreases as peers are leaving the system (Figure 3.5(a)), the average playback bitrate increases as the initial seeder can support their bandwidth request (Figure 3.5(c)). Figure 3.5(b) shows how almost all the peers leave the swarm because of waiting too long when using the KP-base and the ZigZag algorithms. The NS-Core algorithm performs well compared to Deftpack, reaching a higher average playback rate (Figure 3.5(c)), but with a relatively higher bandwidth waste (40%), and for fewer peers (Figure 3.5(a)). In comparison to Deftpack, the NS-Core algorithm has a lower number of stall events (10%), but they last longer (Figure 3.5(d)), which causes roughly double the number of peers to leave the system before completing their playback (Figure 3.5(b)).

3.5.3. Over-seeded Scenario

In this section, the simulation results for the over-seeded scenario are presented to show how Deftpack performs in comparison to the other algorithms in non-critical situations. The results of those simulations are presented in Table 3.4. The most important results are the cumulative playback bitrate and the abort rate (columns *PlaybackBW* and *Abort* in Table 3.4).

Deftpack clearly outperforms the other algorithms in steady state over-seeded scenarios. In the flashcrowd scenario with 150 seeders, Deftpack seems to perform generally worse than the other algorithms as it has a lower cumulative playback bitrate and longer start-up times. However, these results are misleading as with Deft pack, most of the peers actually reach the end of the stream (see the *Abort* column in Table 3.4), even if displaying a lower quality. This scenario can be compared with the one presented in Section 3.5.2 where it is explained how this behaviour provides a higher QoE for the viewer.

3.6. Conclusion

In this chapter we present a new piece-picking algorithm for downloading layered content in P2P networks, called Deftpack, which reduces stalling of the video playback while ensuring that the user receives the best possible quality. In our simulations, Deftpack is compared to other piece-picking algorithms in cases in which the content is under- and over-provisioned. In the under-provisioned scenarios, simulating real-world situations, Deftpack outperforms the other algorithms by reducing stalls and providing the highest QoE to most of the peers. For the over-provisioned scenarios, in which multiple seeders provide more than sufficient bandwidth to all peers, Deftpack clearly outperforms the other algorithms by providing the best average playback bandwidth while having the majority of the peers reaching the end of the playback.

Throughout all the simulations, it is shown how Deftpack allows peers with heterogeneous connections to share the same content in an efficient way. Overall, Deftpack significantly outperforms existing solutions, reducing the number of stalls, and providing the best overall QoE to the user. Furthermore, Deftpack has been integrated into the Next-Share system [69], which is to our knowledge the first open-source P2P system with full SVC support.

4 The Peer-to-Peer Streaming Peer Protocol (PPSPP)

The Peer-to-Peer Streaming Peer Protocol (PPSPP) is a protocol for disseminating the same content to a group of interested parties in a streaming fashion. PPSPP has been recently published as an official IETF Internet standard [37] as RFC7574 [60]. It supports streaming of both prerecorded (on-demand) and live audio/video content. It is based on the P2P paradigm, where clients consuming the content are put on equal footing with the servers initially providing the content, to create a system where everyone can potentially provide upload bandwidth. It has been designed to provide short time till playback for the end user, and to prevent disruption of the streams by malicious peers. PPSPP has also been designed to be flexible and extensible. It can use different mechanisms to optimize peer uploading, to prevent freeriding, and to work with different peer discovery schemes (centralized trackers or Distributed Hash Tables). It supports multiple methods for content integrity protection and chunk addressing. Designed as a generic protocol that can run on top of various transport protocols, it currently runs on top of UDP using Low Extra Delay Background Transport (LEDBAT) for congestion control [149]. LEDBAT is a ”lower-than-best-effort” delay-based congestion control that fully utilises the available network bandwidth without interfering with the higher priority traffic. Hence P2P users redistribute the content for a longer time, since keeping a P2P client active does not have any drawback. PPSPP, and its reference implementation Libswift, was the first P2P protocol to move to this new congestion control and to distribute the content using UDP. This, and other characteristics such as a low time till-playback and its compact way of addressing pieces of content, are the key characteristics that make it stand out over the other protocols that were proposed in the Peer-to-Peer Streaming Protocol (PPSP) working group.

This chapter presents the official PPSPP draft published by the IETF. For clarity, only the core sections are reported, and many low-level details are omitted, such as the description of the terminology, the on-wire definition of the messages, the description of the message options, the description of the UDP encapsulation, and on-wire examples of peer communication. For full details we refer the reader to the official PPSPP document, RFC7574 [60]. The three authors of the standard, including the author of this thesis, have collaborated in designing, implementing, testing, and describing the protocol, fulfilling the requirements for its approval at the IETF's standard track. The PPSPP protocol was originally designed by Victor Grishchenko. Arno Bakker has guided the standardization process, and the author of this thesis has focused on the reference implementation, designing some of its algorithms, and testing it in various environments. This chapter introduces the protocol that is referenced and used as the main P2P protocol throughout the rest of this thesis. In the following two chapters we first describe its reference implementation, Libswift, in Chapter 5, and then analyse its performance and compare it to the most popular P2P protocols in Chapter 6.

Purpose

PPSPP has been designed to provide short time-till-playback for the end user and to prevent disruption of the streams by malicious peers. Central in this design is a simple method of identifying content based on self-certification. In particular, content in PPSPP is identified by a single cryptographic hash that is the root hash in a Merkle hash tree calculated recursively from the content [58, 121]. This self-certifying hash tree allows every peer to directly detect when a malicious peer tries to distribute fake content. The tree can be used for both static and live content. Moreover, it ensures only a small amount of information is needed to start a download and to verify incoming chunks of content, thus ensuring short start-up times.

PPSPP has also been designed to be extensible for different transports and use cases. Hence, PPSPP is a generic protocol that can run directly on top of UDP, TCP, and other protocols. As such, PPSPP defines a common set of messages that make up the protocol, which can have different representations on the wire depending on the lower-level protocol used. When the lower-level transport allows, PPSPP can also use different congestion control algorithms.

At present, PPSPP is set to run on top of UDP using LEDBAT for congestion control [149]. Using LEDBAT enables PPSPP to serve the content after playback (seeding) without disrupting the user who may have moved to different tasks that use his network connection. Please refer to Section 8 of [60] for more detail on the UDP encapsulation.

PPSPP is also flexible and extensible in the mechanisms it uses to promote client contribution and prevent freeriding, that is, how to deal with peers that only download content but never upload to others. It also allows different schemes for chunk addressing and content integrity protection, if the defaults are not fit for a particular use case. In addition, it can work with different peer discovery schemes, such as centralized trackers and fast Distributed Hash Tables [110]. Finally, in its default setup, PPSPP maintains only a small amount of state per peer. A reference implementation of PPSPP over UDP is available [14].

The protocol defined in this chapter assumes that a peer has already discovered a list of (initial) peers using, for example, a centralized tracker [128]. Once a peer has this list of peers, PPSPP allows the peer to connect to other peers, request chunks of content, and discover other peers disseminating the same content. The design of PPSPP is based on our research into making BitTorrent [5] suitable for streaming content [59]. Most PPSPP messages have corresponding BitTorrent messages and vice versa. However, PPSPP is specifically targeted towards streaming audio/video content and optimizes time-till-playback. It was also designed to be more flexible and extensible.

4.1. Overall Operation

The basic unit of communication in PPSPP is the message. Multiple messages are multiplexed into a single datagram for transmission. A datagram (and hence the messages it contains) will have different representations on the wire depending on the transport protocol used (see Section 8 of [60]).

The overall operation of PPSPP is illustrated in the following examples. The examples assume that the content distributed is static, UDP is used for transport, the Merkle Hash Tree scheme is used for content integrity protection, and that a specific policy is used for selecting which chunks to download.

4.1.1. Example: Joining a Swarm

Consider a user who wants to watch a video. To play the video, the user clicks on the play button of a HTML5 <video> element shown in his PPSPP-enabled browser. Imagine this element has a PPSPP URL (to be defined elsewhere) identifying the video as its source. The browser passes this URL to its peer-to-peer protocol handler. Let's call this protocol handler Peer A. Peer A parses the URL to retrieve the transport address of a peer-to-peer streaming protocol tracker and swarm metadata of the content. The tracker address may be optional in the presence of a decentralized tracking mechanism. The mechanisms for tracking peers are outside of the scope of this document.

Peer A now registers with the tracker following the peer-to-peer streaming protocol tracker specification [128] and receives the IP address and port of peers already in the swarm, say, Peer B, C, and D. At this point the PPSPP starts operating. Peer A now sends a datagram containing a PPSPP HANDSHAKE message to Peer B, C, and D. This message conveys protocol options. In particular, Peer A includes the ID of the swarm (part of the swarm metadata) as a protocol option because the destination peers can listen for multiple swarms on the same transport address.

Peers B and C respond with datagrams containing a PPSPP HANDSHAKE message and one or more HAVE messages. A HAVE message conveys (part of) the chunk availability of a peer; this, it contains a chunk specification that denotes what chunks of the content Peer B and C have, respectively. Peer D sends a datagram with a HANDSHAKE and HAVE messages, but also with a CHOKE message. The latter indicates that Peer D is not willing to upload chunks to Peer A at present.

4.1.2. Example: Exchanging Chunks

In response to Peers B and C, Peer A sends new datagrams to B and C containing REQUEST messages. A REQUEST message indicates the chunks that a peer wants to download; thus, it contains a chunk specification. The REQUEST messages to Peers B and C refer to disjoint sets of chunks. Peers B and C respond with datagrams containing HAVE, DATA, and, in this example, INTEGRITY messages. In the Merkle hash tree content protection scheme (see Section 4.4.1), the INTEGRITY messages contain all cryptographic hashes that Peer A needs to verify the integrity of the content chunk sent in the DATA message. Using these hashes peer A verifies that the chunks received from Peers B and C are correct against the trusted swarm ID. Peer A also updates the chunk availability of B and C using the information in the received HAVE messages. In addition, it passes the chunks of video to the user's browser for rendering.

After processing, Peer A sends a datagram containing HAVE messages for the chunks it just received to all its peers. In the datagram to Peers B and C, it includes an ACK message acknowledging the receipt of the chunks and adds REQUEST messages for new chunks. ACK messages are not used when a reliable transport protocol is used. When, for example, Peer C finds that Peer A obtained a chunk (from Peer B) that Peer C did not yet have, Peer C's next datagram includes a REQUEST for that chunk.

Peer D also sends HAVE messages to Peer A when it downloads chunks from other peers. When Peer D is willing to accept REQUESTs from Peer A, Peer D sends a datagram with an UNCHOKE message to inform Peer A. If Peer B or C decides to choke Peer A they send a CHOKE message and Peer A should then re-request from other peers. Peers B and C may continue to send HAVE, REQUEST, or periodic keep-alive messages such that Peer A keeps sending them HAVE messages.

Once Peer A has received all content (video-on-demand use case), it stops sending messages to all other peers that have all content (a.k.a. seeders). Peer A can also contact the tracker or another source again to obtain more peer addresses.

4.1.3. Example: Leaving a Swarm

To leave a swarm in a graceful way, Peer A sends a specific HANDSHAKE message to all its peers (see Section 8.4 of [60]) and deregisters from the tracker following the tracker specification [128]. Peers receiving the datagram should remove Peer A from their current peer list. If Peer A crashes ungracefully, peers should remove A from their peer list when they detect it no longer sends messages (see Section 3.12 of [60]).

4.2. Messages

No error codes or responses are used in the protocol; absence of any response indicates an error. Invalid messages are discarded, and further communication with the peer SHOULD be stopped. The rationale is that it is sufficient to classify peers as either good or bad and only use the good ones. A good peer is a peer that responds with chunks; a peer that does not respond, or does not respond in time is classified as bad. The idea is that, in PPSPP, the content is available from multiple sources (unlike HTTP), so a peer should not invest too much effort in trying to obtain it from a particular source. This classification in good or bad allows a peer to deal with slow, crashed, and (silent) malicious peers.

Multiple messages MUST be multiplexed into a single datagram for transmission. Messages in a single datagram MUST be processed in the strict order in which they appear in the datagram. If an invalid message is found in a datagram, the remaining messages MUST be discarded.

For the sake of simplicity, one swarm of peers deals with one content file or stream only. There is a single division of the content into chunks that all peers in the swarm adhere to, determined by the content publisher. Distribution of a collection of files can be done either by using multiple swarms or by using an external storage mapping from the linear byte space of a single swarm to different files, transparent to the protocol. In other words, the audio/video container format used is outside the scope of this document.

4.2.1. HANDSHAKE

For Peer P to establish communication with Peer Q in Swarm S the peers must first exchange HANDSHAKE messages by means of a handshake procedure. The initiating Peer P needs to know the metadata of Swarm S, which consists of:

- the swarm ID of the content (see Sections 4.4.1 and 4.5),
- the chunk size used,
- the chunk addressing method used,
- the content integrity protection method used, and
- the Merkle hash tree function used (if applicable).
- If automatic content size detection (see Section 4.4.6) is not used, the content length is also part of the metadata (for static content.)

This document assumes the swarm metadata is obtained from a trusted source. In addition, Peer P needs to know a transport address for Peer Q, obtained from a peer discovery/tracking protocol. The payload of the HANDSHAKE message contains a sequence of protocol options. The protocol options encode the swarm metadata just described to enable an end-to-end check to see whether the peers are in the right swarm. Additionally, the options encode a number of per-peer configuration parameters. The complete set of protocol options are specified in Section 7 of [60]. The HANDSHAKE message also contains a channel ID for multiplexing communication and security (see Section 4.2.11). A HANDSHAKE message MUST always be the first message in a datagram.

Handshake Procedure
The handshake procedure for peer, Peer P to start communication with another peer, Peer Q, in Swarm S is now as follows.

1. The first datagram the initiating peer P sends to peer Q MUST start with a HANDSHAKE message. This HANDSHAKE message MUST contain:

    - A channel ID, chanP, randomly chosen as specified in Section 13.1 of [60].
    - The metadata of Swarm S, encoded as protocol options, as specified in Section 7 of [60]. In particular, the initiating Peer P MUST include the swarm ID.
    - The capabilities of peer P, in particular, its supported protocol versions, ”Live Discard Window” (in case of a live swarm) and ”Supported Messages”, encoded as protocol options.

    This first datagram MUST be prefixed with the (destination) channel ID 0; see Section 3.11 of [60]. Hence, the datagram contains two channel IDs: the destination channel ID prefixed to the datagram and the channel ID chanP included in the HANDSHAKE message inside the datagram. This datagram MAY also contain some minor additional payload, e.g., HAVE messages to indicate P's current progress, but MUST NOT include any heavy payload (defined in Section 1.3 of [60]), such as a DATA message. Allowing minor payload minimizes the number of initialization round trips, thus improving time-till-playback. Forbidding heavy payload prevents an amplification attack (see Section 13.1 of [60]).

2. The receiving Peer Q checks the HANDSHAKE message from Peer P. If any check by Peer Q fails, or if Peers P and Q are not in the same swarm, Peer Q MUST NOT send a HANDSHAKE (or any other) message back, as the message from Peer P may have been spoofed (see Section 13.1 of [60]). Otherwise, if Peer Q is interested in communicating with Peer P, Peer Q MUST send a datagram to Peer P that starts with a HANDSHAKE message. This reply HANDSHAKE MUST contain:

    - A channel ID, chanQ, randomly chosen as specified in Section 13.1 of [60].
    - The metadata of Swarm S, encoded as protocol options, as specified in Section 7 of [60]. In particular, the responding Peer Q MAY include the swarm ID.
    - The capabilities of Peer Q, in particular, its supported protocol versions, its ”Live Discard Window” (in case of a live swarm) and ”Supported Messages”, encoded as protocol options.

    This reply datagram MUST be prefixed with the channel ID chanP sent by Peer P in the first HANDSHAKE message (see Section 4.2.11). This reply datagram MAY also contain some minor additional payload, e.g., HAVE messages to indicate Peer Q's current progress, or REQUEST messages (see Section 4.2.7), but it MUST NOT include any heavy payload.

3. The initiating Peer P checks the reply datagram from Peer Q. If the reply datagram is not prefixed with (destination) channel ID chanP, Peer P MUST discard the datagram. Peer P SHOULD continue to process datagrams from Peer Q that do meet this requirement. This check prevents interference by spoofing, see Section 13.1 of [60]. If Peer P's channel ID is echoed correctly, the initiator Peer P knows that the addressed Peer Q really responds.
4. Next, Peer P checks the HANDSHAKE message in the datagram from Peer Q. If any check by Peer P fails, or Peer P is no longer interested in communicating with Peer Q, Peer P MAY send a HANDSHAKE message to inform Q it will cease communication. This closing HANDSHAKE message MUST contain an all zeros channel ID and a list of protocol options. The list MUST either be empty or contain the maximum version number Peer P supports, following the min/max versioning scheme defined in [RFC6709], Section 4.3.1. The datagram containing this closing HANDSHAKE message MUST be prefixed with the (destination) channel ID chanQ. Peer P MAY also simply cease communication.
5. If the addressed peer, Peer Q, does not respond to initiating Peer P's first datagram, Peer P MAY resend that datagram until Peer Q is considered dead, according to the rules specified in Section 3.12 of [60].
6. If the reply datagram by Peer Q does pass the checks by Peer P, and Peer P wants to continue interacting with Peer Q, Peer P can now send REQUEST, PEX_REQ, and other messages to Peer Q. Datagrams carrying these messages MUST be prefixed with the channel ID chanQ sent by Peer Q. More specifically, because Peer P knows that Peer Q really responds, Peer P MAY start sending Peer Q messages with heavy payload. That means that Peer P MAY start responding to any REQUEST messages that Peer Q may have sent in this first reply datagram with DATA messages. Hence, transfer of chunks can start soon in PPSPP.
7. If Peer Q receives any datagram (apparently) from Peer P that does not contain channel ID chanQ, Peer Q MUST discard the datagram but SHOULD continue to process datagrams from Peer P that do meet this requirement. Once Peer Q receives a datagram from Peer P that does contain the channel ID chanQ, Peer Q knows that Peer P really received its reply datagram, and the three-way handshake and channel establishment is complete. Peer Q MAY now also start sending messages with heavy payload to Peer P.
8. If Peer P decides it no longer wants to communicate with Peer Q, or vice versa, the peer SHOULD send a closing HANDSHAKE message to the other, as described above.

4.2.2. HAVE

The HAVE message is used to convey which chunks a peer has available for download. The set of chunks it has available may be expressed using different chunk addressing and availability map compression schemes, described in Section 4.3. HAVE messages can be used both for sending a complete overview of a peer's chunk availability as well as for updates to that set.

In particular, whenever a receiving Peer P has successfully checked the integrity of a chunk, or interval of chunks, it MUST send a HAVE message to all peers Q1..Qn it wants to allow to download those chunks. A policy in Peer P determines when the HAVE is sent. Peer P may send it directly, or Peer P may wait either until it has other data to send to Peer Qi or until it has received and checked multiple chunks. The policy will depend on how urgent it is to distribute this information to the other peers. This urgency is generally determined in turn by the chunk picking policy (see Section 4.6.1). In general, the HAVE messages can be piggybacked onto other messages. Peers that do not receive HAVE messages are effectively prevented from downloading the newly available chunks; hence, the HAVE message can be used as a method of choking.

The HAVE message MUST contain the chunk specification of the received and verified chunks. A receiving peer MUST NOT send a HAVE message to peers for which the handshake procedure is still incomplete, see Section 13.1 of [60]. A peer SHOULD NOT send a HAVE message to peers that have the complete content already (e.g., in video-on-demand scenarios).

4.2.3. DATA

The DATA message is used to transfer chunks of content. The DATA message MUST contain the chunk ID of the chunk and chunk itself. A peer MAY send the DATA messages for multiple chunks in the same datagram. The DATA message MAY contain additional information if needed by the specific congestion control mechanism used. At present, PPSPP uses LEDBAT [149] for congestion control, which requires the current system time to be sent along with the DATA message, so the current system time MUST be included.

4.2.4. ACK

ACK messages MUST be sent to acknowledge received chunks if PPSPP is run over an unreliable transport protocol. ACK messages MAY be sent if a reliable transport protocol is used. In the former case, a receiving peer that has successfully checked the integrity of a chunk, or interval of chunks C, MUST send an ACK message containing a chunk specification for C. As LEDBAT is used, an ACK message MUST contain the one-way delay, computed from the peer's current system time received in the DATA message. A peer MAY delay sending ACK messages as defined in the LEDBAT specification [149].

4.2.5. INTEGRITY

The INTEGRITY message carries information required by the receiver to verify the integrity of a chunk. Its payload depends on the content integrity protection scheme used. When the Merkle Hash Tree scheme is used, an INTEGRITY message MUST contain a cryptographic hash of a subtree of the Merkle hash tree and the chunk specification that identifies the subtree.

As a typical example, when a peer wants to send a chunk and Merkle hash trees are used, it creates a datagram that consists of several INTEGRITY messages containing the hashes the receiver needs to verify the chunk and the actual chunk itself encoded in a DATA message. What are the necessary hashes and the exact rules for encoding them into datagrams is specified in Sections 4.4.3, and 4.4.4, respectively.

4.2.6. SIGNED_INTEGRITY

The SIGNED_INTEGRITY message carries digitally signed information required by the receiver to verify the integrity of a chunk in live streaming. It logically contains a chunk specification, a timestamp, and a digital signature. Its exact payload depends on the live content integrity protection scheme used, see Section 4.5.1.

4.2.7. REQUEST

While bulk download protocols normally do explicit requests for certain ranges of data (i.e., use a pull model, for example, BitTorrent [5]), live streaming protocols quite often use a push model without requests to save round trips. PPSPP supports both models of operation.

The REQUEST message is used to request one or more chunks from another peer. A REQUEST message MUST contain the specification of the chunks the requester wants to download. A peer receiving a REQUEST message MAY send out the requested chunks (by means of DATA messages). When Peer Q receives multiple REQUESTs from the same Peer P, Peer Q SHOULD process the REQUESTs in the order received. Multiple REQUEST messages MAY be sent in one datagram, for example, when a peer wants to request several rare chunks at once.

When live streaming via a push model, a peer receiving REQUESTs also MAY send some other chunks in case it runs out of requests or for some other reason. In that case, the only purpose of REQUEST messages is to provide hints and coordinate peers to avoid unnecessary data retransmission.

4.2.8. CANCEL

When downloading on-demand or live streaming content, a peer can request urgent data from multiple peers to increase the probability of it being delivered on time. In particular, when the specific chunk picking algorithm (see Section 4.6.1), detects that a request for urgent data might not be served on time, a request for the same data can be sent to a different peer. When a Peer P decides to request urgent data from a Peer Q, Peer P SHOULD send a CANCEL message to all the peers to which the data has been previously requested. The CANCEL message contains the specification of the chunks Peer P no longer wants to request. In addition, when Peer Q receives a HAVE message for the urgent data from Peer P, Peer Q MUST also cancel the previous REQUEST(s) from Peer P. In other words, the HAVE message acts as an implicit CANCEL.

4.2.9. CHOKE and UNCHOKE

Peer A can send a CHOKE message to Peer B to signal it will no longer be responding to REQUEST messages from Peer B, for example, because Peer A's upload capacity is exhausted. Peer A MAY send a subsequent UNCHOKE message to signal that it will respond to new REQUESTs from Peer B again (Peer A SHOULD discard old requests). When Peer B receives a CHOKE message from Peer A, it MUST NOT send new REQUEST messages and it cannot expect answers to any outstanding ones, as the transfer of chunks is choked. When Peer B is choked but receives a HAVE message from Peer A, it is not automatically unchoked and MUST NOT send any new REQUEST messages. The CHOKE and UNCHOKE messages are informational as responding to REQUESTs is OPTIONAL, see Section 4.2.7.

4.2.10. Peer Address Exchange

PEX_REQ and PEX_RES Messages

Peer Exchange (PEX) messages are common in many peer-to-peer protocols. They allow peers to exchange the transport addresses of the peers they are currently interacting with, thereby reducing the need to contact a central tracker (or Distributed Hash Table) to discovery new peers. The strength of this mechanism is therefore that it enables decentralized peer discovery: after an initial bootstrap, a central tracker is no longer needed. Its weakness is that it enables a number of attacks, so it should not be used on the Internet unless extra security measures are in place.

PPSPP supports peer-address exchange on the Internet and in benign private networks as an OPTIONAL feature (not mandatory to implement) under certain conditions. The general mechanism works as follows. To obtain some peer addresses, a Peer A MAY send a PEX_REQ message to Peer B. Peer B MAY respond with one or more PEX_REScert messages. Logically, a PEX_REScert reply message contains the address of a single peer Ci.Peer B MUST have exchanged messages with Peer Ci in the last 60 seconds to guarantee liveliness. Upon receipt, Peer A may contact any or none of the returned peers Ci. Alternatively, peers MAY ignore PEX_REQ and PEX_REScert messages if uninterested in obtaining new peers or because of security considerations (rate limiting) or any other reason. The PEX messages can be used to construct a dedicated tracker peer.

To use PEX in PPSPP on the Internet, two conditions must be met:

1. Peer transport addresses must be relatively stable.
2. A peer must not obtain all its peer addresses through PEX.

The full security analysis for PEX messages can be found in Section 13.2 of [60]. Physically, a PEX_REScert message carries a swarm-membership certificate rather than an IP address and port. A membership certificate for Peer C states that Peer C at address (ipC,portC) is part of Swarm S at Time T and is cryptographically signed by an issuer. The receiver Peer A can check the certificate for a valid signature by a trusted issuer, the right swarm, and liveliness and only then consider contacting C. These swarm-membership certificates correspond to signed node descriptors in secure decentralized peer sampling services [108].

Several designs are possible for the security environment for these membership certificates. That is, there are different designs possible for who signs the membership certificates and how public keys are distributed. Section 13.2.2 of [60] describes an example where a central tracker acts as the Certification Authority.

In a hostile environment, such as the Internet, peers must also ensure that they do not end up interacting only with malicious peers when using the peer-address exchange feature. To this extent, peers MUST ensure that part of their connections are to peers whose addresses came from a trusted and secured tracker (see Section 13.2.3 of [60]).

In addition to the PEX_REScert, there are two other PEX reply messages. The PEX_RESv4 message contains a single IPv4 address and port. The PEX_RESv6 message contains a single IPv6 address and port. They MUST only be used in a benign environment, such as a private network, as they provide no guarantees that the host addressed actually participates in a PPSPP swarm.

Once a PPSPP implementation has obtained a list of peers (either via PEX, from a central tracker, or via a Distributed Hash Table (DHT)), it has to determine which peers to actually contact. In this process, a PPSPP implementation can benefit from information by network or content providers to help improve network usage and boost PPSPP performance. How a peer-to-peer (P2P) system like PPSPP can perform these optimizations using the Application-Layer Traffic Optimization (ALTO) protocol is described in detail in [53], Section 7.

4.2.11. Channels

It is increasingly complex for peers to enable communication between each other due to NATs and firewalls. Therefore, PPSPP uses a multiplexing scheme, called channels, to allow multiple swarms to use the same transport address. Channels loosely correspond to TCP connections and each channel belongs to a single swarm, as illustrated in 4.1. As with TCP connections, a channel is identified by a unique identifier local to the peer at each end of the connection (cf. TCP port), which MUST be randomly chosen. In other words, the two peers connected by a channel use different IDs to denote the same channel. The IDs are different and random for security reasons, see Section 13.1 of [60].

In the PPSP-over-UDP encapsulation (Section 8.3 of [60]), when a Channel C has been established between Peer A and Peer B, the datagrams containing messages from Peer A to Peer B are prefixed with the four-byte channel ID allocated by Peer B, and vice versa for datagrams from Peer B to A. The channel IDs used are exchanged as part of the handshake procedure, see Section 8.4 of [60]. In that procedure, the channel ID with value 0 is used for the datagram that initiates the handshake. PPSPP can be used in combination with Session Traversal Utilities for NAT (STUN) [158].

4.2.12. Keep Alive Signaling

A peer SHOULD send a keep alive message periodically to each peer it is interested in, but has no other messages to send to them at present. The goal of the keep alives is to keep a signaling channel open to peers that are of interest. Which peers

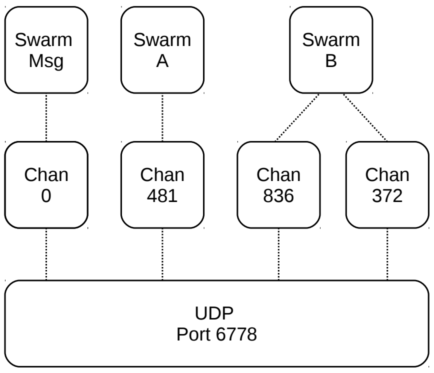


Figure 4.1: Network stack of a PPSPP peer that is reachable on UDP port 6778 and is connected via channel 481 to one peer in Swarm A and two peers in Swarm B via channels 836 and 372, respectively. Channel ID 0 is special and is used for handshaking.

those are is determined by a policy that decides which peers are of interest now and in the near future. This document does not prescribe a policy, but examples of interesting peers are (a) peers that have chunks on offer that this client needs or (b) peers that currently do not have interesting chunks on offer (because they are still downloading themselves, or in live streaming) but gave good performance in the past. When these peers have new chunks to offer, the peer that kept a signaling channel open can use them again. Periodically sending keep alive messages prevents other peers declaring the peer dead. A guideline for declaring a peer dead when using UDP consists of a three minute delay since that last packet has been received from that peer and at least three datagrams having been sent to that peer during the same period. When a peer is declared dead, the channel to it is closed, no more messages will be sent to that peer and the local administration about the peer is discarded. Busy servers can force idle clients to disconnect by not sending keep alives. PPSPP does not define an explicit message type for keep alive messages. In the PPSP-over-UDP encapsulation they are implemented as simple datagrams consisting of a four-byte channel ID only, see Sections 8.3 and 8.4 of [60].

4.3. Chunk Addressing Schemes

PPSPP can use different methods of chunk addressing, that is, support different ways of identifying chunks and different ways of expressing the chunk availability map of a peer in a compact fashion.

All peers in a swarm MUST use the same chunk addressing method.

4.3.1. Start-End Ranges

A chunk specification consists of a single (start specification,end specification) pair that identifies a range of chunks (end inclusive). The start and end specifications can use one of multiple addressing schemes. Two schemes are currently defined: chunk ranges and byte ranges.

4.3.2. Chunk Ranges

The start and end specification are both chunk identifiers. Chunk identifiers are 32-bit or 64-bit unsigned integers. A PPSPP peer MUST support this scheme.

4.3.3. Byte Ranges

The start and end specification are 64-bit byte offsets in the content. The support for this scheme is OPTIONAL.

4.3.4. Bin Numbers

PPSPP introduces a novel method of addressing chunks of content called ”bin numbers” (or ”bins” for short). Bin numbers allow the addressing of a binary interval of data using a single integer. This reduces the amount of state that needs to be recorded per peer and the space needed to denote intervals on the wire, making the protocol lightweight. In general, this numbering system allows PPSPP to work with simpler data structures, e.g., to use arrays instead of binary trees, thus reducing complexity. The support for this scheme is OPTIONAL.

In bin addressing, the smallest binary interval is a single chunk (e.g., a block of bytes that may be of variable size), the largest interval is a complete range of 2^{63} chunks. In a novel addition to the classical scheme, these intervals are numbered in a way that lays them out into a vector nicely, which is called bin numbering, as follows. Consider a chunk interval of width W. To derive the bin numbers of the complete interval and the subintervals, a minimal balanced binary tree is built that is at least W chunks wide at the base. The leaves from left-to-right correspond to the chunks 0,..,W-1 in the interval, and have bin number 2I where I is the index of the chunk (counting beyond W-1 to balance the tree). The bin number of higher-level node P in the tree is calculated as follows:

$$binP = (binL + binR)/2 \tag{4.1}$$

where binL is the bin of node P’s left-hand child and binR is the bin of node P’s right-hand child. Given that each node in the tree represents a subinterval of the original interval, each such subinterval now is addressable by a bin number, a single integer. The bin number tree of an interval of width W=8 looks like the one presented in Figure 4.2.

So bin 7 represents the complete interval, bin 3 represents the interval of chunk C0..C3, bin 1 represents the interval of chunks C0 and C1, and bin 2 represents chunk C1. The special numbers 0xFFFFFFFF (32-bit) or 0xFFFFFFFFFFFFFFFF (64-bit) stands for an empty interval, and 0x7FFF...FFF stands for ”everything”. When bin numbering is used, the ID of a chunk is its corresponding (leaf) bin number in

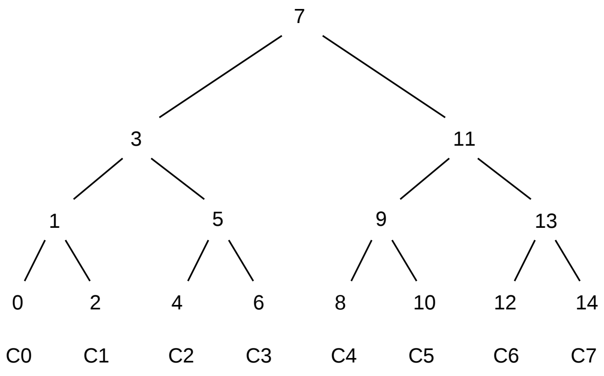


Figure 4.2: The bin number tree of an interval of width W=8.

the tree, and the chunk specification in HAVE and ACK messages is equal to a single bin number (32-bit or 64-bit).

4.3.5. In Messages

In HAVE Messages

When a receiving peer has successfully checked the integrity of a chunk or interval of chunks it MUST send a HAVE message to all peers it wants to allow download of those chunks from. The ability to withhold HAVE messages allows them to be used as a method of choking. The HAVE message MUST contain the chunk specification of the biggest complete interval of all chunks the receiver has received and checked so far that fully includes the interval of chunks just received. So the chunk specification MUST denote at least the interval received, but the receiver is supposed to aggregate and acknowledge bigger intervals, when possible.

As a result, every single chunk is acknowledged a logarithmic number of times. That provides some necessary redundancy of acknowledgements and sufficiently compensates for unreliable transport protocols.

Implementation note

> To record which chunks a peer has in the state that an implementation keeps for each peer, an implementation MAY use the efficient ”binmap” data structure, which is a hybrid of a bitmap and a binary tree, discussed in detail in [98].

In ACK Messages

PPSPP peers MUST use ACK messages to acknowledge received chunks if an unreliable transport protocol is used. When a receiving peer has successfully checked the integrity of a chunk or interval of chunks C, it MUST send an ACK message containing the chunk specification of its biggest, complete interval covering C to the sending peer (see HAVE).

4.4. Content Integrity Protection

PPSPP can use different methods for protecting the integrity of the content while it is being distributed via the peer-to-peer network. More specifically, PPSPP can use different methods for receiving peers to detect whether a requested chunk has been maliciously modified by the sending peer. In benign environments, content integrity protection can be disabled.

For static content, PPSPP currently defines one method for protecting integrity, called the Merkle Hash Tree scheme. If PPSPP operates over the Internet, this scheme MUST be used. If PPSPP operates in a benign environment, this scheme MAY be used. So the scheme is mandatory to implement, to satisfy the requirement of strong security for an IETF protocol [144]. An extended version of the scheme is used to efficiently protect dynamically generated content (live streams), as explained below and in Section 4.5.1.

The Merkle Hash Tree scheme can work with different chunk addressing schemes. All it requires is the ability to address a range of chunks. In the following description abstract node IDs are used to identify nodes in the tree. On the wire, these are translated to the corresponding range of chunks in the chosen chunk addressing scheme.

4.4.1. Merkle Hash Tree Scheme

PPSPP uses a method of naming content based on self-certification. In particular, content in PPSPP is identified by a single cryptographic hash that is the root hash in a Merkle hash tree calculated recursively from the content [58]. This self-certifying hash tree allows every peer to directly detect when a malicious peer tries to distribute fake content. It also ensures only a small the amount of information is needed to start a download (the root hash and some peer addresses). For live streaming, a dynamic tree and a public key are used, see below.

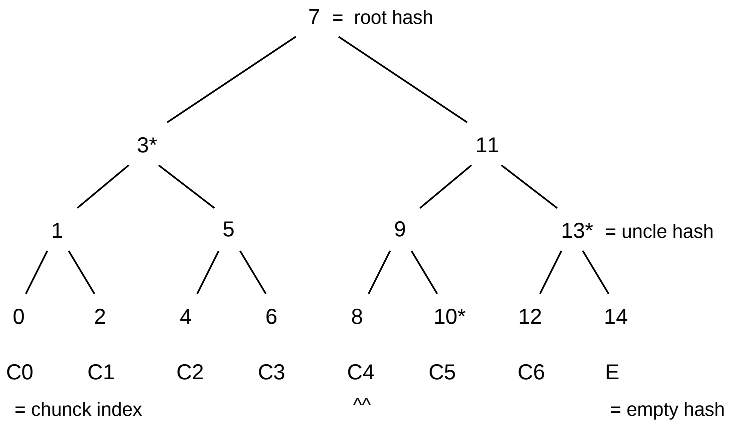


Figure 4.3: Merkle hash tree of a content file with N=7 chunks.

The Merkle hash tree of a content file that is divided into N chunks is constructed as follows. Note the construction does not assume chunks of content to be fixed size. Given a cryptographic hash function, more specifically an MDC [120] , such as SHA-256, the hashes of all the chunks of the content are calculated. Next, a binary tree of sufficient height is created. Sufficient height means that the lowest level in the tree has enough nodes to hold all chunk hashes in the set, as with bin numbering. The figure below shows the tree for a content file consisting of 7 chunks. As with the content addressing scheme, the leaves of the tree correspond to a chunk and, in this case, are assigned the hash of that chunk, starting at the leftmost leaf. As the base of the tree may be wider than the number of chunks, any remaining leaves in the tree are assigned an empty hash value of all zeros. Finally, the hash values of the higher levels in the tree are calculated, by concatenating the hash values of the two children (again left to right) and computing the hash of that aggregate. If the two children are empty hashes, the parent is an empty all-zeros hash as well (to save computation). This process ends in a hash value for the root node, which is called the ”root hash”. Note the root hash only depends on the content and any modification of the content will result in a different root hash.

4.4.2. Content Integrity Verification

Assuming a peer receives the root hash of the content it wants to download from a trusted source, it can check the integrity of any chunk of that content it receives as follows. It first calculates the hash of the chunk it received, for example, chunk C4 in the previous figure. Along with this chunk, it MUST receive the hashes required to check the integrity of that chunk. In principle, these are the hash of the chunk's sibling (C5) and that of its ”uncles”. A chunk's uncles are the sibling Y of its parent X, and the uncle of that Y, recursively until the root is reached. For chunk C4, the uncles are nodes 13 and 3 and the sibling is 10; all marked with a * in the figure. Using this information, the peer recalculates the root hash of the tree and compares it to the root hash it received from the trusted source. If they match, the chunk of content has been positively verified to be the requested part of the content. Otherwise, the sending peer sent either the wrong content or the wrong sibling or uncle hashes. For simplicity, the set of sibling and uncle hashes is collectively referred to as the ”uncle hashes”.

In the case of live streaming, the tree of chunks grows dynamically and the root hash is undefined or, more precisely, transient, as long as new data is generated by the live source. Section 4.5.1 defines a method for content integrity verification for live streams that works with such a dynamic tree. Although the tree is dynamic, content verification works the same for both live and predefined content, resulting in a unified method for both types of streaming.

4.4.3. The Atomic Datagram Principle

As explained above, a datagram consists of a sequence of messages. Ideally, every datagram sent must be independent of other datagrams: each datagram SHOULD be processed separately, and a loss of one datagram must not disrupt the flow of datagrams between two peers. Thus, as a datagram carries zero or more messages, both messages and message interdependencies SHOULD NOT span over multiple datagrams.

This principle implies that as any chunk is verified using its uncle hashes, the necessary hashes SHOULD be put into the same datagram as the chunk's data. If this is not possible because of a limitation on datagram size, the necessary hashes MUST be sent first in one or more datagrams. As a general rule, if some additional data is still missing to process a message within a datagram, the message SHOULD be dropped.

The hashes necessary to verify a chunk are, in principle, its sibling's hash and all its uncle hashes, but the set of hashes to send can be optimized. Before sending a packet of data to the receiver, the sender inspects the receiver's previous acknowledgements (HAVE or ACK) to derive which hashes the receiver already has for sure. Suppose the receiver had acknowledged chunks C0 and C1 (the first two chunks of the file), then it must already have uncle hashes 5, 11, and so on. That is because those hashes are necessary to check C0 and C1 against the root hash. Then, hashes 3, 7, and so on must also be known as they are calculated in the process of checking the uncle hash chain. Hence, to send chunk C7, the sender needs to include just the hashes for nodes 14 and 9, which let the data be checked against hash 11, which is already known to the receiver.

The sender MAY optimistically skip hashes that were sent out in previous, still-unacknowledged datagrams. It is an optimization trade-off between redundant hash transmission and the possibility of collateral data loss in the case in which some necessary hashes were lost in the network so some delivered data cannot be verified and thus had to be dropped. In either case, the receiver builds the Merkle hash tree on-demand, incrementally, starting from the root hash, and uses it for data validation.

In short, the sender MUST put into the datagram the hashes he believes are necessary for the receiver to verify the chunk. The receiver MUST remember all the hashes it needs to verify missing chunks that it still wants to download. Note that the latter implies that a hardware-limited receiver MAY forget some hashes if it does not plan to announce possession of these chunks to others (i.e., does not plan to send HAVE messages.)

4.4.4. INTEGRITY Messages

Concretely, a peer that wants to send a chunk of content creates a datagram that MUST consist of a list of INTEGRITY messages followed by a DATA message. If the INTEGRITY messages and DATA message cannot be put into a single datagram because of a limitation on datagram size, the INTEGRITY messages MUST be sent first in one or more datagrams. The list of INTEGRITY messages sent MUST contain an INTEGRITY message for each hash the receiver misses for integrity checking. An INTEGRITY message for a hash MUST contain the chunk specification corresponding to the node ID of the hash and the hash data itself. The chunk specification corresponding to a node ID is defined as the range of chunks formed by the leaves of the subtree rooted at the node. For example, node 3 in Figure 4.3 denotes chunks 0, 2, 4, and 6, so the chunk specification should denote

Chunk	Node IDs of hashes sent
0	2,5,11
2	- (receiver already knows all)
4	6
6	-
8	10,13 (hash 3 can be calculated from 0,2,5)
10	-
12	14
14	-
Total	# hashes 7

Table 4.1: Overhead for the Example Tree.

that interval. The list of INTEGRITY messages MUST be sorted in order of the tree height of the nodes, descending (the leaves are at height 0). The DATA message MUST contain the chunk specification of the chunk and the chunk itself. A peer MAY send the required messages for multiple chunks in the same datagram, depending on the encapsulation.

4.4.5. Discussion and Overhead

The current method for protecting content integrity in BitTorrent [5] is not suited for streaming. It involves providing clients with the hashes of the content's chunks before the download commences by means of metadata files (called .torrent files in BitTorrent.) However, when chunks are small, as in the current UDP encapsulation of PPSPP, this implies having to download a large number of hashes before content download can begin. This, in turn, increases time-till-playback for end users, making this method unsuited for streaming.

The overhead of using Merkle hash trees is limited. The size of the hash tree expressed as the total number of nodes depends on the number of chunks the content is divided (and hence the size of chunks) following this formula:

$$\text{nnodes} = \text{math.pow}(2,\text{math.log}(\text{nchunks},2)+1)$$

In principle, the hash values of all these nodes will have to be sent to a peer once for it to verify all of the chunks. Hence, the maximum on-the-wire overhead is hashsize * nnodes. However, the actual number of hashes transmitted can be optimized as described in Section 4.4.3.

To see a peer can verify all chunks whilst receiving not all hashes, consider the example tree in Section 4.4.1. In the case of a simple progressive download, of chunks 0, 2, 4, 6, etc., the sending peer will send the following hashes:

So the number of hashes sent in total (7) is less than the total number of hashes in the tree (16), as a peer does not need to send hashes that are calculated and verified as part of earlier chunks.

4.4.6. Automatic Detection of Content Size

In PPSPP, the size of a static content file, such as a video file, can be reliably and automatically derived from information received from the network when fixed-sized chunks are used. As a result, it is not necessary to include the size of the content file as the metadata of the content for such files. Implementations of PPSPP MAY use this automatic detection feature. Note this feature is the only feature of PPSPP that requires that a fixed-size chunk is used. This feature builds on the Merkle hash tree and the trusted root hash as swarm ID as follows.

Peak Hashes

The ability for a newcomer peer to detect the size of the content depends heavily on the concept of peak hashes. The concept of peak hashes depends on the concepts of filled and incomplete nodes. Recall that when constructing the binary trees for content verification and addressing the base of the tree may have more leaves than the number of chunks in the content. In the Merkle hash tree, these leaves were assigned empty all-zero hashes to be able to calculate the higher-level hashes. A filled node is now defined as a node that corresponds to an interval of leaves that consists only of hashes of content chunks, not empty hashes. Reversely, an incomplete (not filled) node corresponds to an interval that also contains empty hashes, typically, an interval that extends past the end of the file. In the following figure, nodes 7, 11, 13, and 14 are incomplete: the rest is filled.

Formally, a peak hash is the hash of a filled node in the Merkle hash tree, whose sibling is an incomplete node. Practically, suppose a file is 7162 bytes long and a chunk is 1 kilobyte. That file fits into 7 chunks, the tail chunk being 1018 bytes long. The Merkle hash tree for that file is shown in Figure 4.4. Following the definition, the peak hashes of this file are in nodes 3, 9, and 12, denoted with an *. E denotes an empty hash.

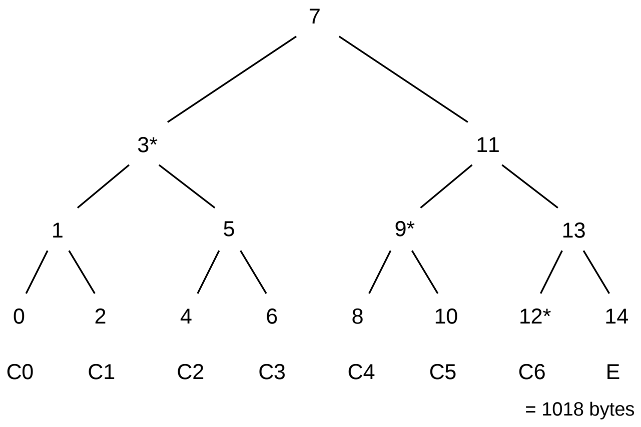


Figure 4.4: Peak hashes in a Merkle hash tree

Peak hashes can be explained by the binary representation of the number of chunks the file occupies. The binary representation for 7 is 111. Every ”1” in binary representation of the file’s packet length corresponds to a peak hash. For this particular file, there are indeed three peaks: nodes 3, 9, and 12. Therefore, the number of peak hashes for a file is also, at most, logarithmic with its size.

A peer knowing which nodes contain the peak hashes for the file can therefore calculate the number of chunks it consists of: thus, it gets an estimate of the file size (given all chunks but the last are fixed size). Which nodes are the peaks can be securely communicated from one (untrusted) peer, Peer A, to another peer, Peer B, by letting Peer A send the peak hashes and their node IDs to Peer B. It can be shown that the root hash that Peer B obtained from a trusted source is sufficient to verify that these are indeed the right peak hashes, as follows.

Peak hashes can be checked against the root hash, as we will now show. (a) Any peak hash is always the left sibling. Otherwise, if it is the right sibling, its left neighbor/sibling must also be a filled node, because of the way chunks are laid out in the leaves, which contradicts the definition of peak hash. (b) For the rightmost peak hash, its right sibling is zero. (c) For any peak hash, the right sibling might be calculated using peak hashes to the left and zeros for empty nodes. (d) Once the right sibling of the leftmost peak hash is calculated, its parent might be calculated. (e) Once that parent is calculated, we might trivially get to the root hash by concatenating the hash with zeros and hashing it repeatedly.

Informally, this fact might be expressed as follows: peak hashes cover all data, so the remaining hashes are either trivial (zeros) or might be calculated from peak hashes and zero hashes.

Finally, once Peer B has obtained the number of chunks in the content, it can determine the exact file size as follows. Given that all chunks except the last are of a fixed size, Peer B just needs to know the size of the last chunk. Knowing the number of chunks, Peer B can calculate the node ID of the last chunk and download it. As always, Peer B verifies the integrity of this chunk against the trusted root hash. As there is only one chunk of data that leads to a successful verification, the size of this chunk must be correct. Peer B can then determine the exact file size as:

(number of chunks -1) * fixed chunk size + size of last chunk

Procedure

A PPSPP implementation that wants to use automatic size detection MUST operate as follows. When Peer A sends a DATA message for the first time to Peer B, Peer A MUST first send all the peak hashes for the content, in INTEGRITY messages, unless Peer B has signaled that it knows the peak hashes by having acknowledged any chunk. If they are needed, the peak hashes MUST be sent as an extra list of uncle hashes for the chunk, before the list of actual uncle hashes of the chunk as described in Section 4.4.3. The receiver, Peer B, MUST check the peak hashes against the root hash to determine the approximate content size. To obtain the definite content size, Peer B MUST download the last chunk of the content from any peer that offers it.

As an example, let’s consider a 7162-bytes file, which fits in 7 chunks of 1 kilobyte, distributed by Peer A. Figure 4.4 shows the relevant Merkle hash tree. Peer B which only knows the root hash of the file after successfully connecting to Peer A, requests the first chunk of data, C0 in Figure 4.4. Peer A replies to Peer B by including in the datagram the following messages in this specific order: first, the three peak hashes of this particular file, the hashes of nodes 3, 9, and 12; second, the uncle hashes of C0, followed by the DATA message containing the actual content of C0. Upon receiving the peak hashes, Peer B checks them against the root hash determining that the file is 7 chunks long. To establish the exact size of the file, Peer B needs to request and retrieve the last chunk containing data, C6 in Figure 4.4. Once the last chunk has been retrieved and verified, Peer B concludes that it is 1018 bytes long, hence determining that the file is exactly 7162 bytes long.

4.5. Live Streaming

The set of messages defined above can be used for live streaming as well. In a pull-based model, a live streaming injector can announce the chunks it generates via HAVE messages, and peers can retrieve them via REQUEST messages. Areas that need special attention are content authentication and chunk addressing (to achieve an infinite stream of chunks).

4.5.1. Content Authentication

For live streaming, PPSPP supports two methods for a peer to authenticate the content it receives from another peer, called ”Sign All” and ”Unified Merkle Tree”.

In the ”Sign All” method, the live injector signs each chunk of content using a private key. Upon receiving the chunk, peers check the signature using the corresponding public key obtained from a trusted source. Support for this method is OPTIONAL.

In the ”Unified Merkle Tree” method, PPSPP combines the Merkle Hash Tree scheme for static content with signatures to unify the video-on-demand and live streaming scenarios. The use of Merkle hash trees reduces the number of signing and verification operations, hence providing a similar signature amortization to the approach described in [159]. If PPSPP operates over the Internet, the ”Unified Merkle Tree” method MUST be used. If the protocol operates in a benign environment, the ”Unified Merkle Tree” method MAY be used. So this method is mandatory to implement.

In both methods, the swarm ID consists of a public key encoded as in a DNSSEC DNSKEY resource record without Base64 encoding [56]. In particular, the swarm ID consists of a 1-byte Algorithm field that identifies the public key’s cryptographic algorithm and determines the format of the Public Key field that follows. The value of this Algorithm field is one of the values in the ”Domain Name System Security (DNSSEC) Algorithm Numbers” registry [87]. The RSASHA1 [56], RSASHA256 [107], ECDSAP256SHA256 and ECDSAP384SHA384 [101] algorithms are mandatory to implement.

Sign All

In the ”Sign All” method, the live injector signs each chunk of content using a private key and peers, upon receiving the chunk, check the signature using the corresponding public key obtained from a trusted source. In particular, in PPSPP, the swarm ID of the live stream is that public key.

A peer that wants to send a chunk of content creates a datagram that MUST contain a SIGNED_INTEGRITY message with the chunk’s signature, followed by a DATA message with the actual chunk. If the SIGNED_INTEGRITY message and DATA message cannot be contained into a single datagram, because of a limitation on datagram size, the SIGNED_INTEGRITY message MUST be sent first in a separate datagram. The SIGNED_INTEGRITY message consists of the chunk specification, the timestamp, and the digital signature.

The digital signature algorithm that is used, is determined by the Live Signature Algorithm protocol option, see Section 7.7 of [60]. The signature is computed over a concatenation of the on-the-wire representation of the chunk specification, a 64-bit timestamp in NTP Timestamp format [68], and the chunk, in that order. The timestamp is the time signature that was made at the injector in UTC.

Unified Merkle Tree

In this method, the chunks of content are used as the basis for a Merkle hash tree as for static content. However, because chunks are continuously generated, this tree is not static, but dynamic. As a result, the tree does not have a root hash, or, more precisely, it has a transient root hash. Therefore, a public key serves as swarm ID of the content. It is used to digitally sign updates to the tree, allowing peers to expand it based on trusted information using the following process.

Signed Munro Hashes The live injector generates a number of chunks, denoted NCHUNKS_PER_SIG, corresponding to fixed power of 2 (NCHUNKS_PER_SIG >= 2), which are added as new leaves to the existing hash tree. As a result of this expansion, the hash tree contains a new subtree that is NCHUNKS_PER_SIG chunks wide at the base. The root of this new subtree is referred to as the munro of that subtree, and its hash as the munro hash of the subtree, illustrated in Figure 4.5. In this figure, node 5 is the new munro, labeled with a $ sign.

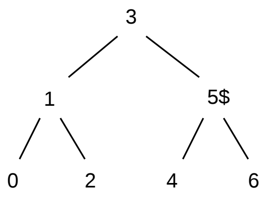


Figure 4.5: Expanded live tree. With NCHUNKS_PER_SIG=2, node 5 is the munro for the new subtree spanning 4 and 6. Node 1 is the munro for the subtree spanning chunks 0 and 2, created in the previous iteration.

Informally, the process now proceeds as follows. The injector signs only the munro hash of the new subtree using its private key. Next, the injector announces the existence of the new subtree to its peers using HAVE messages. When a peer, in response to the HAVE messages, requests a chunk from the new subtree, the injector first sends the signed munro hash corresponding to the requested chunk. Afterwards, similar to static content, the injector sends the uncle hashes necessary to verify that chunk, as in Section 4.4.1. In particular, the injector sends the uncle hashes necessary to verify the requested chunk against the munro hash. This differs from static content, where the verification takes places against the root hash. Finally, the injector sends the actual chunk.

The receiving peer verifies the signature on the signed munro using the swarm ID (a public key) and updates its hash tree. As the peer now knows the munro hash is trusted, it can verify all chunks in the subtree against this munro hash, using the accompanying uncle hashes as in Section 4.4.1.

To illustrate this procedure, lets consider the next iteration in the process. The injector has generated the current tree shown in Figure 4.5 and it is connected to several peers that currently have the same tree and all posses chunks 0, 2, 4, and 6. When the injector generates two new chunks, NCHUNKS_PER_SIG=2, the hash tree expands as shown in Figure 4.6. The two new chunks, 8 and 10, extend the tree on the right side, and to accommodate them, a new root is created: node 7. As this tree is wider at the base than the actual number of chunks, there are currently two empty leaves. The munro node for the new subtree is 9, labeled with a $ sign.

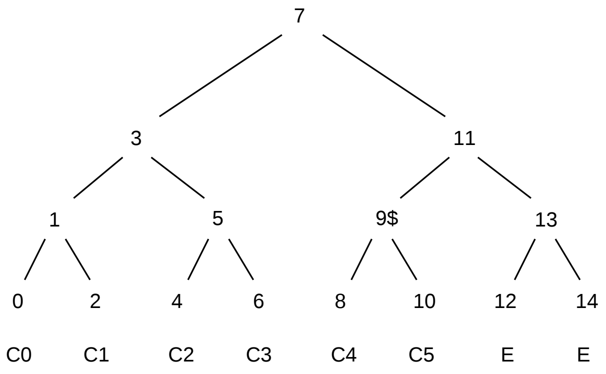


Figure 4.6: Expanded live tree. With NCHUNKS_PER_SIG=2, node 9 is the munro of the newly added subtree spanning chunks 8 and 10.

The injector now needs to inform its peers of the updated tree, communicating the addition of the new munro hash 9. Hence, it sends a HAVE message with a chunk specification for nodes 8 + 10 to its peers. As a response, Peer P requests the newly created chunk, e.g., chunk 8, from the injector by sending a REQUEST message. In reply, the injector sends the signed munro hash of node 9 as an INTEGRITY message with the hash of node 9, and a SIGNED_INTEGRITY message with the signature of the hash of node 9. These messages are followed by an INTEGRITY message with the hash of node 10 and a DATA message with chunk 8.

Upon receipt, Peer P verifies the signature of the munro and expands its view of the tree. Next, the peer computes the hash of chunk 8 and combines it with the received hash of node 10, computing the expected hash of node 9. He can then verify the content of chunk 8 by comparing the computed hash of node 9 with the munro hash of the same node he just received; hence, Peer P has successfully verified the integrity of chunk 8. This procedure requires just one signing operation for every NCHUNKS_PER_SIG chunks created, and one verification operation for every NCHUNKS_PER_SIG received, making it much cheaper than "Sign All". A receiving peer does additionally need to check one or more hashes per chunk via the Merkle Hash Tree scheme, but this has less hardware requirements than a signature verification for every chunk. This approach is similar to signature amortization via Merkle Tree Chaining [159]. The downside of this scheme is in an increased latency. A peer cannot download the new chunks until the injector has computed the signature and announced the subtree. A peer MUST check the signature before forwarding the chunks to other peers [81].

The number of chunks per signature NCHUNKS_PER_SIG MUST be a fixed power of 2 for simplicity. NCHUNKS_PER_SIG MUST be larger than 1 for performance reasons. There are two related factors to consider when choosing a value for NCHUNKS_PER_SIG. First, the allowed CPU load on clients due to signature verifications, given the expected bitrate of the stream. To achieve a low CPU load in a high bitrate stream, NCHUNKS_PER_SIG should be high. Second, the effect on latency, which increases when NCHUNKS_PER_SIG gets higher, as just discussed. Note how the procedure does not preclude the use of variable sized chunks.

This method of integrity verification provides an additional benefit. If the system includes some peers that saved the complete broadcast, as soon as the broadcast ends, the content is available as a video-on-demand download using the now stabilized tree and the final root hash as swarm identifier. Peers that saved all the chunks, can now announce the root hash to the tracking infrastructure and instantly seed the content.

Munro Signature Calculation The digital signature algorithm used is determined by the Live Signature Algorithm protocol option, see Section 7.7 of [60]. The signature is computed over a concatenation of the on-the-wire representation of the chunk specification of the munro node (see the previous paragraph), a timestamp in 64-bit NTP Timestamp format [68], and the hash associated with the munro node, in that order. The timestamp is the time signature that was made at the injector in UTC.

Procedure Formally, the injector MUST NOT send a HAVE message for chunks in the new subtree until it has computed the signed munro hash for that subtree.

When Peer B requests a chunk C from Ppeer A (either the injector or another peer), and Peer A decides to reply, it must do so as follows. First, Peer A MUST send an INTEGRITY message with the chunk specification for the munro of chunk

C and the munro's hash, followed by a SIGNED_INTEGRITY message with the chunk specification for the munro, timestamp, and its signature in a single datagram, unless Peer B indicated earlier in the exchange that it already possess a chunk with the same corresponding munro (by means of HAVE or ACK messages). Following these two messages (if any), Peer A MUST send the necessary missing uncles hashes needed for verifying the chunk against its munro hash, and the chunk itself, as described in Section 4.4.4, sharing datagrams if possible.

Secure Tune In When a peer tunes in to a live stream, it has to determine what is the last chunk the injector has generated. To facilitate this process in the Unified Merkle Tree scheme, each peer shares its knowledge about the injector's chunks with the others by exchanging their latest signed munro hashes, as follows.

Recall that, in PPSPP, when Peer A initiates a channel with Peer B, Peer A sends a first datagram with a HANDSHAKE message, and Peer B responds with a second datagram also containing a HANDSHAKE message (see Section 4.2.1). When Peer A sends a third datagram to Peer B, and it is received by Peer B, both peers know that the other is listening on its stated transport address. Peer B is then allowed to send heavy payload like DATA messages in the fourth datagram. Peer A can already safely do that in the third datagram.

In the Unified Merkle Tree scheme, Peer A MUST send its rightmost signed munro hash to Peer B in the third datagram, and in any subsequent datagrams to Peer B, until Peer B indicates that it possess a chunk with the same corresponding munro or a more recent munro (by means of a HAVE or ACK message). Peer B may already have indicated this fact by means of HAVE messages in the second datagram. Conversely, when Peer B sends the fourth datagram or any subsequent datagram to Peer A, Peer B MUST send its rightmost signed munro hash, unless Peer A indicated knowledge of it or more recent munros. The rightmost signed munro hash of a peer is defined as the munro hash signed by the injector of the rightmost subtree of width NCHUNKS_PER_SIG chunks in the peer's Merkle hash tree. Peer A MUST NOT send the signed munro hash in the first datagram of the HANDSHAKE procedure and Peer B MUST NOT send it in the second datagram as it is considered heavy payload.

When a peer receives a SIGNED_INTEGRITY message with a signed munro hash but the timestamp is too old, the peer MUST discard the message. Otherwise, it SHOULD use the signed munro to update its hash tree and pick a tune-in in the live stream. A peer may use the information from multiple peers to pick the tune-in point.

4.5.2. Forgetting Chunks

As a live broadcast progresses, a peer may want to discard the chunks that it already played out. Ideally, other peers should be aware of this fact so that they will not try to request these chunks from this peer. This could happen in scenarios where live streams may be paused by viewers, or viewers are allowed to start late in a live broadcast (e.g., start watching a broadcast at 20:35 where it actually began at 20:30).

PPSPP provides a simple solution for peers to stay up to date with the chunk availability of a discarding peer. A discarding peer in a live stream MUST enable the Live Discard Window protocol option, specifying how many chunks/bytes it caches before the last chunk/byte it advertised as being available (see Section 7.9 of [60]). Its peers SHOULD apply this number as a sliding window filter over the peer's chunk availability as conveyed via its HAVE messages.

Three factors are important when deciding for an appropriate value for this option: the desired amount of playback buffer for peers, the bitrate of the stream, and the available resources of the peer. Consider the case of a fresh peer joining the stream. The size of the discard window of the peers it connects to influences how much data it can directly download to establish its prebuffer. If the window is smaller than the desired buffer, the fresh peer has to wait until the peers downloaded more of the stream before it can start playback. As media buffers are generally specified in terms of a number of seconds, the size of the discard window is also related to the (average) bitrate of the stream. Finally, if a peer has few resources to store chunks and metadata, it should choose a small discard window.

4.6. Extensibility

4.6.1. Chunk Picking Algorithms

Chunk (or piece) picking entirely depends on the receiving peer. The sending peer is made aware of preferred chunks by the means of REQUEST messages. In some (live) scenarios, it may be beneficial to allow the sender to ignore those hints and send unrequested data.

The chunk picking algorithm is external to the PPSPP and will generally be a pluggable policy that uses the mechanisms provided by PPSPP. The algorithm will handle the choices made by the user consuming the content, such as seeking or switching audio tracks or subtitles. Example policies for P2P streaming can be found in [153], and [65].

4.6.2. Reciprocity Algorithms

The role of reciprocity algorithms in peer-to-peer systems is to promote client contribution and prevent freeriding. A peer is said to be freeriding if it only downloads content but never uploads to others. Examples of reciprocity algorithms are tit-for-tat as used in BitTorrent [75] and Give-to-Get [125]. In PPSPP, reciprocity enforcement is the sole responsibility of the sending peer.

5 Libswift: the PPSPP Reference Implementation

In Chapter 4 we presented the Peer-to-Peer Streaming Protocol (PPSPP), the official IETF Internet standard for distributing streaming content over a P2P network [60]. In this chapter we present the technical details of this protocol that are not covered by the official protocol description, and details of the implementation that are left to the developer. As any standard protocol description, PPSPP [60] defines in great detail how the protocol must behave and how information is exchanged between communicating nodes. To work at its best, a transmission protocol needs to provide several features such as a well defined and extensible communication mechanism, flow and congestion control, a good interface to external programs, and it has to make efficient use of the available resources, without being too intrusive. While the official PPSPP document clearly describes many of those features such as the communication pattern, the message structure, and the congestion control, some details of the implementation are not presented as they are out of scope for the protocol description. In addition, PPSPP presents several novel ideas and data structures for increasing its efficiency, of which the implementation details have not been discussed before.

Over the last several years, we have designed and implemented the reference implementation of PPSPP and its novel ideas and data structures, called Libswift. It is a useful reference to analyse and validate the protocol's properties and behaviour. Libswift can be used either as a stand-alone client, or as a library, which allows for an easy integration into existing applications, offering several interfaces, from a simple command line interface to more complex remote control interfaces, e.g. via HTTP or a socket. Libswift has also been widely used for the experimental research in the context of the P2P-Next project [59, 91, 97, 129, 133, 135, 141]. In the following chapter, we will deeply analyse the performance of the initial implementation of Libswift [135]. Driven by the results of experiments and experiences with real-world deployment, we have made several design choices in Libswift, that are not directly relevant to the protocol description of PPSPP, and therefore, are not discussed in the PPSPP document, RFC7574. The aim of this chapter is to provide insight on some of these design choices and approaches to solving implementation problems. We describe these choices with the goal of assisting others who are interested in implementing the protocol.

In the remainder of this chapter we describe four aspects of Libswift's implementation. Section 5.1 describes the *binmap*, a novel data structure described in the PPSPP document. While it has already been introduced and discussed in previous work [98, 135], Section 5.1 presents some implementation details and clearly defines the role of binmaps in Libswift. Section 5.2 presents Libswift's *state machine* which governs its communication pattern. This is particularly important in a highly dynamic environment such as P2P networks. Section 5.3 describes Libswift's content request approach, which is based on a pull mechanism. While PPSPP is designed to work following both a push and pull mechanism [60], our reference implementation mostly relies on a pull mechanism. This choice is mostly motivated by the unstructured network in which Libswift operates, as peers are usually organised in a mesh type structure. In a structured network, e.g., in a live distribution scenario where peers are organised in a tree structure, a push mechanism might be prefered [104, 160]. Finally, Section 5.4 presents some experimental results of Libswift in challenging environments. Our testing environments are characterised by lossy, high-latency and low-bandwidth networks. The goal of Section 5.4 is to analyse and discuss the behaviour of LEDBAT [149], which is used as Libswift's congestion control algorithm.

5.1. Bins and Binmaps for Data Availability

In PPSPP [60], *bins*, or binary numbers, are introduced as a way of addressing ranges of content. Bins are the nodes and leaves of the *binmap*, a binary tree built on top of the content. This novel approach has been first designed and introduced in the early development stages of Libswift [98]. It is an optional feature and can be replaced with piece or byte ranges, which are actually the default means of requesting content given their simplicity. Our reference implementation, Libswift, supports both approaches, but internally it handles data only through bins and binmaps. Bins are used in conjunction with time values in order to keep track of incoming requests, outstanding requests, and data that has been sent and is waiting to be acknowledged. Binmaps are used to store information about the availability of content, to identify what has been retrieved, verified, announced by neighbour peers, and so on.

In this section, we do not describe those data structures in detail, as they have been extensively presented in PPSPP and in previous work [60, 98, 135]. Instead we give an overview of how binmaps fit in the context of the reference implementation, Libswift, and how they are used.

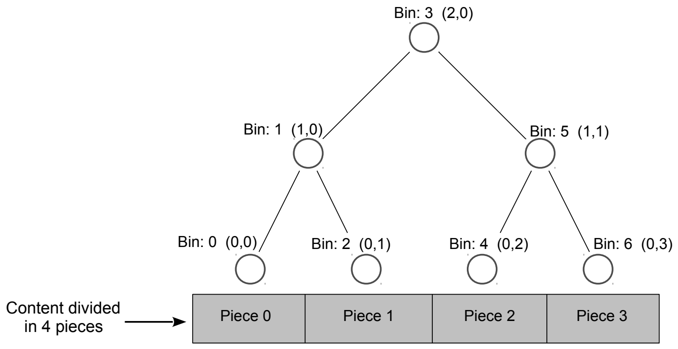


Figure 5.1: Binmap – format: *bin number* (*layer*, *offset*)

5.1.1. Binmap

In P2P systems, the content is divided into a number of pieces in order to optimise its distribution. Those pieces are represented by the leaves of the binmap, a binary tree built on top of the content. Each node of the tree, called *bin*, addresses either a single piece or a range of pieces of the content:

- Single pieces are addressed by leaf nodes, represented by bins with even numbers.
- Piece ranges are addressed by higher layer nodes, represented by bins with odd numbers.

To demonstrate the behaviour of a binmap, we will use the binmap depicted by Figure 5.1. This binmap is used to address content which is divided into four pieces (which are depicted by bin 0, 2, 4 and 6). Pieces are numbered from left to right, starting with 0 (i.e., the *relative offset*).

Bin numbers are expressed either in a compact form as a single integer, or in an extended form as the layer number and its relative offset, i.e., (*layer*, *offset*). When addressing content through a bin number, we refer to the subtree of that bin, either a single piece or a range of pieces. More precisely, the bin, identified by (layer, offset), addresses the range of pieces with numbers in:

$$[\mathit{offset} \cdot 2^{\mathit{layer}}, (\mathit{offset} + 1) \cdot 2^{\mathit{layer}}),$$

Hence, the entire content of Figure 5.1 can be addressed either as bin *3*, or as bin *(2,0)*, which addresses piece range $[0, 4)$. The bins uniquely address the nodes of the tree, that can be either full, if the content is available, or empty. The bin number can be calculated based on its position in the tree:

$$\mathit{bin} = 2^{(\mathit{layer}+1)} \cdot \mathit{offset} + 2^{\mathit{layer}} - 1$$

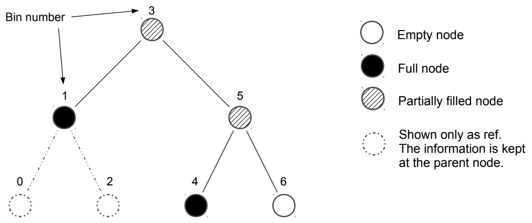


Figure 5.2: Visual representation of storing the availability information in a binmap.

Since Libswift is written in C++, the bin representation allows for efficient binary operations which provide fast tree traversal and binmap comparison. Examples of such operations in C++, where b is the bin number, are the following:

- Check if the bin is a leaf of the tree: $!(b \ \& \ 1)$ (not an odd number)
- Return the beginning of the range of a bin: $(b \ \& \ (b+1)) \gg 1$, e.g. *bin(0,0)* from *bin(2,0)*.
- Return the length of the range of a bin: $(b+1) \ \& \ -(b+1)$, e.g. 4 for *bin(2,0)*.
- Return the sibling bin: $b \,\hat{}\, (b \,\hat{}\, (b+1)+1)$, e.g. *bin(1,1)* for *bin(1,0)*.

Each node of the tree, referenced by a bin, holds a status, which is either *full*, meaning that the range of content referenced by the bin has been retrieved, *empty*, the opposite of full, or *partially filled*, meaning that only part of the bin has been retrieved. Figure 5.2 gives a graphical representation on the statuses of the binmap's nodes. Figure 5.2 also shows another important characteristic of the binmap, the aggregation of information to higher layers. In this example, bins 0 and 2 have been retrieved, hence the information is propagated to bin 1 which is full. This allows to free the memory required to store the status of bin 0 and 2, making the binmap a very lightweight data structure when distributing large content. Furthermore, it allows to quickly determine if a range of content, or bin, has been partially or fully retrieved.

5.1.2. Availability

Every P2P system requires a way of sharing information about what parts of the content are available at each peer. Peers exchange information with each other on what has already been downloaded, often driven by an incentive mechanism such as tit for tat [75].

The standard choice for storing availability information in P2P systems, e.g., in BitTorrent [5], is the bitmap. In contrast, we decided to implement and develop the binmap data structure, as it provides several advantages over the standard bitmap, such as low hardware requirements, tree compression in case of redundant information (e.g., one side of the tree being completely full or empty), fast tree traversal, and easy comparison [98]. In Libswift, binmaps are used for two purposes: representing the availability of content 1) between two peers, and 2) in the swarm.

Availability of Content Between Two Peers

Each peer stores one binmap representing the content it already retrieved and has locally available, and one binmap for each peer it connects to, representing the content that the other peer announces to have locally available. Requests for content and content availability announcements are performed by comparing the local binmap and the binmap the other peer announces.

Availability of Content in the Swarm

Binmaps are also used to represent the overall availability of content in a swarm. In a swarm, each piece of content is available 1 to N times, where N is the total number of peers in the swarm. The local view of the availability of pieces is given by what neighbour peers announce to have, hence it is limited by the number of connections a peer establishes, defined as n. In order to efficiently select the rarest pieces to retrieve, we introduce a novel data structure, called the *availability array*. A peer creates an availability array for each swarm it joins that has more than two active peers. It is an array of binmaps, where the index represents the data replication in the swarm. For example, if only one peer announces to have bin x, than x is filled at the binmap with index 1, and the bins that are filled in the binmap at the n-th index of the availability array represent the content that has been announced by n peers.

The availability array has an important property, which is the complementarity of its binmaps: the combination (defined as intersection) of all the binmaps in the availability array always results in one full binmap. An example of an availability array is provided in Figure 5.3

- n has already been defined,
- we call the index of the elements of the availability array *rarity index*, as it indicates the content replication in the swarm.

Imagine that the example of the availability array presented in Figure 5.3 refers to a swarm in which the content is divided into four pieces, following the example of Figure 5.1. It shows that no peer has announced to have bin 6 (*bin(0,3)*), as it is filled at index 0, one peer has bin 4 (*bin(0,2)*) and two peers have bin 1 (*bin(1,0)*), the first 2 pieces of content. The binmaps for indices 3 to n are not shown as the first 3 indexes are complementary and create a full binmap, meaning that the intersection of these binmaps results in a full tree. In other words, the replication of all 4 pieces of the content is known, hence the remaining indexes of the availability array necessarily point to empty binmaps.

The availability array is updated if either a new announcement is received, e.g, a connection with a new peer is established and it announces what it already retrieved, an existing peer updates its announcement, or if the connection with a peer is closed. Figure 5.4 depicts the situation after an announcement for bin 5 (*bin(1,1)*)

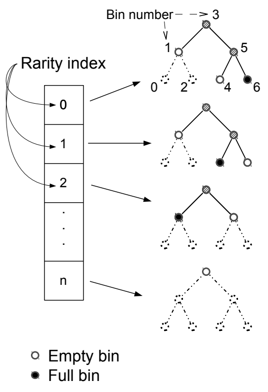


Figure 5.3: The availability array.

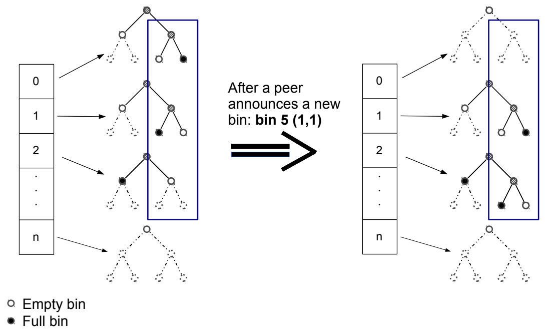


Figure 5.4: The availability array modifications after a new announce for a bin is received. The new announce is is for bin 5, hence only the right hand side of the trees change.

is received. The availability array from Figure 5.3 is now updated as one peer has piece 3 available, and two peers have piece 2 available.

The size of the availability array, that is, the length of the array, is a customisable parameter and directly affects the accuracy of the availability view. By default, it is set to n, the maximum number of connections that will be initialised by a peer. In some circumstances, such as on hardware-limited devices which simultaneously participate in several popular swarms, it can be reduced to a lower size in order to lower memory and computational requirements. In case more connections are established than can be represented by the availability array, some loss of accuracy occurs as the omitted information is combined on the last index of the array.

5.2. Libswift's State Machine

The official PPSPP document [60] defines how the protocol should react to general dynamics of a P2P network environment, such as peers unexpectedly leaving the system, but it does not define how the protocol should behave in other circumstances, such as intermitted content requests. This section aims to clarify how in Libswift we determine when to send data, when to wait, and when to close a connection. Contrary to the client-server scenario, P2P protocols need more than a single sending state in order to provide efficient data transmission. This is mainly caused by the absence of a single uninterrupted byte stream: generally, peers want to keep active connections with each other, as even peers that do not have any useful data to offer might retrieve new pieces of content at any time. In Libswift, peers have three different sending states:

- *Ping-pong*
- *Congestion avoidance*
- *Keep-alive*

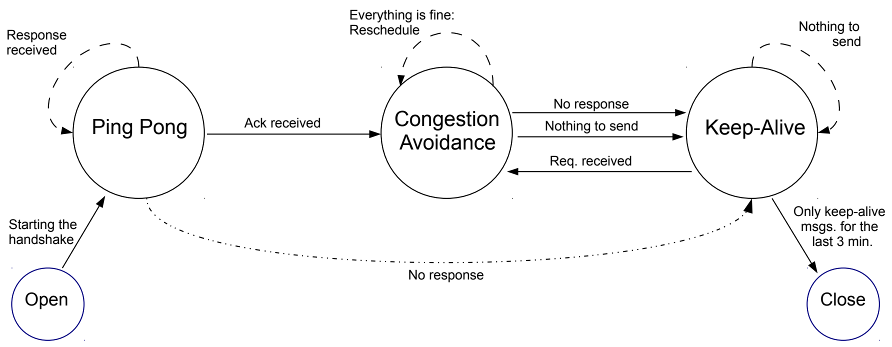


Figure 5.5: Libswift's state machine.

For each connection a peer establishes, it switches between the three sending states based on the requests and acknowledgements it receives. Figure 5.5 presents the state diagram of the state machine. In the remainder of this section we analyse each sending state in detail.

5.2.1. Ping-Pong State

When initialising a connection, during the initial handshake, a peer starts in the ping-pong state. During this state, the peer sends a packet and waits for a reply before sending the following packet. The sending state changes when either no reply is received within a certain period, moving to the keep-alive state, or when an acknowledgement is received, in which case the state changes to congestion avoidance.

5.2.2. Congestion Avoidance State

The congestion avoidance state is a sending state in which a peer is actively sending data to another peer in the network. It is based on a congestion window (*cw*), defined as the number of allowed outstanding packets at any time. It is divided into two stages:

1. Initially a *slow start* approach is applied, as defined in [54],
2. then the state moves to the *congestion control* stage.

During the slow-start stage, *cw* is initialised to one packet, and increased by one for each acknowledgement received. The congestion window is linearly increased until one of the following happens:

- An acknowledgement is not received on time, following the retransmission time-out presented in [54]. Result: *cw* is halved.

- The following applies: $si < 100ms$, where *si* is the sending interval, defined as rtt/cw, and *rtt* is the estimated round-trip time (RTT) average. Result: the stage moves from the slow-start stage to the congestion control stage.

The congestion control stage follows the Low Extra Delay Background Transport (LEDBAT) standard [149], which adjusts the sending rate based on the one-way delay samples (*owd*). The owd is measured over the data path, hence it is the time required for data packets to arrive from the sender to the receiver. LEDBAT is a delay-based congestion control which aims to utilise the available bandwidth without interfering with concurrent traffic, and has been selected given its widespread adoption [5, 15, 40, 41]. Previous work provides an in-depth analysis of LEDBAT [70, 140, 151].

The connection will stay in the congestion avoidance state and switches to the keep-alive state if one of the following conditions occurs:

- No packet has been received during the last $8 \cdot rtt$ seconds.
- *si* is greater than the maximum of 4 seconds and $4 \cdot rtt$.
- There are no outstanding requests to be served.

For the first two conditions, the sending state switches to keep-alive mode as expected, resetting the *cw* and other parameters, while the third condition presents an exception. The third condition differs from the other two as the connection might still be active, and should not be restored as it would take longer to redetermine the optimal sending rate. There might not be any outstanding requests for three different reasons:

1. The peer's requests might be lost, if on a lossy network such as wireless or mobile links, or they might be dropped by a router if the link is heavily congested.
2. It frequently occurs during a live streaming transmission when the peer's download speed is higher than the video stream bitrate. In this case the peer can retrieve the content faster than it is generated at the source, therefore it will not have enough requests for content to fully utilise its link.
3. The requesting peer might already have the pieces announced by the sending peer, but considering the P2P environment, new pieces might become available at any time, hence the connection must be kept alive.

Hence, in case new requests are received within a short period of time, set to $4 \cdot rtt$ as default but possibly changed based on the specific environment, the sending state switches back to the congestion avoidance state resuming the transmission at the same rate as before.

5.2.3. Keep-Alive State

The keep-alive state and its transition to other states are presented in the official document [60]. We discuss it in this document for clarity. If there is nothing to be sent, as no request has been received, a peer stays in the keep-alive state. In this state, a peer sends packets with no payload, containing only the keep-alive signal, to notify the other peer of his presence. A typical scenario is a leecher communicating with a seeder. In this scenario the leecher stays in keep-alive state while the seeder switches between sending states based on the leecher's incoming requests and acknowledgements. If two connected peers are both in keep-alive state, e.g., two seeders, the interval between keep-alive packets increases over time, up to a maximum of 3 minutes, after which the connection is closed. A peer switches to the congestion avoidance state once it receives a new request for content.

5.3. Pull Mechanism

Our reference implementation applies a pull mechanism in order to retrieve data from other peers. By directly requesting the data, the receiver peer (RP) has control over what will be received from any of his neighbours, referred to in this section as sending peers (SPs). In the official document [60] there is no description of the specific pull mechanism strategy, as it is left to the implementer to decide, hence this section presents the approach implemented in Libswift.

The general goal in a P2P network is to retrieve the content as soon as possible, which is achieved by maximising the rate of incoming data. Hence, the aim of the RP is to guarantee that the SP always has an outstanding queue of requests that need to be served, assuring that it never stays idle. If the SP runs out of requests to serve, it will switch to the *keep-alive* state and the flow of data is halted until new requests are received (see Section 5.2.2). On the other hand, the RP should not overwhelm the SP with requests, as the number of requests affects the reaction time to sudden network changes, such as a congested links or peers leaving the network. Furthermore, some other aspects need to be considered, which might affect the efficiency of data retrieval for the individual peer or for the entire network. For example, in a streaming context, it was shown to be more beneficial to retrieve the content at a download rate close to the stream bit-rate [79], rather than retrieving it as fast as possible like in a file-sharing context.

In the remainder of this section we present the strategies implemented in Libswift in order to optimise the rate of incoming traffic in every scenario, whether it is static or dynamic content. It is left to the implementer to choose the strategy that best fits his needs.

5.3.1. RTT Estimation

In many protocols that apply a pull mechanism, the standard way of determining the amount of content that should be requested relies on the time required for a request to be served, hence by estimating the round-trip time (RTT). In Libswift, we do not rely exclusively on the RTT estimation, as it might be unreliable for the RP. In LEDBAT, only the peer that is actively sending data has a correct estimation of the RTT, as it is calculated based on the received acknowledgements (acks). Since acks are sent by the RP as soon as possible [149], the SP calculates the RTT as the time interval between sending the data packet and receiving its acknowledgement. The SP needs to keep a correct RTT estimation as it is used in Libswift to determine the retransmission time-out, using the same approach as TCP [131].

On the other hand, the RP cannot estimate the RTT accurately. The only way for the RP to calculate the RTT would be to analyse the time elapsed between requesting a chunk of content and receiving it. This time is influenced by several factors, as the time needed for the SP to serve the request depends on:

- The number of outstanding requests.
- The processing time at the SP, as it might vary depending on the number of hashes that need to be sent.
- The number of concurrent streams, as the SP divides the available bandwidth on its upload link over all the RPs from the swarms in which it is participating.

All this information is unavailable to the RP, hence the RTT cannot be used as the main parameter to determine how much content should be requested.

5.3.2. The Data Inter-Arrival Period

In Libswift, we rely on the data inter-arrival period, at the RP, to determine how much content should be requested by the RP to the SP. The data inter-arrival period (dip) is defined as the amount of time elapsed between successive arrivals of packets containing data messages. Hence, the dip is the most accurate way of estimating the actual sending rate of the SP. The RP determines whether to request content to the SP based on Algorithm 1.

Algorithm 1 Determine the amount of data to be requested.

```
1: rt ← max(1s, 4 * RTT)                 ▷ rt: request time, usually 1 second
2: rp ← max(1, rt/dip)                   ▷ rp: request packets, number of pkts in rt sec.
3: ra ← max(0, rp − ro)                  ▷ ra: allowed requests; ro: outstanding req.
4: rs ← min(ra, rate allowed)            ▷ rs: req. size; rate allowed: take rate limits into
   account
5: if rs > rg then                       ▷ rg: req. granularity, avoids req. fragmentation.
6:     req ← Pick(rs)
7:     if req = 0 then
8:         return                        ▷ end-game: nothing can be requested
9: return req
```

The first line of Algorithm 1 might be misleading as the RTT estimation is used. We consider the RTT estimation only for extremely slow connections where the RTT is greater than 250ms. The RTT value is always initialised during the handshake, as peers reply to handshakes immediately, and is updated only through acks. Therefore, even peers communicating with seeders have a rough estimation of the RTT, but it is never updated and unreliable. The RTT is only used to guarantee that there are enough outstanding requests, while lines 1 to 5 of Algorithm 1 calculate how much content to request considering the number of outstanding requests, the potential rate limit, and the desired requests granularity (a configurable parameter). The aim of Algorithm 1 is to limit the size of the request, or bin layer, while the *Pick* function of line 6, described in the following section, determines which bin to request.

5.3.3. Piece Selection

In Libswift we provide different piece-picking algorithms depending on the required retrieval pattern, e.g., static content (file-sharing), time-critical content (VoD) or dynamically generated content (live stream). The piece-picking function *Pick* (line 6 of Algorithm 1) is called each time new content needs to be requested from another peer. Libswift features four piece-picking algorithms, which differ in the way they prioritise ranges of content:

1. *Sequential*, the simplest piece-picker as it just retrieves the content sequentially. A small variation can be introduced by setting some randomness when traversing the binmap.

2. *Rarest-first*, selects the rarest bin, within the limits of rs from Algorithm 1, that the SP has to offer across the entire content.

3. *Video-on-Demand*, divides the content in different sets of priorities, from high to low, selecting the rarest content the SP has to offer starting from the highest priority set. The algorithm has been presented in [135].

4. *Live stream*, returns the largest bin available from the current playback position in the stream.

The rarest-first and Video-on-Demand algorithms apply the same strategy in order to select the rarest available bin that can be retrieved from the SP. They only differ in the range of pieces which are considered at each execution. They iterate through the availability array (*Avail*), described in Section 5.1.2, from the lowest to the highest index. While iterating through *Avail*, from the rarest to the most popular content, the algorithm performs a three-way comparison with the binmap representing the content that the SP has to offer, and the binmap representing what the RP has already retrieved, both defined in Section 5.1.2. The three-way comparison can be expressed using the set intersection and complement as $(\text{Avail[i]} \cap \text{SP}_{binmap}) \setminus \text{RP}_{binmap}$, where i is the index of the availability array addressing what i neighbour peers announce to have. An example of selecting the rarest content from a seeder is presented in Figure 5.6. In this example RP selects the second piece of content, bin 2 or (0,1), as it is the only full bin in the resulting binmap. In case the resulting binmap has more than one full bin, the RP selects the bin to request depending on which piece picking algorithm is used, e.g., the bin closer to the current playback position or randomly.

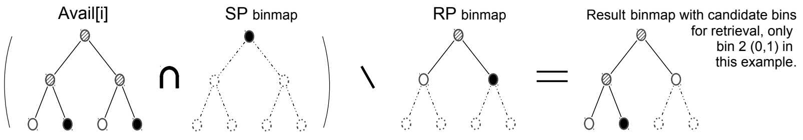

Avail[i]
SP binmap
RP binmap
Result binmap with candidate bins for retrieval, only bin 2 (0,1) in this example.

Figure 5.6: An example of selecting the rarest content to request from a sending peer.

As a final note regarding the pull mechanism, it should be noted that the number of requests sent to each peer also influences the rate at which data will be retrieved, as lowering the number of requests automatically leads to a lower link utilisation. This characteristic can be exploited in order to prioritise specific connections over others, e.g., connections which feature a low delay can be dedicated to the most time-critical requests (as for missing content close to the current playback position).

5.4. Experiments

In this section we provide an evaluation of Libswift. We analyse its behaviour by emulating different network conditions, and present the results.

5.4.1. Experimental Set-up

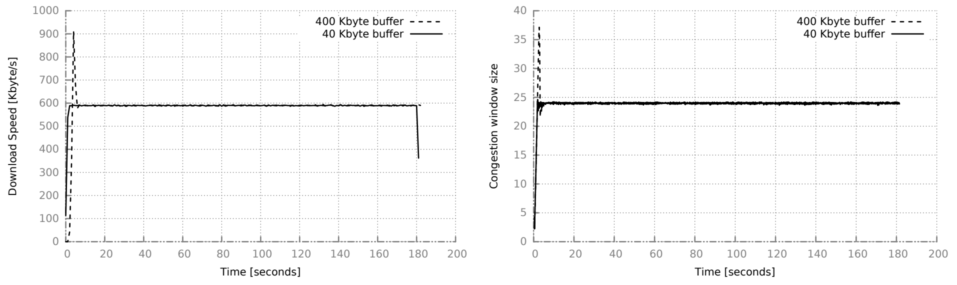
To simplify our analysis we emulate only two peers exchanging content. Analysing only two peer allows us to correctly identify the behaviour of the protocol, highlighting its downsides when operating in challenging environments. In previous work we have analysed the behaviour of Libswift in big swarms [135]. In our emulations one peer is the content provider, the *seeder*, while the other peer is the content consumer, the *leecher*. The leecher is initialised once the seeder loads the content, has generated the required hashes, and is ready to distribute it. The leecher network link speed is set to 1 Gbit/s (hence it is practically not limited) and we vary the connection speed limit on the seeder's network link. We configure the end-to-end delay between the leecher and seeder to be 60ms. The content is 100 MByte in size and is divided in 1 Kbyte chunks, the default value.

We consider the *completion time* as the most important metric to evaluate the performance of our protocol. We present the *download progress* for each experiment, showing both the completion time and the download progress over time. We also present the *download speed* over time, discussing the efficiency of the protocol in exploiting the available bandwidth. The download speed is directly related to LEDBAT's *congestion window*, as the congestion control dictates the sending rate, hence we also present it in some experiments to clarify Libswift's behaviour.

Our experimental environment uses LXC containers[1] and netem[2] to emulate various network conditions. An LXC container is a lightweight environment which provides full resource isolation and resource control for an application or a system. LXC containers can be considered a lightweight virtualization method for running

[1]`https://linuxcontainers.org/`

[2]`http://www.linuxfoundation.org/collaborate/workgroups/networking/netem`



(a) The leecher download speed over time. (b) The seeder's congestion window over time.

Figure 5.7: The behaviour of Libswift over a 5 Mbit/s link (the curves almost overlap).

multiple isolated Linux systems (containers) on the same host. All experiments were conducted on a single machine with a single dual Intel Xeon CPU 2.40GHz and 8GB of memory, running Debian with kernel 3.12. We used a customised experiment framework, called Gumby[3], to conduct the experiments in an automated fashion. For the specific container setup details, we refer the reader to the Gumby documentation[4].

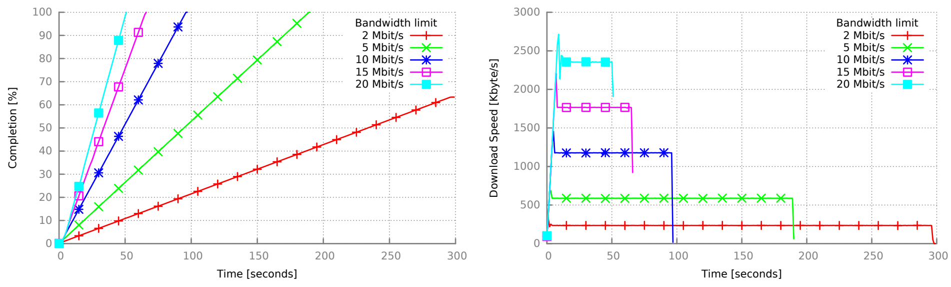
We start each peer in the experiment in its own container. To emulate various network conditions, we limit outgoing and incoming traffic to the containers using netem. Netem allows the definition of network traffic control rules on a network device. By setting these rules for the network devices inside the container, we can specify traffic rules for one peer without affecting the network speed of the other peer in the experiment. For every container, we can specify the download and upload rate in bits/s, the delay in ms and the percentage of lost packets on the outgoing traffic. In addition, we can specify the burst sizes for the incoming and outgoing connection.

5.4.2. Network efficiency

In this section we show how Libswift efficiently exploits the available bandwidth. Figure 5.7 shows its performance with a network link limited to 5 Mbit/s. Figure 5.7(a) presents the seeder's upload speed while Figure 5.7(b) presents its congestion window. Both metrics are important to analyse the protocol's performance. The stability of the sender's congestion window shows that LEDBAT correctly identifies the optimal sending rate, which results in a stable download speed for the leecher. In both figures we present two experiments which show how buffers on the data path, usually introduced by routers, influence the time required to reach the optimal network speed. A stable download speed is especially important in streaming applications, where it is used to determine if it is possible to provide a constant playback for the end user. LEDBAT, hence Libswift, increases the sending rate

[3]`https://github.com/tribler/gumby`

[4]`https://github.com/Tribler/gumby/tree/devel/experiments/libswift`



(a) The download progress over time.

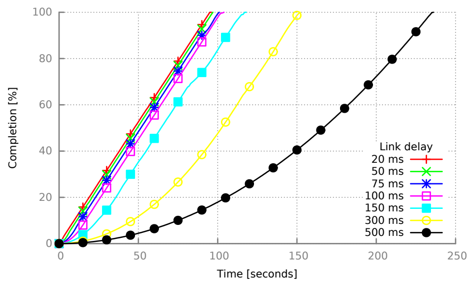
(b) The download speed over time.

Figure 5.8: The behaviour of Libswift with a a network link capacity ranging from 2 to 20 Mbit/s.

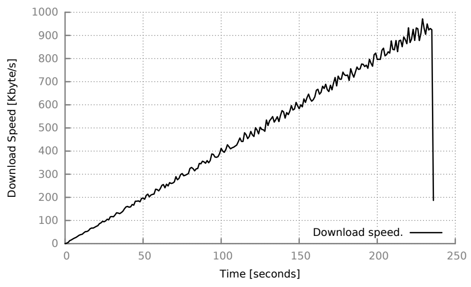
until either a packet is lost, or the one-way-delay starts to increase. This behaviour can be seen at the beginning of the transmission before the sending rate reaches a stable value, causing the small initial spike shown in Figures 5.7(a) and 5.7(b). The intensity of the spike, and the time required to reach a stable speed depend on two factors, the end-to-end latency and the intermediate buffers. The higher the latency, the longer it will take to receive acknowledgements for sent data. Hence it will take longer to adapt the sending rate. This behaviour is investigated in the following section. The buffers present on the data path, e.g., router's buffers, interfere with the sending rate calculation. Small buffers lead to a quicker adaptation, while large buffers delay the adaptation process. In Figure 5.7 we present the effect of changing the router's buffer from 40Kbyte, the continuous line, to 400Kbyte, the dashed line. While a 40KByte buffer quickly leads to the optimal link utilization, the larger 400Kbyte buffer introduces an initial spike, as LEDBAT requires more time to determine the proper upload speed. Figure 5.8 presents Libswift's behaviour with different bandwidth capacities. This experiment shows that the sending rate properly adapts to the available bandwidth.

5.4.3. Delay

We investigate the effect of changing the delay between the seeder and the leecher. The leecher's delay is fixed at 20ms, both on the outgoing and incoming packets, while the seeder's delay on the data path varies from 25ms to 500ms depending on the experiment. Figure 5.9(a) shows the download progress over time, and clearly illustrates how a longer delay influences the time required to reach the optimal speed and fully utilise the network link. The delays shown in Figure 5.9(a) are the one-way-delay (owd) values on the data path, as those influence LEDBAT, hence the protocol's behaviour. The difference in download progress for owd delays lower than 100ms is rather marginal. For end-to-end delays greater that 150ms the congestion control, LEDBAT [149], requires a long time to adapt to the available bandwidth. Figure 5.9(b) shows how with a delay of 500ms it never actually reaches a sending rate of 10 Mbit/s when sending the 100 MByte test file.

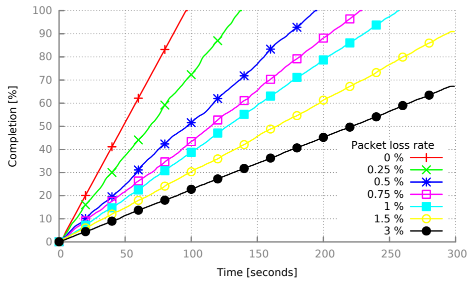


(a) The download progress with different end-to-end delays.

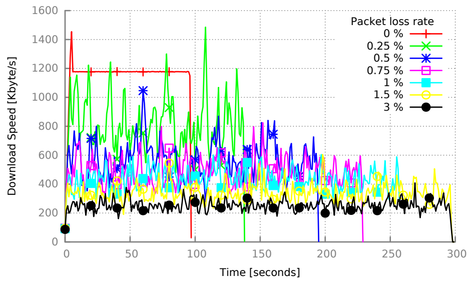


(b) The download speed over a 10 Mbit/s link with 500ms latency.

Figure 5.9: The behaviour of Libswift with different end-to-end delays over a 10 Mbit/s link.



(a) The leecher's download progress over time.



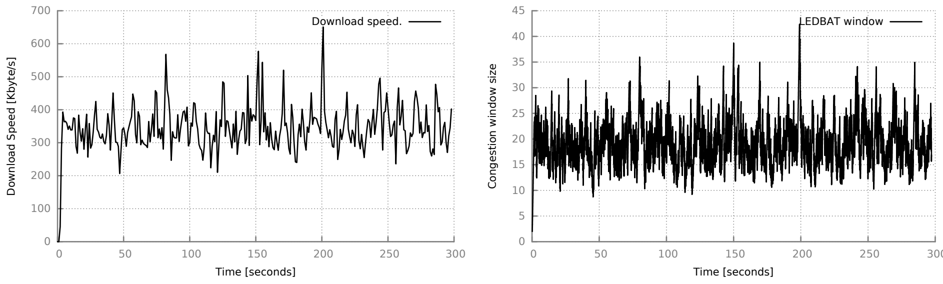
(b) The leecher's download speed.

Figure 5.10: The behaviour of Libswift depending on the packet loss rate over a 10 Mbit/s link.

This behaviour is the result of LEDBAT's design decisions, as it was designed as a low delay congestion control. LEDBAT tries to reach a *target* delay on the data path. The target delay is defined as the maximum queueing delay that LEDBAT may introduce in the network. With large delays, LEDBAT does not increase the sending rate as fast as with low delays, in the eventuality that the higher delay values might be caused by other traffic. LEDBAT's RFC specifies that the target delay needs to be less than 100ms, which is the same value used in Libswift and in the UTP protocol [41], used by other P2P protocols such as BitTorrent [5]. This explains the performance degradation when the delay is greater than 100ms, see Figure 5.9(a). In the real world, a target delay value of 100ms has proven to be a suitable candidate for peer-to-peer communications. The main exception is for high latency network links, such as satellite links, where a longer time is required to reach optimal download rates.

5.4.4. Packet Loss

An important performance characteristic that needs to be considered when using LEDBAT, as we do in Libswift, is its behaviour on a lossy network. LEDBAT does



(a) The leecher's download speed over time.

(b) The congestion window over time.

Figure 5.11: The behaviour of Libswift with a 1.5% packet loss over a 10 Mbit/s link.

not distinguish between packet losses and congestion. Hence, packet losses are interpreted as a sign of a congested network link, causing the congestion window to be reduced, therefore decreasing the sending rate. Figure 5.10 shows the performance degradation related to the packet loss rate. We present the result of 7 experiments, each characterized by a different packet loss rate. The packet loss rates we investigate are between 0% and 3%. Figure 5.10(a) presents the effects on the download progress, while Figure 5.10(b) clearly illustrates how the protocol is able to reach a stable sending rate only when the link presents no packet loss.

Figure 5.11 shows in more detail the relation between the leecher's download rate (Figure 5.11(a)) and the seeder's congestion window (Figure 5.11(b)). Here, the performance degradation on lossy networks becomes clear, as opposed to the behaviour on a perfect network (see Figure 5.7).

5.4.5. Performance Limitations

The biggest limitation of Libswift is the computational requirement of the content integrity protection scheme, which is particularly evident on high capacity network links. Libswift applies a data atomic principle, which states that every packet transferring data should hold all the information needed to verify the data it contains. This means that for every packet containing a data chunk, the sender needs to include all the hashes required to verify it. A peer starts retrieving the content with only the root hash as content identifier, which is therefore the only hash it can initially use to verify incoming data. The receiver, after retrieving a data packet, needs to verify the data against a valid hash, might it be the root hash or a hash of an intermediate node which has been previously validated. Only after validation, it will send an acknowledgement to the sender and it will notify other peers of the newly downloaded chunk. This content validation is the most resource consuming process in Libswift, taking up to 50% of its execution time [141].

The default piece/chunk size in PPSPP is 1 Kbyte, as it fits well within the boundaries of the average Ethernet MTU, which is 1500 bytes. Furthermore, a small piece size has also proven to be a good candidate for live streaming systems [125].

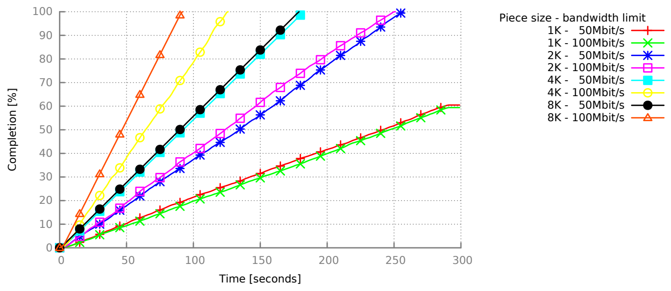


Figure 5.12: The download progress over time of Libswift on high capacity networks for varying piece sizes and link capacities.

Libswift is a single-threaded application, and the performance it can achieve with the default piece size of 1 KByte is limited by the capabilities of the CPU. While it should not be a limiting factor for live streams, as the protocol can easily retrieve a high bitrate stream on a low-end CPU, 20 Mbit/s in our experiments, the piece size might be increased to reach higher download speeds.

Figure 5.12 presents the download progress over two high speed links, 50 Mbit/s and 100 Mbit/s, when varying the piece size from 1 Kbyte to 8 Kbyte. For the experiments presented in Figure 5.12, the test file is 1 GByte in size and both clients are executed on the same CPU. Using the default piece size of 1 Kbyte does not allow Libswift to fully exploit the available bandwidth when running on a low-end CPU. This behaviour can be observed in Figure 5.12, where the lines for 1 Kbyte piece size show how the protocol only retrieves ∼60% of the 1 GByte test file during the first 300 seconds. On this specific CPU, to fully utilise a 50 Mbit/s link, the piece size should be increased to 4 Kbyte, while a piece size of 8 Kbyte is a better candidate for links with (more than) 100 Mbit/s capacity.

The downside of increasing the chunk size is that in case a packet containing data is dropped in the network, more content needs to be retransmitted. Considering the case of a link with 1500 Byte MTU, if the content is divided in 1 Kbyte chunks and one packet is lost, only that packet will need to be retransmitted. On the other hand, if the content is divided in 4 Kbyte chunks, the transmission of a single data chunk involves up to 4 packets, including the required hashes and other messages, and the loss of a single packet might require the retransmission of all 4 packets.

5.5. Conclusions and Future Work

The Peer-to-Peer Streaming Peer Protocol (PPSPP) is a P2P transmission protocol for disseminating the same content to a group of interested parties in a streaming fashion. PPSPP has been published as an official IETF Internet standard and is described in detail in RFC7574 [60]. While the official Peer-to-Peer Streaming Peer Protocol document defines in great detail how the protocol must behave, the details of the implementation are left to the developer.

In this chapter, we have discussed several design choices for Libswift, our reference implementation of PPSPP. First, we have discussed the implementation of the availability map, our data structure for representing the overall availability of content in a swarm. Second, we have discussed how we have made Libswift suitable for reacting to the dynamics in a P2P network environment. Finally, we discussed the pull mechanism used by Libswift to retrieve data from other peers. In addition, we have presented a number of experiments in which we demonstrate the behaviour of Libswift under various network conditions.

Future work involves fine tuning Libswift and developing a simple GUI for running and configuring Libswift.

6 Performance Analysis of the Libswift P2P Streaming Protocol

In the previous two chapters we presented the Peer-to-Peer Streaming Peer Protocol standard (PPSPP) in Chapter 4, and the details of its reference implementation Libswift in Chapter 5. In this chapter we conclude the description of the protocol by providing a performance comparison of Libswift with the most popular BitTorrent-based clients on both high-end and power-limited devices. PPSPP is a completely new transport protocol, and its design is not compatible with the BitTorrent protocol. Nevertheless, we use the same terminology to describe the behaviour of both protocols, e.g. swarm, peer, seeder, and leecher, as those are terms that refer to P2P networks in general, and not to a specific system and its protocol. The comparison with BitTorrent-based systems is an insightful addition to the analysis of the PPSPP protocol as BitTorrent-based systems are currently by far the most popular way of sharing content in a P2P fashion, and are a natural candidate for the design of P2P-assisted systems. BiTorrent has been designed for *file downloading* with the goal of retrieving the *entire* content as soon as possible, but much work has been done on adapting BitTorrent-like protocols to fit the needs of video-on-demand and video streaming [18, 69, 99, 117, 148]. In this chapter we also describe some of the modifications we did on the Libswift reference implementation in order to better fit the streaming context. Most of these changes have affected the download scheduler (usually called the piece picker), the upload policies, and the overlay networks, but have relied on the standard transport protocols in the internet.

We propose Libswift [13, 97] as an appropriate candidate for *media streaming* applications. Libswift is a content-centric multi-party transport protocol that has been designed with flexibility and lightness in mind. It allows NAT traversal, small message-passing overhead, and has a novel data structure that allows a very small per-connection cost. It differs from most of the existing P2P systems as it lies at the transport layer rather than at the application layer. While most of the existing solutions use standard transport layer protocols designed for client-server networks such as TCP, Libswift provides a transport protocol specifically designed for P2P networks, removing features that are not needed for video distribution such as in-order retrieval. Furthermore, we present an algorithm for selecting peers from which to request time-critical data, an algorithm for ordering data requests that guarantees upload fairness to all requesting peers that applies a form of Weighted Fair Queuing, and a downloading algorithm that takes peer locality and robustness into account. Each of these three algorithms exploits information available at the transport layer of Libswift.

We have implemented a realistic testing framework that executes the P2P clients that we compare on a multicluster machine, which gives us a chance to evaluate Libswift's behaviour in a real world environment, rather than relying on simulations. Furthermore, we provide a comparison of the Libswift client with popular BitTorrent-based clients, providing more insight into its strengths and weaknesses. The remainder of this chapter is organised as follows. First, in Section 6.1, we provide a high level description of the design features of the Libswift protocol. Second, in Section 6.2, we present a peer selection and upload request ordering policy for video-on-demand that use Libswift's transport-layer features in an essential way. Third, in Sections 6.3 and 6.4, we perform an experimental evaluation of Libswift's performance on high-end and power-constrained devices, and compare it with popular, state-of-the-art BitTorrent-based protocols.

6.1. Libswift

This section describes several design characteristics of Libswift which are required to understand the algorithms and experiments presented in this chapter. A full description of the protocol is available at [60], and in Chapter 4.

6.1.1. Design Features

The Libswift protocol has been designed with simplicity and lightness in mind. It can be run over any transmission protocol [97], allowing each provider to choose its own mechanism depending on the specific needs. Libswift operates at the transport layer rather than the application layer, and given its small footprint, it can be easily integrated into the operating-system kernel as an additional transport protocol. It features very short start-up times and it has been designed for in-order and out-of-order download with a particular focus on real-time streaming.

For congestion control, the Low Extra Delay Background Transport (LEDBAT) [149] approach has been implemented and integrated into our reference implementation. LEDBAT has been designed to provide the best possible ”background” transmission, avoiding interference with TCP foreground traffic on the same network link. The algorithm has proven to be successful and has already being integrated into some BitTorrent-based clients [40]. This behaviour allows Libswift to be run in the background, without the side effects of a noticeable delay when traffic generated by other applications, such as a web browser, needs access to the link. LEDBAT is the ideal congestion control mechanism for file-sharing and video-on-demand (VoD) content, while for live-streaming a different mechanism might be applied.

Content integrity is provided by employing Merkle hash trees [58, 121]. All the hashes needed to validate the received packets are sent along with the actual content, allowing for immediate verification. In our implementation, the Merkle tree scheme is applied for every download scenario, offline file downloading, VoD, and live streaming, but the modularity of Libswift allows for an easy integration of different mechanisms. As a last note, Libswift has been designed to support both push and pull strategies, but for the scope of the experiments presented in this chapter, we implement Libswift as a push protocol, providing consistent and reliable results.

The reference implementation currently consists of about 8K lines of code, and has not yet been fully optimised. It uses the UDP protocol because of its simplicity, lack of connection set-up delays, and retransmission. UDP also fulfils the requirements of time critical applications, such as video streaming, and we therefore consider it to be the best candidate for our reference implementation.

6.1.2. Binmap

Every P2P system needs a way of representing the local availability of the downloaded content to other peers participating in the swarm. BitTorrent, which splits the content into a number of data blocks, usually about a thousand, addresses this issue by using a bitmap in which every bit represents a successfully downloaded block. Peers exchange their bitmaps to detect who already retrieved any missing blocks. While this approach guarantees a constant and easy way of sharing information, it does not scale and is not flexible enough to provide any level of aggregation of information. Being a plain array of bits, the cost of sharing this information is constant and does not scale over time.

In Libswift, to prevent the propagation to every node of redundant information, a binary tree is built on top of the bitmap of which every node represents a specific data range. The nodes in this tree, which is called a *binmap* [98], represent the availability of their left and right child nodes. If a consecutive range of data has the same status, either available or not, this information can be aggregated in a node in the tree that is at a higher layer, offering an easy way to identify missing ranges of data without having to inspect the entire binmap. Once the information of a node, also called binary interval or *bin*, is aggregated in a parent node, it is removed from the tree, thus reducing the size of the tree and space in memory. Each bin in the binmap is represented by a 16-bit value, therefore, while the leaves of the binmap have a value of either all 0 or all 1, their information is propagated to their parent bins, up to the fourth layer. This structure is repeated at the higher levels of the tree, with the root of the tree indicating whether all blocks are available or not. The novel binmap data structure merges the concept of bitmaps with the one of binary intervals, and represents an efficient way of storing information about the data availability of each connection, reducing the per-connection costs. This approach reduces the overhead in peer communication, allowing to request data and announce the local availability with a single number, the bin number.

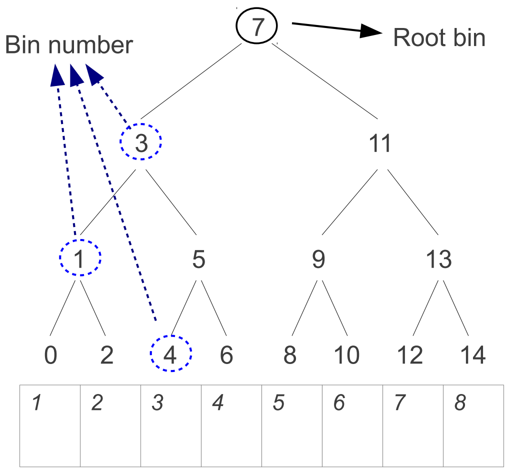


Figure 6.1: The binmap for a file divided into 8 blocks.

An example of such a binmap for a file consisting of 8 blocks is presented in Figure 6.1. In this example bin 7, called the root bin, represents the entire content from block 1 to block 8, while for instance bin 3 represents blocks 1 to 4. The default block size in Libswift is 1 KByte, but the system can easily increase the size depending on the environment and size of the content. The choice of 1 KByte data packet allows to avoid fragmentation while adding to the message the local availability, acknowledgements for previously received packets and requests for new data blocks, all represented by bin numbers.

6.1.3. Peer Communication

As previously stated, Libswift operates at the transport layer, and as it is specifically targeted at video distribution in P2P systems, it only has messages for discovering new peers, for establishing a connection, and for requesting content.

Peer discovery in Libswift is done through Peer exchange (PEX), enabling peers to exchange peer addresses with each other in order to reduce the load on the (optional) trackers. The communication between peers is performed with a small set of messages. These represent the initial handshake (HANDSHAKE), the announcement of the local availability of a piece (HAVE), acknowledgements of received packets (ACK), packets containing data (DATA) with the uncle hashes needed for verification (HASH), and requests for new content (HINT). The quality of the connection with each peer is constantly being monitored and predicted, based on the arrival of data, the round trip time (RTT), and the stability of the end-to-end link.

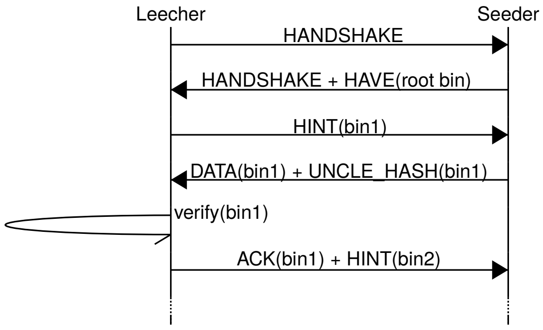


Figure 6.2: Sequence diagram of the initial interaction.

The binmap data structure used to record the content availability presented in the previous section allows for a very low per-peer communication cost. The request for hardware resources is highly dynamic and is directly influenced by the level of data fragmentation in the swarm.

Figure 6.2 shows the initial interaction between a peer and a seeder. The key characteristic is the aggregation of information within the same message. This design allows peers to start requesting data already at their second interaction (HINT), greatly reducing the warm-up time and therefore leading to short start-up times in streaming environments.

6.2. Download Scheduling

In this section we present the design principles of Libswift's download algorithm, which has been designed with a focus on providing video-on-demand capabilities, such as fast pre-fetching or buffering, and low start-up delays. The download algorithm, presented in Section 6.2.3, selects *fast* peers from which the high-priority data is requested. Section 6.2.1 presents the ranking system used to select fast peers and Section 6.2.2 presents the upload strategy introduced to achieve fairness, to have the opportunity to limit the upload capacity, and to provide a better bandwidth estimation.

6.2.1. Peer Ranking

P2P clients usually select peers to which to send requests for data randomly. The importance of selecting the right peers when playing a video stream retrieved through a P2P network has been proven by many existing systems [85, 99]. Many alternative upload strategies have been proposed [52, 117] that allow requesting *urgent* data, which is prioritised over other requests, from a selected group of peers. This behaviour guarantees that urgent data will be sent before their deadline, but it assumes an environment without selfish peers. This can however not be assumed for the Libswift system, because as the code and its policies are easily accessible to the users, selfish peers might modify their policies and mark all of their requests as urgent. While most of the solutions design an overlay network to organise the peers, we have direct access to the transport layer, and therefore, we can take advantage of direct information about the actual received bandwidth and other values such as the packet loss rate. Exploiting this information, we can select peers that provide the best end-to-end links, reducing the probability of requesting urgent data from slow or unreliable peers.

The quality of a link is calculated based on the measured bandwidth speed, derived from the RTT, the packet inter-arrival time, and the packet loss rate. The information we need is already available as every peer continuously monitors the acknowledgements for the data it sends to other peers in the swarm, and, in order to apply the LEDBAT congestion control [149], the average RTT, the bandwidth capacity, and the packet loss rate are constantly updated.

Every peer then ranks all its neighbour peers according to the value of

$$R_i = S_i - k \cdot S_i \cdot L_r \quad \text{, where} \quad S_i = P \cdot \frac{1 - RTT_i}{DIP_i}, \tag{6.1}$$

where R_i is defined as the expected bandwidth of peer i, DIP represents the packet inter-arrival time, P represents the packet size, L_r represents the loss ratio defined as the ratio between the number of lost and received packets, and k is a variable parameter that determines the influence of the packet loss rate on the bandwidth estimation. The variable k is initialised at 1, and incremented if the peer shows to have an unstable connection, resulting in big variations of the derived RTT and DIP values. Peers then sort their neighbour peers according to decreasing expected bandwidth, and select the first n peers for which

$$\sum_{i=1}^{n} R_i \geq x \cdot BR_v, \tag{6.2}$$

where BR_v is the average video bit-rate and x is a dynamic factor that varies based on the experienced playback behaviour, e.g., the number of playback stalls. These n peers represent the set of *fast peers*, to which the download algorithm will turn when requesting urgent data.

6.2.2. Upload Policy

The design of Libswift [60] does not specify a specific upload strategy when serving incoming requests. The requesters decide how much data to request based on the measured network delay, following the LEDBAT approach [149]. The initial design of LEDBAT did not take fairness in serving requests into account, but recently its authors have presented several strategies to do so [70]. The proposed solutions provide a fair distribution of the available bandwidth to concurrent LEDBAT streams, always aiming to fully utilise the available bandwidth. While this approach is well suited for high-end devices, maximising the upload stream is not always the best approach for low-end devices, such as mobile phones and set-top boxes, where resources are usually expensive. In order to be able to limit the upload capacity, we approach the problem of fairness from a different perspective. One of the main goals of Libswift is to provide a protocol that, given its small footprint, can be easily integrated on low-power devices such as mobile phones and set-top boxes. To

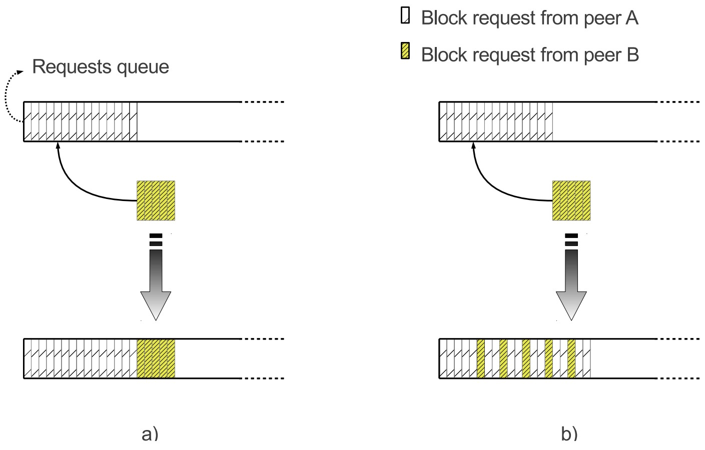


Figure 6.3: Scheduler for incoming requests. a) The original scheduler. b) The new scheduler for fairness and load distribution.

limit the upload bandwidth utilisation in environments where resources are scarce, either because of network costs or to offload the hardware during a video playback, we present a new approach that schedules the stream of outgoing packets equally among the connected peers.

We replace the original upload queue management that was serving requests following a FIFO approach, see Figure 6.3-a, with a more elaborate one to provide fairness and a good level of bandwidth estimation. We apply the well known Weighted Fair Queuing (WFQ) scheduling technique [80], that fairly distributes the available bandwidth to all the connected peers. At the beginning, all the neighbour peers receive the same number of requests for chunks. The higher the network speed with a neighbour peer, the more requests are sent. The weight used in the WFQ algorithm is assigned per peer depending on the number of outstanding requests. If the upload speed of the sender is limited, and the limit is reached, the sender will delay serving the requests, introducing a delay specific to each peer, and proportional to the number of his outstanding requests. The requester, on the other hand, will notice an increasing delay in the responses and will decrease the rate of his requests. Ideally, the algorithm reaches a stable state in which all requests are served with the same delay, and all peers send the same number of requests. If the requester is expecting urgent data, it will cancel his outstanding requests, that might be asked from other connected peers, and reduce the size of the next request to guarantee its delivery on time.

In this way it becomes trivial to predict load changes on the sending peers, and it gives a more accurate bandwidth estimation needed for the peer scheduler, see above.

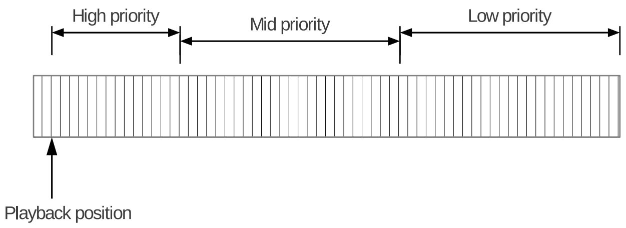


Figure 6.4: Content divided into three different priority sets.

6.2.3. The download algorithm

For the download algorithm, each peer maintains for every swarm it participates in a so-called *availability tree* in which it keeps track of its view of the global data availability in the swarm. In fact, the availability tree of a peer represents the data availability among all peers it is connected to. It has the same structure and size as the binmap in its fully expanded form, and it keeps track of the percentage of availability of each bin, or node, of the binmap tree. The cost of maintaining this tree is very small as it is only updated when new availability information is received from a neighbour peer. The availability tree allows the download algorithm to detect, and target for download, bins that have a low level of replication. This approach will balance the data availability in a swarm, and will facilitate data exchange between peers.

The download algorithm divides the video stream into three sets, a high-, a medium-, and a low-priority set [125], see Figure 6.4. The high-priority set starts at the current playback position and selects data in-order, as it represents the most urgent data in a video stream. If nothing can be downloaded from the high-priority set, the algorithm selects data from the mid- and low-priority sets in a rarest first fashion to increase their availability in the swarm. In the high-priority sets data are retrieved as soon as possible, requesting content form the set of *fast peers* defined in the previous section, which provide the best end-to-end link quality. The size of the requests depends on the number of fast peers currently in the set and the queue delay calculated given by LEDBAT. In particular it is defined as:

$$\text{size} = \min\left(\frac{W_{ledbat}}{2}, \frac{H}{F}\right), \tag{6.3}$$

where W_{ledbat} is the size of the congestion window, H is the size of the high-priority window, and F is the number of fast peers currently in the set. The algorithm keeps track of each request that has been sent, and assigns a deadline to it based on the current playback position and the loss ratio. The deadline for a packet from the high priority set is always assigned with a safety margin, allowing to forward the same request to a different peer in case that it is not retrieved on time.

The higher the speed and the reliability calculated from previous data transmissions, the bigger the requested data blocks. This approach is similar to the windowing system in the TCP protocol. Libswift constantly monitors several parameters, such as packet losses and average round trip times (RTT), trying to fully utilise the available link capacity. When downloading from the low-priority set, the algorithm iterates through the local binmap to identify missing data, through the remote peers' binmaps to see what data can be retrieved, and through its availability tree, selecting the rarest bin in the swarm.

6.3. Experimental set-up

In this section we present the environment, the scenarios, and the metrics we use to evaluate the performance of Libswift in comparison to existing solutions. Section 6.3.1 describes the experimental set-up for assessing the download algorithm presented in Section 6.2, Section 6.3.2 does so for the comparison of Libswift with BitTorrent-based protocols, and Section 6.3.3 does so for the evaluation of the performance of Libswift on power-limited devices.

6.3.1. Algorithm Comparison

In this section we describe the environment and scenarios we use to evaluate the impact of the newly introduced scheduling algorithm. We perform a series of experiments in which we analyse the behaviour of Libswift in terms of efficiency, reliability, and its capabilities of providing deadline-based streaming content. Table 6.1 presents the scenarios we use to investigate Libswift's behaviour, characterized by different proportions of seeders and leechers, with the size of the swarm ranging from 32 to 208 peers. We assume peers to be selfish, leaving the swarm as soon as they complete the download or reach the end of the playback. We compare the algorithms in this scenario, since swarm that provide an over-supply of bandwidth do not allow to easily identify misbehaving algorithms. The only seeder in the swarm is the initial content provider, which distributes a video stream encoded with an average bitrate of about 1Mbit/s.

To simulate the content playback, we implement a video player that starts consuming the content provided by the download engine as soon as the initial 5-second buffer has been filled. This initial set of data is requested by the media player in an aggressive way in order to reduce the time till playback, thus stressing the download engine. After the buffer has been filled, the player requests the content at its average bitrate, simulating normal playback. The media player stops consuming content only when it reaches the end of the stream or when the engine cannot provide the requested content, in which case it will pause and resume once the buffer has been filled again. This behaviour completely decouples the media player from the transport layer, as the download engine has no control over the rate at which the content is consumed. Furthermore, it simulates a real-world scenario, in which existing mediaplayers can use Libswift as their transport protocol.

We compare our VoD algorithm with two alternative approaches, a baseline algorithm, referred to as *linear*, and a standard priority-based VoD algorithm, referred to as *std-vod*. The first retrieves data in order, while the second divides the content into the same priority sets as our algorithm, retrieving data in order from the

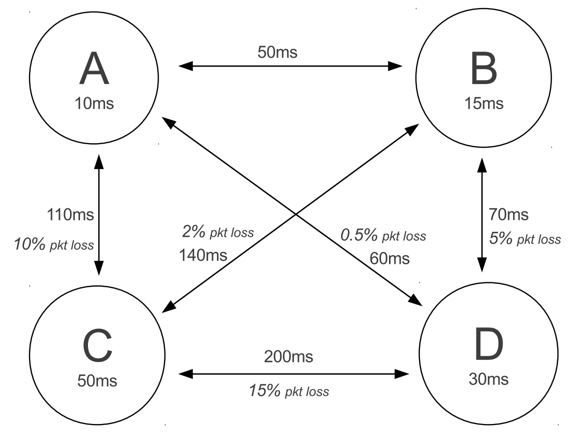


Figure 6.5: Artificial delay and packet loss rate introduced between group of peers.

high-priority set and in a rarest-first fashion from the mid- and low-priority sets. Both algorithms do not apply any kind of peer selection, and forward the requests to random connected peers.

To evaluate the performance of our VoD algorithm presented in the previous section, we divide the leechers into four groups distinguished by different connection speeds. For each group we introduce an artificial delay and a packet loss rate for each outgoing network packet, except for the seeder for which no packet loss rate is introduced. Packets sent by the seeder are characterised by the peer's delay and an additional 5ms latency. The delays assigned to each group, and introduced between the groups, are presented in Figure 6.5. The seeder distributes the content over a 20Mbit/s network link, while the upload rate of the leechers is limited to 1.2Mbit/s.

Our experiments are performed on the Distributed ASCI supercomputer [61] (DAS4), which has dual quad-core 2.5 GHz machines connected with Gbps links, to guarantee a stable environment providing consistent results and avoiding bottlenecks introduced by limited network capabilities and lack of hardware resources. Depending on the specific experiment presented in Table 6.1, we execute up to 8 clients simultaneously on each DAS node, considering that our current prototype implementation is single threaded and each process fits well within the boundaries of a single core's capabilities.

6.3.2. Libwift versus BitTorrent

To evaluate the performance of the Libswift protocol we compare it to the most popular P2P protocol, BitTorrent [5]. For our comparison we have selected the uTorrent client [40], as the most widely used P2P client [164], and the libtorrent library [15], which is currently in use by several popular projects (e.g., Limewire

Table 6.1: Experiment scenarios showing the number of leechers in the swarm.

Libswift vs. BitTorent. 4 Leechers per Node										
4	8	16	24	32	48	64	96	128	176	208

Libswift VoD algorithm comparison. 8 Leechers per Node								
	32	48	64	96	128	176	208	

[16], Deluge torrent [7] and Miro [18]) for its lightness and good performance. For our experiments we select the latest Windows version of uTorrent, as it proves to be more stable and provides more consistent results than the outdated Linux version of the client, and we implement a small client composed of few lines of code as an interface to the libtorrent library. For both BitTorrent-based clients we disable optional features, such as DHT support or the decline of connections from clients running the same host, where possible. Furthermore, we enable features such as prioritising partial pieces, to increase sharing.

In the experiments we measure the time needed to complete the download and the bootstrap time, that is, the initial start-up delay until the playback starts, which has a great impact on the quality of experience (QoE). Furthermore, we investigate the capacity of the clients to provide consistent results across all the peers in the swarm. During this set of experiments we do not implement a player that consumes the content in real-time, as those functionalities are not easily accessible on all clients. Libswift downloads data according to the algorithm of Section 6.2, with the high-priority window set to 10 seconds, which corresponds to 11.4MB, while the remaining content is retrieved in a rarest-first fashion. Libtorrent has policies to download the entire content either sequentially or in rarest-first mode. Even through we could set the download policy to retrieve the initial set of pieces in order, and then change it to rarest first, we decided to retrieve the entire content in a rarest first fashion, as this represents libtorrent's default behaviour. UTorrent, on the other hand, offers a streaming feature, but unfortunately it cannot be enabled from the integrated web interface we use to retrieve the download progress. Thus, when measuring the start-up delay for uTorrent and libtorrent, we consider the first 11.4MB downloaded from the *entire* content, giving the clients a clear advantage over Libswift, as the retrieved content does not need to be at the beginning of the file.

Also for these experiments, we use the DAS4. For the BitTorrent clients we execute the content provider and seeder on the same node as the tracker, using the popular opentracker [23], while the clients are executed on remaining nodes following the scenarios presented in Table 6.1. For Libswift, even through a tracker is considered to be optional and can be used to speed up the peer discovery phase, we consider Peer Exchange (PEX), described in Section 4.2.10, to be sufficient for our needs and rely on the initial seeder for the dissemination of peer addresses to new-comers.

We run each of the scenarios of Table 6.1 applying two inter-arrival patterns. In the first, peers join the swarm roughly at the same time, simulating a flashcrowd scenario. In the second arrival pattern, the peers are started following a Poisson interinter-arrival process with a rate of 1 peer per second, simulating a steady-state scenario. In all the presented scenarios, peers are initialised only once the seeder and the tracker are ready to receive incoming requests. The testing framework schedules the arrival of the first peer 30 seconds after the seeder and tracker have been initialised, giving the seeder enough time to perform the needed integrity checks of the 1GB seeding content. We sample the progress with 1 second accuracy, starting once the .torrent file is added to the client in the case of uTorrent, or at execution time for Libswift and Libtorrent as they receive it as an execution argument.

6.3.3. Libswift on Power Limited devices

In this section we present the metrics and scenarios we use to evaluate Libswift when executed on power-limited devices. In order to do so, we analyse its performance when running Libswift on a 450Mhz set-top box developed by Pioneer in the context of the P2P-Next project [25], and on a Samsung Galaxy Nexus smart phone running the Android platform.

During the P2P-Next project, the Pioneer Digital Design Center in London developed a low end set-top box (STB), called NextShareTV, featuring HD streaming provided through a P2P network. The initial implementation used a BitTorrent based download engine to provide the stream to the decoder, and ultimately to an HD TV. Later versions adopt Libswift as the main P2P client. To evaluate both download engines, we compare the ability to reach high download bitrates given the hardware constraints. Experiments are run by the Pioneer R&D team in an isolated environment where 7 STB peers receive the stream from the initial content provider, represented by a PC client. All the peers are connected to each other over a 1Gbit/s link, and, as for our previous experiments, clients are started within 1 second to each other. Furthermore, different from the previously presented scenarios, the retrieved stream is actually sent to a player and displayed on the connected TV, increasing the load on the hardware.

On the Android platform we evaluate the performance of Libswift in comparison to the most popular VoD application, YouTube. We implement a small interface between the Libswift protocol and the default media player available on the Android platform, to compare Libswift's capabilities to download and simultaneously playback. To evaluate the performance of the compared applications, we measure the power consumption when downloading, while playing, a 58 second 720p video content composed of about 14MByte over the same wireless network link. To provide an equal comparison of Libswift with the YouTube application, we chose the same video encoded in mp4 format. This allows Libswift to stream the content to the integrated media player using the same native decoder used by the YouTube application. Considering the fact that in the Android environment each application runs independently in its own private virtual machine, measuring the CPU and memory load would not give us sufficient inside to the actual resource utilization. We therefore measure the power consumption, physically bypassing the battery, and consider it to be the most reliable metric to analyse the resources needed to execute each application. We apply a 0.33 Ohm high side shunt providing a 99% accuracy

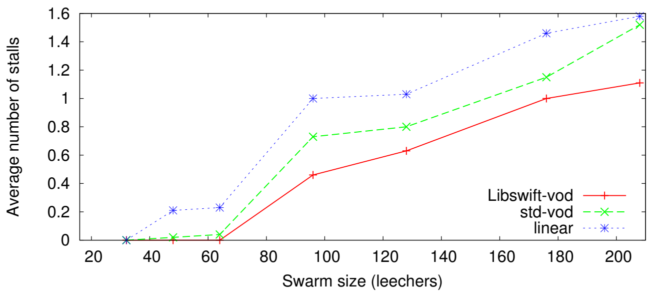


Figure 6.6: Average number of stalls experienced during playback.

[95] for our measurements, and use a NI USB-6009 data acquisition card to retrieve the battery variations. Our results are averaged over 3 consecutive measurements, where between each measurement we reboot the device and wait 30 seconds after the application is initialised, starting the download only once it provides a constant reading. Even through the battery voltage readings are known to degrade over time, and can not be avoided [102], we consider the difference not significant enough to draw our conclusions.

6.4. Experimental results

In this section we present the experimental results of Libswift. Section 6.4.1 presents the results of retrieving streaming content when different download algorithms are applied. Section 6.4.2 presents the results of the comparison with existing BitTorrent-based P2P solutions, while Section 6.4.3 presents the performance on Libswift when run on hardware-limited devices.

6.4.1. VoD Algorithm Comparison

In this section we discuss the effect that different download policies have on the QoE of the final viewer. We consider the number of stalls experienced during playback as the most important metric to evaluate the final QoE. Figure 6.6 presents the average number of stalls that occurred during playback. The worst QoE is obviously caused by the linear approach. This approach, despite its simplicity, causes a very low level of scalability and information exchange between peers, as only peers further on in the playback are able to provide useful data to newcomers. The main reason for failing in establishing a fully scalable distributed network, is the lack of global knowledge about data availability in the swarm. The standard VoD algorithm, on the other hand, holds global knowledge about data availability in the swarm, using the same availability tree data structure presented in Section 6.2.3. The effect of retrieving data in a rarest first fashion, to increase its availability in the swarm, becomes quite evident in scenarios with more than 96 leechers. Libswift's download

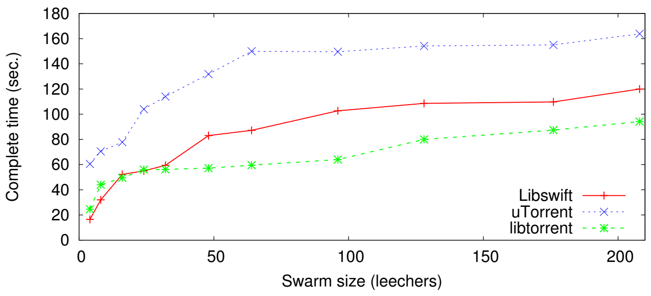


Figure 6.7: The completion time in the flashcrowd scenario.

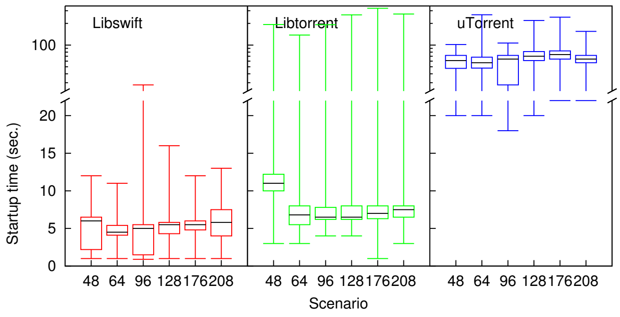


Figure 6.8: The start-up time in the flashcrowd scenario.

algorithm clearly outperforms the other approaches, selecting the best candidate peers to which request urgent data before its deadline, and in general, increasing the scalability of the system. As faster peers, belonging to group A and B of Figure 6.5, retrieve the content from the high priority set, they are selected by slow peers, from group C and D, as the best candidates for providing urgent data and increasing their QoE. We do not show the start-up time, as it proves to be very similar for all three algorithm, ranging from 3 to 7 seconds on average. This experiment also clearly demonstrates how P2P networks represent an efficient solution to increase the scalability of a system, taking fully advantage of the viewers available bandwidth.

6.4.2. Libswift vs. BitTorrent

For each of the figures presented in this section, the time refers to the relative time of each client. The time starts once the torrent file is scheduled for download, for the BitTorrent-based clients, or once the client has been launched, for Libswift. The presented results are averaged over 10 runs for each scenario and for each client.

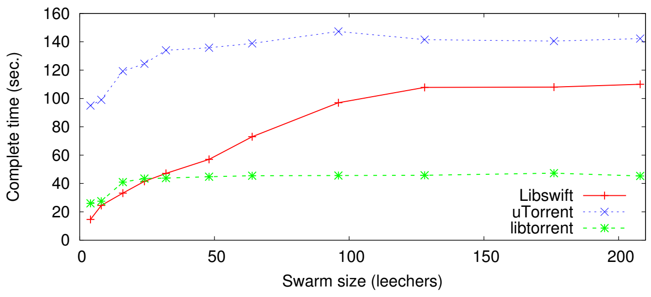


Figure 6.9: The completion time in the steady-state scenario.

Figure 6.7 shows the average time needed to complete the download in a flash-crowd scenario. On average Libswift performs right between the two compared clients, and all three clients present a similar level of scalability. For swarms that are characterised by a leecher to seeder ratio smaller than 32:1, Libswift presents better performance than Libtorrent. Both clients fully utilise the available bandwidth link, and the slightly faster download speed presented by Libswift is given by the smaller overhead in peer communication, as messages are sent within the same packet containing the actual data. For swarms that are characterised by a ratio of leechers to seeders higher than 48:1, libtorrent clearly outperforms Libswift, given its highly optimised code and algorithm that select rare pieces for download.

Figure 6.8 presents the interquartile range and the average start-up delay for the six biggest scenarios presented in Table 6.1. We only investigate the scenarios with swarms composed of more than 48 peers, as those present more interesting results. The vertical axis of Figure 6.8 presents the start-up delay using two different scales. The first 20 seconds are represented linearly, while the remainder is displayed on a logarithmic scale. This allows us to show in great detail the most interesting results, providing a higher QoE to the final viewer, while keeping worst results on the same graph.

It is clear that on average, Libswift and Libtorrent appear to have similar performance, but there is a great difference in the variability of the results. While all Libswift clients start almost always within the first 15 seconds, a small percentage of Libtorrent clients will have to wait more than one minute before being able to start the playback. UTorrent clients, on the other hand, experience a lower QoE, having to wait more than a minute on average before retrieving the first 10 seconds of video and being able to start their playback.

Figure 6.9 shows the average download time when peers join the swarm following a Poisson inter-arrival rate of 1 peer per second. In this scenario, Libswift presents slightly better results than in the previous one. This is clearly caused by the VoD oriented download scheduler. In the absence of a flashcrowd, peers are able to download all the data from the high-priority window, prioritised over the rest of

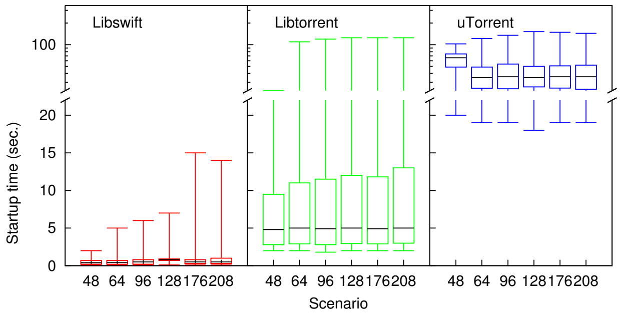


Figure 6.10: The start-up time in the steady-state scenario.

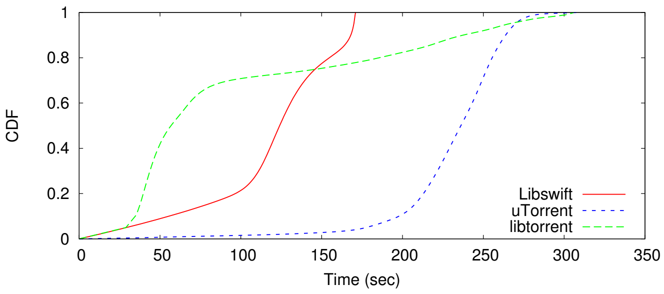


Figure 6.11: CDF of the completion time in a steady-state scenario with 128 leechers

the content, and start requesting data in a rarest first fashion, increasing the co-operation between peers. This behaviour is more evident in Figure 6.10, where the vast majority of Libswift peers retrieve the initial data needed to start the playback within the first second from their execution. As for the results presented in Figure 6.8, Libtorrent peers fill the initial buffer within the first 10 seconds, but a small minority still needs more than one minute for the buffer to fill. UTorrent clients also perform better when their arrival is linearly distributed over time, reducing by ∼20 seconds their average start-up delay, reaching a more acceptable value of 40 second.

Finally, Figure 6.11 presents the CDF of the completion time in the steady state scenario with 128 leechers and 1 seeder. Clearly, the average values presented in Figure 6.9 do not always reflect the real behaviour, as 25% of Libtorrent peers will experience a quite higher complete time. As a last note, while uTorrent appears in

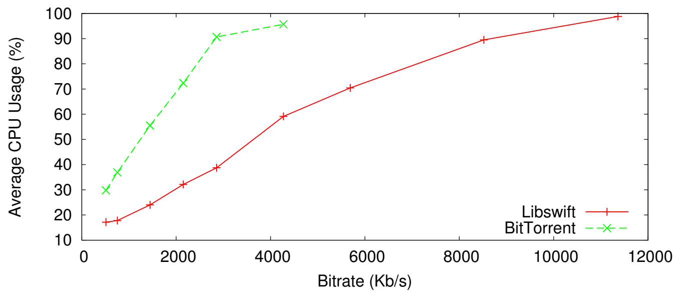


Figure 6.12: A comparison of the NextShareTV STB running Libswift and a python-based BiTorrent client as download engine.

our results to be the worst client, it is also the more mature one, offering a quite more extensive set of functionalities compared to Libtorrent or Libswift. When only one client per node is executed, libswift is the fastest, retrieving the 1GB file in only 9 seconds and fully utilising the Gbit connection, but its performance degrades a bit when the link is shared with other clients or applications. Obviously, our implementation being just an initial prototype, we can not compete with the maturity of Libtorrent and uTorrent. Nevertheless, we analyse the suitability of the clients in a highly demanding streaming context with the sole aim of retrieving the content as soon as possible.

6.4.3. Libswift on Power Limited devices

The low footprint of Libswift allows for an easy integration on hardware-limited devices.

Figure 6.12 presents the results of running Libswift as the transport protocol for the NextShareTV STB. During the P2P-Next project, also real-world experiments have been conducted, but unfortunately at that time only the original python-based BitTorrent client had been integrated into the STB. It is obvious that the performance of Libswift is far superior than the original BitTorrent client, even though much has been gained by moving from python code, needing more resources, to C++ code. Nevertheless, the integration of Libswift allows the STB to reach higher bitrates, offering a footprint of only 1MB for the code, and 0.5% of the content size for the Merkle hashes.

Figure 6.13 presents the power consumption when running the Libswift and the YouTube Android applications on the same hardware. The measurements are taken while downloading and playing the same 720p content in the same network conditions. The experiment shows that the YouTube application maintains a lower, more constant, power consumption, which is due to the fact that the application retrieves the content at a lower rate. On the other hand, the Libswift application is just a proof-of-concept, and has not been optimised for such a use. Therefore,

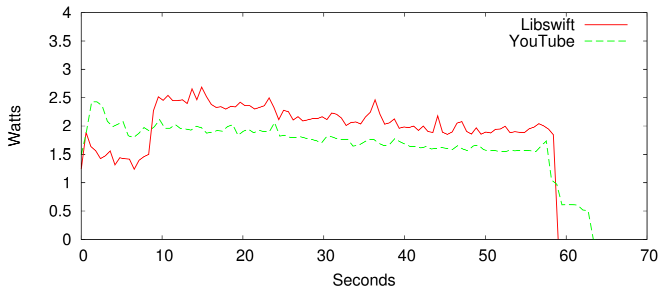


Figure 6.13: The power consumption of Libswift and YouTube when playing a 720p video stream on an Android smart phone.

it retrieves the content in an aggressive way, using fewer resources, but leading to a higher power consumption during playback. This is obvious by the significant increase in power consumption once the playback actually starts, around the 10 seconds. The aim of this experiment is not to prove the superiority of Libswift over the commercial application; instead, we want to show that Libswift, if optimised, can be a valuable alternative to provide the same QoE while distributing the content over a P2P network.

6.5. Conclusions and Future Work

In this chapter we have analysed Libswift, the reference implementation of the P2P Streaming Peer Protocol, which is an official IETF Internet standard, further explaining some of its design characteristics and goals. Furthermore, we have presented a download algorithm, specifically crafted for retrieving streaming content, that reaches a high level of QoE by exploiting the information available at the transport layer. We demonstrate that Libswift is a valuable alternative to standard P2P systems in terms of speed and efficiency, while still providing the key functionalities of a streaming protocol such as low warm-up times. We have validated the efficiency of our download algorithm by running realistic experiments on a multi-cluster machine, simulating a media player that consumes the content in real time. Results show that requesting urgent data from peers that provide a good end-to-end network link leads to a better QoE, reducing the occurrence and duration of stalls during playback.

To investigate the performance of Libswift in general terms of content retrieval and streaming capabilities, we provide a comparison with two of the most popular BitTorrent-based clients, uTorrent and Libtorrent. We showed that Libswift outperforms the compared clients when the ratio of seeders to leechers is relatively low, and in general provides more consistent results among all peers of the swarm.

On the other hand, it is yet not fully optimised, and it does not reach the same level of scalability as the highly optimised Libtorrent. Future work will focus on improving the scalability of Libswift, by investigating different strategies to keep track of the availability of data in the swarm in high churn scenarios. We also analysed the time it takes to start retrieving the content, and provided detailed results that prove the low start-up time required by Libswift, making it an ideal candidate for time-critical applications. Furthermore, we provided an evaluation of Libswift's performance when run on hardware-limited devices, such as mobile phones and set-top boxes, demonstrating its adaptability as a transport protocol in different environments.

7 Hiding User Content Interest while Preserving P2P Performance

Traditional P2P systems leak sensitive information about the content of interest of its users. Users publicly advertise what they have downloaded in order to distribute the content more efficiently. The drawback of these designs is that they allow malicious peers to easily identify the interests of the user of the system. It has been shown how crawling P2P networks can easily yield to the identification of its participants [112]. Several privacy-enhancing solutions have been proposed in order to obscure the behaviour of P2P users [62, 83, 105, 138]. The majority of those employ techniques such as onion routing [96], relaying requests through different peers, and applying multiple layers of encryption. Due to the lack of alternatives, users in need of privacy have adopted such systems for P2P file-sharing [119], joining communities which may be involved in illegal activities. The majority of users do not need to hide their activity from the ISPs and governments, as they are not interested in engaging in illegal activities, but are simply reluctant to share their interests with other participants of the system. The reasons for doing so might be various, from religious to political affiliation or sexual orientation. Our goal is to provide a solution to such users, increasing their privacy by providing a *plausible deniability* property. Malicious peers monitoring the dynamics of the system, should not be able to infer the real content of interest of a user [1]. It is plausible that a peer is downloading content for its own use, but it is equally possible that it is part of some cover traffic. An attacker, either observing or actively probing a peer, cannot conclude with high certainty whether the peer is participating in the swarm because of user interest, nor whether the peer is directly interested in the requests it is generating.

[1]We use “peer” when referring to the software, and “user” when referring to the human.

In this chapter, we propose a new system that uses cover traffic to provide plausible deniability and enhance performance. Peers create several *deception downloads*, as they always participate in multiple swarms at the same time, generating cover traffic for their *sincere downloads*. In our system it is not possible to distinguish deception downloads from sincere downloads, hence the plausible deniability property. To show the privacy properties of our system, we analyse possible attacks and quantify the probability of an attacker revealing the content interest of a target peer considering several certainty levels. Our system successfully resists both passive and active attacks by an attacker controlling a large number of peers. Note that we do not protect against an attacker that can monitor all traffic generated by a peer, as it is out of the scope of this chapter. To test the performance of our system, we run experiments on a cluster computer using the publicly released P2P client Libswift, presented in Chapters 5 and 6, the reference implementation of the Peer-to-Peer Streaming Peer Protocol (PPSPP), described in Chapter 4 and [60]. The results show that the download speed penalty is negligible when considering the performance of the system as a whole, reducing the performance penalty for long term users.

7.1. Related Work

Several solutions have been proposed to enhance privacy in P2P systems, providing different degrees of privacy to users. Most of the proposed solutions rely either on *onion routing* [96] or on *mix networks* [71]. While many solutions have not been specifically designed with P2P systems in mind, such as Tor [83] or Crowds [138], P2P users have adopted those solutions given their high level of anonymity.

Chaum [71] proposed the first system to obscure the source and the destination of data transfers using relay nodes. His main ideas were picked up by Crowds, which allows anonymised browsing and messaging by tunnelling messages through a random set of peers participating in the system.

Tor [83] applies onion routing techniques to hide information from external attackers in a system that has been designed to offer anonymity while keeping a low latency. Tor does so by tunnelling messages through a set of volunteer relay peers. Recent measurements [119] show that Tor is a popular choice for P2P users, with 40% of the traffic of an exit node representing BitTorrent data. While Tor guarantees a high level of anonymity, it lacks incentives for peers to act as relay nodes, limiting its scalability, and therefore applicability to P2P environments.

Specific solutions for P2P networks have also been proposed. Tarzan [93] applies a similar approach as Tor, but in a decentralised manner specifically designed for P2P networks. It supports two types of messages, one to create and maintain the virtual circuits of relay peers that will forward messages from the source to the destination, and one containing the encrypted data messages.

BitBlender [62] also presents some similarities to Tor, routing the P2P traffic over a selected set of peers, called “blender” peers. Upon joining a P2P swarm, peers can choose to join a dedicated blender server that will take care of forwarding their requests to the appropriate peers over a set of randomly selected relay peers.

The solutions above presented, and others such as Anonymizer [2], require a centralised node to organise the relay paths. While a good level of anonymity can be guaranteed with this approach, it is often inapplicable to real-world scenarios and lacks incentives for peers to donate their resources.

OneSwarm [105] has a more general approach. It uses social networks to select "friends" to which personal information can be revealed. Peers can choose to communicate only with peers they actually trust, avoiding to share information with "wild" peers. While this approach provides a high level of privacy, in reality the great variety of available content makes it difficult to have trusted peers interested in the same content at the same time, limiting its applicability in the real world.

SwarmScreen [73] and The BitTorrent Anonymity Marketplace [127] propose systems in which incentives are offered to the participating peers to distribute their resources over multiple swarms, obscuring their real interest from external observers. We present a similar approach, as it provides incentives for investing resources in several swarms, obscuring the users' real interests. On the other hand, our work presents a greatly enhanced solution, as peers need to invest less resources, with the advantage of benefiting from a higher level of privacy.

7.2. System Design

In this section we describe the design of our system, including details of the caching strategy and of the relay mechanism.

7.2.1. Overview

Our design builds upon the most popular P2P swarming system, BitTorrent [5]. A peer in a swarm can have the role of a *leecher* or a *seeder*, just like in BitTorrent, but also the role of *helper*. Helpers participate in swarms to generate cover traffic, and not because users are interested in the specific content shared by the swarm. The shared content in a swarm is divided into pieces and peers make requests for individual pieces. As opposed to BitTorrent, peers do not advertise their whole set of downloaded pieces to other peers, but only a subset, to avoid the identification of the seeders. However, peers can request pieces that are not advertised, and such requests might be relayed to other peers. This behavior provides plausible deniability, as helper peers are indistinguishable from normal leechers and seeders.

We do not rely on a specific incentive mechanism, as our design is compatible with all incentive mechanisms based on the download-to-upload ratio. Peers are motivated to download any piece of content that is requested by others and likely to be upload more than once. Furthermore, peers are motivated to cache and relay in order to create more cover traffic and enhance their privacy, benefiting themselves and the system. To maximize efficiency, we allow peers to download subsets of content and prohibit the chaining of relayed requests, i.e., all relayed requests must be directed to peers that announce to have the content.

The success of our system lies on the uniformity of its peers' behavior, independently of their role in the system. Seeders, leechers and helper peers all appear to behave in the same way, and original requests are identical to relayed requests, hence they are not distinguishable by an attacker.

Finally, our system also makes use of *private swarms* [66]. Consider a regular public swarm sharing some content. Any peer in the public swarm can also be a member of one or more private swarms where it shares the same content with a restricted group of peers that is not publicly discoverable. This removes the attacker's certainty of knowing every participant of the system, hence preventing him from identifying a peer's interest even when controlling a large fraction of the public swarm.

To summarise, the system can be divided in four different components which contribute to creating the cover traffic. First, peers create several deception downloads, as they always participate in multiple swarms at the same time, Section 7.2.2. Second, peers relay requests for content on behalf of other peers, creating more cover traffic, Section 7.2.3. Third, peers always perform truthful announces, but they announce only a subset of what they have downloaded, Section 7.2.4. Finally, the system relies on an incentive mechanism and the creation of private clusters of peers, Section 7.2.5.

7.2.2. Caching Strategy

Peers join swarms that are not of interest to users, acting as helper peers, retrieving and distributing a part of the content. We define this behavior as *caching*.

Caching content in under-supplied P2P swarms has proven to provide big benefits to the community, and is thus of value to a P2P system regardless of its privacy-enhancing effect. The caching strategy problem consists of two parts. The first concerns the content replication problem, which consists of directing the helper peers to the swarms that need resources [162]. This is outside of the scope of this work (see Section 7.2.5). The second part regards the cache replacement policy, which investigates how to maintain a cache up to date. While much research has focused on finding the optimal caching strategy [155–157], recent work [161] has shown that an optimal solution provides only marginal benefits at a much higher cost compared to the simple solution of replacing the least recently used piece (LRU).

Our caching strategy applies an LRU policy over the entire cache, containing pieces from multiple swarms, causing the announce field to increase and decrease over time. A piece that has been selected for removal, being the least recently used, will be deleted only after a certain time interval, reducing the response delay and relay traffic might it be requested by one of its neighbors. Every peer has a local view of the rarity of each piece, which depends on the pieces its neighbors are currently announcing. By applying an LRU replacement policy, we ensure that information about the local view of the pieces rarity is not revealed to the swarm. The LRU policy fits well in the context of our system, as replacing pieces that haven't been recently requested maximizes a peer's resource investment. Furthermore, since requests for pieces are performed based on their rarity, a peer adds to its cache content what is considered rare by its neighbors.

The cache of a peer has a fixed size and is filled with the rarest content among all the swarms in which the peer is participating, including the swarm which is of

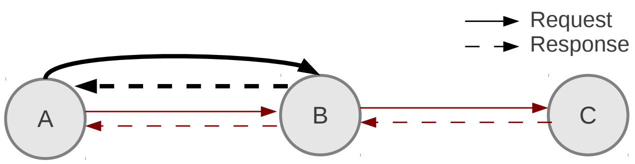


Figure 7.1: Peer B acting as a relay for peer A. If B has the requested piece, it uploads it directly (black line); otherwise, it downloads it from C, and then uploads it to A (red line).

interest to the user. When the user starts a new download, the algorithm removes the previous download and one extra swarm from the cache, replacing them with the new download, and one new swarm in which the peer joins as a helper. Our algorithm prioritizes swarms in which there are less than two complete replicas of the content publicly announced. This approach favors newly created swarms, reducing the relay traffic, and occasionally the load in the case of a flashcrowd.

7.2.3. Relay Mechanism

In our system, all peers act as relays for their neighbors. This prevents attackers from identifying a user's interest simply by observing the pattern of piece requests.

Upon receiving a request for a piece that it is not announcing, a peer reacts in one of two ways, as depicted in Figure 7.1. It either responds directly to the request if the piece is locally available, after first introducing a realistic random delay to simulate the retrieval from a third peer. Or, it acts as a relay and retrieves the piece from a neighbor that is announcing it. If the peer retrieves the piece, it may also cache it if it is rare enough, as the peer might serve it to others afterwards. When a peer completely retrieves the content of interest and starts seeding it, it will always relay a fraction of the incoming requests, even if the content is locally available. The fraction of relayed requests should be small enough to avoid performance degradation while reducing the probability of revealing the identity of seeders.

Most importantly, relaying hides the real intentions of a user from an external observer, unable to know all the peers in the system, given the possibility of private swarm connections. An external observer will never be able to decide with a high level of certainty if a peer makes a request itself or is relaying it for one of its neighbors.

7.2.4. The announce field

Peer requests for pieces are driven by what neighbors declare to store locally, also called their *announce fields*. In a P2P system, peers usually announce exactly what they have downloaded to maximize the probability of being selected by other peers when establishing new connections. In our design, instead, to provide plausible deniability, a peer only announces a subset of the pieces that it has locally available.

A trivial strategy for a relay is to combine and replicate the announce fields of its neighbors. Hence, it receives the most requests possible that it can potentially serve through relaying. While this might seem like a good solution, it involves a series of complications, such as relay loops and a misleading assumption of data

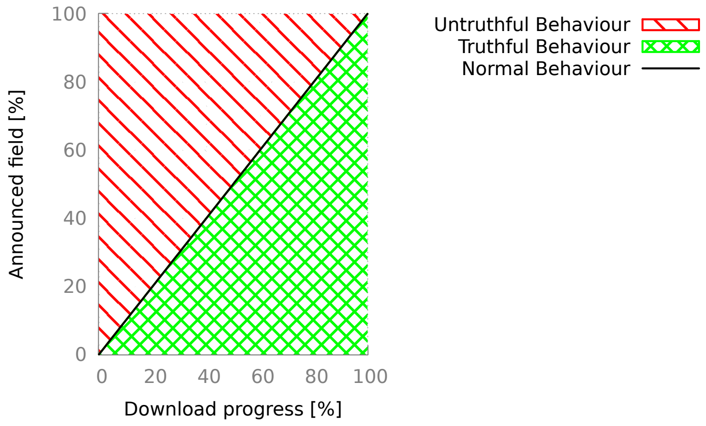


Figure 7.2: Possible announce strategies for peers, based on what they have downloaded related to what they announce.

availability in the swarm, increasing the complexity of the whole system. Relay loops are easily formed if every peer replicates its neighbors announce fields, because requests might be forwarded only to relays, never reaching the peer that actually has the piece. These problems can be easily overcome if a centralised authority controls the relay peers [2, 62, 83], but our goal is to build a completely distributed system. Some of those limitations can be removed, as described in [93], but not without affecting the overall performance of the system. Therefore, to keep the swarm healthy, it is necessary for peers to announce only what they have actually downloaded. Furthermore, by assuming a truthful announce, it becomes trivial to detect malicious peers. See Figure 7.2 for an overview of all announce strategies.

Helper peers can simply announce all the pieces they download. This solution, investigated by Nielson and Wallach [127], requires helper peers to invest considerable resources in the swarms they join. Furthermore, it forces helper peers to stay longer in the swarm to gain back their investment. Instead, peers have a great interest in downloading only a small subset of the content, especially the rarest parts, as they have a better chance to increase their sharing ratio.

If all the peers in the system announce only a subset of their download, as depicted in Figure 7.3, helper peers become undistinguishable from seeders and leechers based solely on the announce field. Hence, our design requires all peers to announce only a subset of their download.

The size of the announce field can vary over time. This behaviour is a direct consequence of the LRU cache replacement policy (see Section 7.2.2).

7.2.5. System Prerequisites

Our system requires two constructs that have been proposed in related work, but are not yet commonly found in deployed P2P systems. These are a cross-swarm incentive mechanism and a system of creating overlapping private swarms.

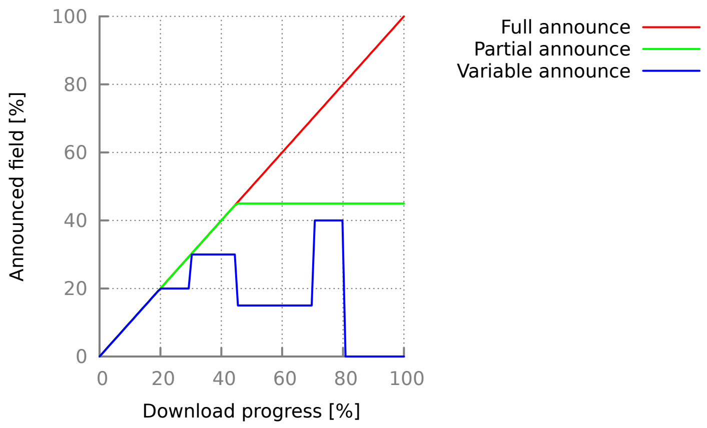


Figure 7.3: Example of announcing only a subset of the download.

Incentive Mechanism

We identify two types of incentives for users to participate in our system. First, creating cover traffic makes it easier for the traffic of each peer to remain private. So any user interested in privacy will want to cache and relay data for others simply to make it harder for an attacker to identify their own traffic. Second, we envisage a community that keeps track of peer reputation and provides differentiated services to peers based on their reputation. In such a community, peers will gain reputation for caching and relaying because these are actions that benefit the community as a whole. Caching increases the data replication as well as the amount of upload bandwidth available in the system. Relaying increases privacy. Thus, peers will engage in these activities out of user self interest, in order to gain reputation. One way the reputation could be used is to offer differentiated service to peers through bandwidth prioritization: in case of congestion, peers with higher reputation will be served first.

Private Swarms

In order to provide plausible deniability to users confronted with some of the attacks described in Section 7.3, we require a system for managing private swarms, such as Closed Swarms [66].

In order to ensure a good level of privacy, the system needs to guarantee that full swarm membership of the participants cannot be determined by any node. If an attacker in a swarm knows every other participant in the swarm, it can infer the identity (i.e., seeder, leecher, or helper) of a single participant under certain circumstances.

The attacker's knowledge can be weakened by users explicitly or implicitly creating partially connected private swarms that share the same content as the public swarm. Access credentials to join these private swarms can be distributed either by a content distributor, or by the users themselves, for example based on friendship relations in a social network [105]. This allows users themselves to select which private swarms they wish to participate in. These possibilities exist in parallel, and one provider of credentials cannot detect other providers for the same content.

7.3. Security Analysis

In this section we present a theoretical analysis of the protection against attackers that our system provides to its users. In Section 7.3.1, we describe the threat model, and provide a solution to general attacks that affect P2P systems. Then, in Section 7.3.2, we provide a detailed analysis of how our system protects its users from a collusion attack.

7.3.1. Threat Model and Common Attacks

In this section we present the threat model we address, and how the system protects users from common attacks against P2P systems. The attacker can impersonate one or multiple peers in the system, but is not able to control the network connection of the peer under attack, i.e., there is no protection against the ISP. We assume the attacker is limited in resources, and therefore cannot impersonate all the peers in the system. Furthermore, evidence must be gathered in *plain view* [154]. We consider an attack to be successful if the forensic investigator can gather enough evidence to obtain a search warrant based on *probable cause* with a very small margin of error. We rely on the analysis presented by Prusty et al. [136] when considering the precision needed for a successful attack, and we consider 80% to be a reasonable value for probable cause, and 60% to be a conservative value for weaker evidence of probable cause.

The majority of attacks that usually threaten P2P systems is solved by the P2P protocol on which our solution is based. Our system is build on top of the PPSPP protocol, which provides defences to common attacks such as forgery, pollution, neighbour selection, denial of service, and omission. An attack that might affect our system is the Sybil attack [86], but it would only compromise the incentive mechanism. Therefore, the incentive mechanism should be designed with this in mind. Systems which use relays are often the target of attacks trying to identify if a target peer is the real source of the content. Attackers simultaneously send the same relay request to the target peer. They can conclude, by analysing the response time, whether the target is the originator of the request or just a relay [136]. Our system is immune to such timing attacks, as peers always cache relayed pieces for a random amount of time, and also add random delays to all responses.

7.3.2. Collusion Attack

In a collusion attack, the attacker impersonates multiple peers in the system, connecting to a subset of peers to detect their real interests. In [136], Prusty et al. present a detailed security analysis of OneSwarm [105], identifying the collusion attack as one of the most successful. Given the similarities of our system to OneSwarm, we follow the same analysis in order to compare the security of both systems when threatened by a collusion attack. The attack works as follows. One attacker peer requests from the target peer content that it is not announcing to the swarm, either

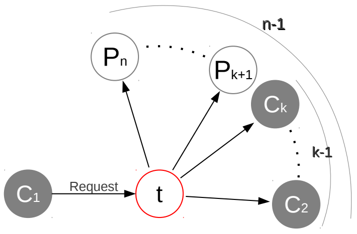


Figure 7.4: A collusion attack where attacker C_1 requests content that is not announced by t, and attackers $C_2, \ldots, C_k$ wait for the request to be forwarded to them.

triggering a relay request, or revealing the target peer has the content locally available, and hence the content is of interest to the user. By controlling as many of the target peer's connections as possible and observing the relay requests, the attacker can determine if the peer is likely to have the content. A graphical representation of the collusion attack is given in Figure 7.4, where k attackers are connected to the target peer T. The higher the number of connections established with a target peer, the more likely it is for the attack to succeed.

The attacker cannot detect all the relay requests because the target peer could redirect the request to any of the $n - k$ non-attacker neighbours. Therefore, for the attack to succeed, the attacker must compute the probability of peer t having the content while none of the attackers was forwarded the request. In our analysis, we consider three values corresponding to the weak probable cause standard (WPC), 60%, probable cause standard (PC), 80%, and the definite cause (DC), 95%, as the OneSwarm standard in [136].

We denote the event "target peer T has the requested piece" by X, and the event "none of the colluding attackers were forwarded the request" by Y. The probability to be computed is given by:

$$P(X|Y) = \frac{P(Y|X)P(X)}{P(Y|X)P(X) + P(Y|\bar{X})P(\bar{X})} \tag{7.1}$$

The probability of the target having the request-ed piece

To identify $P(X)$ we only consider the worst case scenario, where attackers join and monitor multiple swarms over a long period of time. In this scenario, the attacker might identify when a peer, which has been observed in the past, re-joins the system. In this case $P(X) \leq 0.5$, as peers always replace at least two swarms from their cache when downloading new content (see Section 7.2.2). The probability $P(X)$ is 0.5 only when the peer successfully retrieves the entire content and keeps seeding it in the swarm, which is unlikely. On the other hand, $P(Y|X)$ is determined by the ratio, r, of requests which are relayed while seeding (see Section 7.2.3), thus $P(Y|X) = 1-r$.

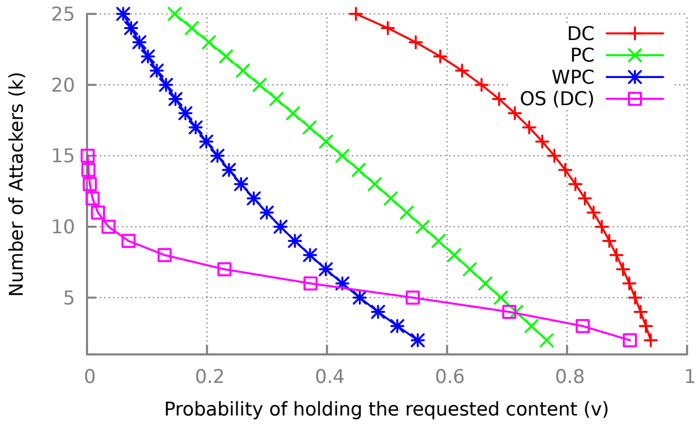


Figure 7.5: The number of attackers connected to the target peer needed for a successful attack depending on the probability of the target peer holding the requested content. (WPC=60%, PC=80%, and DC=95% certainty). Derived from Equation 7.4 ($n = 25, s = 5, r = \frac{1}{10}$, 5 possible private connections). Also the results of OneSwarm are showed.

Number of colluding attackers

We investigate the effect of the number of connected attackers on the effectiveness of the attack. The probability of forwarding the request to one of the $n - k$ honest neighbours, $P(Y|\bar{X})$, is given by:

$$P(Y|\bar{X}) = \frac{P(X_h)(n-k)}{P(X_a)k + P(X_h)(n-k)}, \tag{7.2}$$

where $P(X_a)$ represents the probability that an attacker is announcing the requested piece, while $P(X_h)$ represents the probability that an honest, non-attacker neighbour, is announcing it. Because the colluding attackers are always announcing the requested piece to maximize the effectiveness of the attack, $P(X_a) = 1$. To derive $P(X_h)$ we consider the system in a steady state and peers uniformly distributed over the swarms. Therefore $P(X_h) = \frac{1}{s}$, as each peer divides its cache resources equally among the s swarms, hence it announces $\frac{1}{s}$ of the content. By replacing and simplifying we obtain:

$$P(Y|\bar{X}) = \frac{n-k}{sk+n-k}. \tag{7.3}$$

To match the analysis in [136], we set $P(X) = v$ and $P(X|Y) = \phi$. Hence Equation 7.1 becomes:

$$\phi = \frac{v(1-r)}{v(1-r) + ((n-k)(sk+n-k)^{-1})(1-v)} \tag{7.4}$$

Figure 7.5 shows the minimum number of attackers connected to the target needed to identify with different degrees of certainty (corresponding the scenarios WPC, PC, and DC) if a user is interested in the content, depending on the probability of the user having the content. In the worst case scenario, when $v = 0.5$, to satisfy the probable cause standard (80% certainty), at least 13 colluding attackers need to be connected to the target. It is almost impossible to obtain the 95% certainty, as the attacker needs to control all the target's public connections.

Percentage of attackers in the system
The probability for a target peer to be connected to enough attackers for the attack to succeed, denoted k, can be modelled by the hypergeometric distribution, since neighbour peers are chosen at random without replacement (as in [136]). We define N as the total number of peers in the swarm, C of which are attackers. The number of untrusted connections a peer has is U, and A is the number of attackers connected to the target. We have:

$$P(A \geq k) = \sum_{i=k}^{U} \frac{\binom{C}{i}\binom{N-C}{U-i}}{\binom{N}{U}}. \tag{7.5}$$

Figure 7.6 shows the probability of a successful attack depending on the percentage of attacker peers in the system.

Comparison with OneSwarm
As reported in [136], performing a collusion attack on One-Swarm leads to greater results than in our system. Figure 7.5, OS (DC), shows how far fewer attackers are required to determine with a 95% certainty if a peer is the origin of the content. There are two differences in the analysis that are relevant in comparing the two systems. The fist relies in the meaning assigned to v. In [136], v represents the popularity of a content, giving a trivial way for attackers to identify peers in popular swarms. In our system the popularity of a file does not influence the success of the attack, as more popular swarms naturally contain more helper peers, given the higher demand for resources. The second difference is in the design of relay requests. In OneSwarm each peer forwards its requests to a subset of its neighbours, based on a probability p which is set to 0.5. The consequence of using a fixed probability becomes more obvious when considering the *false positive rate*. The false positive rate (FPR) represents the probability of a target peer requesting content from a trustful peer instead of an attacker. The FPR is given by $P(Y|\bar{X})$. In OneSwarm $P(Y|\bar{X}) = (1-p)^{k-1}$, hence the FPR is less than 1.6% only with $k >= 7$ [136], making the FPR quite irrelevant to the attacker. On the other hand, our system provides a greater protection to its users, as the FPR greatly increases when the number of connected attackers decreases. For the number of attackers needed for PC, 13, the FPR is 20.7%, while for WPC, 4 attackers, it is 56.5%. Only within the DC scenario the FPR decreases to an acceptable level, but it is irrelevant, as it implies that the attacker controls all the peers in the system, see Figure 7.6 with $k >= 25$.

7.4. Experiments

In this section we present the set up and the results of our experiments. We implement our solution in a real application, and evaluate the performance by comparing it with other systems.

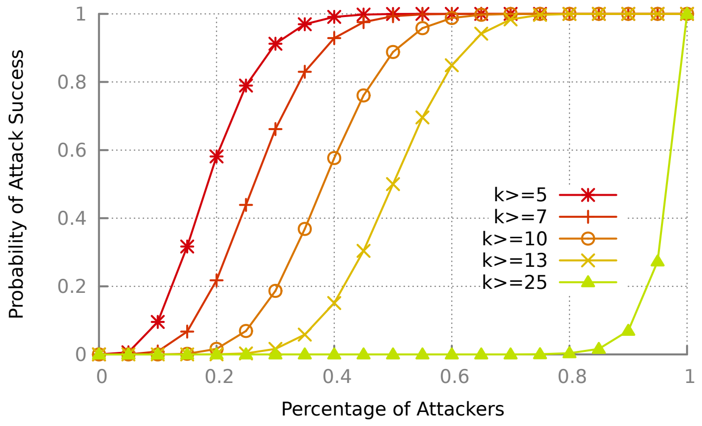


Figure 7.6: Probability of a target peer being connected to enough attackers for a successful collusion attack depending on the percentage of attackers in the swarm. Derived from Equation 7.5 ($u = 25$).

7.4.1. Setup

We have implemented our system in Libswift, the reference implementation of the P2P Straming Peer Protocol (PPSPP) [60]. We have performed experiments by means of emulations, running our application on real hardware. Our experiments have been performed on the Distributed ASCI Supercomputer, DAS4 [61], composed of 32 nodes, each with 16 cores. We analyse the behaviour retrieving 10 files, all of the same size of 100 MB, which are distributed over 10 different swarms. For each of the 10 swarms we define a different *popularity*, which we model using the Zipf distribution with a 0.66 exponent, as proposed by Zhang et al. [164].

Peers are distributed among the nodes of the cluster, 16 leechers on each node, and have an upload capacity of 5.56 Mbit/s and a download capacity of 13.69 Mbit/s, conforming to the analysis by Ookla of April 2013 [19]. The 10 files are distributed by 10 initial seeders with an upload capacity of 5.56 Mbit/s, to stimulate cooperation between peers. We run experiments with different number of peers in the system, and the results we present are for 512 peers. Peers join the system following a Poisson distribution with an arrival rate of 1 peer per second, and leave the system after successfully retrieving the content of interest, and reaching a sharing ratio of 1. Once a peer leaves the system, f downloaded files are deleted from the local storage. The file which represents the content of interest is always deleted, and the remaining $f-1$ are random files from the cache, as presented in Section 7.2.2. Thereafter it re-joins the system selecting f new swarms following the Zipf distribution, one being the content of interest and the remaining to fulfil the caching requirements. We execute each scenario until each peer completely retrieves at least 4 files of interest, defining each successful retrieval as an *iteration*. To simulate the incentive mechanism, we simply follow the Zipf probability distribution. By replacing each peer with a new one, we create a *steady state*, as a constant number of peers is always present in the system.

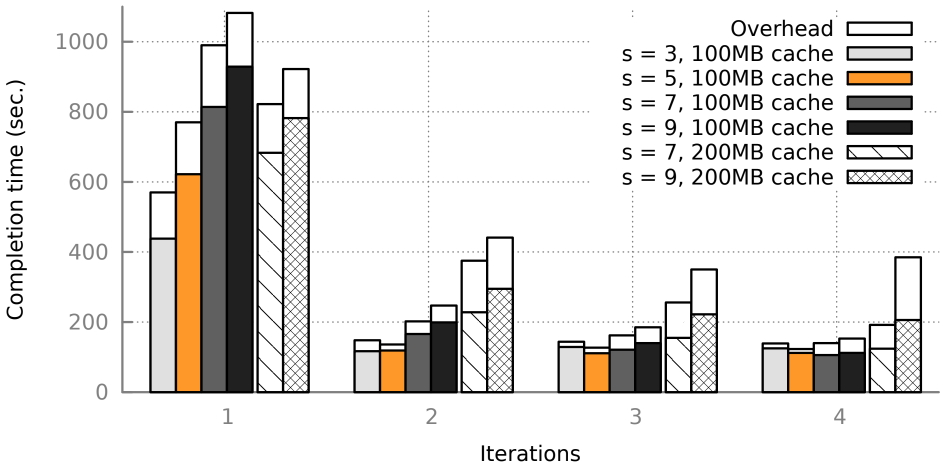


Figure 7.7: Average completion time depending on the size of the cache, and the number of cached files (s). ($f = 2$)

We investigate several scenarios that affect the privacy and performance of the peers. The performance is evaluated based on the *completion time*, defined as the time needed to retrieve the content of interest, being the most valuable metric to the user. The privacy of the system is evaluated based on tree different parameters, the size of the cache, the number of swarms in which a peer is helping (s), and the number of swarms which are replaced in the cache when re-joining the system (f). Furthermore we compare our solution with the standard PPSP protocol, and the solution presented in [127].

7.4.2. Results

To evaluate our system we analyse the average time needed to retrieve the content of interest using different parameters. We also show the overhead, defined as the additional time needed to reach a sharing ratio of 1, or regaining the resource investment. Given the peers' asymmetric network links, they will require more time to reach a sharing ratio of 1. A common characteristic of all the results we show is the higher completion time during the first iteration, as peers simultaneously compete for the same resources. From the second iteration the completion time drastically decreases, as peers join over-supplied swarms, where the majority of peers are waiting to reach the designated sharing ratio.

Figure 7.7 shows the effects of varying the number of swarms s and the size of the cache across different iterations. Increasing the number of swarms in which a peer is helping decreases the performance and increases the overhead, without always providing an increased privacy for the user. Considering the scenario in which the cache is the same size as the content, equal to 100 MB, increasing the number of swarms better protects the user from an attacker observing a snapshot of the system. However that decrease his privacy if the attacker monitors the frequency of his

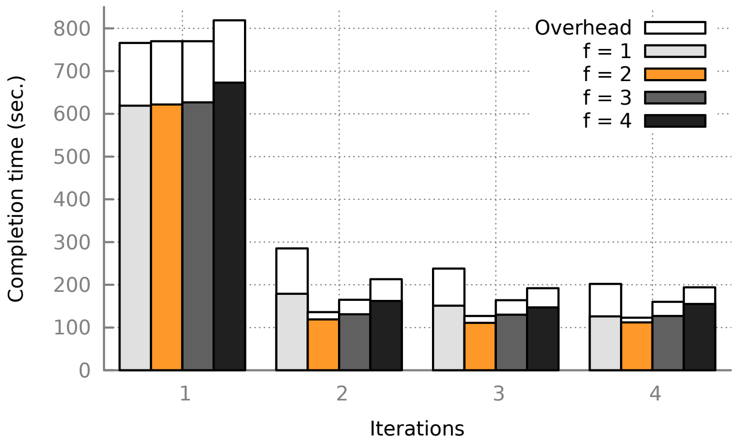


Figure 7.8: Completion time based on the number of files deleted from the cache (f). ($s = 5$, cache size $=$ 100 MB.)

requests, simplifying the detection of his content of interest. From our experiments, see Figure 7.7, when the cache is as big as the content, participating in 5 swarms ($s = 5$) provides the best trade-off between privacy and performance. This is due to two main reasons. First, participating in 5 swarms leads to less overhead over time. Second, the security analysis presented in the previous sections is based on a peer participating in 5 swarms, hence proving that it provides a good level of privacy. If a user participates in fewer swarms, his privacy is reduced proportionally, as it becomes easier to determine his real content of interest. The aim of the experiment presented in Figure 7.7 is to show how participating in less swarms does not lead to a better overall performance after the first iteration. In order to gain a higher privacy by increasing s, peers need to proportionally increase the size of the cache, e.g. double the cache size when $s = 9$. While increasing s and the cache size seems like a good way of increasing privacy, the performance loss are more than proportional, see Figure 7.7 comparing the completion time for $s = 5$ and 100MB cache, and $s = 9$ and 200MB cache. Moreover increasing the size of the cache might be prohibitive when the content of interest is several GB in size, rather than 100 MB as in our experiments.

Figure 7.8 presents the effect of varying the number of files (f) deleted before re-joining the system on the completion time over different iterations. The content of interest is always deleted, hence $f \geq 1$, and the greater f the more privacy is gained in case an attacker monitors several swarms and is able to identify when a peer re-joins the system. Peers should always delete at leasts 1 file from their cache, hence $f \geq 2$, but it is interesting to notice how keeping the cached files also decreases the system performance. Our experiments show that replacing more files from the cache has limited impact on the performance, while providing a greater privacy to the users. On the other hand, we consider $f = 2$ to be a good parameter in order to meet our goal, providing plausible deniability to the users. Even with

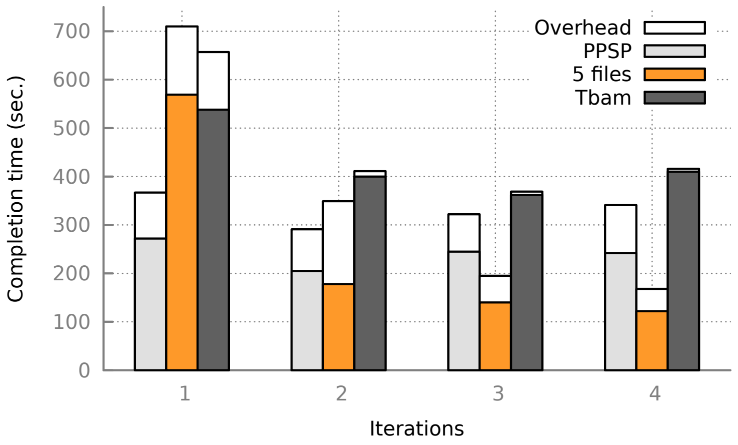


Figure 7.9: Completion time in comparison to the standard PPSP protocol, and the solution presented in The BitTorrent Anonimity Marketplace (tbam) [127]. ($s = 5$, $f = 2$, cache size = 100 MB.)

this low value an attacker is not be able to identify with a high degree of certainty the real interest of a peer, as presented in Section 7.3.2 where $f = 2$ has been used for the analysis. Figure 7.8 shows how users wanting more privacy can increase f without suffering from a noticeable performance loss.

Finally in Figure 7.9 we present a comparison of our system with the standard PPSP protocol [60], representing a traditional P2P system, and the solution present in The BitTorrent Anonimity Marketplace (tbam) [127], as it closely relates to our system. In [127], peers always retrieve 4 additional files in order to create cover traffic. This trivial solution has a big performance drawback when compared to the other systems. The PPSP protocol, while being the most efficient solution for single file retrieval in over-supplied swarms, suffers in under-supplied environments where peers struggle to achieve the desired sharing ratio of 1. Our solution on the other hand provides the best performance for its users, while providing the same privacy as [127], and a much improved privacy solution over a standard P2P system [60]. We do not compare against other systems providing anonymity [62, 83, 93, 105], as adding multiple encryption layers over the PPSP protocol will definitely decrease its performance.

7.5. Conclusion

In this chapter, we propose a solution for hiding the interest of users in content that does not decrease the performance of P2P file sharing systems. Our solution is incentive compatible, as both peers and the system as a whole benefit from caching and relaying. The usage of private swarms is a feature that increases the privacy for every peer, but it is not a requirement for every peer to participate in a private swarm. The potential existence of private swarms is enough for plausible deniability.

Attackers targeting a peer in a coordinated attack cannot reveal its role in the system. Leechers, seeders and helpers are indistinguishable, as are relayed and direct content requests.

Finally, our experimental results show that, after a warm-up period, the steady state performance of our system is close to that of the reference PPSPP implementation it is based on. We show how our caching strategy has little impact on the swarm's performance, between 5% and 10% overhead, while providing a higher level of privacy to all system participants.

In future work, we would like to deploy our solution in a P2P file sharing client and offer users a selectable privacy level, from completely open to completely closed and all the shades in between. Furthermore, we want to extend the caching and relaying algorithms with specialized versions for video-on-demand and live video streaming.

8 Conclusion

In this thesis we have presented the design and analysis of several solutions for improving Peer-to-Peer (P2P) streaming systems. We focused on the challenges that involve the dissemination of time-critical streaming content over fully distributed P2P systems. The work presented in this thesis can be divided into three areas, upon which we draw our conclusions. First, we have designed a system for distributing adaptive streaming content over P2P networks, in Chapter 2, and presented an enhancement of its download algorithm in Chapter 3. Secondly, we have introduced the novel P2P protocol specifically designed for distributing streaming content in Chapter 4, presented its reference implementation in Chapter 5, and analysed its performance in Chapter 6. Lastly, we have addressed the privacy concerns of the users of P2P systems, presenting a solution that increases their privacy with little impact on the performance of the system as a whole in Chapter 7. Our solutions for each of those three research areas can be used independently, or can coexist within the same P2P system. They are all open-source and have been implemented on top of Libswift, the reference implementation of the Peer-to-Peer Streaming Peer Protocol standard (PPSPP), described in RFC7574 [60]. In this chapter, we present our conclusions, some reflections on the value of the work presented in this thesis, and suggestions for future work.

8.1. Conclusions

Based on the work presented in this thesis we can draw the following conclusions with respect to the Research Questions stated in Section 1.3.

1. [**RQ1-RQ2**] It is possible to provide a high Quality of Experience with a fully distributed P2P streaming system. The best Quality of Experience in a video streaming scenario is achieved by combining a framework for encoding and decoding the original content with an algorithm that smartly selects the best video quality based on the available network bandwidth and display capabilities. In Chapter 2 we presented the first open-source system for providing

scalable video content over a fully distributed P2P system. We described all the modules of the producer- and consumer-side of the architecture, including the encoding and decoding of the video stream. We evaluated the performance of our solution in a series of test scenarios. Our framework can be easily integrated in existing P2P systems, and supports third-party media consumption solutions. In Chapter 3 we presented Deftpack, an improved self-adapting download algorithm specifically crafted to retrieve scalable video content. We showed how Deftpack enhances the Quality of Experience of the users by reducing the start-up time and the number of stalls during the video playback, outperforming available alternatives.

2. [**RQ3-RQ4**] Designing a new protocol in order to specifically address the video streaming scenario leads to a better result than trying to modify an existing one. In recent years, much effort has been invested in trying to adapt existing P2P protocols to the streaming scenario leading to acceptable, but never completely satisfactory results. We have shown that in order to overcome the limitations of existing approaches, a new protocol specifically designed with streaming in mind leads to better results. In Chapter 4 we presented the Peer-to-Peer Streaming Peer Protocol (PPSPP), the Internet standard for P2P streaming systems. Despite the popularity of other P2P protocols such as BitTorrent, we showed that they do not provide the best solution for distributing time-critical content such as video streams. In Chapter 5 we presented Libswift, the reference implementation of PPSPP. We described some of its design choices and investigated its behaviour in network challenging environments. Finally, in Chapter 6, we described Libswift's download algorithm for retrieving streaming content that provides a high Quality of Experience by exploiting the information available at the transport layer. We compared our protocol with the most popular BitTorrent-based P2P clients and validated the effectiveness of our solution. Furthermore, we evaluated its performance when run on hardware-limited devices such as cell phones.

3. [**RQ5**] Users of P2P systems do not need to trade performance for a higher level of privacy. Traditional P2P systems that offer some level of privacy are characterised by an inherent performance drawback. In Chapter 7 we showed how users can gain privacy by collaborating in the retrieval of content, without affecting the performance of the entire system. Our solution uses caching and relaying in order to hide the user's content interest. We analysed this solution and showed that it has little impact on the swarm's performance, increasing the availability of content after an initial warm-up period.

8.2. Reflection on the State of P2P Streaming

In this section we reflect on the value of the work presented in this thesis, in the light of the recent decline of interest in P2P research. Since the pioneering days of Napster, over 17 years have passed [38]. The early gold rush is now over, several official Internet standards have been published, thousands of open-source P2P projects have failed to gain traction, de-facto standards have become pervasive, and conferences have both emerged and declined. This is an excellent moment for reflection and for a discussion of likely developments over the coming years.

Academic interest on P2P has seen a rapid decline in interest, and important venues such as the *IEEE P2P Conference* and the *International Workshop on P2P Systems* have been discontinued. While deployed P2P systems like BitTorrent [5] keep on being very active, and their network utilisation still represents 15-20% of the current consumer Internet traffic [48], they have seen little change in their implementation over the last decade. The personal opinion of the author is that this is partially due to the fact that many areas of P2P research never saw their application in those mainstream systems. However, the same does not necessarily hold true for P2P video streaming. This field is still very active, and some of the results of the research on P2P video streaming is being adopted in various existing systems [27, 39]. One of the underling reasons is that video streaming distribution systems are mostly designed and built by content providers for their own specific streaming service, which are mostly closed source and do not require the critical public adoption. It might be argued that many content providers have moved away from P2P-assisted networks to alternatives such as Content Delivery Networks (CDNs), which is indeed a reality for some [33]. This choice is driven by rapidly decreasing infrastructure costs, and by the fact that the use of P2P networks is often does not justified because of the increase in complexity and operation costs, with the added inherent risk for the users' Quality of Experience (QoE). It should be noted, however, that, as presented in Section 1.1.1, the quality and size of the video content is increasing at the same rate as the infrastructure capabilities, as well as taking full advantage of the more powerful hardware. As an example, it is possible today to stream ultra-HD video while over mobile network [49] and view it on a 5-inch mobile phone [1].

In Chapters 2 and 3 of this thesis, we have presented a system to distribute adaptive streaming content and a download algorithm designed to efficiently retrieve such content. We are not aware of any content provider that has adopted the system as presented in this thesis. Nevertheless, almost every content provider, from Google to NetFlix, is nowadays providing their content in multiple qualities, mostly using adaptive streaming technologies such as DASH [106]. The variety of adaptive streaming solutions and the importance of the companies that support them, show how the interest in maintaining a dynamic service and in trying to provide the best possible QoE in a very competitive environment is driving full adoption by content providers.

The following three chapters, Chapters 4-6, focus on the design, the reference implementation, and the evaluation of the novel P2P Streaming Peer Protocol, which was recently accepted as standard RFC7475 at the IETF [60]. The need for a standardised approach to P2P streaming is obvious because video streaming is the major area where P2P technology is still evolving. Clear indications can be found in projects such as BitTorrent Live Streaming [4], where the first showcase of the technology is currently being made available for the mobile environment, and the rapid diffusion of platforms such as Popcorn Time [27], a video streaming service based on BitTorrent with millions of users [11]. Furthermore, reference implementations of the P2P Streaming Protocol are actively being developed [14, 36], and several companies [31, 34] are using the protocol to enhance their streaming distribution coverage.

Regarding Chapter 7, we are not aware of any system that is currently using the privacy solution presented in this thesis. Nevertheless, P2P systems that focus on privacy and anonymity are still very actively being developed, and P2P users are constantly seeking for the best solution to satisfy their privacy needs. Examples of the application of such systems range from overcoming censorship and avoiding surveillance to anonymous banking systems [3]. The privacy solution presented in this thesis might never be entirely deployed, but some of the ideas might influence the design of future privacy-aware P2P clients.

There is evidence for the re-emergence of the P2P paradigm. A new generation of developers [9, 12] is working on generic distributed approaches for building decentralised applications with unbounded scalability. However, the billion plus users of Facebook and Youtube are unlikely to leave cloud technology behind in favour of P2P systems. For embedded devices and self-driving cars, central control is not a very appealing option. Cars can simply never rely on a reliable Internet connection. During snow storms, clouds become inaccessible and the only viable communication reaches the closes car on the street. Therefore, self-aware fully distributed solutions will need to be developed in order for such technology to succeed at a global scale. My prediction is that this will lead to a re-discovery of P2P technology, perhaps under a new name.

8.3. Suggestions for Future Work

In this thesis we have focused on improving the distribution of video streaming content over fully distributed P2P systems. As discussed in the previous section, the key for the success and continuation of the research on P2P video streaming lays in the continued interest and popularity of video streaming. Future research should provide practical solutions that can easily be integrated with existing P2P clients, such as Tribler [39], While our work is a step forward in providing better solutions, it is definitely not exhaustive and there is much room for further improvement. Below we highlight possible directions for future work.

1. In Chapter 2 we presented a framework for distributing scalable video content, with the intention of improving the users Quality of Experience. The Quality of Experience can be further improved by changing the way scalable video coding is integrated. When designing our solution we put a great effort in providing backward compatibility with existing BitTorrent-based clients. While this allows an easy integration into existing systems, it provides several drawbacks in terms of efficiency. For instance, we required the content to be divided using a fixed piece size, which limits the scalability layers of the video stream. Our current solution only supports the scalability layers defined by the Scalable Video Coding (SVC) standard, e.g. scalability in terms of resolution and fidelity. A desirable further step could be the adoption of temporal

scalability and quality scalability, also known as medium-grain scalability. Unfortunately, the adoption of such scalability layer would require some major changes to our system, and to the underlying BitTorrent protocol as it would require the pieces to be of an arbitrary size. If backwards compatibility is not a strict requirement, it would be interesting to investigate the advantages of using a dynamic piece size, exploiting all the features of scalable video coding.

2. In Chapters 4, 5, and 6, we described and analysed the Peer-to-Peer Streaming Protocol. In order to evaluate our solution we performed extensive emulation of its reference implementation on real hardware, simulating real-world scenarios. While this is a good method for estimating the protocol's capabilities, a real-world deployment is required in order to understand the real benefit of our solution. Several companies, e.g., Swarmplanet [34], are currently adopting our protocol and reference implementation for distributing live content to millions of users. It would be beneficial to analyse real-world traces from such companies to better understand its advantages and drawbacks, and how it can be further improved to provide the best P2P streaming experience.

3. In Chapter 7 we presented a privacy solution for P2P systems. It would be interesting to deploy our solution in P2P file sharing clients, offering users several levels of privacy. Studies could be performed to analyse the users preferred level of privacy, and the system could be adjusted accordingly. Furthermore, our solution only supports file-sharing applications, and should be extended to support streaming applications.

Bibliography

[1] 4k smartphones. http://4k.com/phone.

[2] Anonymizer. http://www.anonymizer.com.

[3] Anonymous p2p. http://en.wikipedia.org/wiki/Anonymous_P2P.

[4] BitTorrent Live. http://live.bittorrent.com.

[5] BitTorrent Protocol 1.0. http://www.bittorrent.org.

[6] CNN. http://www.cnn.com.

[7] Deluge Torrent. http://deluge-torrent.org.

[8] Dynamic streaming in flash media server. http://www.adobe.com/devnet/adobe-media-server/articles/dynstream_advanced_pt1.html.

[9] Ethereum. http://www.ethereum.org.

[10] The history of the BBC: The first TV era. http://www.teletronic.co.uk/tvera.htm.

[11] Inside popcorn time. http://www.wired.com/2015/03/inside-popcorn-time-piracy-party-hollywood-cant-stop.

[12] Ipfs. http://ipfs.io.

[13] Libswift. http://libswift.org.

[14] Libswift: official reference implementation of rfc7574. https://github.com/libswift.

[15] Libtorrent. http://www.libtorrent.org.

[16] Limewire. http://www.limewire.com.

[17] Microsoft Smooth Streaming. http://www.microsoft.com/silverlight/smoothstreaming.

[18] Miro. http://www.getmiro.com.

[19] Net index. http://netindex.com.

[20] Netflix. http://netflix.com.

[21] Nipkow disk. http://en.wikipedia.org/wiki/Paul_Gottlieb_Nipkow.

[22] Octoshape. http://www.octoshape.com.

[23] Opentracker. http://erdgeist.org/arts/software/opentracker.

[24] The P2P-Next Living Lab. http://livinglab.eu.

[25] The P2P-Next project. http://www.p2p-next.org.

[26] Peer to Peer Streaming Protocol working group. http://datatracker.ietf.org/wg/ppsp.

[27] Popcorn time. http://www.popcorn-time.to.

[28] PPLive. http://www.pplive.com.

[29] PPSTream. http://www.ppstream.com.

[30] QQLive. http://v.qq.com.

[31] SkunkWerks. http://skunkwerks.at.

[32] SOPCast. http://www.sopcast.org.

[33] Spotify abandoning p2p. http://torrentfreak.com/spotify-starts-shutting-down-its-massive-p2p-network-140416.

[34] SwarmPlanet. http://swarmplanet.com.

[35] The Swarmplayer. http://swarmplayer.p2p-next.org.

[36] Swirl: reference implementation of rfc7574. https://github.com/skunkwerks/swirl.

[37] The Internet Engineering Task Force (IETF). https://www.ietf.org.

[38] Timeline of file sharing. http://en.wikipedia.org/wiki/Timeline_of_file_sharing.

[39] Tribler. http://www.tribler.org.

[40] uTorrent BitTorrent client. http://www.utorrent.com.

[41] uTorrent transport protocol. http://www.bittorrent.org/beps/bep_0029.html.

[42] VLC. http://www.videolan.org.

[43] Vodo. http://www.vodo.net.

[44] Wikipedia foundation. http://www.wikipedia.org.

[45] Youtube. http://youtube.com.

[46] *JSVM 9.15 Software Manual*, 2009.

[47] Cisco visual networking index: Forecast and methodology, 2013-2018. White paper, June 2014. http://www.cisco.com.

[48] Cisco visual networking index: Forecast and methodology, 2014-2019. White paper, May 2015. http://www.cisco.com.

[49] Cisco visual networking index: Global mobile data traffic forecast update, 2015-2020. White paper, Feb. 2016. http://www.cisco.com.

[50] I. 13818-1. *Generic coding of moving pictures and associated audio information: Systems*, 2000.

[51] I. 14496-1. *MPEG-4 Part 14: MP4 File Format Version 2*, 2003.

[52] U. Abbasi, G. Simo, and T. Ahmed. Differentiated chunk scheduling for P2P video-on-demand system. In *Consumer Communications and Networking Conference (CCNC)*, 2011.

[53] R. Alimi, Y. Yang, and R. Penno. Alto protocol. *Resource*, 501:35, 2014.

[54] M. Allman, V. Paxson, and E. Blanton. TCP congestion control, RFC5681, 2009.

[55] S. Alstrup and T. Rauhe. Introducing Octoshape-a new technology for largescale streaming over the Internet. *IETE Technical Review*, 303, 2005.

[56] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. Resource records for the DNS security extensions, RFC4034, 2005.

[57] P. Baccichet, T. Schierl, T. Wiegand, and B. Girod. Low-delay peer-to-peer streaming using scalable video coding. In *Packet Video*, 2007.

[58] A. Bakker. Merkle hash torrent extension. bittorrent enhancement proposal 30, mar 2009. . http://bittorrent.org/beps/bep_0030.html.

[59] A. Bakker, R. Petrocco, M. Dale, J. Gerber, V. Grishchenko, D. Rabaioli, and J. Pouwelse. Online video using BitTorrent and HTML5 applied to Wikipedia. In *10th International Conference on Peer-to-Peer Computing (P2P)*, 2010.

[60] A. Bakker, R. Petrocco, and V. Grishchenko. Peer-to-Peer Streaming Peer Protocol (PPSPP), RFC7574, 2015.

[61] H. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, T. Kielmann, J. Maassen, R. Van Nieuwpoort, J. Romein, L. Renambot, T. Rühl, et al. The distributed ASCI supercomputer project. *ACM SIGOPS Operating Systems Review*, 34(4):76–96, 2000.

[62] K. Bauer, D. McCoy, D. Grunwald, and D. Sicker. BitBlender: Light-weight anonymity for BitTorrent. In *Workshop on Applications of private and anonymous communications*, 2008.

[63] A. Berson. *Client-server architecture.* Number IEEE-802. McGraw-Hill, 1992.

[64] A. Bharambe, C. Herley, and V. Padmanabhan. Analyzing and improving a BitTorrent networks performance mechanisms. In *International Conference on Computer Communications (INFOCOM)*, 2006.

[65] T. Bonald, L. Massoulié, F. Mathieu, D. Perino, and A. Twigg. Epidemic live streaming: optimal performance trade-offs. In *ACM SIGMETRICS Performance Evaluation Review*, number 1, 2008.

[66] N. Borch, K. Michell, I. Arntzen, and D. Gabrijelcic. Access control to BitTorrent swarms using closed swarms. In *ACM workshop on Advanced Video Streaming Techniques for Peer-to-Peer Networks and Social Networking*, 2010.

[67] Y. Borghol, S. Ardon, N. Carlsson, and A. Mahanti. Toward efficient on-demand streaming with bittorrent. In *NETWORKING*, 2010.

[68] J. Burbank, D. Mills, and W. Kasch. Network Time Protocol version 4: protocol and algorithms specification, RFC5905, 2010.

[69] N. Capovilla, M. Eberhard, S. Mignanti, R. Petrocco, and J. Vehkapera. An architecture for distributing scalable content over peer-to-peer networks. In *2nd International Conferences on Advances in Multimedia (MMEDIA)*, 2010.

[70] G. Carofiglio, L. Muscariello, D. Rossi, and S. Valenti. The quest for LEDBAT fairness. *Global Telecommunications Conference (GLOBECOM)*, pages 1–6, 2010.

[71] D. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–90, 1981.

[72] N. Chiluka. *Securing Social Media: A Network Structure Approach.* PhD thesis, TUDelft, The Netherlands, June 2013.

[73] D. Choffnes, J. Duch, D. Malmgren, R. Guierma, F. Bustamante, and L. Amaral. Swarmscreen: Privacy through plausible deniability in P2P systems. Technical report, Northwestern EECS, 2009.

[74] Y.-h. Chu, S. G. Rao, and H. Zhang. A case for end system multicast (keynote address). In *Performance Evaluation Review of ACM SIGMETRICS*, number 1, 2000.

[75] B. Cohen. Incentives build robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer systems*, volume 6, 2003.

[76] B. Cohen. Peer-to-peer live streaming, Mar. 14 2013. US Patent App. 13/603,395.

[77] L. Cohen. Electrical signaling, Sept. 6 1927. US Patent 1,641,608.

[78] L. D'Acunto. *Peer-to-peer Video-on-Demand Systems*. PhD thesis, TUDelft, The Netherlands, June 2012.

[79] L. D'Acunto, N. Andrade, J. Pouwelse, and H. Sips. Peer selection strategies for improved QoS in heterogeneous BitTorrent-like VoD systems. In *International Symposium on Multimedia (ISM)*, 2010.

[80] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *Computer Communication Review of ACM SIGCOMM*, number 4, 1989.

[81] P. Dhungel, X. Hei, K. Ross, and N. Saxena. Pollution in P2P live video streaming. *International Journal of Computer Networks and Communications*, 1:99–110, 2009.

[82] Y. Ding, J. Liu, D. Wang, and H. Jiang. Peer-to-peer video-on-demand with scalable video coding. *Computer Communications*, 33(14):1589–1597, 2010.

[83] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. Technical report, DTIC Document, 2004.

[84] M. Dischinger, A. Haeberlen, K. Gummadi, and S. Saroiu. Characterizing residential broadband networks. In *Internet Measurement Comference*, 2007.

[85] T. Do, K. Hua, and M. Tantaoui. P2VoD: Providing fault tolerant video-on-demand streaming in peer-to-peer environment. In *International Conference on Communications*, 2004.

[86] J. R. Douceur. The sybil attack. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, 2002.

[87] D. Eastlake et al. Domain name system security extensions, IANA. 1999.

[88] M. Eberhard, A. Kumar, S. Mignanti, R. Petrocco, and M. Uitto. A framework for distributing scalable content over peer-to-peer networks. *International Journal On Advances in Internet Technology*, 4(1 and 2):1–13, 2011.

[89] M. Eberhard, A. Palo, A. Kumar, R. Petrocco, L. Mapelli, and M. Uitto. NextSharePC: an open-source BitTorrent-based P2P client supporting SVC. In *3rd Multimedia Systems Conference*, 2012.

[90] M. Eberhard, T. Szkaliczki, H. Hellwagner, L. Szobonya, and C. Timmerer. Knapsack problem-based piece-picking algorithms for layered content in peer-to-peer networks. In *Workshop on Advanced Video Streaming Techniques for Peer-to-Peer Networks and Social Networking*, 2010.

[91] P. Eittenberger, K. Schneider, and U. Krieger. Performance evaluation of next generation content delivery proposals. In *7th International Conference on Next Generation Mobile Apps, Services and Technologies (NGMAST)*, 2013.

[92] R. Finlayson. A more loss-tolerant RTP payload format for MP3 audio. RCF 3119, 2001.

[93] M. Freedman and R. Morris. Tarzan: A peer-to-peer anonymizing network layer. In *9th ACM Conference on Computer and Communications Security*, 2002.

[94] A. Gelman, S. Halfin, and W. Willinger. On buffer requirements for store-and-forward video on demand service circuits. In *Global Telecommunications Conference (GLOBECOM)*, 1991.

[95] P. Glatz, L. Hormann, C. Steger, and R. Weiss. A system for accurate characterization of wireless sensor networks with power states and energy harvesting system efficiency. In *Pervasive Computing and Communications Workshops (PERCOM)*, 2010.

[96] D. Goldschlag, M. Reed, and P. Syverson. Onion routing. *Communications of the ACM*, 42(2):39–41, 1999.

[97] V. Grishchenko et al. On the design of a practical information-centric transport. Technical report, TUDelft, the Netherlands, PDS-2011-006, 2011.

[98] V. Grishchenko and J. Pouwelse. Binmaps: Hybridizing bitmaps and binary trees. Technical report, TUDelft, the Netherlands, PDS-2011-005, 2011.

[99] Y. Guo, S. Mathur, K. Ramaswamy, S. Yu, and B. Patel. PONDER: performance aware P2P video-on-demand service. In *Global Telecommunications Conference (GLOBECOM)*, 2007.

[100] X. Hei, Y. Liu, and K. Ross. IPTV over P2P streaming networks: the mesh-pull approach. *Communications Magazine*, 46(2):86–92, 2008.

[101] P. Hoffman. Elliptic Curve Digital Signature Algorithm (DSA) for DNSSEC, RFC6605, 2012.

[102] L. Hormann, P. Glatz, K. Hein, and R. Weiss. State-of-charge measurement error simulation for power-aware wireless sensor networks. In *Wireless Communications and Networking Conference (WCNC)*, 2012.

[103] C. Huang, J. Li, and K. Ross. Can internet video-on-demand be profitable? *ACM SIGCOMM Computer Communication Review*, 37(4):133–144, 2007.

[104] Q. Huang, H. Jin, and X. Liao. P2P live streaming with tree-mesh based hybrid overlay. In *International Conference on Parallel Processing Workshops (ICPPW)*, 2007.

[105] T. Isdal, M. Piatek, A. Krishnamurthy, and T. Anderson. Privacy-preserving P2P data sharing with OneSwarm. In *ACM SIGCOMM Computer Communication Review*, volume 40, 2010.

[106] ISO/IEC 23009-1:2014. *Information technology – Dynamic adaptive streaming over HTTP (DASH) – Part 1: Media presentation description and segment formats*. ISO, Geneva, Switzerland.

[107] J. Jansen. Use of SHA-2 algorithms with RSA in DNSKEY and RRSIG resource records for DNSSEC, RFC5702, 2009.

[108] G. Jesi, A. Montresor, and M. van Steen. Secure peer sampling. *Computer Networks*, 54(12):2086–2098, 2010.

[109] X. Jiang, Y. Dong, D. Xu, and B. Bhargava. GnuStream: a P2P media streaming system prototype. In *International Conference on Multimedia and Expo (ICME).*, 2003.

[110] R. Jimenez, F. Osmani, and B. Knutsson. Sub-second lookups on a large-scale Kademlia-based overlay. In *11th International Conference on Peer-to-Peer Computing (P2P)*, 2011.

[111] Kamm and Baird. John Logie Baird: A Life.

[112] S. Le Blond, C. Zhang, A. Legout, K. Ross, and W. Dabbous. I know where you are and what you are sharing: exploiting P2P communications to invade users' privacy. In *Conference on Internet Measurement (SIGCOMM)*, 2011.

[113] T.-C. Lee, P.-C. Liu, W.-L. Shyu, and C.-Y. Wu. Live video streaming using P2P and SVC. In *Management of Converged Multimedia Networks and Services*, 2008.

[114] B. Li, S. Xie, Y. Qu, G. Keung, C. Lin, J. Liu, and X. Zhang. Inside the new Coolstreaming: Principles, measurements and performance implications. In *International Conference on Computer Communications (INFOCOM)*, 2008.

[115] W. Liang, R. Wu, J. Bi, and Z. Li. PPStream characterization: measurement of P2P live streaming during Olympics. In *Symposium on Computers and Communications, (ISCC)*, July 2009.

[116] K. Lin, N. Liu, and X. Luo. An optimized P2P based algorithm using SVC for media streaming. In *3rd International Conference on Communications and Networking in China (ChinaCom)*, 2008.

[117] Z. Liu, Y. Shen, K. Ross, S. Panwar, and Y. Wang. Layerp2p: using layered video chunks in P2P live streaming. *Transactions on Multimedia*, 11(7):1340–1352, 2009.

[118] S. Martello and P. Toth. *Knapsack problems*. Wiley New York, 1990.

[119] D. McCoy, K. Bauer, D. Grunwald, T. Kohno, and D. Sicker. Shining light in dark places: Understanding the Tor network. In *Privacy Enhancing Technologies*, 2008.

[120] A. Menezes, P. Van Oorschot, and S. Vanstone. *Handbook of applied cryptography.* CRC press, 2010.

[121] R. Merkle. *Secrecy, Authentication, and Public Key Systems.* PhD thesis, Stanford, USA, 1979.

[122] M. Meulpolder, L. D'Acunto, M. Capota, M. Wojciechowski, J. Pouwelse, D. Epema, and H. Sips. Public and private BitTorrent communities: a measurement study. In *International workshop on Peer-To-Peer Systems (IPTPS)*, 2010.

[123] S. Mirshokraie and M. Hefeeda. Live peer-to-peer streaming with scalable video coding and networking coding. In *Conference on Multimedia systems (SIGMM)*, 2010.

[124] K. Mokhtarian and M. Hefeeda. Efficient allocation of seed servers in peer-to-peer streaming systems with scalable videos. In *17th International Workshop on Quality of Service (IWQoS)*, 2009.

[125] J. Mol, J. Pouwelse, M. Meulpolder, D. Epema, and H. Sips. Give-to-get: Free-riding-resilient video-on-demand in P2P systems. In *Multimedia Computing and Networking (MMCN)*, 2008.

[126] J. D. Mol. *Free-riding Resilient Video Streaming in Peer-to-Peer Networks.* PhD thesis, TUDelft, The Netherlands, January 2010.

[127] S. Nielson and D. Wallach. The BitTorrent Anonymity Marketplace. *arXiv preprint arXiv:1108.2718*, 2011.

[128] M. Nunes, J. Taveira, G. Yingjie, J. Xia, and R. Cruz. PPSP Tracker Protocol–Base Protocol (PPSP-TP/1.0), Jan. 2012. IETF draft.

[129] F. Osmani, V. Grishchenko, R. Jimenez, and B. Knutsson. Swift: The missing link between peer-to-peer and information-centric networks. In *First Workshop on P2P and Dependability (P2P-Dep)*, 2012.

[130] R. Pantos and W. May. Apple HTTP live streaming, Apr. 2014. IETF draft.

[131] V. Paxson and M. Allman. Computing TCP's retransmission timer, RFC2988, 2000.

[132] R. Petrocco, C.-P. Bezemer, J. Pouwelse, and D. Epema. Libswift: the PPSPP reference implementation. Technical report, TUDelft, the Netherlands, PDS-2014-003, 2014.

[133] R. Petrocco, M. Capotă, J. Pouwelse, and D. Epema. Hiding user content interest while preserving P2P performance. In *Annual ACM Symposium on Applied Computing (ACM)*, 2014.

[134] R. Petrocco, M. Eberhard, J. Pouwelse, and D. Epema. Deftpack: A robust piece-picking algorithm for scalable video coding in P2P systems. In *International Symposium on Multimedia (ISM)*, 2011.

[135] R. Petrocco, J. Pouwelse, and D. Epema. Performance analysis of the Libswift P2P streaming protocol. In *12th International Conference on Peer-to-Peer Computing (P2P)*, 2012.

[136] S. Prusty, B. Levine, and M. Liberatore. Forensic investigation of the OneSwarm anonymous filesharing system. In *Conference on Computer and Communications Security*, 2011.

[137] R. Rahman. *Peer-to-Peer System Design: A Socio-economic Approach*. PhD thesis, TUDelft, The Netherlands, September 2011.

[138] M. Reiter and A. Rubin. Crowds: Anonymity for web transactions. *Transactions on Information and System Security (TISSEC)*, 1(1):66–92, 1998.

[139] R. Rejaie and A. Ortega. PALS: peer-to-peer adaptive layered streaming. In *International workshop on Network and Operating Systems Support for Digital Audio and Video*, 2003.

[140] D. Rossi, C. Testa, and S. Valenti. Yes, we LEDBAT: Playing with the new BitTorrent congestion control algorithm. In *Passive and Active Measurement*, 2010.

[141] G. Sanchez. Libswift-PPSPP information centric router: SHA1 accelerator. Master's thesis, TUDelft, August 2013.

[142] S. Saroiu, P. Gummadi, and S. Gribble. Measurement study of peer-to-peer file sharing systems. In *Electronic Imaging*, 2001.

[143] T. Schierl and S. Wenger. Signaling media decoding dependency in the session description protocol. RCF 5538, 2009.

[144] J. Schiller. Strong security requirements for Internet Engineering Task Force standard protocols, RFC3365, 2002.

[145] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: a transport protocol for real-time applications, RFC3550, 2003.

[146] H. Schulzrinne, A. Rao, and R. Lanphier. Real Time Streaming Protocol (RTSP), RFC2326, 1998.

[147] H. Schwarz, D. Marpe, and T. Wiegand. Overview of the scalable video coding extension of the H. 264/AVC standard. *Transactions on Circuits and Systems for Video Technology*, 17(9):1103–1120, 2007.

[148] A. Sentinelli, L. Celetto, D. Lefol, C. Palazzi, G. Pau, T. Zahariadis, and A. Jari. Survey on P2P overlay streaming clients. In *Future Internet Assembly*, 2009.

[149] S. Shalunov, G. Hazel, J. Iyengar, and M. Kuehlewind. Low Extra Delay Background Transport (LEDBAT), RFC6817, 2012.

[150] T. Stockhammer, M. Hannuksela, and T. Wiegand. H.264/AVC in wireless environments. *Transactions on Circuits and Systems for Video Technology*, 13(7):657–673, 2003.

[151] C. Testa and D. Rossi. On the impact of uTP on BitTorrent completion time. In *International Conference on Peer-to-Peer Computing (P2P)*, 2011.

[152] M. Uitto and J. Vehkaperä. Spatial enhancement layer utilisation for SVC in base layer error concealment. In *International Mobile Multimedia Communications Conference (ICST)*, 2009.

[153] A. Vlavianos, M. Iliofotou, and M. Faloutsos. BiToS: Enhancing BitTorrent for supporting streaming applications. In *International Conference on Computer Communications (INFOCOM)*, 2006.

[154] R. Walls, B. Levine, M. Liberatore, and C. Shields. Effective digital forensics research is investigator-centric. In *USENIX Workshop on Hot Topics in Security (HotSec)*, 2011.

[155] C. Wang, L. Xiao, Y. Liu, and P. Zheng. Distributed caching and adaptive search in multilayer P2P networks. In *International Conference on Distributed Computing Systems*, 2004.

[156] J. Wang et al. New efficient replacement strategies for P2P cooperative proxy cache systems. Citeseer. 2004.

[157] A. Wierzbicki, N. Leibowitz, M. Ripeanu, and R. Woźniak. Cache replacement policies for P2P file sharing protocols. *European Transactions on Telecommunications*, 15(6):559–569, 2004.

[158] D. Wing, P. Matthews, J. Rosenberg, and R. Mahy. Session Traversal Utilities for (NAT)(STUN), RFC5389, 2008.

[159] C. K. Wong and S. S. Lam. Digital signatures for flows and multicasts. In *International Conference on Network Protocols*, 1998.

[160] D. Wu, Y. Liu, and K. Ross. Queuing network models for multi-channel P2P live streaming systems. In *International Conference on Computer Communications (INFOCOM)*, 2009.

[161] J. Wu and B. Li. Keep cache replacement simple in peer-assisted vod systems. In *International Conference on Computer Communications (INFOCOM)*, 2009.

[162] W. Wu and J. Lui. Exploring the optimal replication strategy in P2P-VoD systems: Characterization and evaluation. *Transactions on Parallel and Distributed Systems*, 23(8):1492–1503, 2012.

[163] B. Zhang. *Understanding Operation and User Behavior in Peer-to-Peer Systems.* PhD thesis, TUDelft, The Netherlands, January 2013.

[164] C. Zhang, P. Dhungel, D. Wu, and K. Ross. Unraveling the BitTorrent ecosystem. *Transactions on Parallel and Distributed Systems*, 22(7):1164–1177, 2011.

[165] M. Zink, O. Künzel, J. Schmitt, and R. Steinmetz. Subjective impression of variations in layer encoded videos. In *Quality of Service (IWQoS)*, 2003.

Summary

Video Streaming is nowadays the Internet's biggest source of consumer traffic. Content providers are constantly improving their distribution infrastructure to satisfy the request for higher quality video streams. Traditional content distribution networks rely on the standard client-server approach, but it involves severe costs. As the current generation is moving from being passive viewers, or content consumers, to active content producers, the traditional content distribution infrastructure needs to change and adapt. Peer-to-Peer systems have proven to be a valuable solution for distributing files to a wide audience, allowing users to share their content directly from their home. The reason for the widespread adoption of P2P systems has been the capacity of splitting the content into small chunks, and distributing the rarest one first. This approach allows for a homogeneous availability of the content in the P2P system, allowing a fast retrieval by its users. However, this mechanism does not fit the needs of video streaming, and it must be adapted in order to efficiently retrieve time-sensitive content. Improving P2P streaming systems to effectively distribute video streaming content is the main focus of this thesis.

In Chapter 2 we present a framework for distributing scalable video content over P2P networks. One of the main challenges of today's video distribution systems, is providing the right video quality to the right audience, considering the diversity of devices and connections. Layered video content can address this challenge by providing the content in different qualities within a single bitstream. This chapter describes the architecture of the NextShare P2P streaming system, designed to support the distribution and consumption of scalable content in a fully distributed P2P network. We describe both the producer- and consumer-side architecture, including the encoding and decoding of the content. Finally, we provide an evaluation of the system's performance in different scenarios.

In Chapter 3 we discuss Deftpack, an improved self-adapting download algorithm specifically crafted to retrieve scalable video content. We describe its design, and show how Deftpack enhances the users' Quality of Experience by reducing the playback start-up time and the number of stalls during the video playback, outperforming available alternatives.

In Chapter 4 we present the Peer-to-Peer Streaming Peer Protocol, PPSPP, a novel protocol recently accepted on the standard track at the IETF as RFC7574. PPSPP has been specifically designed to distribute multimedia content in a P2P fashion, and to overcome the limitations of existing P2P protocols. PPSPP is a content-centric multi-party transport protocol that has been designed with flexibility and lightness in mind. It allows for a short time till playback, a small message-passing overhead, and defines a novel data structure that reduces the per-connection cost. Also, it provides NAT/Firewall traversal, and prevents malicious attacks by other peers. In Chapter 4 we only report the most essential sections directly ex-

tracted from the official IETF standard, while in Chapter 5 we present Libswift, its reference implementation. Chapter 5 reports the technical details of the PPSPP protocol that are not covered by the official protocol description, and details of the implementation that are left to the developer. We discuss features and algorithm that are at the core of Libswift, and analyse its behaviour in network challenging environments.

In Chapter 6 we thoroughly investigate the performance of Libswift, comparing it with the mainstream BitTorrent protocol, both on high-end and power-limited devices. We present some enhancements on the upload and download policy of Libswift, a needed addition that improves its performance in streaming environments. In the past, much work has gone in adapting BitTorrent-like protocols to fit the needs of video streaming. This chapter provides a valuable insight into the benefits of designing a a protocol from the ground up with streaming in mind. We compare Libswift with the two major BitTorrent clients, uTorrent and Libtorrent, and with the YouTube Android application.

In Chapter 7, we present a novel solution for hiding the user's content of interest on a P2P network. Our solution can be implemented on top of existing P2P networks, and, while it increases the users' privacy, it does not significantly affect their performance. In a P2P network, peers need to announce what they have already retrieved to their neighbour, hence traditional P2P systems are notorious for leaking sensitive information. Existing privacy-aware solutions are characterised by an inherent performance drawback, as they try to achieving complete anonymity, making users reluctant to join. On the other hand, our solution does not try to guarantee complete anonymity, but rather plausible deniability. In this chapter we describe the techniques we use for archiving this result, and provide an analysis of possible attacks. Furthermore, using actual P2P software, we show that our solution has very little impact on the system's performance.

Finally, in Chapter 8, we present our conclusion, a reflection on the value of research on P2P considering the recent decline of interest, and suggestions for future work.

Samenvatting

Video streaming representeert tegenwoordig het meeste internet-verkeer van consumenten. Leveranciers van media verbeteren continu hun distributie-infrastructuur om te voldoen aan de vraag naar video streams van hogere kwaliteit. Traditioneel is het distributienetwork gebaseerd op een client-server model, maar dit brengt hoge kosten met zich mee. Met de verandering van de huidige generatie van passieve kijkers naar media-producenten, bestaat er de noodzaak voor verandering en aanpassing van de distributie-infrastructuur. Peer-to-Peer systemen hebben bewezen een waardevolle oplossing te zijn om bestanden te distribueren naar een breed publiek, zodat gebruikers hun media direct vanaf thuis kunnen delen. De reden voor het wijd verspreide gebruik van P2P systemen is de mogelijkheid om de media in kleine stukken op te splitsen, en de zeldzaamste stukken het eerst te verspreiden. Deze aanpak zorgt voor een homogene beschikbaarheid van de media in het P2P systeem, waardoor gebruikers deze op een snelle manier kunnen ophalen. Dit mechanisme is echter niet geschikt voor video streaming, en moet aangepast worden om tijdsafhankelijke fragmenten efficiënt te verkrijgen. De verbetering van P2P systemen voor de effectieve distributie van streaming video is het onderwerp van deze proefschrift.

In Hoofdstuk 2 presenteren we een structuur voor systemen voor de schaalbare distributie van video media over P2P netwerken. Een van de voornaamste uitdagingen van hedendaagse video distributiesystemen is het leveren van de juiste video kwaliteit voor de juiste gebruikers met het oog op de diversiteit van apparatuur en netwerkverbindingen. Gelaagde video media beantwoordt aan deze uitdaging door verschillende kwaliteiten van de media in een enkele bitstream op te nemen. Dit hoofdstuk beschrijft de architectuur van het NextShare P2P streaming systeem, dat ontworpen is voor de distributie van schaalbare media in een volledige gedistribueerd P2P netwerk. We beschrijven de architectuur van de kant van zowel de producent als de consument, inclusief het coderen en decoderen van de media. Tenslotte presenteren we een evaluatie van de prestaties van het systeem in verschillende scenario's.

In Hoofdstuk 3 bespreken we Deftpack, een verbeterd, zelf-adaptief download-algoritme dat speciaal gemaakt is om schaalbare videos over te sturen. We beschrijven het ontwerp, en tonen aan dat Deftpack de kwaliteit van de gebruikerservaring verbetert door de opstarttijd en het aantal onderbrekingen van videos te reduceren, waarmee het beter is dan alternatieve oplossingen.

In Hoofdstuk 4 presenteren we het Peer-to-Peer Streaming Peer Protocol (PPSPP), een nieuw protocol dat recent als standaard is geaccepteerd door de IETF als RFC7574. PPSPP is speciaal ontworpen voor de distributie van multimedia op een P2P-manier en om de beperkingen van bestaande P2P protocollen te overkomen. PPSPP is een inhoud-centrisch, flexibel en lichtgewicht transport-protocol tussen meerdere partijen. Het zorgt voor korte opstarttijden, een kleine overhead bij het versturen van berichten, en het definieert een nieuwe datastructuur die de kosten per verbinding reduceert. Bovendien biedt het NAT/Firewall traversal en voorkomt het kwaadaardige aanvallen door andere gebruikers. In dit hoofdstuk beschrijven we alleen de meest essentiële onderdelen, direct genomen uit de officiale IETF standaard, terwijl we in Hoofdstuk 5 Libswift, de referentie implementatie van PPSPP presenteren. Hoofdstuk 5 beschrijft de technische details van het PPSPP protocol die niet gespecificeerd zijn in het officiële protocol, en details die aan de ontwikkelaar worden overgelaten. We bespreken eigenschappen en algoritmen in de kern van Libswift, en we analyseren het gedrag van Libswift in uitdagende netwerk-omgevingen.

In Hoofdstuk 6 onderzoeken we grondig de prestaties van Libswift en vergelijken het met het bekende BitTorrent protocol, beide op apparaten met veel en weining voorzieningen. We presenteren enkele verbeteringen in de upload- en download-algorithmen van Libswift, wat een welkome toevoeging om de prestaties te verbeteren in streaming-omgevingen. In het verleden is veel werk is gedaan om BitTorrent-achtige protocollen aan te passen aan de behoeften van video streaming. Dit hoofdstuk biedt een waardevolle inzicht in de voordelen om een heel nieuw protocol te ontwerpen voor video streaming. We vergelijken Libswift met twee belangrijke BitTorrent cliënten, uTorrent en Libtorrent, en met YouTube onder Android.

In Hoofdstuk 7 presenteren we een nieuwe oplossing om de media-interesses van de gebruiker te verbergen in een P2P netwerk. Onze oplossing kan worden toegepast in bestaande P2P netwerken, en terwijl het de privacy van de gebruiker vergroot heeft het nauwelijks invloed op de prestaties. In een P2P netwerk moeten peers aan hun buren aankondigen wat ze al opgehaald hebben, en hierdoor zijn traditionele P2P systemen berucht voor het lekken van gevoelige informatie. Bestaande privacy-bewuste oplossingen worden gekarakteriseerd door een verslechtering in prestaties aangezien ze volledige anonymiteit willen bereiken, waardoor gebruikers er niet graag gebruik van maken. Aan de andere kant probeert onze oplossing geen volledige anonymiteit te bieden, alleen geloofwaardige ontkenning. In dit hoofdstuk beschrijven we technieken om deze resultaten te bereiken, en bieden we een analyse van mogelijke aanvallen. Bovendien laten we met een implementatie zien dat onze oplossing nauwelijks invloed heeft op de presaties van het systeem.

Tenslotte presenteren we in Hoofdstuk 8 onze conclusies, een reflectie op de waarde van P2P-onderzoek gezien de recente afname in interesse, en suggesties voor toekomstig werk.

Curriculum Vitæ

Riccardo Petrocco was born in Rome, Italy, on July 26th, 1983. After received his BSc degree in Computer Science at Roma Tre University, he enrolled at Delft University of Technology in 2006. There he received his MSc in Computer Engineering in 2008. During his MSc thesis project, he developed a system for encoding and distributing multi bit-rate video streaming content over a P2P network. The results of this project led him to join the Parallel and Distributed Systems Group of Delft University of Technology in 2009 as a researcher and scientific developer. In 2011, he started his PhD on peer-to-peer streaming networks. The work presented in this thesis is based on the research carried out between 2011 and 2014.

Riccardo is currently working as a data engineer for Spotify in Stockholm. His free time is dedicated to his partner, Victoria, to his friends and to his numerous interests. He has a strong passion for motorcycles, and when he can not ride them in the exploration of the world, he spends his time restoring old motorcycles from the 60s and 70s. Other passions include climbing, photography and music. Above all, he enjoys new experiences and adventures, whether travelling or by meeting new people.

Publications of the author

- M. Foivos, G. Kreitz, R. Petrocco, B. Zhang, J. Widmer. Passive mobile bandwidth classification using short lived TCP connections, In *8th Wireless and Mobile Networking Conference*, 2015.
- R. Petrocco, M. Capotă, J. Pouwelse, D. Epema. Hiding user content interest while preserving P2P performance. In *Annual ACM Symposium on Applied Computing*, 2014, Best paper award.
- R. Petrocco, C.P. Bezemer, J. Pouwelse, D. Epema. Libswift: the PPSPP reference implementation, *Technical report PDS-2014-003, Delft University of Technology*, 2014.
- R. Petrocco, J. Pouwelse, D. Epema. Performance analysis of the Libswift P2P streaming protocol, In *12th International Conference on Peer-to-Peer Computing*, 2012.
- M. Eberhard, A. Palo, A. Kumar, R. Petrocco, L. Mapelli, M. Uitto. NextSharePC: an open-source BitTorrent-based P2P client supporting SVC, In *3rd Multimedia Systems Conference*, 2012.
- R. Petrocco, M. Eberhard, J. Pouwelse, D. Epema. Deftpack: A robust piece-picking algorithm for scalable video coding in P2P systems, In *Symposium on Multimedia*, 2011.
- M. Eberhard, A. Kumar, S. Mignanti, R. Petrocco, M. Uitto. A framework for distributing scalable content over Peer-to-Peer networks, In *International Journal On Advances in Internet Technology*, 40(1 and 2):1-13, 2011.
- N. Capovilla and M. Eberhard and S. Mignanti and R. Petrocco and J. Vehkapera. An Architecture for distributing scalable content over Peer-to-Peer networks. In *2nd International Conferences on Advances in Multimedia*, 2010, Best paper award.
- A. Bakker, R. Petrocco, M. Dale, J. Gerber, V. Grishchenko, D. Rabaioli, J. Pouwelse. Online video using BitTorrent and HTML5 applied to Wikipedia, In *10th International Conference on Peer-to-Peer Computing*, 2010.